Light at the end of the tunnel

Synthesis-based engineering for road tunnels

Lars Moormann

Department of Mechanical Engineering
EINDHOVEN UNIVERSITY OF TECHNOLOGY
Eindhoven, The Netherlands 2022

EINDHOVEN
UNIVERSITY OF
TECHNOLOGY

The work described in this thesis was carried out at the Eindhoven University of Technology and has been financially supported by Rijkswaterstaat.

A catalogue record is available from the Eindhoven University of Technology Library.
ISBN: 978-90-386-5626-7

Typeset using LaTeX.
Reproduction: ADC Dereumaux
Cover photo: Eerste Heinenoordtunnel [beeldbank.rws.nl, Rijkswaterstaat]

Light at the end of the tunnel
Synthesis-based engineering for road tunnels

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische
Universiteit Eindhoven, op gezag van de rector magnificus prof.dr.ir.
F.P.T. Baaijens, voor een commissie aangewezen door het College
voor Promoties, in het openbaar te verdedigen op
donderdag 15 december 2022 om 16:00 uur

door

Lars Moormann

geboren te Heerlen

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

voorzitter:	prof.dr.ir. A.A. van Steenhoven
1^e promotor:	prof.dr. W.J. Fokkink
2^e promotor:	dr.ir. L.F.P. Etman
copromotor:	dr.ir. J.M. van de Mortel-Fronczak
leden:	prof.dr. J.E.R. Cury (Federal University of Santa Catarina)
	dr. R.J. Leduc (McMaster University)
	dr.ir. T.A.C. Willemse
	prof.dr.ir. J.P.M. Voeten
adviseur:	ing. P. Maessen (Rijkswaterstaat)

Het onderzoek dat in dit proefschrift wordt beschreven is uitgevoerd in overeenstemming met de TU/e Gedragscode Wetenschapsbeoefening.

Abstract

Road tunnels play an important role in the traffic infrastructure in the Netherlands, either to bypass geological obstacles such as rivers, or to improve traffic flow and quality of life in urban areas. Rijkswaterstaat (RWS), the executive branch of the Dutch Ministry of Infrastructure and Water Management, responsible for the construction and maintenance of these road tunnels, is planning to renovate several existing ones. Part of such a renovation pertains to the supervisory controller of the tunnel. The functions of this supervisory controller are to monitor the situation to detect an emergency and to correctly handle it. Road tunnels are large and intricate systems that consist of many components, such as smart lighting, ventilation, evacuation systems, fire extinguishing tools, and flood prevention systems, which makes the design of their supervisory controllers a difficult and error-prone task. Furthermore, a road tunnel is at all times monitored by a human operator through a command interface. As commands given through this interface must be correctly carried out by the supervisory controller, this interface should be correctly integrated in the supervisory controller design.

The aim of the research described in this thesis is to investigate the applicability of supervisory control theory (SCT) for the design of supervisory controllers for road tunnels. SCT enables the design engineer to automatically derive a supervisory controller, which reduces the realization time and eliminates human errors. This procedure, called supervisor synthesis, uses as input a model of the system's behavior, called the plant model, and a model of the desired behavior, called the requirements model, and produces a supervisory controller that is correct-by-construction with respect to the models. Furthermore, a formal (synthesized) model of the supervisory controller supports the automatic generation of implementation code. Using SCT may decrease the time-to-market and increase the evolvability of the supervisory controller, while preserving a high quality.

This project focuses on the complete process of supervisory controller design, including the steps for modeling, supervisor synthesis, simulation, and implementation. Furthermore, it investigates the characteristics of road tunnels and how those characteristics could be exploited in the supervisory controller design process. In this context, the following five main contributions are made.

First, modeling using SCT is a cumbersome task for systems with large numbers of components, and requires extensive knowledge on SCT, e.g. regarding modeling formalisms and synthesis algorithms. To facilitate this modeling task, a parameter-based modeling approach is proposed that enables a designer to automatically generate the required models for synthesis and simulation by defining the parameters of a road tunnel. The gained benefits include more efficient modeling while requiring less knowledge on SCT. The parameter-based modeling platform is implemented as a prototype configuration tool and validated by means of simulation. Its applicability is shown in a case study for a family of 22 tunnels in the Netherlands.

Second, the complexity of the supervisor synthesis problem grows exponentially with the number of system components and the number of requirement models of this system. For large-scale systems, such as road tunnels, this may result in a lengthy or even unsolvable synthesis procedure. In this thesis, a new method is proposed for reducing the model of the system before synthesis to decrease the required computational time and effort. The method consists of five steps for model reduction, that are mainly based on symmetry in dependency graphs of the system. Dependency graphs visualize the components in the system and the relations between these components. In a case study for the Eerste Heinenoord tunnel, the model can be reduced to an 80% smaller model before synthesis, which made the initially unsolvable synthesis problem solvable in two minutes.

Third, in practice, supervisory controllers are often not implemented on a single programmable logic controller (PLC), but on a set of PLCs. Such a distributed implementation can improve PLC performance, can reduce maintenance and renovation effort, and can decrease the required amount of cables, but introduces communication between PLCs that may be delayed in time. In this thesis, a novel method is proposed to distribute a synthesized supervisor for implementation on multiple physical controllers. Dependency structure matrices are used to determine an optimal distribution of a discrete-event system, and the supervisor is distributed accordingly using an existing localization method. Communication delays between the distributed components of a supervisor may affect its behavior, due to changes in the order of events. Therefore, a delay-robustness check needs to be performed and where needed mutex locks are employed to make the distributed supervisor delay robust. The controller performance is analyzed and a general performance optimization is achieved through a parameter study and a mutex implementation evaluation.

Fourth, a synthesized supervisor is not directly implementable on a hardware platform, often a PLC. To do this, the properties for an implementable controller need to be verified, the supervisory controller model needs to be translated to PLC code, and interface models need to be created to allow validation through hardware-in-the-loop (HIL) simulation. In this thesis, several contributions are made relating to the implementation process of supervisory controllers, including the design and implementation of resource controllers, a relaxation in the property check for implementable controllers, and an algorithm for automatic PLC code generation that is adaptable for any desired target implementation platform. Furthermore, it is shown how the integration of a digital twin, which is a digital copy of the physical system, in HIL simulation allows for more intuitive and extensive validation procedures.

Fifth, several real-life case studies are performed for road tunnels in the Netherlands, including the Koning Willem-Alexander tunnel, the Eerste Heinenoord tunnel, and the Swalmen tunnel. These case studies have shown the applicability of SCT for the design of road tunnel supervisory controllers, and demonstrated the effectiveness of the aforementioned contributions.

Samenvatting

Tunnels spelen een belangrijke rol in het wegennetwerk van Nederland. Zo worden ze gebruikt om geologische obstakels als rivieren te passeren, of om verkeersstromen en levenskwaliteit te verhogen in stedelijke gebieden. Rijkswaterstaat (RWS), de uitvoerende tak van het Nederlandse Ministerie van Infrastructuur en Waterstaat, is verantwoordelijk voor het bouwen en onderhouden van deze tunnels, en is van plan om veel van de bestaande tunnels in Nederland te renoveren. Een deel van zo'n renovatieproject beslaat het vernieuwen van het besturingssysteem van de tunnel. De functies van dit besturingssysteem zijn het monitoren van de situatie in de tunnel om een calamiteit te detecteren, en om gedetecteerde calamiteiten correct af te handelen. Tunnels zijn grote en ingewikkelde systemen die bestaan uit vele componenten, zoals slimme verlichting, ventilatie, evacuatiesystemen, brandblusapparaten, en overstromingspreventiesystemen, die het ontwerp van het besturingssysteem moeilijk en foutgevoelig maken. Daarnaast wordt een tunnel te allen tijde gecontroleerd door een menselijke bedienaar die gebruik maakt van een bedienscherm. Het koppelvlak met dit bedienscherm moet correct meegenomen worden in het ontwerp van het besturingssysteem.

Het doel van dit onderzoek is om de toepasbaarheid van supervisory control theory (SCT) voor het ontwerp van tunnel-besturingssystemen te bekijken. Met SCT kunnen ontwerpers automatisch een besturingssysteem genereren, waardoor de realisatietijd en de door de mens gemaakte fouten verminderd worden. Dit generatieproces, dat supervisor synthese wordt genoemd, gebruikt als input een model van het mogelijke gedrag, genaamd het plant-model, en een model van het gewenste gedrag, genaamd het eisen-model. Synthese levert een besturingssysteem op dat per constructie correct is ten opzichte van de geleverde modellen. Daarnaast zorgt het formele (gesynthetiseerde) model van het besturingssysteem ervoor dat automatische generatie van implementatiecode mogelijk is. Door gebruik te maken van SCT kan de ontwikkeltijd van het besturingssysteem verlaagd worden, kan de herbruikbaarheid verhoogd worden, terwijl een hoge kwaliteit behouden wordt.

Dit onderzoek focust zich op het gehele ontwerpproces van een besturingssysteem, inclusief het modelleren, supervisor-synthese, simulatie en implementatie. Daarnaast wordt onderzocht wat de specifieke kenmerken van tunnels zijn en hoe

die mogelijk benut kunnen worden in het ontwerpproces van het besturingssysteem. In dit onderzoek zijn de volgende vijf hoofdbijdragen geleverd.

Ten eerste is het modelleren voor SCT een moeizame taak voor systemen die bestaan uit veel componenten, en is er veel kennis nodig van SCT, met name met betrekking tot modelleerformalismen en synthese-algoritmen. Om dit modelleerproces te vereenvoudigen is er een parameter-gebaseerde aanpak voorgesteld. Deze aanpak zorgt ervoor dat de ontwerper automatisch de benodigde modellen voor synthese en simulatie kan generen door de parameters van een bepaalde tunnel in te voeren. De voordelen die hiermee behaald worden zijn een efficiënter modelleerproces terwijl er minder kennis van SCT nodig is. Het parameter-gebaseerde raamwerk is geïmplementeerd als een prototype configuratieprogramma. Het programma is gevalideerd aan de hand van simulatie, en de toepasbaarheid van het programma is aangetoond door een industriële toepassing te laten zien voor een familie van 22 tunnels in Nederland.

Ten tweede neemt de complexiteit van supervisor-synthese exponentieel toe wanneer het aantal componenten in het systeem toeneemt. Voor grootschalige systemen, zoals tunnels, kan dit resulteren in een langdurige en soms zelfs onoplosbare synthese-procedure. In dit onderzoek is een nieuwe methode opgesteld om het model van het systeem te reduceren voordat synthese wordt toegepast om de rekentijd en -moeite terug te brengen. Deze methode bestaat uit vijf stappen voor modelreductie, die met name gebaseerd zijn op afhankelijkheidsgrafen. Een afhankelijkheidsgraaf visualiseert de componenten van het systeem en de relaties tussen deze componenten. In een industriële toepassing voor de Eerste Heinenoordtunnel mochten meer dan 80% van de modellen weggehaald worden voordat synthese werd uitgevoerd, waardoor het oorspronkelijk onoplosbare syntheseprobleem in 2 minuten oplosbaar is gemaakt.

Ten derde worden besturingssystemen in veel gevallen niet geïmplementeerd op één programmeerbare logische besturing (Eng. programmable logic controller, PLC), maar op een set van PLC's. Zo'n gedistribueerde implementatie kan de prestaties van de besturing verbeteren, kan de benodigde moeite voor onderhoud en renovatie verminderen en kan de vereiste hoeveelheid kabels reduceren. Een gedistribueerde implementatie zorgt er echter voor dat communicatie tussen de PLC's nodig is waarin tijdsvertragingen kunnen voorkomen. In dit onderzoek is een nieuwe methode opgesteld om een besturingssysteem te distribueren met het doel om het te implementeren op een set van PLC's. De methode maakt gebruik van 'Dependency Structure Matrices' (DSM's) om een verdeling van het systeem vast te stellen. Het besturingssysteem wordt daarna aan de hand van deze verdeling gelokaliseerd. Vertragingen in de communicatie tussen de PLC's kunnen het gedrag van het besturingssysteem beïnvloeden, aangezien de volgorde waarin signalen verstuurd worden kan veranderen. Om dit tegen te gaan wordt een vertragings-robuustheid controle uitgevoerd om uit te vinden welke delen van het besturingssysteem niet robuust tegen vertraging zijn. Bij die delen worden wederzijdse uitsluiting-sloten (Eng. mutex locks) toegevoegd om het gedistribueerde besturingssysteem toch robuust tegen vertraging te maken. De prestaties van de geïmplementeerde besturing zijn geanalyseerd en geoptimaliseerd door een parameteronderzoek en een evaluatie van de mutexlock-implementatie.

Ten vierde is een gesynthetiseerd besturingssysteem niet direct implementeerbaar op een PLC. Om dit wel te bewerkstelligen moeten de eigenschappen van een implementeerbaar besturingssysteem vastgesteld worden, het model van het besturingssysteem moet vertaald worden naar PLC code en koppelvlakmodellen moeten gemaakt worden om validatie aan de hand van hardware-in-de-loop (HIL) simulatie beschikbaar te maken. In dit onderzoek zijn verschillende bijdragen geleverd met betrekking tot het implementatieproces van een besturingssysteem. Dit betreft het ontwerp en de implementatie van de besturings- en/of regelsystemen van de fysieke componenten (Eng. resource controllers), een afname in striktheid in het vaststellen van één van de eigenschappen van een implementeerbaar besturingssysteem en een algoritme om automatisch PLC code te generen dat aanpasbaar is voor elk gewenst implementatieplatform. Daarnaast wordt de integratie van een Digital Twin in het ontwerpproces van het besturingssysteem beschreven. Zo'n Digital Twin zorgt ervoor dat het validatieproces met de HIL simulaties intuïtiever en uitgebreider wordt.

Ten vijfde zijn er meerdere industriële toepassingen gedemonstreerd voor verschillende tunnels in Nederland. Deze tunnels zijn de Koning Willem-Alexandertunnel, de Eerste Heinenoordtunnel, en de Swalmentunnel. Deze industriële toepassingen laten zien dat SCT toepasbaar is voor het ontwerp van een tunnelbesturingssysteem, en demonstreren de effectiviteit van de eerdergenoemde onderzoeksbijdragen.

Dankwoord

Dit proefschrift beschrijft het resultaat van de samenwerking van Rijkswaterstaat (RWS) en de Technische Universiteit Eindhoven (TU/e). In dit project, een samenwerking tussen overheid en universiteit, zijn methoden en theorieën ontwikkeld die bruikbaar zijn voor het besturen van tunnels. De combinatie van theoretisch onderzoek en praktische toepassing heeft mij vier boeiende jaren gebracht. Ik bedank Han Vogel en Pascal Etman, projectleiders van respectievelijk RWS en TU/e, voor de geboden mogelijkheid om dit project uit te voeren.

Patrick Maessen, mijn begeleider bij RWS, bedank ik voor de fijne samenwerking, al vanaf de eerste dag van mijn afstudeerproject in januari 2018. Hij heeft mij vertrouwd gemaakt met de verschillende facetten die een rol spelen bij het ontwerpen en het utiliseren van tunnels met hun besturing. Ook zijn deelname aan mijn promotiecommissie stel ik op prijs. Johan Naber bedank ik voor zijn kennis van tunnels, voor zijn bijdragen tijdens onze discussies en voor het organiseren van rondleidingen in tunnels.

Wan Fokkink, mijn 1^{e} promotor, bedank ik voor zijn enthousiaste betrokkenheid. Hij heeft mij geholpen om de resultaten van mijn onderzoek zowel intuïtief als wiskundig correct op te schrijven. Pascal Etman, mijn 2^{e} promotor, bedank ik voor zijn positieve en opbouwende opmerkingen. Hij heeft mij de waarde getoond van het plaatsen van onderzoek in een bredere context. Asia van de Mortel-Fronczak, mijn copromotor en dagelijkse begeleider, bedank ik voor alle begeleiding in de afgelopen jaren. Zij heeft mij onder andere geleerd hoe wetenschappelijke artikelen te schrijven, en gaf altijd constructieve opmerkingen op papers en presentaties. Koos Rooda, adviseur bij het project, bedank ik voor zijn hulp en inzet bij mijn project. Ik bedank hem met name voor de ondersteuning bij het structureren van mijn onderzoek en het paraat hebben van een passend spreekwoord of gezegde zal mij bijblijven. Albert Hofkamp bedank ik voor zijn ondersteuning bij het schrijven van wetenschappelijke algoritmen en voor het implementeren van de bijbehorende code.

I thank the other members of the doctorate committee, prof. José Cury, dr. Ryan Leduc, dr. Tim Willemse, and prof. Jeroen Voeten, for reading my thesis, providing useful feedback, and for taking part in this committee.

Ik bedank alle collega's waar ik de afgelopen jaren mee in het laboratorium heb gezeten. Martijn, Ferdie, Sjoerd en Jeroen bedank ik voor de fijne samenwerking in de RWS projecten. Onze discussies en jullie hulp heb ik altijd op prijs gesteld. Aida, Fabio, Joshua, Nick, Maurizio, Merlijne, Olaf, Sander en Thomas bedank ik voor de gezellige momenten in het laboratorium.

Ik dank alle studenten die ik tijdens het onderzoek heb begeleid voor hun werkzaamheden. De resultaten van hun onderzoek hebben bijgedragen aan dit project. Met name wil ik Jaap Schouten, Jeroen van Hegelsom en Julian Schokking bedanken voor hun bijdragen.

Mijn beste vrienden Bas, Daan, Boyan en Sander bedank ik voor alle avonden en weekenden die we samen door hebben gebracht met stapavonden, films, games, vakanties en andere activiteiten. De (ex-)teamgenoten van de volleybalverenigingen van Hajraa en Nuvo'68 bedank ik voor de gezellige momenten waarbij ik mijn energie kwijt kon tijdens trainingen en wedstrijden.

Ik bedank mijn familie ontzettend voor hun steun, niet alleen tijdens mijn onderzoek, maar al 28 jaar. Jacqueline en Frans in Limburg, Victor en Bregtje in Amsterdam, en mijn broers en zussen Celine, Estelle, Julius, Bente, Roos en Wout, bedank ik voor hun liefde, interesse en ondersteuning.

Ten slotte wil ik Karlijn Fransen bedanken voor alle liefde en betrokkenheid. We hebben elkaar leren kennen tijdens onze promotie projecten en terugkijkend ben ik heel gelukkig dat ik jou heb ontmoet. Ik hoop dat we samen nog heel veel mooie momenten mogen beleven.

Lars Moormann
November 2022

List of publications

Peer-reviewed journal contributions

Moormann, L., van de Mortel-Fronczak, J.M., Fokkink, W.J., Maessen, P., and Rooda, J.E. Supervisory control synthesis for large-scale systems with isomorphisms. In: *Control Engineering Practice*, 115:104902, 2021.

Moormann, L., Schouten, R.H.J., van de Mortel-Fronczak, J.M., Fokkink, W.J., and Rooda, J.E. Synthesis and implementation of distributed supervisory controllers with communication delays. In: *Transactions on Automation Science and Engineering.* Submitted.

Peer-reviewed conference contributions

Moormann, L., Maessen, P., Goorden, M.A., van de Mortel-Fronczak, J.M., and Rooda, J.E. Design of a tunnel supervisory controller using synthesis-based engineering. In: *ITA-AITES World Tunnel Congress, WTC2020 and 46th General Assembly*, pages 573–578, 2020.

Moormann, L., Goorden, M.A., van de Mortel-Fronczak, J.M., Fokkink, W.J., Maessen, P., and Rooda, J.E. Efficient validation of supervisory controllers using symmetry reduction. In: *15th IFAC Workshop on Discrete Event Systems (WODES)*, pages 288–295, 2020.

Moormann, L., van de Mortel-Fronczak, J.M., Fokkink, W.J., and Rooda, J.E. Exploiting symmetry in dependency graphs for model reduction in supervisor synthesis. In: *2020 IEEE 16th International Conference on Automation Science and Engineering (CASE)*, pages 659–666. IEEE, 2020.

Moormann, L., van de Mortel-Fronczak, J.M., and Rooda, J.E. Design of a parameter-based modeling platform for road tunnel supervisory controllers. In: *2021 IEEE Conference on Control Technology and Applications (CCTA)*, pages 1024–1030. IEEE, 2021.

Moormann, L., Schouten, R.H.J., van de Mortel-Fronczak, J.M., Fokkink, W.J., and Rooda, J.E. Synthesis and implementation of distributed supervisory controllers with communication delays. In: *2021 IEEE 17th International Conference on Automation Science and Engineering (CASE)*, pages 1268–1275. IEEE, 2021.

Moormann, L., van Hegelsom, J., van de Mortel-Fronczak, J.M., Maessen, P., Fokkink, W.J., and Rooda, J.E. Digital twins for the validation of road tunnel controllers. In: *ITA-AITES World Tunnel Congress, WTC2022 and 47th General Assembly.* 2022.

Moormann, L., Hofkamp, A.T., van de Mortel-Fronczak, J.M., Fokkink, W.J., and Rooda, J.E. Derivation and hardware-in-the-loop testing for a road tunnel controller. In: *16th IFAC Workshop on Discrete Event Systems (WODES)*, 2022.

Moormann, L., Etman, L.F.P., Fokkink, W.J., van de Mortel-Fronczak, J.M., and Rooda, J.E. Supervisory controller design for road tunnels using DSM techniques. In: *24th International DSM Conference*, 2022.

Goorden, M.A., Moormann, L., Reijnen, F.F.H., Verbakel, J.J., van Beek, D.A., Hofkamp, A.T., van de Mortel-Fronczak, J.M., Reniers, M.A., Fokkink, W.J., Rooda, J.E., and Etman, L.F.P. The road ahead for supervisor synthesis. In: *Proceedings of the 6th Symposium on Dependable Software Engineering*, pages 1-16. Springer, 2020.

Fokkink, W.J., Goorden, M.A., Hendriks, D., van Beek, D.A., Hofkamp, A.T., Reijnen, F.F.H, Etman, L.F.P., Moormann, L., van de Mortel-Fronczak, J.M., Reniers, M.A., Rooda, J.E., van der Sander, L.J., Schiffelers, R.R.H., Thuijsman, S.B., Verbakel, J.J., Vogel, J.A. Eclipse ESCET™: The Eclipse Supervisory Control Engineering Toolkit. In: *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2023. Submitted.

Non peer-reviewed conference contributions

Moormann, L., Goorden, M.A., van de Mortel-Fronczak, J.M., Rooda, J.E., Etman, L.F.P., and Maessen, P. Supervisory control synthesis for tunnels: The Koning Willem-Alexandertunnel case. In: *Proceedings of the 38th Benelux Meeting on Systems and Control*, pp 154, 2019.

Moormann, L., van de Mortel-Fronczak, J.M., Fokkink, W.J., and Rooda, J.E. Model reduction for supervisor synthesis. In: *Proceedings of the 39th Benelux Meeting on Systems and Control*, pp 82, 2020.

Moormann, L., van de Mortel-Fronczak, J.M., Fokkink, W.J., and Rooda, J.E. Hardware-in-the-loop testing for the Swalmen tunnel using automatically generated PLC code. In: *Proceedings of the 41st Benelux Meeting on Systems and Control*, pp 105, 2022.

Contents

Chapter 1

Introduction

Road tunnels play an important role in traffic infrastructure around the world. They are used to bypass geological obstacles such as rivers or mountains, to improve traffic flow in dense traffic networks, or to increase quality of life in urban areas. In the Netherlands, which is the country with the highest population density in Europe and the country with the densest waterway network, tunnels are commonly used in the traffic infrastructure.

A distinction is made between tunnels and underpasses: any underpass that is longer than 500 meters is identified as a tunnel, on which specific European safety guidelines are imposed regarding the safety and monitoring systems, see European Commission (2021). In the Netherlands, a length of 250 meters is used for this distinction, and each tunnel is at all times monitored by an individual human operator.

There are 27 tunnels in total throughout the Netherlands, of which 20 are state-owned, maintained, and operated by Rijkswaterstaat (RWS). An overview of these tunnels can be found at Rijkswaterstaat (2022). RWS is the executive branch of the Dutch Ministry of Infrastructure and Waterway management. The remaining 7 tunnels are smaller tunnels maintained by provinces or municipalities.

Besides the maintenance of existing tunnels, RWS is also responsible for the construction of the new ones. Currently, 4 new tunnels are planned for construction, see Rijkswaterstaat (2022). Figure 1.1 shows the Eerste Heinenoord tunnel, which is a tunnel near Rotterdam that bypasses the river the Oude Maas that is currently under renovation. An example of a tunnel that was recently constructed in the Netherlands is the Koning Willem-Alexander tunnel, shown in Figure 1.2. This tunnel passes underneath the city of Maastricht to increase traffic flow through the city, and improve the quality of life for the residents.

The function of a road tunnel is to allow road users to get from one end of the tunnel to the other in a safe environment. To this end, a tunnel consists of a civil structure, often underground, and a road surface. For the purpose of

Figure 1.1: The Eerste Heinenoord tunnel passing underneath the river the Oude Maas. Image from `https://beeldbank.rws.nl`.

Figure 1.2: The Koning Willem-Alexander tunnel in Maastricht, the Netherlands. Image from `https://www.rijkswaterstaat.nl/wegen/wegenoverzicht/a2/koning-willem-alexandertunnel-a2-n2`.

maintaining a safe environment, the tunnel contains technical installations, e.g. smart lighting, ventilation, evacuation systems, fire extinguishing tools, and flood prevention systems. These technical installations are coordinated by a controller.

As reported in Rijkswaterstaat (2021b), many of the tunnels in the Netherlands that were built in the 50s and 60s of the previous century are due for renovation. A renovation of a tunnel often consists of replacement of part of the civil structure, the technical installations, and the software. In the past, road tunnels were designed as standalone "works of art", in which unique characteristics were embraced. However, recent renovation projects showed that the uniqueness of a road tunnel increases the required renovation effort and costs, as specific solutions are required at each unique tunnel. In an attempt to tackle this problem, RWS released the national tunnel standard (*Landelijke Tunnel Standaard* (LTS) in Dutch) in 2012, which can be found in Rijkswaterstaat (2021a). This documentation provides standardized processes and functional requirements for the construction and maintenance of tunnels, as well as requirements for the design of the control system. In 2016, this LTS was updated to Version 1.2 by applying practical experiences. This version is the current standard for tunnels. The creation of this standard has resulted in a more structured construction process and applying this standard will result in more standardized tunnels throughout the Netherlands.

However, the implementation of the LTS has not entirely solved the problem regarding the tunnel control system design. Even though the LTS has brought standardization in the design of tunnels, many differences still exist between these tunnels. These differences are mainly the result of decisions made by the third parties that built the tunnel or adjustments that needed to be made due to local circumstances. These deviations from the standard tunnel often have impact on the control system of that tunnel. The control system thus needs to be redesigned every time a new tunnel is designed, as well as every time a tunnel is slightly adjusted or renovated. The design process of such a control system takes a lot of time, including not only the design phase, but also the validation phase, the verification phase, and the implementation phase. Furthermore, unique control systems impede the operation and monitoring of multiple tunnels from a central location, as each tunnel requires specific knowledge of its control system functionality. RWS is therefore looking for control system design methods that allow for higher standardization and evolvability.

To this end, RWS initiated the MultiWaterWerk (MWW) project. The MWW project is a collaboration project with the Eindhoven University of Technology, in which, among other things, control system design methods are investigated while evaluating their applicability for RWS. Initially, the MWW project focused on the design process of waterway locks, and was later extended to movable bridges, tunnels, and roadside units. The focus of the project lies in establishing a shift from an Engineer-to-Order production method to a Configure-to-Order one, as reported in Wilschut (2018). In an Engineer-to-Order production method, each new system is designed from scratch after an order has been received. This method can tailor to exact specifications of the customer, but has low standardization and evolvability capabilities. In a Configure-to-Order production method, a basic product design is required. Once an order is received, this basic design is configured to fit the specific needs of the customer. Since all products are derived from the same basic design, standardization and evolvability of the Configure-to-Order method is much higher compared to the Engineer-to-Order method. In Rijkswaterstaat (2020), RWS argues that a Configure-to-Order design method is likely to have a higher reliability for correctness, and problems can be solved more quickly.

In the case of the design of control systems for road tunnels, Configure-to-Order seems a suitable method, as the tunnels throughout the Netherlands largely resemble each other and the control system requirements are standardized as defined in the LTS. In Goorden (2019), formal model-based methods are investigated that can be used for the development of control systems of large-scale infrastructural systems. Specifically, the focus lies on synthesis-based methods that allow for automatic generation of control systems, which reduce the time-to-market and immediately provide guarantees for correctness. In Reijnen (2020), the implementation process of synthesized controllers is investigated and applied, among others, in a case study for the Oisterwijksebaan bridge, where the synthesized controller is tested on the real system. Goorden et al. (2020) reports on recent developments in synthesis-based design methods in the context of the MWW project, and showcases various industrial applications.

1.1 Control systems of road tunnels

The road tunnel and its control system form a so-called cyber-physical system, described in Lee (2008), which is an integration of computational and physical processes. A cyber-physical system consists of (networked) computers that monitor and control a physical system, and where the state of the physical system is reported back to the controller through feedback loops. However, the tunnel system is not only automatically controlled through the controller algorithms, but also by human road traffic operators. There is at all times an operator monitoring the tunnel, who can intervene when deemed necessary through an operator interface. This graphical user interface contains an overview of the tunnel system in its current state. The road traffic controller can send commands to the tunnel through the operator interface, such as to close off the tunnel or to change a specific traffic light. The control structure of a road tunnel system is schematically visualized in Figure 1.3.

Layer 1 consists of the mechanical components of the tunnel that need to be controlled. These are often driven by actuators and monitored through sensors, as is shown in Layer 2. In Layer 3, the resource controller is shown. This is a local controller that is responsible for the low-level control of a specific component, such as signal processing or motion control. The combination of Layers 1 through 3 is referred to as the plant. Layer 4 contains the supervisory controller, which receives sensor information and sends signals to the actuators, both directly or via the resource controllers. The supervisory controller is typically implemented on a programmable logic controller (PLC). The supervisory controller is able to make control decisions based on the current state of the plant, but also receives commands from the operator interface in Layer 5. This is the interface that is

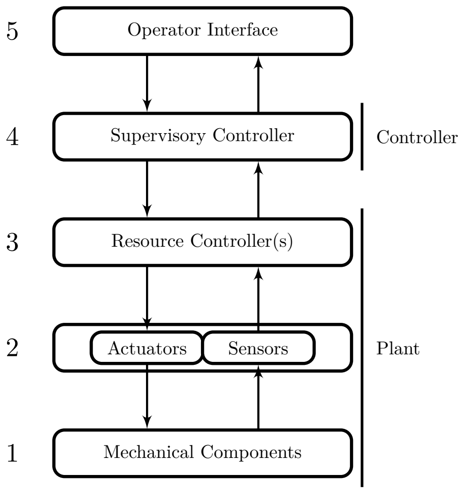


Figure 1.3: Control structure of a road tunnel system.

used by a human operator, both to monitor the system and to send commands to it.

As the main purpose of the control system of a road tunnel is to maintain a safe environment for road users, demands in terms of safety, quality, and functionality are high. Nonetheless, designing a control system for a road tunnel is challenging due to the large number of components in the system and the large number of control dependencies between them. As an example, the supervisory controller of the Eerste Heinenoord tunnel drives 152 actuators based on inputs from 332 sensors and 285 operator buttons. Traditionally, PLC code of such a supervisory controller is developed by hand and is tested manually. The PLC coding process leaves much room for error, and typically has a high time-to-market as errors found once the software has been implemented require a large step backwards in the design process. In A2Maastricht (2018), the control system design team of the Koning Willem-Alexander tunnel reports how one of the largest controller design challenges was a result of errors found after the software was implemented.

Within the MWW project, RWS aims to develop methods for the specification, design, realization, implementation, and maintenance of supervisory control systems that give higher guarantees for safety, a lower time-to-market, and increased standardization and evolvability.

1.2 Problem description

Supervisory control theory (SCT), also known as the Ramadge-Wonham framework, introduced in Ramadge and Wonham (1987) is a model-based method for automatically synthesizing supervisors. This method is applicable for systems for which the uncontrolled behavior of the components is given, and the desired behavior can be specified in a set of formal requirements. From this model of the uncontrolled behavior, called the plant, and the model of the requirements, a supervisory controller can be derived. This derivation step is called supervisor synthesis. In SCT, the control engineer specifies *what* the control system should do, not *how* it should do that. *How* the control system implements the given requirements is automatically determined during supervisor synthesis. In Baeten et al. (2016), the synthesis-based engineering (SBE) method is introduced, in which SCT is integrated in the engineering process for supervisory controllers.

There are several advantages of adopting a formal method such as SCT. First and foremost, using supervisor synthesis gives the guarantee that the specified requirements are adhered to. This removes the need to verify whether the requirements are implemented correctly. As safety is often of great concern, this guaranteed correctness is essential in the design of supervisory controllers. The second advantage is that the plant model and the requirements model give a consistent and unambiguous specification of the uncontrolled behavior and the desired behavior, respectively, in comparison to textual documents. Third, as the models of the plant and the requirements are executable models, they can be used for the purpose of simulation. This can give the control engineer insight into the controlled system, and allows for early validation. Finally, a formal (synthesized) model of the supervisory controller supports automatic generation of implementation code.

Even though research on the topic of supervisory control synthesis has been carried out for several decades now, the number of industrial-size case studies is still relatively low. There are multiple reasons for this. First of all, as stated in Wonham et al. (2018), few control engineers have the knowledge required to design component models and requirement models for the purpose of supervisor synthesis, as they are often trained to be programmers instead of supervisory control engineers. Furthermore, there exist relatively few clear guidelines for developing proper component models and requirement models, as mentioned in Grigorov et al. (2011) and Zaytoon and Riera (2017). The lack of adequate commercially available tooling for modeling and synthesis is another reason for the low number of industrial applications. Finally, synthesizing a supervisor for a large-scale cyber-physical system with many components, such as a road tunnel, can be a computationally intensive, sometimes even intractable, task.

The aim of this thesis is to show that SCT is suitable for the design and implementation of supervisory controllers for large-scale road tunnel systems. Furthermore, this thesis aims to provide extensions to SCT, e.g. to improve the scalability of supervisory synthesis, to develop methods for distributed supervisor implementation, and to increase testing possibilities. Doing so can contribute to a decreased cost and a decreased time-to-market of the construction or renovation process of infrastructural systems, while preserving a high quality.

1.3 Research questions

The following research questions are posed to investigate the applicability of SCT for the design of supervisory controller for road tunnels.

Research question 1

> *What is a suitable way to model road tunnels and their requirements for the purpose of supervisor synthesis?*

Supervisor synthesis relies on a model of the plant and a model of the control requirements. The behavior that should be captured in these models is usually established in documents, but the way it is realized in the models is often up to the control engineer. These modeling decisions can affect different aspects of the synthesis-based engineering process, e.g. the readability of the models, the reusability of the models, the complexity of the synthesis problem, and the difficulty of adapting the model for the purpose of implementation. Therefore, a suitable way of modeling the components and requirements of road tunnels is essential.

Research question 2

> *How can the characteristics of road tunnels be exploited in the synthesis-based engineering process?*

Road tunnels are infrastructural systems that have certain distinctive characteristics compared to other types of infrastructural systems. Specifically, they have a high degree of symmetry, both lengthwise, as many components are repeated over the length of the tunnel, and broadwise, as the traffic tubes of the tunnel resemble each other. It is, therefore, interesting to investigate how these characteristics affect the synthesis-based engineering process, and whether they can be exploited.

Research question 3

> *Is it possible to synthesize a supervisor for large-scale systems like road tunnels?*

Road tunnels are systems that have a large number of components with many, possibly complicated, dependencies between them. Case studies have shown that a complete tunnel is modeled using over 500 components, and can have an uncontrolled state space of over 10^{200} states. The question is, therefore, whether existing synthesis algorithms can solve the synthesis problem for road tunnels in a tractable time using commonly available computing power, and whether they can be improved to simplify the synthesis problem.

Research question 4

> *How can a supervisory controller correctly be synthesized for the purpose of implementation on multiple PLCs?*

For road tunnels, supervisory controllers are often implemented on multiple hardware platforms, often PLCs. Supervisor synthesis, however, results in a single supervisory controller. Therefore, it has to be investigated how a supervisory controller can be distributed for the purpose of implementation on multiple PLCs, without losing the properties that are guaranteed by supervisor synthesis. Moreover, a distributed supervisory controller often requires communication between the PLCs, which can have time delays. These delays should be coped with, and should not affect the controlled behavior of the system.

Research question 5

> *Are the methods for synthesizing, implementing, and testing a supervisory controller applicable to an existing, real-life, road tunnel?*

To assess whether synthesis-based engineering is suitable for the design of supervisory controllers for road tunnels, case studies should be performed that cover the complete process including synthesis, implementation, and testing. Moreover, the aspects of scalability, adaptability, and reusability can only be evaluated when multiple case studies are carried out.

1.4 Main contributions

This thesis has the following five main contributions. While it mainly focuses on the design of road tunnel supervisory controllers, many of the contributions are also applicable to similar application domains that can be modeled as discrete-event systems. Examples of such other application domains include: theme park vehicles (Forschelen et al. (2012)), manufacturing lines (Reijnen et al. (2018)), waterway locks (Goorden et al. (2019a)), movable bridges (Reijnen et al. (2020b)), roadside units (Verbakel et al. (2021)), and automotive systems (Korssen et al. (2017)).

Contribution 1

During the various case studies for road tunnels in the Netherlands it has been observed that there are many similarities between these tunnels in terms of layout, components, and control requirements. Inspired by previous works shown in

Grigorov et al. (2011) and Reijnen et al. (2020b), a parameter-based modeling method is proposed that allows design engineers to configure a tunnel by only specifying its parameters. From this configuration, all models that are required for synthesis and model simulation can then be generated automatically. Using this method, a road tunnel can be modeled more efficiently while requiring almost no knowledge on SCT. A prototype tool that implements this parameter-based modeling method has been developed, and its applicability is shown in a case study pertaining 22 road tunnels in the Netherlands. This contribution relates to Research questions 1 and 2.

Contribution 2

Road tunnels are large-scale systems due to the many actuators, sensors, and operator buttons they contain. As the complexity of supervisor synthesis problems increases exponentially with the number of components in the system, the synthesis problem for a road tunnel can be computationally intensive or even unsolvable. To facilitate this synthesis problem, a new method is proposed for reducing the model of the system before synthesis is performed. The model reduction steps in this method are mainly based on symmetry in the system. The system structure is analyzed using dependency graphs, and symmetrical components are identified and removed from the model. The method is applied for the synthesis of a supervisory controller for the Eerste Heinenoord tunnel, where the model can be reduced to an 80% smaller model, which made an initially unsolvable synthesis problem solvable in some minutes. This contribution relates to Research questions 2 and 3.

Contribution 3

Large-scale systems such as road tunnels are typically not controlled by a single PLC, but by a set of PLCs. This can improve the performance of the individual PLCs, increase the clarity and readability of the PLC code, and increase availability of the control system. However, distributing a supervisory controller over multiple PLCs requires communication between the PLCs, which have time delays. This thesis proposes a method for distributing a supervisory controller. The method consists of two main steps: obtaining a distribution of the system components, and distributing the supervisory controller accordingly. Dependency structure matrices are used to obtain a distribution that is suitable for a distributed implementation, and existing localization techniques are applied to distribute the supervisory controller. Furthermore, a delay-robustness check is proposed that analyzes the communication between the PLCs and, in case of delay-critical communication, mutual exclusion algorithms are applied to make the communication delay robust. This contribution relates to Research questions 4 and 5.

Contribution 4

A synthesized supervisory controller is not directly implementable on a PLC. To do this, the properties for an implementable controller need to be verified and the supervisory controller model needs to be translated to PLC code. Furthermore, to validate implemented PLC code in a simulation environment, hardware-in-the-loop (HIL) simulation is performed. For HIL simulation, interface models need to be created so that the operator interface and the virtual system can be simulated. This thesis provides several contributions to the supervisor implementation process, including the design and implementation of resource controllers, and a relaxation in the property check for implementable controllers. Furthermore, a contribution is made towards an algorithm for automatic PLC code generation. Moreover, it is shown how digital twins can be integrated in the HIL simulation, which allows for a more intuitive validation process with more extensive possibilities for simulation scenarios. This contribution relates to Research question 5.

Contribution 5

Several real-life case studies have been performed to show the applicability of SBE for the design of supervisory controllers for road tunnels. Specifically, supervisory controllers have been designed for the Koning Willem-Alexander tunnel, the Eerste Heinenoord tunnel, and the Swalmen tunnel. Furthermore, a case study has been performed for 22 tunnels in the Netherlands to show the effectiveness of the parameter-based modeling tool. Through the various case studies, the process of modeling, synthesis, implementation, and validation are investigated extensively, underscoring the scalability, adaptability, evolvability of the SBE method. This contribution relates to Research questions 1 and 5.

1.5 Thesis outline

This thesis is structured as follows. In Chapter 2, the preliminaries regarding SCT and SBE are described. Specifically, the subjects of SBE, modeling of discrete-event systems, modeling of requirements, supervisor synthesis, and implementation of supervisors are covered. Chapter 3 introduces road tunnels systems, their function of their control system, and the operator interface. In Chapter 4, the plant model and the requirements model for road tunnels are described, and the parameter-based modeling method is proposed. Chapter 5 continues with the synthesis process for road tunnels. Specifically, the model reduction steps that are used to synthesize a supervisory controller for a road tunnel are detailed. Subsequently, the implementation process of the synthesized supervisor and the validation process using HIL simulation are described in Chapter 6. In Chapter 7, the new method for distributing a supervisory controller for the purpose of implementation on multiple PLCs is described, including the steps of obtaining a distribution and localizing the supervisory controller accordingly. Chapter 8 covers the incorporation of digital twins in the SBE process, both in the model simulation step and the HIL simulation step. Finally, in Chapter 9, conclusions are drawn, answers to the research questions are given, and recommendations for future work are provided.

Chapter 2

Supervisory control

This chapter gives an introduction to supervisory control. First, the synthesis-based engineering method for supervisory controllers is introduced. Subsequently, it is shown how discrete-event systems and requirements are modeled, and the supervisor synthesis process is explained. Finally, the steps toward supervisor implementation are described.

2.1 Synthesis-based engineering

Traditionally, supervisory controllers are designed in a document-based manner. Here, the desired system specifications, both for the physical system and the control system, are described in documents. For large-scale systems with many specifications this can result in documents with thousands of pages, which are often unclear and inconsistent, as stated e.g. in Weber and Weisbrod (2002).

In the last decades, it has become more common to use executable models when designing systems. Such design methods, called model-based engineering methods, can increase the quality of a system, decrease development costs, and lower the time-to-market, as advocated in Ramos et al. (2011). A survey of model-based engineering methods is provided in Estefan (2007). Furthermore, executable models enable testing of realized components together with yet to realize, virtual, components, in HIL simulation, as detailed in Bullock et al. (2004).

Model-based engineering methods enable early validation through simulation and controller testing, but the correctness of the controller still very much depends on the test engineer, as is also mentioned in Taipale et al. (2011). In the 1980s, research was started on the control of discrete-event systems. The results of this research are found in Ramadge and Wonham (1987) and Ramadge and Wonham (1989). The idea is to define the possible system behavior and the controller specifications using formal models. This enables the use of synthesis algorithms to automatically generate the supervisory controller. It is mathematically proven that the generated controller adheres to the specified controller requirements. This means that verification is no longer necessary.

The synthesis-based engineering process is schematically visualized in Figure 2.1. The process starts on the left-hand side, where the high-level system requirements H_R are defined in documents. Now, a division is made in the controller requirements C_R and the plant requirements P_R. The documented controller requirements are directly formalized by means of a controller requirements model C_R. In parallel, from the documented plant requirements, a plant design P_D is created. Subsequently, from this plant design, a plant model P is created. From the requirements model C_R and the plant model P, a supervisor C is synthesized. For the purpose of simulation, a hybrid plant model P_H is derived from P by enriching it with continuous time behavior. In the final steps, an implementable controller C is generated from the supervisor, which is implemented on an implementation platform, and the plant P is realized by building the actual system.

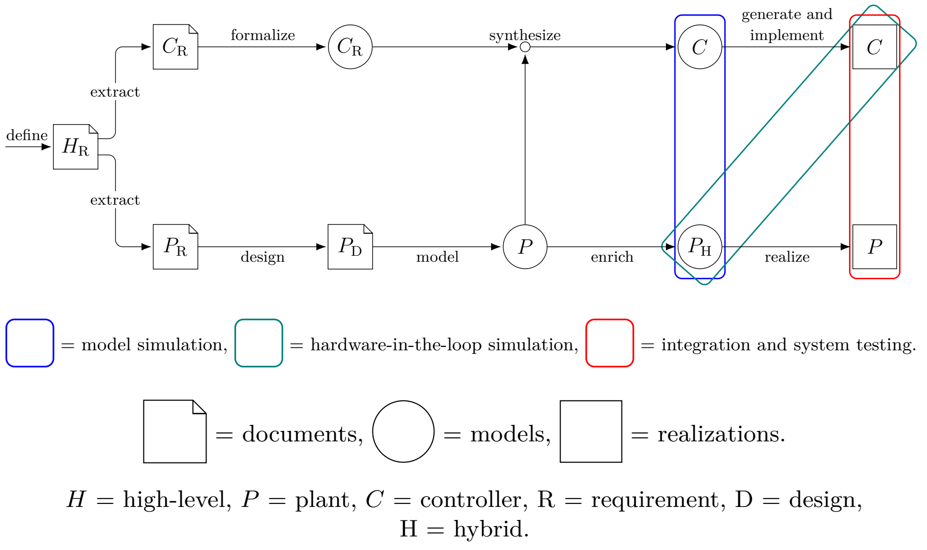


Figure 2.1: Schematic overview of the synthesis-based engineering process, adapted from Baeten et al. (2016).

Advantages of synthesizing the supervisor include the guarantee that the controller always adheres to the defined requirements. An important step, however, remains the validation of the controller. While the synthesized controller is guaranteed to adhere to the requirements, it is still possible to define incorrect or incomplete requirements. There are several validation stages in the synthesis-based design process, as can be seen in Figure 2.1. At these stages, the controlled behavior of the system is validated. The first validation stage is model simulation, which uses the hybrid model of the plant and the synthesized controller. If any error is found during the validation process, it is traced back in the controller requirements model or the plant model, and a new supervisor is synthesized. The second validation stage is HIL simulation, where the controller is implemented on the hardware platform, and combined with the hybrid plant model to perform simulations with interfaces that visualize the operator interface and the virtual system. The final validation stage is integration and system testing, which is performed when the plant is realized as well.

2.2 Modeling of discrete-event systems

In the context of supervisory control theory, the plant behavior is often modeled using either (extended) finite state automata or Petri nets, as described in Cheng and Krishnakumar (1993) and Murata (1989), respectively. In this thesis, extended finite state automata are used to model the uncontrolled plant behavior as previous work in Goorden et al. (2019b) and Reijnen et al. (2020a) has shown that they are suitable for the modeling of infrastructural systems. This section first introduces the definition of finite state automata, and then proceeds with the definition of extended finite state automata.

All modeling, synthesis, simulation, and code generation procedures in this project are performed using CIF. CIF is a hybrid automata-based language and the associated toolset is part of the Eclipse Supervisory Control Engineering Toolkit (Eclipse ESCET™) project[1].

2.2.1 Finite state automata

Finite state automata (FAs) are used to model discrete-event systems in a formal way. In discrete-event systems, as described in Cassandras and Lafortune (2008), the dynamic behavior of a system is modeled by a discrete set of states and the transitions between these states, which are driven by occurrences of events. An FA P can be denoted as a five-tuple:

$$P = (Q, E, f, q_0, Q_m). \tag{2.1}$$

Here, Q is the finite set of states in the automaton, and E is the finite set of events. Every transition in the automaton is labeled by an event, and the partial transition function $f \subseteq Q \times E \times Q$ describes for each transition its starting state, the event that labels it, and its end state. An FA is called deterministic if for

[1]See `https://eclipse.org/escet`. 'Eclipse', 'Eclipse ESCET' and 'ESCET' are trademarks of Eclipse Foundation, Inc.

each state and each event there exists at most one end state. In this thesis, only deterministic FAs are considered as infrastructural systems are naturally deterministic. Furthermore, every FA has exactly one initial state, denoted by q_0. Finally, every FA contains at least one marked state, and the set of marked states is denoted by Q_m. The marked states in the automaton indicate the accepting or safe states of that automaton, such as the idle state of a machine or the end state of a process.

The event set E can be partitioned into the set E_c of controllable events and the set E_u of uncontrollable events. Controllable events are events that can be enabled by the supervisor, such as turning an actuator on or off, whereas uncontrollable events cannot be influenced but only observed, such as a sensor turning on or off. The observable events in E are denoted by E_o. The set of all finite strings of events from E, including the empty string ϵ, is denoted by E^*.

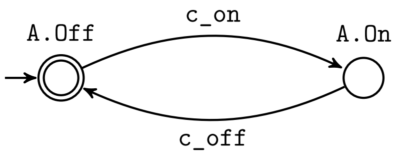


Figure 2.2: Example of an automaton for actuator `A`.

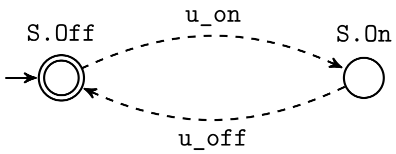


Figure 2.3: Example of an automaton for sensor `S`.

Automata are often displayed graphically to represent them more intuitively. Two examples of visualized automata are shown in Figures 2.2 and 2.3, showing the model of an actuator and the model of a sensor, respectively. In the graphical representation, states are indicated by circles, and transitions are indicated by arrows that are labeled by an event. A distinction is made between controllable events and uncontrollable events, where controllable events are represented by solid arrows, such as in Figure 2.2, and uncontrollable events by dashed arrows, such as in Figure 2.3. The initial state of an automaton is indicated by an unconnected incoming arrow, and marked states are indicated by double concentric circles. For the actuator and the sensor shown in the figures, the initial and only marked state is the `A.Off` state and the `S.Off` state, respectively.

For systems that consist of numerous individual components, it is often difficult to model their full behavior using a single automaton due to the many different states and even more transitions between them. Instead, each component is modeled individually by an automaton. The combined behavior can then be determined by calculating the synchronous product, denoted by $P = P_1 \mid\mid P_2 \mid\mid ... \mid\mid P_n$. The synchronous product, as e.g. in Cassandras and Lafortune (2008), is a composition operation that determines the joint behavior of a set of automata that operate concurrently. When there are shared events in the set of automata, the automata synchronize over these events in the synchronous product. The synchronous product of the actuator and the sensor is shown in Figure 2.4.

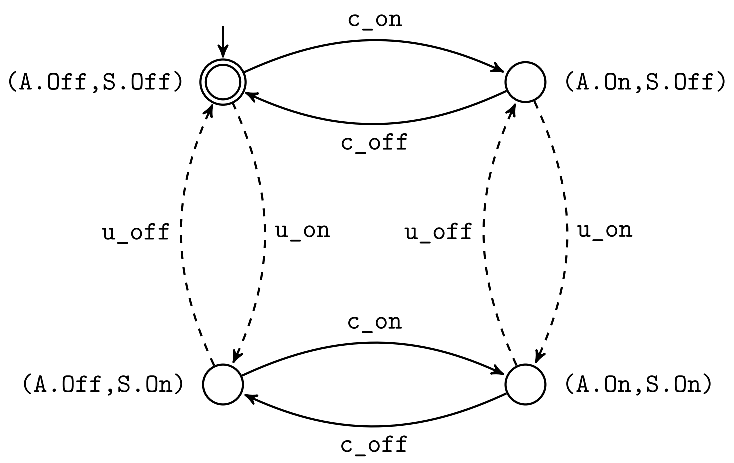


Figure 2.4: Synchronous product of actuator `A` and sensor `S`.

When a system is modeled using a set of automata, this representation is called the *Composed System Representation (CSR).* Furthermore, automata are said to be asynchronous if they do not have common events, i.e., automata P_1 and P_2 are asynchronous if $E_1 \cap E_2 = \emptyset$. Otherwise, they are said to be synchronous. In Ramadge and Wonham (1989), the *Product System Representation (PSR)* is introduced as the CSR where all component models are pairwise asynchronous. It is shown that there always exists a PSR for a CSR, which can be obtained by taking the composition of pairs of component models in the CSR that are either directly or transitively synchronous, i.e., P_1 is transitively synchronous with P_3 when P_1 and P_2 are synchronous and P_2 and P_3 are synchronous. In this case, the composition of $\{P_1, P_2, P_3\}$ is taken. The representation that contains the largest number of automata and only contains pairwise asynchronous component models is defined in de Queiroz and Cury (2000) as the *Most Refined Product System Representation (MRPSR).*

2.2.2 Extended finite state automata

Extended finite state automata (EFAs) are FAs where variables are included. The variables are associated with the transitions in the automaton, either to be used to evaluate conditions under which a transition is enabled, or in an update when a transition takes place. Previous works that use EFAs for modeling systems can be found in Grigorov et al. (2011), Chen and Lin (2001), Sköldstam et al. (2007), and Malik et al. (2011). A model consists of multiple interacting EFAs $\mathcal{E} = \{E_1, ..., E_m\}$ together with a set of variables $X_{\mathcal{E}} = \{x_1, ..., x_n\}$.

With each variable x, a finite discrete domain $\mathrm{dom}(x)$ of values is associated.

A valuation $v \in V$ is a mapping $v : X_{\mathcal{E}} \rightarrow \cup_{x \in X_{\mathcal{E}}} \text{dom}(x)$ with $v(x) \in \text{dom}(x)$ for each $x \in X_{\mathcal{E}}$. Here, V is the set of all valuations, with $v_0 \in V$ as the initial valuation and $V_m \subseteq V$ as the marked valuation.

An EFA is defined as the seven-tuple

$$P = (Q, X, E, \rightarrow, q_0, x_0, Q_m), \tag{2.2}$$

where $X \subseteq X_{\mathcal{E}}$ denotes the set of local variables in the automaton, with x_0 as initial evaluation of those variables, and $\rightarrow$ is the extended transition function. The other notations are the same as defined in the five-tuple for FAs.

The state space of an EFA is represented by $Q \times V$, which contains all combinations of locations and variable values. The initial state is therefore (q_0, v_0) and the marked states are $Q_m \times V_m$. The extended transition function $\rightarrow$ is similar to the partial transition function f of FAs, but extended with guard expressions (conditions) and variable assignments (updates). The formal definition of the extended transition function is therefore $\rightarrow \subseteq Q \times E \times C \times U \times Q$ with a natural extension to E^*, where C is the set of all conditions, and U is the set of all updates. With $\rightarrow (q, e, g, u)!$ we denote that there exists a transition enabled in q, that is labeled by event e, guard g, and update u.

A transition in an EFA can have a condition that defines when it is enabled. This condition is called a guard, which is a function $g : V \rightarrow \{\texttt{True}, \texttt{False}\}$. For clarity, $v \models g$ is used instead of $g(v)$. The transition corresponding to a guard can only be taken when the guard evaluates to true. Each EFA in the system contains a variable that represents its current location, resulting that locations of other EFAs can be used in the guard. A guard can, for example, be $x < 3 \wedge \texttt{A.On}$, meaning that the corresponding transition is enabled when $v(x) < 3$ and EFA `A` is in the location `On`.

Besides a guard, a transition in an EFA can also have one or more updates. An update is a function $u : V \rightarrow V$, in which a variable is assigned a value when that transition is taken. Only local variables of an EFA can be updated by transitions in that EFA. Updates are defined as $X := c$, where ':=' denotes the assignment of value c to variable x. An update can, for example, be $x := x - 1$, meaning that the value of x is decreased by 1.

We define the explicit state transition relation $\mapsto$, taken from Sköldstam et al. (2007). The explicit state transition relation is written as $(q, v) \overset{e}{\mapsto}_{g/u} (q', v')$, which indicates the transition that starts from location q and valuation v, labeled by event e, guard g, and update u, and that ends in location q' and valuation v'. It is extended to strings in E^* in the usual recursive way. The language $L(G)$ of an EFA G is $L(G) = \{s \in E^* | (q_0, v_0) \overset{s}{\mapsto}\}$. The marked language $L_m(G)$ is $L_m(G) = \{s \in E^* | (q_0, v_0) \overset{s}{\mapsto} (q, v), q \in Q_m, v \in V_m\}$.

Furthermore, the synchronous product operation for EFAs G_1 and G_2, $G_1 = (Q_1, X_1, E_1, \rightarrow_1, q_{0,1}, x_{0,1}, Q_{m,1})$ and $G_2 = (Q_2, X_2, E_2, \rightarrow_2, q_{0,1}, x_{0,2}, Q_{m,2})$, is $G_1 || G_2 = (Q_1 \times Q_2, X_1 \times X_2, E_1 \cup E_2, \rightarrow, (q_{0,1}, q_{0,2}), (x_{0,1}, x_{0,2}), Q_{m,1} \times Q_{m,2})$, where the transition relation $\rightarrow$ is defined as:

$$\rightarrow ((q_1, q_2), e, (g_1 \wedge g_2), (u_1 \oplus u_2)) :=$$
$$\begin{cases} (\rightarrow_1 (q_1, e, g_1, u_1), \rightarrow_2 (q_2, e, g_2, u_2)) & \text{if } \rightarrow_1 (q_1, e, g_1, u_1)! \quad \wedge \ \rightarrow_2 (q_2, e, g_2, u_2)! \\ (\rightarrow_1 (q_1, e), q_2) & \text{if } \rightarrow_1 (q_1, e, g_1, u_1)! \quad \wedge \ e \notin E_2 \\ (q_1, \rightarrow_2 (q_2, e)) & \text{if } e \notin E_1 \quad \wedge \ \rightarrow_2 (q_2, e, g_2, u_2)! \\ \text{undefined} & \text{otherwise} \end{cases}$$

The update $u_1 \oplus u_2$ denotes that the valuations of the variables from X_1 and X_2 are updated according to u_1 and u_2, respectively.

Another operation used in SCT is projection $P : E^* \rightarrow E'^*$ for $E' \subseteq E$. It takes a string formed from events of E and erases the ones that do not belong to E'. Projection is in a natural way extended to sets of strings.

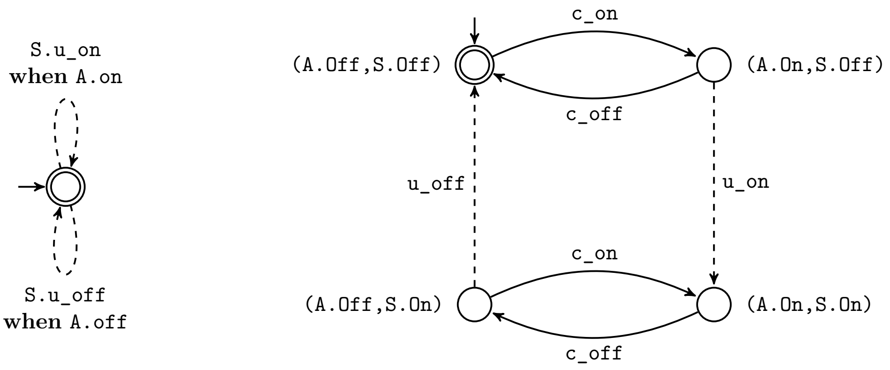


Figure 2.5: The physical relation between actuator `A` and sensor `S` (left), and the synchronous product `A` || `S` (right).

An example of an EFA that contains guards, related to the actuator and sensor from the previous section, is shown on the left-hand side of Figure 2.5. This EFA shows the physical relation between the actuator and the sensor. The sensor is in this case a feedback to the controller to indicate whether the actuator is on or off. The physical relation, therefore, expresses that the sensor can only turn on when the actuator is in the location `A.On`, and vice versa. In this EFA, these conditions are represented by the guards. Guards are denoted by the keyword **when** in the visualized EFA.

To incorporate the physical relation between the actuator and the sensor in the combined behavior, the synchronous product of the actuator, the sensor, and their physical relation is calculated. This synchronous product is shown on the right-hand side of Figure 2.5. Here we see, compared to Figure 2.4, that the events `u_on` and `u_off` are no longer enabled in the locations where their guards evaluate to false.

An example of an EFA that contains updates is shown on the left-hand side of Figure 2.6, where the actuator automaton of Figure 2.2 is extended with the variable `Q`. In this example, variable `Q` represents the output variable for the actuator that indicates whether the actuator is in the location `A.On` or `A.Off`. `Q` is initially set to `False`, as is indicated at the initial arrow at the `A.Off` location in Figure 2.6. Furthermore, `Q` is set to `True` and `False` when the events `c_on` and `c_off` occur, respectively, as is indicated by the update expressions of these events.

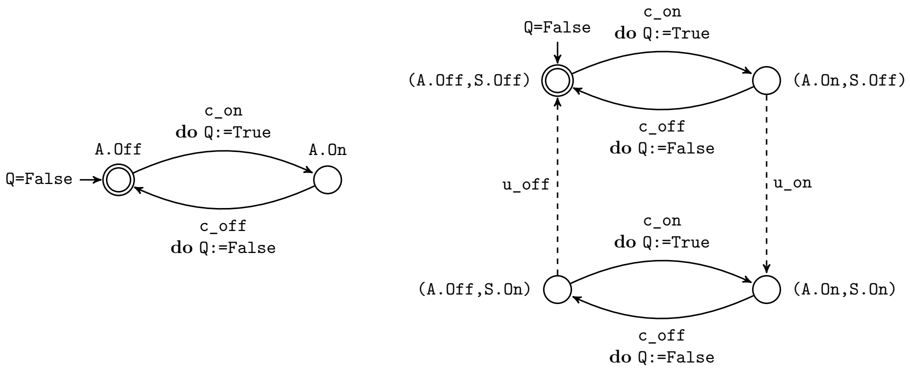


Figure 2.6: The automaton of actuator `A` extended with variable `Q`, and the new synchronous product of actuator `A`, sensor `S`, and their physical relation (right).

2.3 Modeling of requirements

The second set of models that is required for supervisor synthesis consists of the requirements. Typically, requirements are modeled in order to prevent unsafe behavior, or to ensure functionality or progress in the system. Requirements can be modeled in several ways. First of all, requirements can be modeled as FAs or EFAs. Secondly, requirements can be modeled as state-based requirements, as described in Markovski et al. (2010), either as event-condition requirements or as state-exclusion requirements. These methods are usually used to prevent unsafe or undesired behavior in the system.

A requirement that is modeled as an FA or EFA is typically used to define the order of a set of events to ensure progress in the system. An example of an FA requirement is shown in Figure 2.7, which defines that actuator `A` and sensor `S` must turn on alternately.

Event-condition requirements define under which conditions a certain event is enabled. This can be done in two variants, being e **`needs`** Y and Y **`disables`** e. The first describes that event e is enabled when Y evaluates to true, whereas the second indicates that e is enabled when Y evaluates to false.

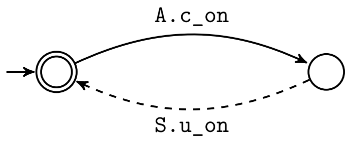


Figure 2.7: The requirement automaton that defines that actuator `A` and sensor `S` must turn on alternately.

requirement `c_on` **needs** `S.Off`

requirement `c_off` **needs** `S.On`

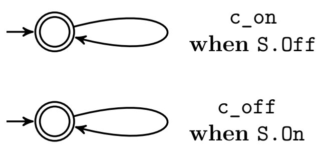


Figure 2.8: Two requirements in event-condition representation (left) and in EFA representation (right).

On the left-hand side of Figure 2.8, two event-condition requirements related to the actuator-sensor example are shown. They define that the actuator is only allowed to switch on when the sensor is off, and may only switch off when the sensor is on. Event-condition requirements can also be represented as EFAs. The EFA representation of the two example requirements are shown on the right-hand side of Figure 2.8. Here, each EFA consists of a single location and a transition that loops back to this same location. Such a transition is called a selfloop. This selfloop is labeled by the event of the event-condition requirement, and the condition is included in the guard of the transition. Note that every event-conditions requirement can be represented as an EFA, but not every EFA can be represented as an event-condition requirement.

requirement `S.On` **disables** `c_on`

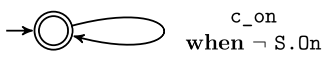


Figure 2.9: Two requirements in event-condition representation (left) and in EFA representation (right).

Event-condition requirements can also be defined using the keyword **`disables`**. In this variant, the requirement defines under which condition the event is not enabled. An example of such a requirement is shown on the left-hand side of Figure 2.9. In that requirement it is defined that when sensor `S` is turned on, event `c_on` is not enabled. Again, this requirement can also be represented as an EFA, as shown on the right-hand side of the figure. Note that the requirement shown in Figure 2.9 is functionally the same as the first requirement shown in Figure 2.8.

The controlled behavior of the plant combined with the requirement models of Figure 2.8 is presented in Figure 2.10. The difference compared to Figure 2.6 is that due to the addition of the requirements mentioned above, the controllable events `c_off` in location (`A.On, S.Off`) and `c_on` in location (`A.Off, S.On`) are disabled, since the requirements do not allow them based on the sensor locations.

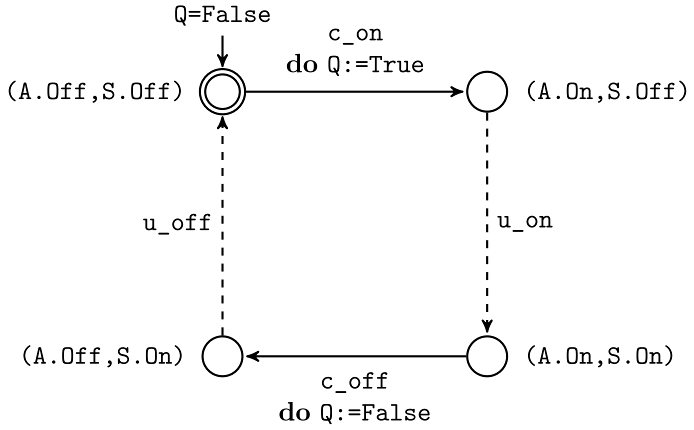


Figure 2.10: Controlled behavior of actuator `A`, sensor `S`, their physical relation, and two requirements.

2.4 Supervisor synthesis

Supervisory control theory, as introduced in Ramadge and Wonham (1987), provides a method to synthesize a supervisory controller from the model of the plant and the model of the requirements. Synthesis guarantees that the supervisor by construction satisfies the following properties:

Safety The supervisor prevents all behavior that conflicts with the specified requirements.

Nonblockingness From every reachable state, there exists a path to reach a marked state.

Controllability The supervisor never disables uncontrollable events.

Maximal permissiveness The supervisor disables as few events as possible, while guaranteeing the three previously mentioned properties.

An example of supervisor synthesis is given regarding a product line system in Figure 2.11, consisting of a machine `M` and a buffer `B`. Figure 2.11 shows the automata models of these components on the left-hand side, as well as control requirement R. Furthermore, the corresponding synchronous product M || B || R is shown on the right-hand side. The machine M can start working on a product by event c_start, and this product is finished after some time, indicated by event u_done. The product then goes to the buffer B, which accepts the product by the same event u_done, and can remove the product by event c_remove.

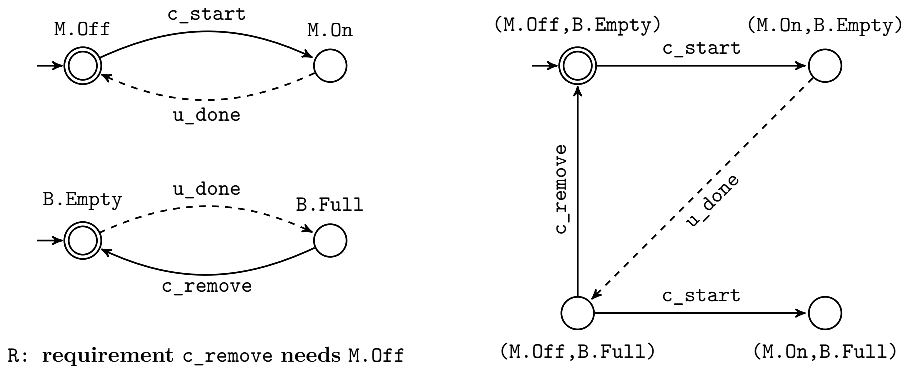


Figure 2.11: Automata models for a machine M, a buffer B, and requirement R (left) and their synchronous product M || B || R (right).

The synchronous product shows that the system reaches a deadlock, i.e. a location that can never be left, when the machine starts on a new product while the buffer is still full. This can be recognized by the location (M.On,B.Full), which has no outgoing transitions and is not a marked location. Performing supervisor synthesis for this system provides a supervisor that prevents this deadlock, as synthesis guarantees *nonblockingness.* Furthermore, this supervisor will only restrict controllable events, in accordance with the *controllability* property.

When synthesis is used, a supervisor is generated to control the plant. The supervisor that is synthesized for the machine-buffer system of Figure 2.11 is shown on the left-hand side of Figure 2.12. Here, event c_start is removed in the location (M.Off,B.Full) to guarantee a nonblocking system.

For systems with large numbers of components, it is often infeasible and undesirable to represent the supervisor in a single FA like the left-hand side of Figure 2.12. Synthesized supervisors are therefore often represented as an additional EFA to the plant model and the requirements model, as is described in Miremadi et al. (2011) and Yang and Gohari (2005). This EFA contains each controllable event of the plant as a selfloop, and contains guards for all controllable events. This guard is either the True predicate, or an additional restriction provided during synthesis in order to ensure nonblockingness and controllability. The guard representation of the supervisor shown on the left-hand side of Figure 2.12 is shown on the right-hand side of Figure 2.12. Here, the guard **when** B.Empty is added to event c_start. As there are no additional

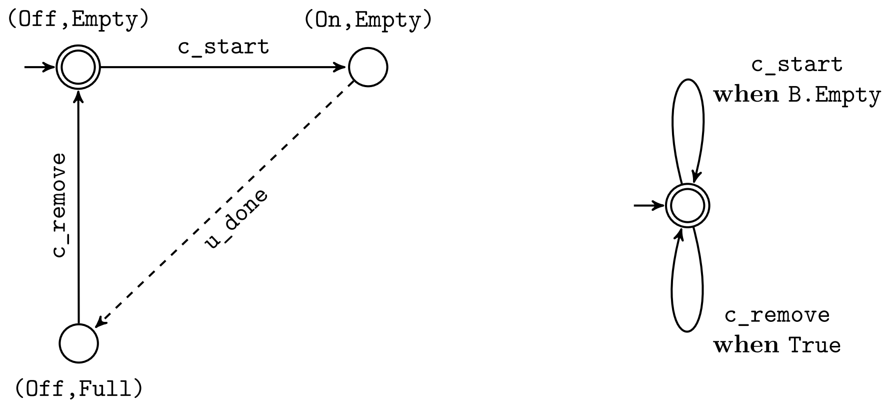


Figure 2.12: Synthesized supervisor for system `M` || `B` || `R` (left) and its guard representation following Miremadi et al. (2011) (right).

restrictions from synthesis for event `c_remove`, this event simply has the guard **when** `True`. Note that, following the representation of Miremadi et al. (2011), the plant model and the requirements model are not included in these guards, since the set of automata models and the set of requirements is included in the supervisor separately from the supervisor EFA.

One of the benefits of this representation, as is also noted in Fabian et al. (2018), is that these guard expressions are more transparent than the traditional FA representation of the supervisor, and can thus give meaningful insight to the design engineer. Furthermore, often supervisor synthesis adds no guards to many of the controllable events, as described in Goorden and Fabian (2019), implying that no additional restrictions are needed to ensure nonblockingness and controllability. Additionally, supervisor synthesis can add guards that always evaluate to false, e.g. in the case of conflicting requirements. Both these cases can be a source of valuable information for the design engineer.

2.4.1 Synthesis approaches

There exist different approaches to synthesizing a supervisor. In Wonham et al. (2018), a brief history on supervisory control of discrete-event systems is given. In this section, a short overview of available synthesis techniques is provided to show which options a control engineer can choose from. The standard form of supervisor synthesis is called monolithic synthesis. An example of an algorithm that provides a monolithic supervisor is described in Ouedraogo et al. (2011). This algorithm provides a monolithic supervisor, which is a single centralized supervisor that adheres to all control specifications. There exist multiple extensions to monolithic synthesis in which a set of supervisors is synthesized, such as decentralized synthesis, distributed synthesis, modular synthesis, hierarchical synthesis, and multilevel synthesis. The main benefits of synthesis approaches that provide a set of supervisors are a lower computational effort and a more understandable design.

In decentralized and distributed synthesis, a set of supervisors is synthesized where each of the supervisors is responsible for a part of the plant. Lafortune (2007) provides a survey on decentralized and distributed synthesis, and denotes the difference between these approaches as follows. In decentralized control (see Lin and Wonham (1990) and Rudie and Wonham (1991)) each supervisor knows the entire system and the entire control specification, but each supervisor sees and controls a different aspect of the behavior of the system. In distributed control (see Su et al. (2010) and Zhang et al. (2016)) each supervisor only knows part of the system and part of the control specification, and communication between supervisors is required to achieve a global specification.

In modular synthesis, each supervisor is responsible for part of the control specifications, i.e. part of the requirements, (see Wonham and Ramadge (1988) and de Queiroz and Cury (2000)). An extension to modular synthesis is using a hierarchical control architecture. Here, separate supervisors are created for parts of the control specifications as well, but control specifications can be defined for specific low-level controllers as well as global specifications for the high-level controllers. Hierarchical control is described and applied in Zhong and Wonham (1990) and da Cunha and Cury (2007). An extension of hierarchical control is multilevel control, as explained and applied in Komenda et al. (2016) and Goorden et al. (2019a), in which an arbitrary number of control layers is possible.

2.5 Implementation of supervisors

The last step in the supervisor design process is the implementation of the supervisor. In Ramadge and Wonham (1987) it is described how a supervisor can be implemented together with a separate controller. Conversely, a synthesized supervisor can be interpreted as a controller, as is described in Balemi et al. (1993) and Vieira et al. (2016), and be implemented on its own. This interpretation is needed due to the functional differences between a supervisor and a controller, which are as follows.

First, a supervisor monitors the plant behavior and decides which events are enabled and which are disabled. A supervisor does not actually decide which event to execute. A controller chooses which events to execute based on which events are enabled. Although a synthesized supervisor is guaranteed to be nonblocking, a controller that is interpreted from this supervisor might not be, due to this functional difference.

Second, for supervisor synthesis it is assumed that there is no time delay between the supervisor and the plant. Contrarily, in a real-time implementation of a controller there exists a small time delay between the controller and the plant. This delay can affect the controlled system behavior, and even nullify the guarantees provided by supervisor synthesis, as described in Fabian and Hellgren (1998) and Zaytoon and Riera (2017).

2.5.1 Programmable logic controllers

PLCs are widely used in industry as the implementation platform for controllers. The PLC consists of a central processing unit that runs the controller code and input and output modules that connect the PLC to the sensors and the actuators in the plant, respectively. An input image and an output image are used to represent the sensor signals and the actuator signals from the modules as variables, such that the signals are usable for the PLC. The operating semantics of a PLC is visualized in Figure 2.13. A PLC operates in cycles, where first the input variables are read, subsequently the controller code is executed, and finally the output variables are written.

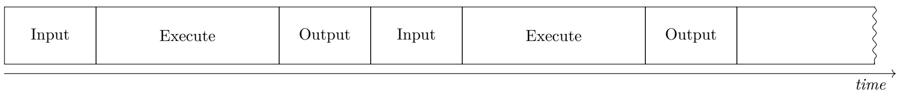


Figure 2.13: Operating semantics of a PLC.

The supervisor can be executed on a PLC, as described in Prenzel and Provost (2018) and Reijnen et al. (2019b). The code execution consists of the following steps:

1. Get the input values (represented by Booleans), determine whether they have changed, and translate these changes to the corresponding uncontrollable events (e.g. sensor inputs).
2. Update the state of the system by executing the uncontrollable events.
3. Determine whether controllable events are enabled in the new state. If so, execute the enabled controllable events and update the system state. Repeat until there are no more controllable events enabled.
4. Update the output values (represented by Booleans) and write these to the output image (e.g. actuator states).

To connect the values of the input and output variables of the PLC to the supervisory controller, a hardware mapping is supplied. This hardware mapping is an extension of the plant model, where an input variable is added for each sensor or button component, and an output variable is defined for each actuator component. The events of the sensor and button automata are connected to the input variables by means of guards, as is shown in the example in Figure 2.14 for input `I`. The output variables are connected to the events of the actuator automata by means of updates, such as the update on the left-hand side of Figure 2.6 for output `Q`.

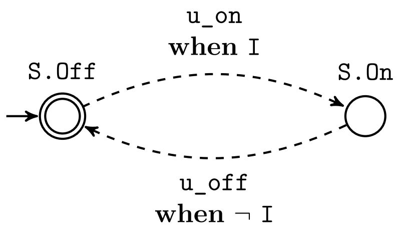


Figure 2.14: Hardware mapping for sensor `S`.

2.5.2 Implementable supervisor properties

As mentioned, there are several functional differences between a supervisor and a controller that can affect the controlled system behavior. To guarantee that a controller expresses the same, desired, behavior as the supervisor, several properties need to be determined, introduced in Malik (2003). When a supervisor adheres to these properties, it is denoted as an implementable supervisor. The properties for an implementable supervisor are described here, and examples taken from Reijnen (2020) are provided.

Finite response The controller always reaches a state where it waits for new inputs from the plant, i.e. it never gets stuck in a loop of controllable event executions.

The controller executes all enabled controllable events one after another until no controllable events are enabled anymore. To prevent a PLC from getting stuck in a loop of controllable events, no forcible controllable event loop may exist in the supervisor. The supervisor on the left-hand side of Figure 2.15 contains such a forcible controllable event loop, consisting of events `b` and `c`, meaning that this supervisor does not have finite response. The supervisor shown on the right-hand side does have finite response as there does not exist such a loop.

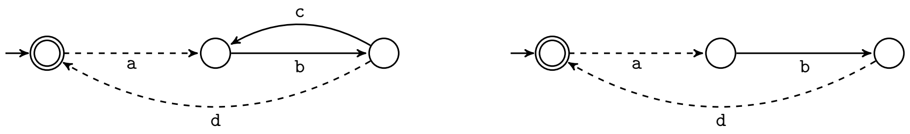


Figure 2.15: A supervisor that does not have finite response (left) and a supervisor that has finite response (right).

Confluence Whenever a choice between multiple controllable events exists, any sequence of controllable events after this choice eventually leads to the same end-state, i.e. the choice of which controllable event to execute does not influence the stable state that is eventually reached.

As a supervisor only enables controllable events and does not decide which event to execute, a controller can decide between multiple enabled controllable events. To ensure that a supervisor has confluence, this decision may not affect the end-state that is reached after all controllable events have been executed. The supervisor on the left-hand side of Figure 2.16 does not have confluence, as the decision between events `b` and `d` results in two different end-states. Contrarily, the supervisor on the right-hand side has confluence, since after event `b` event `e` can be executed, after which the same end state is reached as with event `d`.

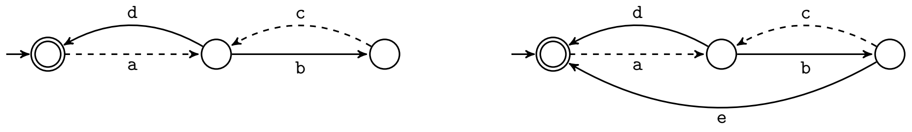


Figure 2.16: A supervisor that does not have confluence (left) and a supervisor that has confluence (right).

Nonblocking under control From any reachable state, a marked state is always reachable by an event sequence that prioritizes controllable events over uncontrollable events.

As a controller executes all controllable events until no more controllable events are enabled, controllable event sequences will be prioritized over uncontrollable events. This can mean that a nonblocking supervisor becomes blocking when it is implemented as a controller. The supervisor on the left-hand side of Figure 2.17 is nonblocking, but requires uncontrollable event `c` to reach the marked state, which will never be taken as the controllable event sequence `b` · `d` will have priority. The blocking behavior when executed is shown on the right-hand side of Figure 2.17, which reveals that the marked state is never reached.

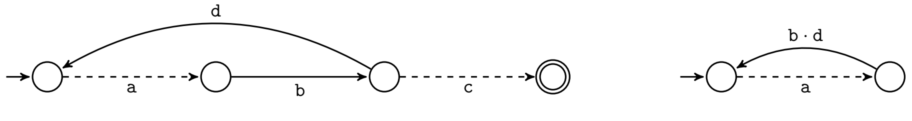


Figure 2.17: A supervisor that does not have nonblocking under control (left) and its blocking behavior when executed (right).

The nonblocking under control property can be verified by enforcing the priority of controllable events over uncontrollable events. When the synthesis procedure adds no additional restrictions, it means that the supervisor is non-blocking under this priority, and thus that the supervisor is nonblocking under control. The automaton that enforces the priority of controllable events is shown in Figure 2.18. Let

$$\Sigma_c = \bigcup_{i=1}^{m} E_{c,i} \quad \text{and} \quad \Sigma_u = \bigcup_{i=1}^{m} E_{u,i} \tag{2.3}$$

be the set of controllable events and the set of uncontrollable events in the system, respectively. Furthermore, let

$$g_c = \bigvee_{e \in \Sigma_c} g_e \tag{2.4}$$

be the guard that evaluates to true when any controllable event is enabled. Using the automaton in Figure 2.18, controllable events are executed (Σ_c) until there are none enabled (**when** $\neg g_c$). It then transitions to the second location where an uncontrollable event is taken (Σ_u). By only marking the second location, it is ensured that a marked location is always reached at the end of a controllable event sequence.

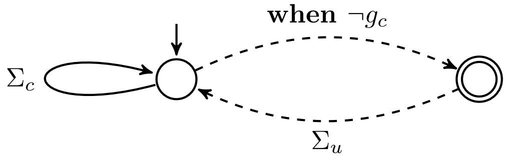


Figure 2.18: The additional automaton used during synthesis that enforces the priority of controllable events.

The finite response property and the confluence property can only be checked if the complete state space of the model is checked, which is infeasible due to state-space explosion. Instead, in Reijnen et al. (2019a), sufficient conditions are defined to determine finite response and confluence that can be checked without calculating the complete state space. Furthermore, algorithms are implemented that check for these sufficient conditions.

2.5.3 Implementation code generation

For the implementation of the supervisory controller on a PLC, controller code has to be derived from the supervisor. One method of code generation is described in Fabian and Hellgren (1998). Here, first the state space of the model is calculated, followed by the generation of the PLC code. In many real-life case studies, calculating the complete state space is infeasible.

In Swartjes et al. (2014), a method is proposed to normalize the EFAs and remove the synchronous behavior. In this procedure, each EFA is transformed such that only one location remains, and the original location information is included by means of a location pointer variable. This location pointer variable is local to its EFA. Subsequently, all the EFAs with synchronous events are merged into one normalized locationless EFA. Note that the original (local) variables are preserved. In Figure 2.19, the result of this procedure is shown for the machine-buffer example from Figure 2.11. As is seen, both automata have a single location, and location pointer variables `LP` are introduced to include the location information of the original automata. These variables are included in the guards and updates of the events. In the synchronization step of the procedure, the synchronized event `u_done` has been removed from the buffer automaton. In this way, for each event, exactly one edge is defined. Thus, the need to calculate the whole state space is omitted.

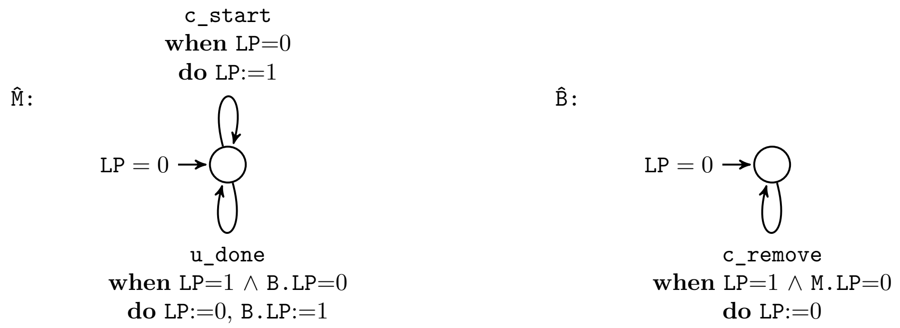


Figure 2.19: The normalized locationless EFAs of machine `M`, buffer `B`, and requirement `R` following the method of Swartjes et al. (2014).

Using the method of Swartjes et al. (2014), a block of PLC code is generated for each event in the system. The drawback of this method is that the model structure is lost, and the code becomes hard to interpret. In Reijnen (2020), an adapted method is proposed, which preserves the structure of the original model. In that method, a block of code and a set of variables is generated for each EFA. Within that block of code, for each transition in the EFA a block of code is generated. This approach allows the control engineer to trace each part of the generated code back to the original controller model.

Chapter 3

Road tunnel systems

The goal of this thesis is to investigate whether synthesis-based engineering is applicable for the design of road tunnel supervisory controllers. To give an extensive overview of a road tunnel system and its desired supervisory controller functionality, in this chapter the road tunnel system is introduced and its interfaces and boundaries are established. Section 3.1 gives a description of the road tunnel system, and shows an overview of the subsystems covered by it. Subsequently, the operator interface of the road tunnel is explained in Section 3.2.

3.1 System description

In the Dutch national tunnel standard (Landelijke Tunnel Standaard, LTS), RWS defines a road tunnel as "an enclosed part of the road, separated from the surrounding environment, with the aim of crossing beneath other infrastructure, often waterways, and/or increasing the quality of life of the surrounding area". RWS discerns three different road tunnel terms depending on the system boundaries. Figure 3.1 visualizes the distinction between these three terms. The description in the LTS concerns the tunnel, whereas the tunnel system includes both the tunnel and technical installations inside that tunnel. Finally, the RWS tunnel system covers the entire set of services, including the tunnel system, service buildings, operating rooms, and surrounding terrain.

For the purpose of supervisory controller design, the system boundaries of the tunnel system in Figure 3.1 are used. The control system of, for instance, the service buildings is left outside this scope.

Under normal circumstances, a tunnel and its controller are mostly idle, as traffic can simply drive through the tunnel. In case of an emergency, however, the controller must coordinate the components in the tunnel to safely handle the emergency and the evacuation. For this reason, the two main functions of

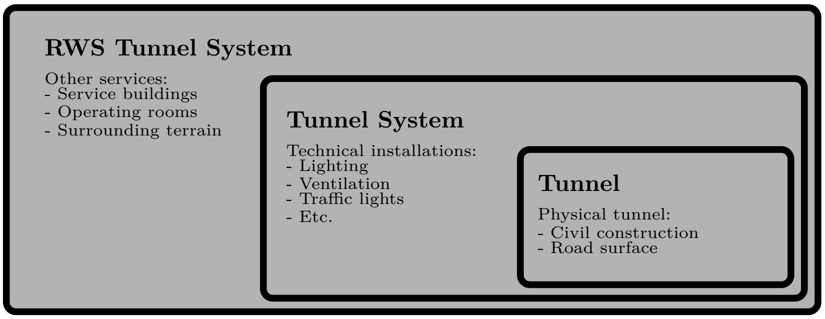


Figure 3.1: Distinctions within the RWS tunnel system definitions.

the controller are to detect when an emergency is happening and to correctly handle it. Emergency detection is done using sensors in the tunnel that monitor, for example, traffic flow, smoke formation, and water flooding. The function of emergency handling can be divided further into sub-functions, being closing the traffic tube, regulating environmental conditions, preparing the escape route, and readying the water cellars. The following subsections describe these functions in more detail.

3.1.1 Detecting emergencies

One of the main functions of the tunnel controller is detecting when an emergency occurs. This is done using three different types of sensors, as well as by using closed-circuit television (CCTV). These components are visualized in Figure 3.2 and indicated by numbers. The first type of sensors is used to detect when traffic is driving too slow or is even standing still in the traffic tube (1). These sensors are induction loops embedded in the road surface that can detect when traffic is moving below a certain threshold. Secondly, emergencies are detected using smoke detection (2). These sensors are used to keep track of visibility in the tunnel. A low visibility indicates smoke formation, and therefore indicates fire in the traffic tube. The third type of sensors is connected to the emergency cabinets in the traffic tube (3). Each cabinet contains a hand-held fire extinguisher, a fire hose, and an emergency phone. There exists a sensor for each component that detects if that component is being used. Furthermore, there is a sensor that detects if the cabinet itself is open or closed. Note that Figure 3.2 only gives an overview of the types of emergency detection systems, and does not represent the actual numbers of components in a tunnel. Emergency cabinets, for example, have a maximum distance of 60 meters between them, meaning that a real-life tunnel contains more than two emergency cabinets.

The controller is designed to detect an emergency when the following conditions hold. First of all, both a standstill detection and a smoke detection are required.

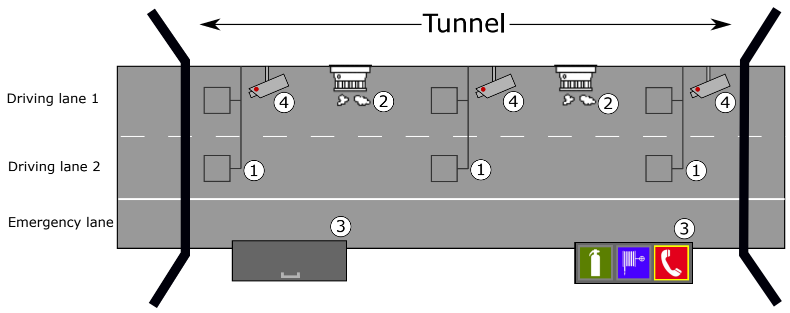


Figure 3.2: Overview of the emergency detection components: Standstill detection (1), smoke detection (2), emergency cabinets (3), and CCTV (4).

Furthermore, at least two of the following conditions must hold:

- Detection of a fire extinguishing tool being used (either a hand-held fire extinguisher or a fire hose).
- Detection of an emergency cabinet being open.
- Detection of an emergency phone being used.

Once an emergency is detected based on these conditions, the tunnel operator is notified of this emergency. The operator can then use the CCTV (4) to assess the situation and either confirm or reject the emergency notification. When the operator does not respond within 30 seconds after the notification, the emergency is confirmed automatically. Through this automatic confirmation process, the control system can function fully autonomously. It is, however, generally undesired that an emergency is declared without human confirmation. Furthermore, the operator is always able to declare an emergency using the operator interface, even when none of the conditions above hold.

When an emergency is declared in a traffic tube, either through automatic detection or manual action, the other traffic tube is designated as the supporting traffic tube. This supporting tube is used for evacuation purposes and to reach the emergency tube by emergency services such as the fire brigade.

Moreover, the operator can control the CCTV whenever desired through the operator interface and by using hardware such as a joystick. This includes viewing a specific camera in the tunnel, controlling the pan-tilt-zoom of a specific camera, and watching all cameras in a tube in sequence. The operator interface is described in more detail in Section 3.2.

3.1.2 Closing the traffic tube

One of the responses of the tunnel controller to an emergency is to close both the emergency traffic tube and the supporting traffic tube. This is done using the four components shown in Figure 3.3. First, the traffic lights (1) are turned to a flashing yellow state, and the triangular J32 sign (2) is turned on to notify drivers that the traffic lights are on. Furthermore, the matrix signs (3) are turned on to impose a reduced speed limit. The traffic lights go through the states of flashing yellow, continuous yellow state, and finally red, each with a specified duration. Once the traffic lights show a red light, the boom barriers (4) will be lowered to physically close the traffic tube.

When opening the traffic tube, the inverse process is followed: the boom barriers are opened, the traffic lights go through the states flashing yellow and off, and the J32 sign and matrix signs are turned off.

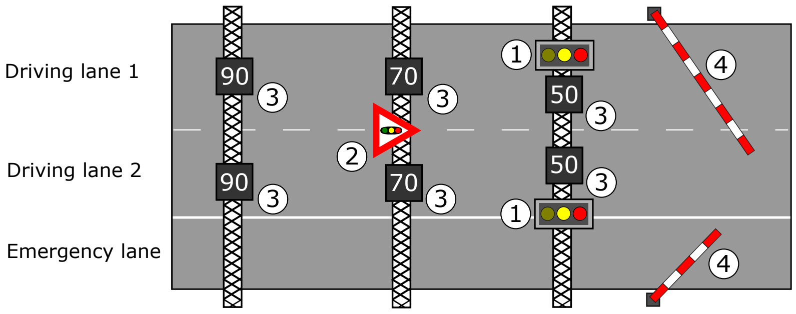


Figure 3.3: Overview of the traffic tube closing components: Traffic lights (1), J32 sign (2), matrix signs (3), and boom barriers (4).

Besides the autonomous closing of the traffic tube, the human operator can also manually send commands to the components shown in Figure 3.3. He can, for instance, open the boom barriers during an emergency to let emergency services enter the tunnel.

3.1.3 Preparing the escape route

The second response of the tunnel controller to a detected emergency is preparing the escape route for evacuation. The LTS describes four possible escape routes that can be implemented in a tunnel, which are visualized in Figure 3.4. The most common route uses a hallway between the two traffic tubes, called the middle-tunnel channel. This hallway can be accessed from either traffic tube through escape doors. The middle-tunnel channel can be left through a head door (a) or through the last door of the supporting traffic tube (b). In the second case, people must wait in the middle-tunnel channel until the supporting traffic tube is closed and free of traffic. A third escape route uses cross connections (c), which are small rooms between the traffic tubes. Similar to route (b), people can wait in these rooms, and evacuate to the supporting traffic tube. Finally, some older tunnels have a dividing wall as escape route (d). This is the most dangerous route, as people directly evacuate to the supporting traffic tube without a place to wait for it to be free of traffic.

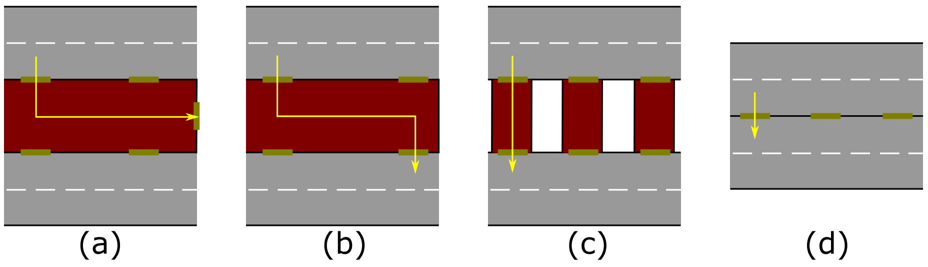


Figure 3.4: Overview of the escape routes: middle-tunnel channel with head door (a), middle-tunnel channel (b), cross connections (c), and dividing wall (d).

Newly constructed or renovated tunnels typically have a middle-tunnel channel as escape route, so this type is explained here in more detail. The other escape route types are less common and, at the same time, are simpler in terms of controller design.

Several components in the middle-tunnel channel are readied when it is being prepared for evacuation. These components are shown in Figure 3.5. Note that Figure 3.5 shows only two sets of escape doors, whereas the actual middle-tunnel channel contains a set of escape doors every 50 meters. When an emergency is detected, the escape doors (1) are unlocked and a broadcasting system (2) plays an audio message to indicate where the escape doors are. Furthermore, signs are turned on inside the middle-tunnel channel to indicate the direction of evacuation (3). Finally, a pressure system (4) is used to increase the air pressure inside the middle-tunnel channel to prevent smoke from entering from the traffic tubes.

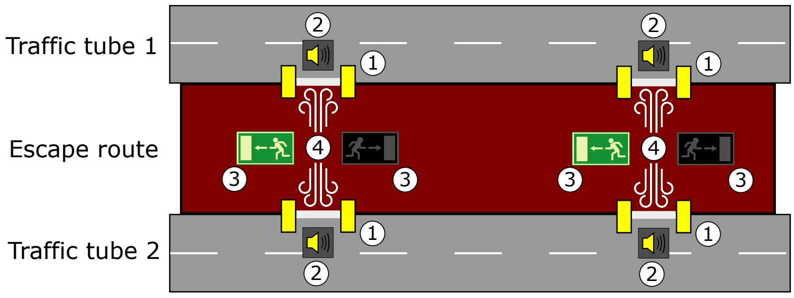


Figure 3.5: Overview of the escape route components: Escape doors (1), broadcasting (2), route indication (3), and pressure system (4).

3.1.4 Regulating environmental conditions

Another function of the tunnel controller is regulating the environmental conditions to maintain sufficient visibility in the tunnel.

First, visibility in the tunnel is maintained by ensuring a sufficient air quality. This is done using ventilation units that regulate the air flow through the traffic tubes and thus clear it from smoke. The rotation speed of the ventilation units depends on the sight measurements of the smoke detection, meaning that they rotate at a higher speed when a higher smoke level is measured. In case of an emergency, the ventilation speed is always set to the maximum setting. One important part of the control functionality is that changes in the rotation speed occur incrementally. This is due to the fact that sudden large changes in rotation speed in all the ventilation units requires too much energy, which could result in a power failure. Moreover, the forces generated by a ventilation unit that switches to its maximum mode ca result in failing fixtures, and thus breaking down of the ventilation units.

Second, visibility in the tunnel is maintained by regulating the light level in the tunnel. The lighting in the traffic tubes has several different settings that vary in light level. The main goal is to have the light level inside the tunnel resemble the light level outside the tunnel to prevent blinded drivers because of a too big difference in light level. The tunnel controller is therefore connected to a light sensor outside of the tunnel. During an emergency, on the contrary, the lighting is set to the highest setting to maximize visibility. Similar to the ventilation system, the light level must be increased and decreased incrementally to prevent power failures and to avoid sudden changes in the light level.

3.1.5 Readying the water cellars

The final function of the tunnel controller in response to an emergency is readying the water cellars that lie below the tunnel. There are two types of water cellars, being the pump cellars for drainage and the water cellar for supplying the fire hoses.

The pump cellars for drainage are shown in Figure 3.6. These cellars are used to drain rain water from the traffic tubes, as well as fluids that might leak during an emergency. As can be seen, the tunnel contains three pump cellars for drainage, consisting of two head pump cellars at each end of the tunnel, and one middle pump cellar located at the center of the tunnel. Each pump cellar contains a set of sensors to measure the water level in the cellar, as indicated by the dashed lines in Figure 3.6. Furthermore, a set of pumps is present at each pump cellar to pump out the water. For the middle pump cellar, the direction can be regulated to which head pump cellar the water is pumped. Depending on the water level in the cellar, one or more pumps are used. Finally, each pump cellar can be set to three different regimes: keeping the cellar empty, storing as much water as possible, or turning the pumps off.

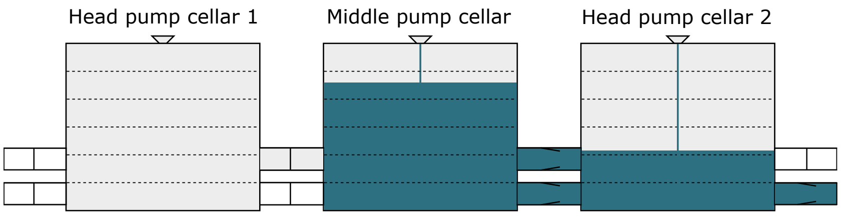


Figure 3.6: Overview of the pump cellars for the draining of rain and leaked fluids.

The second type of water cellar is shown in Figure 3.7. It is needed to supply the fire hoses in the traffic tubes with water. It consists of two sensors to measure the water level, and one pump to fill the cellar. The pump is turned on when the water cellar is not full, and an emergency is detected or one of the fire hoses is being used. The pump is turned off once the water cellar is full.

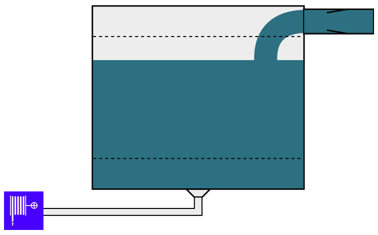


Figure 3.7: Overview of the water cellar connected to the fire hoses.

3.2 Operator interface description

Every tunnel in the Netherlands is at all times monitored and operated by a human operator through an operator interface. The operator monitors the situation in the tunnel through CCTV images and information displayed on the operator interface. This information includes visualization of the tunnel state, e.g. the current state of a traffic light, and notifications shown in the interface, e.g. a sensor detection for slowly driving traffic. The second purpose of the operator interface is operating the tunnel. Operation of the tunnel is done through the operator interface by sending commands to specific components in the tunnel. The tunnel control system is able to work fully autonomously, though a human operator can at any moment intervene when deemed necessary.

In the LTS, RWS defines specifications for the tunnel operator interfaces in the Netherlands. Furthermore, based on these specifications, the LTS gives a standard design for the operator interface. Because of the standard design, each tunnel is operated through a similar interface, which facilitates operation and maintenance.

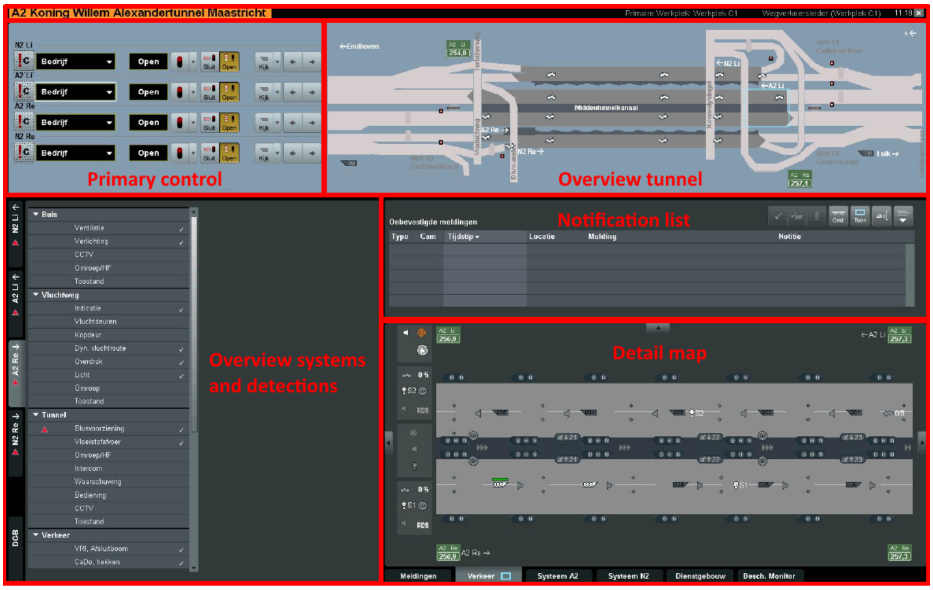


Figure 3.8: Overview of the operator interface of the KWA tunnel.

The operator interface of the Koning Willem-Alexander (KWA) tunnel, which is based on the standard design of the LTS, is shown in Figure 3.8. This interface is divided in the following sections, as indicated by the red boxes in the figure:

- **Primary control**
  The primary control section contains important information about the current tunnel state and buttons that need to be easily accessible. These buttons are mainly related to closing the tunnel and operating the CCTV.

- **Overview tunnel**
  The overview tunnel section is a schematic view of the tunnel and its direct surroundings. Icons in this view indicate the state of components like the boom barriers and ventilation units.

- **Overview systems and detections**
  The overview systems and detections section gives a complete overview of all systems in the tunnel. It contains all buttons that can be used for operating a specific component.

- **Notification list**
  The notification list section shows all notifications received from the supervisory controller, such as alarm notifications or system failures.

- **Detail map**
  The detail map section gives a more detailed overview of a specific section of the tunnel, as requested by the road traffic controller.

An operator interface, as designed following the guidelines of RWS, has no control functionality, meaning that all decision making and information processing is done by the supervisory controller or resource controllers. The operator interface merely sends signals to these controllers, and visualizes inputs received from them. It is, however, important to include the operator interface in the design process of the supervisory controller. Firstly, the supervisory controller can only be correctly designed when the appropriate input and output signals with regard to the operator interface are considered. Secondly, validation of the supervisory controller by means of simulation holds more value when the same operator interface is used as in the realized tunnel.

Chapter 4

Parameter-based modeling

One of the main challenges of applying SCT in practice is the knowledge and effort that is required to create correct models. For industrial systems, which often consist of a large number of components, modeling can be a cumbersome task. Furthermore, few control engineers have the required knowledge on SCT, as they are trained to be software programmers instead of supervisory control engineers.

In this chapter, the design of supervisors for a product platform is proposed, with the aim to make modeling in SCT more efficient and more accessible. A product platform is described in Meyer and Lehnerd (1997) and Harland et al. (2020) as a collection of modules and components that are common to a number of products. This commonality is introduced intentionally to gain several benefits, such as a higher development speed and lower development costs.

The modeling platform that is proposed in this chapter is a parameter-based platform. It is inspired by the concept of decision support systems, as described in Minch and Burns (1983) and Overstreet and Nance (1985), in which parameters are used to define a system within a product platform. Our platform, however, differs from decision support systems as it focuses on model generation instead of decision making and model management. Our modeling platform can be used to generate the models that are required for supervisor synthesis and simulation from the set of parameters that define the system. A template library is used in this approach to instantiate modules in the product platform based on the entered parameters.

This chapter is based on: Moormann, L., van de Mortel-Fronczak, J.M., and Rooda, J.E. Design of a parameter-based modeling platform for road tunnel supervisory controllers. In *2021 IEEE Conference on Control Technology and Applications (CCTA)*, pages 1024–1030. IEEE, 2021.

The use of templates is comparable to modeling approaches described in Grigorov and Rudie (2010), Grigorov et al. (2011), and Malik et al. (2011). In Grigorov et al. (2011), extensive research is presented on template-based design, in which discrete-event systems are modeled using a set of templates for the components and the requirements. They conclude that template-based design greatly improves the accessibility of modeling for supervisor synthesis as well as the speed of modeling itself. In the approach of Grigorov et al. (2011), a template library is used, which is a repository of discrete-event models that can be instantiated to model the system. Our platform uses such a template library as well, though instead of manually instantiating all components in the system, the templates are instantiated automatically based on the parameters of the system. In Grigorov and Rudie (2010), the approach of Grigorov et al. (2011) is extended by introducing techniques for the parametrization of templates. With these techniques, the behavior of a template can be altered by specifying a parameter, e.g. the capacity of a buffer. This, however, differs from our parameter-based modeling platform, as the parameters in Grigorov and Rudie (2010) are used to construct the behavior of a specific component, while our platform also includes high-level parameters that construct the complete system. In Malik et al. (2011), the techniques of Grigorov and Rudie (2010) are extended and implemented to support parameterized modules, which consist of a set of components with defined interfaces. Our platform also uses such modules, though in our method the parameters are not only used to define the modules, but also to automatically perform the instantiation process. In Grigorov et al. (2011), Grigorov and Rudie (2010), and Malik et al. (2011), this instantiation process of the templates is a manual, and thus more time-consuming, task.

Several benefits are obtained by allowing the control engineer to define a system using a set of parameters and, subsequently, generate the models for supervisor synthesis. First, modeling systems within the product platform becomes a quick and almost effortless process as long as the system parameters are known, and adaptations can be made quickly by changing the corresponding parameters. Second, no knowledge on SCT is required from the control engineer, as the models that are needed for supervisor synthesis and simulation are generated automatically. Third, the quality of the controller increases as the possibility of human errors is reduced further since the models are not created manually.

The concept of a modeling platform to automatically generate models for supervisor synthesis and simulation is not new, as this has been shown before in Reijnen et al. (2020b). The platform, however, differs from that work as the type of systems that is considered in this thesis requires a different modeling approach, namely a parameter-based approach. The systems considered in Reijnen et al. (2020b) contain a relatively low number of components. Furthermore, there exists more variability between the configuration of the modules and components in those systems, thus requiring more freedom for the user. The disadvantage of this freedom is that it becomes more time consuming as the system gets larger, and more knowledge on modeling formalisms and synthesis algorithms in SCT is required as interfaces between modules, i.e. the requirement models, need to be specified by the user. The systems considered in our modeling platform contain large numbers of modules and components, and the variation in configurations and interfaces between these modules and components can be captured in the parameters. Therefore a platform is chosen where, compared to Reijnen et al. (2020b), the user has less freedom in configuration design of the system. In return, the modeling process is more efficient, and much less knowledge on SCT is required as both the plant components and the requirements are generated automatically.

The product platform in this chapter considers supervisory controllers for road tunnel systems. Road tunnels are infrastructural systems consisting of many components that have to cooperate together correctly to ensure a safe environment for road users. Creating a parameter-based product platform for road tunnels is a suitable approach, since all road tunnels resemble each other as they contain the same modules and components. The road tunnels can, however, vary in configuration and dimensions, which is captured in the parameter set that defines the system.

This chapter is organized as follows. In Section 4.1, the component-based modeling method for the plant model and requirements model is described. Moreover, a running example is introduced, pertaining a pump-cellar system, to showcase the component-based modeling method. Subsequently, in Section 4.2, the parameter-based modeling method is introduced and applied for the running example. The parameter-based modeling method is implemented in a prototype tool, described in Section 4.3, and its applicability is shown in a case study for a family of tunnels, as shown in Section 4.4. Finally, in Section 4.5 some concluding remarks are made.

4.1 Component-based modeling method

There exist several approaches to modeling the plant behavior, varying in, e.g., abstraction level and the interfaces with adjacent systems. In this section, the approach of component-based modeling is used. Component-based modeling is described in Gössler and Sifakis (2005) and Majdara and Wakabayashi (2009), and applied for, respectively, correct-by-construction network design and automated fault tree generation. In this approach, the chosen abstraction level and interfaces are based on the input and output connections of the control unit. This holds for both the connections to the plant and the connections to the operator interface. One of the benefits of this approach is that this gives a clear interface with the surrounding systems that, depending on the type of project, might already be fixed. Furthermore, the abstraction level of inputs and outputs enables the option to automatically generate implementation code from the synthesized supervisor.

A third benefit is that using this approach, many of the modeled components become loosely coupled, i.e. they do not share events and each represent a specific component in the system, such as individual actuators and sensors. This makes the modeling of the components and the requirements less error-prone and more intuitive, and is therefore faster and requires less effort. Component-based modeling has been used successfully for the purpose of supervisory control theory in previous works, as can be seen in Huang et al. (2015), Kovács et al. (2012), and Reijnen et al. (2020a).

As mentioned, the abstraction level at which the components are modeled is based on the input and output connections of the controller. There are, however, two exceptions to this rule. First, certain models are included to model the interactions between components that limit their physical behavior, as is discussed in Gössler and Sifakis (2005). An example of such a physical relation between two components is shown in Figure 2.5 in Section 2.2. Second, an exception to the input–output abstraction level involves information that needs to be memorized by the controller. For instance, as described in Chapter 3, the tunnel controller detects an emergency when a certain combination of conditions is met. The controller must memorize that an emergency was detected, even when the conditions no longer hold. An automaton is therefore modeled that represents this memory, which is neither an input nor an output of the controller.

The loosely coupled component models also allow for modeling using templates, as introduced in Grigorov et al. (2011). Such a template represents the behavior of a certain frequently occurring component, such as an actuator, and is instantiated for each of the actuators in the system. Therefore, using templates makes the modeling of systems with reoccurring components much more efficient.

When defining the requirements for supervisory synthesis, the requirements should relate to events and locations in the discrete-event plant model. In the component-based modeling method, this means that the requirements typically refer to the input and output connections of the controller, and in some cases to a memory component. While this may seem like a restriction on the requirement modeling process, practice shows that such requirements are defined more intuitively and more clearly when they directly relate an output to one or more inputs. Furthermore, the component-based modeling of the plant results in a larger set of smaller requirements that each relate to a specific component. These smaller requirements are typically more straightforward and transparent than larger, often convoluted, requirements.

From the discrete-event plant model and the requirements model, a supervisor is synthesized. Supervisors are often validated by means of simulation, and while it is possible to simulate the discrete-event model of the plant, it is often worth the effort to create a more advanced, hybrid, model that more closely resembles the real system. This hybrid model consists of the discrete-event model of the plant as well as hybrid automata, as introduced in Henzinger (2000). This hybrid plant model ensures that output signals of the supervisor are correctly processed by the plant and that the correct input signals are provided to the supervisor. Furthermore, the hybrid plant model enriches the discrete-event plant model with continuous time behavior. In component-based modeling, the hybrid plant model is created more easily as each input signal and each output signal is represented by an individual automaton in the discrete-event plant model. This allows the design engineer to also create a component-based hybrid plant model, where a separate hybrid automaton is created for each signal.

As an example, the hybrid plant model of a boom barrier is discussed. The supervisor sends output signals to the actuators of the boom barrier to either open or close it, and receives input signals from the boom barrier about the current position. The hybrid plant model of this boom barrier includes a continuous variable that represents the current height of the boom barrier. When the supervisor gives the output signal to open the boom barrier, the value of this variable is increased to indicate a rising boom barrier, and once the variable surpasses a specified value, the hybrid plant model gives the input signal to the supervisor that the boom barrier is fully opened. Subsequently, the supervisor decides that the boom barrier must be stopped, and gives the corresponding output signal.

Besides the continuous-time enrichment of the discrete plant, the hybrid plant often also includes a visualization. To this end, a scalable vector graphics (SVG) file is created, as described in Quint (2003), in which object properties are connected to events, locations, and variables in the hybrid model. Such a visualization gives a clearer and more intuitive representation of the current state of the system. Moreover, objects in the SVG can be linked to events in the plant model such that input signals can be given during a simulation. For instance, when a button exists in the GUI, a clickable object can be created in the SVG that represents that button. An event then is connected to this clickable object, such that is it executed when the object is clicked in the SVG during a simulation. This way, the correct operator interface can be created, which makes validating the supervisor through simulations easier and more meaningful.

4.1.1 Running example: Pump-cellar system

Throughout this chapter, a running example is used to demonstrate the proposed methods of component-based and parameter-based modeling, described in Sections 4.1 and 4.2, and the proposed configuration tool, described in Section 4.3. The running example concerns the pump-cellar system of a road tunnel, as is introduced in Chapter 3. Such a pump-cellar system consists of several pump cellars, which each have a set of sensors to determine the water level in the cellar, and a set of pumps to empty the cellar. An example of a pump-cellar system is shown in Section 3.1, Figure 3.6, which consists of three pump cellars that each have five sensors and two pumps.

The plant model

Following the component-based modeling method, a separate model is created for each component in the system based on the inputs and outputs of the controller. In the pump-cellar system, the outputs of the controller are analyzed, which in the case of the pump-cellar system are the two pumps in each of the pump cellars. An automaton is therefore created for each individual pump `P`. The automaton of pump $\mathtt{P}_1$ is shown in the top-left corner of Figure 4.1. Next, the inputs of the controller are the five sensors in each pump cellar that determine the water level. Once again, an automaton model is created for each of these sensors. The automaton for sensor $\mathtt{S}_1$ is shown in the middle at the top of Figure 4.1.

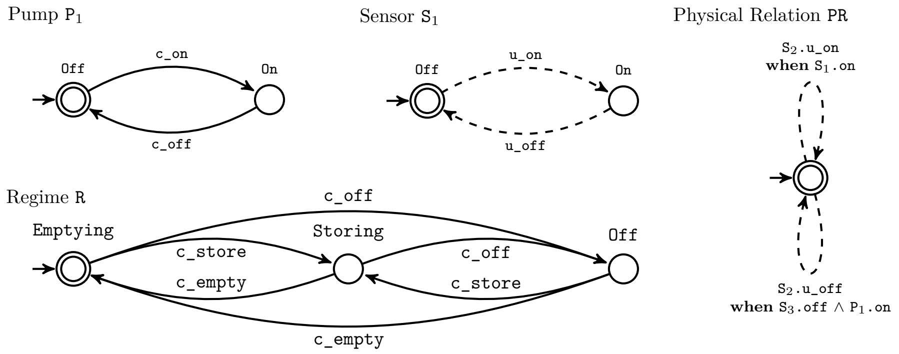


Figure 4.1: Discrete-event plant model of the components of a pump cellar.

One of the two exceptions to the input-output abstraction level in the component-based modeling method are the additional models created to capture the physical behavior between components in the system. For the pump-cellar system, physical relations exist between the sensors in a pump cellar, as well as between the sensors and the pumps in that cellar. On the right-hand side of Figure 4.1, an example of this physical relation is shown specifically for the second sensor in a pump cellar. This physical relation adds guards to the events of the sensor, as these events are only physically possible under certain conditions. First, the sensor can only turn on ($\mathtt{S}_2$`.u_on`) when the sensor below is already on ($\mathtt{S}_1$`.on`), since the water level can never be above S_2 and below S_1 at the same time. Similarly, a guard is added to turning the sensor off ($\mathtt{S}_2$`.u_off`) in relation to the sensor above ($\mathtt{S}_3$). Moreover, there exists a physical relation between a sensor and the pumps in a pump cellar, namely the relation that a sensor can only turn off when the pump is emptying that cellar ($\mathtt{P}_1$`.on`). Both these relations are included in the guard of the $\mathtt{S}_2$`.u_off` event in the automaton on the right-hand side of Figure 4.1. Note that these physical relations are modeled under the assumption of nominal behavior, i.e. the actuators and sensors in the plant never fail.

The second exception to the input-output abstraction level concerns the memorized information of the controller. For the pump-cellar system, the controller must memorize the current pumping regime `R` of each pump cellar. There are three different regime options that indicate if the cellar should be as empty as possible (`Emptying`) or as full as possible (`Storing`), or if the pumps should be off regardless of the water level (`Off`). The automaton for this regime `R` is shown in the bottom-left corner of Figure 4.1.

The requirements model

The requirements of a water pump cellar define the conditions under which the pump regime changes, and under which the pumps are turned on and off. Here, an example is given of the requirements for a pump cellar with 2 pumps and 5 sensors. These requirements are listed below in textual form.

1. The pumping regime may only be set to `Storing` when
   (a) the traffic tube is in emergency mode, or
   (b) the corresponding button is pushed.
2. The pumping regime may only be set to `Emptying` when
   (a) the traffic tube is in operational mode, or
   (b) the corresponding button is pushed.
3. The pumping regime may only be set to `Off` when
   (a) the corresponding button is pushed.
4. Pump 1 may only be turned on when
   (a) the pumping regime is `Storing` and Sensor 5 is on, or
   (b) the pumping regime is `Emptying` and Sensor 2 is on.
5. Pump 2 may only be turned on when
   (a) the pumping regime is `Storing` and Sensor 5 is on, or
   (b) the pumping regime is `Emptying` and Sensor 3 is on.
6. Pump 1 may only be turned off when
   (a) the pumping regime is `Storing` and Sensor 4 is off, or
   (b) the pumping regime is `Emptying` and Sensor 1 is off, or
   (c) the pumping regime is `Off`.
7. Pump 2 may only be turned off when
   (a) the pumping regime is `Storing` and Sensor 4 is off, or
   (b) the pumping regime is `Emptying` and Sensor 1 is off, or
   (c) the pumping regime is `Off`.

The requirements for the pump cellar are modeled as event-condition requirements, as this closely resembles the textual form as seen above, and is therefore the most intuitive approach. Table 4.1 gives an overview of these requirements for the pump cellar, and shows the condition for each controllable event. In these requirements, the pumping regime is denoted as `R`, the pumps as $\mathtt{P}_1$ and $\mathtt{P}_2$, and the sensors as $\mathtt{S}_1$ (lowest) through $\mathtt{S}_5$ (highest).

Table 4.1: Event-condition requirements for a water pump cellar with 2 pumps and 5 sensors.

Req.	Event	Condition **(a)**	Condition **(b)**	Condition **(c)**
1	`R.c_store`	`TrafficTubeMode.Emergency` $\vee$	`ButtonStoring.Pushed`	
2	`R.c_empty`	`TrafficTubeMode.Operational` $\vee$	`ButtonEmptying.Pushed`	
3	`R.c_off`	`ButtonOff.Pushed`		
4	$\mathtt{P_1.c\_on}$	$(\mathtt{R.Storing} \wedge \mathtt{S_5.On}) \vee$	$(\mathtt{R.Emptying} \wedge \mathtt{S_2.On})$	
5	$\mathtt{P_2.c\_on}$	$(\mathtt{R.Storing} \wedge \mathtt{S_5.On}) \vee$	$(\mathtt{R.Emptying} \wedge \mathtt{S_3.On})$	
6	$\mathtt{P_1.c\_off}$	$(\mathtt{R.Storing} \wedge \mathtt{S_4.Off}) \vee$	$(\mathtt{R.Emptying} \wedge \mathtt{S_1.Off}) \vee$	`R.Off`
7	$\mathtt{P_2.c\_off}$	$(\mathtt{R.Storing} \wedge \mathtt{S_4.Off}) \vee$	$(\mathtt{R.Emptying} \wedge \mathtt{S_1.Off}) \vee$	`R.Off`

The hybrid plant model

For the purpose of simulation, a hybrid plant model is created. For a water pump cellar, the hybrid plant model introduces a continuous variable that represents the water level in the tank. A relation is modeled between the sensors and this water-level variable, as shown on the left-hand side in Figure 4.2. This ensures that the sensor events occur automatically during simulation when the water level reaches a certain height. Furthermore, the option to let rain water fill the pump cellars during the simulation is added by modeling the occurrence of rain, as shown in the middle of Figure 4.2. Finally, a differential equation of the water level is modeled, as shown on the right-hand side of Figure 4.2, to connect the water-level variable to the rain and to the mode of pumps.

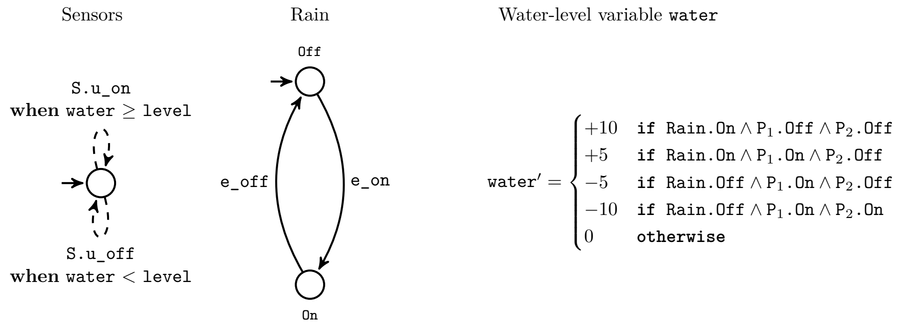

Sensors
Rain
Water-level variable water
S.u_on
when water ≥ level
S.u_off
when water < level
Off
e_off
e_on
On
water′ =
+10 if Rain.On ∧ P1.Off ∧ P2.Off
+5 if Rain.On ∧ P1.On ∧ P2.Off
−5 if Rain.Off ∧ P1.On ∧ P2.Off
−10 if Rain.Off ∧ P1.On ∧ P2.On
0 otherwise

Figure 4.2: Hybrid plant model of a pump cellar.

To create an intuitive simulation model, often a visualization is created and connected to the discrete-event plant model and the hybrid plant model. It consists of an SVG, in which, e.g., the color, visibility, position, and rotation of objects are linked to locations and variables in the discrete and hybrid automata through SVG declarations. For instance, in the case of the pump-cellar system, the modes of the pumps and the rising or falling water level are animated this way. The visualization used during simulation of the pump-cellar system is shown in Section 3.1, Figure 3.6. The SVG declarations specify how high the water is visualized depending on the continuous variable of the water level. Furthermore, the visibility of the inflow depends on the occurrence of rain, and finally the visibility of the outflow depends on the activity of the pumps.

4.2 Parameter-based modeling method

In this section, the process of parameter-based modeling is described, the modules of road tunnel systems are explained, and the way modules are connected in different tunnel configurations is elaborated.

The modeling method that is proposed in this chapter is a parameter-based modeling method. The general process is visualized in Figure 4.3. The input files in this process are shown on the left-hand side, being the template library and the parameter list. The template library contains model templates, such as described in Grigorov and Rudie (2010) and Grigorov et al. (2011), for all common modules and parts in the product platform. These templates can consist of both discrete-event automata and requirements for the discrete-event plant model, and hybrid automata for the hybrid plant model. The parameter list contains a set of parameters that defines a system within the product platform. In this method, the required effort of the design engineer shifts from creating a large set of models to creating a library of templates. An additional benefit of creating such a library is seen when multiple similar systems have to be modeled. In that case, the effort of creating the template library is only required once, and it can then be used to model all systems belonging to the same class.

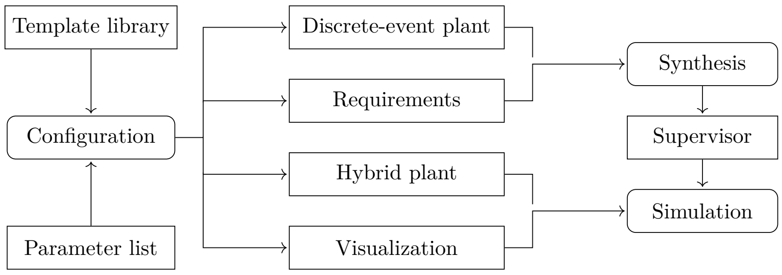


Figure 4.3: Overview of the parameter-based modeling process.

The template library and the parameter list are used as the inputs for the configuration tool. This tool processes the data and generates the models and visualizations that are required in the synthesis-based engineering process. These files are shown in the middle of Figure 4.3. They contain the discrete-event plant model, the requirements model, the hybrid plant model, and the SVG that visualizes the system.

The configuration tool creates the plant model directly from the template library. The template library contains a template definition, such as the definition for an actuator shown in Figure 4.4. A component in the plant model is created by the configuration tool by using this definition and supplying the name of the component and the required parameters. The parameters of the example in Figure 4.4 define selectable marked locations with `OffMarked` and `OnMarked`. If, for example, a model for a lamp with the `On` location as the only marked location is needed, this is expressed as `Lamp : Actuator(False,True)`.

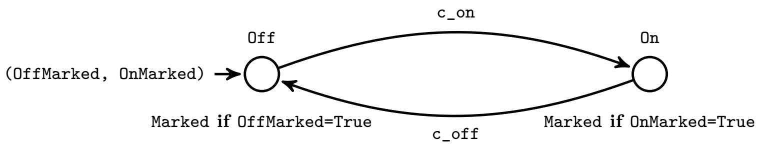


Figure 4.4: Template definition of an actuator with selectable marked locations.

Requirements can be added by the configuration tool in a similar way, by creating a template definition of one or more requirements, but this is often done without a template. As most requirements are related to multiple component models to ensure correct cooperation, and thus depend on many different parameters, the contents and the length of the requirements need to be more versatile. Instead of using a template, the requirement is created in the configuration tool itself. For this purpose, the requirements are defined in the algorithm of the configuration tool, and for each requirement it is indicated how they depend on the components in the system. An example of such a requirement is shown in Equation (4.1), where $\mathtt{S}_i\mathtt{.on}$ and $\mathtt{S}_j\mathtt{.on}$ are two different sensor components created by the configuration tool. However, depending on the parameters of the system, there might be more sensors that are needed in this requirement, so the length of the requirement needs to be versatile.

$$\mathtt{A.c\_on}\ \textbf{needs}\ \mathtt{S}_i\mathtt{.on} \wedge \mathtt{S}_j\mathtt{.on}\ \cdots \tag{4.1}$$

As described in Chapter 2, a supervisor can be synthesized from the discrete-event plant model and the requirements model, which is shown at the right-hand side of Figure 4.3. To validate if the discrete-event plant model and the requirements model are correct, simulations are performed using the supervisor, the hybrid plant model, and the visualization SVGs. Errors can be found during simulation, either when requirements are too relaxed so undesired behavior is still possible, or when requirements are too strict so some desired functionality is prevented. The error is often corrected in the template library, but the error can also lie in the configuration tool or the parameter list. After the error is corrected, new files are generated using the configuration tool, a new supervisor is synthesized, and simulations are performed again, making the validation an iterative process.

The product platform of road tunnels consists of a set of common modules. To generate the models for synthesis and simulation, a template library is created for these modules that contains templates for the discrete-event plant, the requirements, the hybrid plant, and the visualization. Furthermore, for each module, a set of parameters is included that defines the instantiation of that module.

The modules of a road tunnel are the traffic tubes, the escape routes, the water pump cellars, and the water tanks. These are the common modules that make up a road tunnel. There are, however, also connections between the different modules. Depending on the configuration of a road tunnel, certain modules, such as the traffic tubes and the escape routes, may or may not be connected. For instance, an escape route is always connected to two traffic tubes. A road tunnel with four traffic tubes and two escape routes thus requires additional information about which escape routes are connected to which traffic tubes. This information is captured in an additional configuration module that always appears exactly once in the road tunnel parameter list. In the next section, an example of this configuration module is provided. Note that the connection between modules can be directional. For example, in the case of a connection between two pump cellars, the direction indicates which cellar pumps water to the other cellar.

4.2.1 Running example: Pump-cellar system

Once again, the pump-cellar system shown in Section 3.1, Figure 3.6, is used as an example, in this case to showcase the parameter-based modeling method. In this section, the modules of the pump-cellar system, the parameter-based modeling of the plant model, the requirements model, and the hybrid plant model are described.

The modules

A pump-cellar system consists of one or more pump-cellar modules. In the example of the system shown in Figure 3.6, the pump-cellar system consists of three pump cellars, where the second pump cellar is able to pump its water to the first and the third pump cellar. To model a pump-cellar system using parameter-based modeling, pump-cellar modules are added to the parameter list. For each module, a set of parameters is required that define that module. In the case of a pump-cellar module, the number of water-level sensors and the number of pumps in that pump cellar are needed.

An overview of the pump-cellar modules in this example is shown at the top of Figure 4.5. In total, three of these modules are added and for each module the required parameters are provided. In this example, each pump cellar contains five sensors and two pumps.

Moreover, the relation between the pump cellars needs to be defined, as this affects the controlled behavior of the system. For this purpose, a configuration

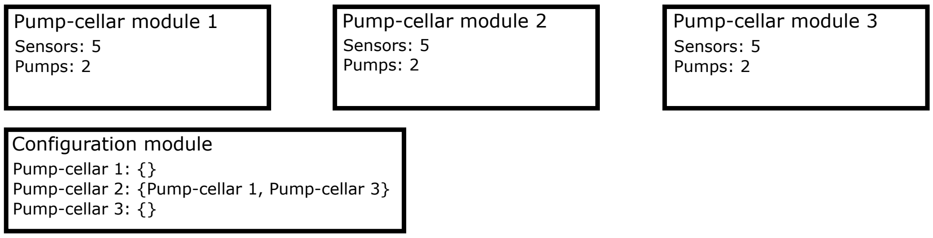


Figure 4.5: Modules of the pump-cellar system.

module is added, as this concerns the configuration between the other modules. This configuration module is shown at the bottom of Figure 4.5. It contains the information of which pump-cellar modules are connected to which other pump-cellar modules. Note that this connection is directional. In this case, the second pump cellar can pump water to the other two cellars.

When a complete tunnel is modeled using this method, the configuration module also contains the information about which escape routes are connected to which traffic tubes, and which traffic tubes a water tank is connected to.

The plant model

The discrete-event plant model of each pump-cellar module consists of a set of pumps $\mathtt{P}_i$, a set of water-level sensors $\mathtt{S}_j$, and a model that describes the current pumping regime R. The model for the pumping regime R is parameter-independent, as this model always appears exactly once in a pump cellar. Contrarily, the number of pump automata and sensor automata in the discrete-event plant model depend on the input parameters. In Figure 4.6, the parameter-based models of the pump and the sensor, adapted from Figure 4.1, are shown. Note that the events and guards in the physical relation automata are parameter-dependent to create the correct physical relations with the other sensors.

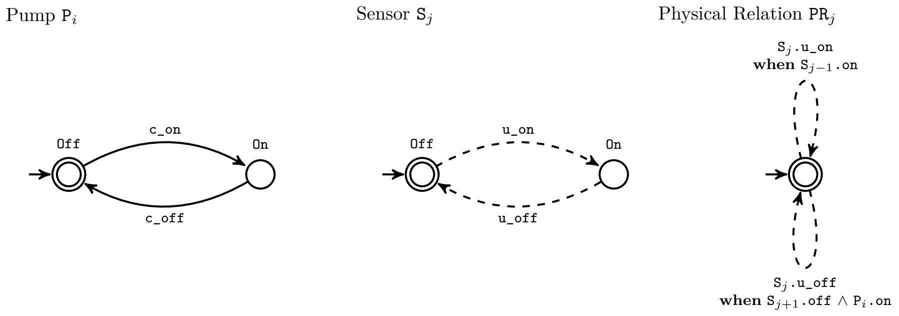

Pump $\mathtt{P}_i$
Off
c_on
On
c_off
Sensor $\mathtt{S}_j$
Off
u_on
On
u_off
Physical Relation $\mathtt{PR}_j$
$\mathtt{S}_j$.u_on
when $\mathtt{S}_{j-1}$.on
$\mathtt{S}_j$.u_off
when $\mathtt{S}_{j+1}$.off $\wedge$ $\mathtt{P}_i$.on

Figure 4.6: Parameter-based discrete-event plant model of a pump cellar.

As the second pump cellar can pump its water to either the first or the third pump cellar, an additional model is needed to capture this pump direction. The automaton for the pump direction in shown in Figure 4.7. Initially, the water is pumped to pump cellar 1, though it can switch to pump cellar 3 when needed.

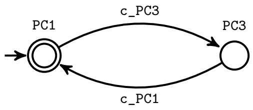


Figure 4.7: Automaton model for the pump direction of pump cellar 2.

The requirements model

Requirements are added to the requirements model for each module, depending on the parameters of that module. Similar to the modeling process of the plant, there are requirements that are parameter-dependent and requirements that are parameter-independent. The requirements for a pump cellar with two pumps and five sensors are shown in Table 4.1. In this case, there always exists one pumping regime model R, so requirements 1-3 are always included once as well, and are thus parameter-independent.

Requirements 4-7 are parameter-dependent, as they depend on the number of pumps and sensors in the system. These requirements are included in the template library, and are instantiated several times depending on the parameters that are provided. In some cases, the parameter only specifies the number of instantiations of a requirement, such as requirements 6 and 7 in Table 4.1. In such a requirement, the corresponding event changes but the condition stays the same. In other cases, both the event and the condition change based on the parameters, such as requirements 4 and 5 in Table 4.1. The way the requirement depends on the parameters must either be defined in the template library, or be assignable in the configuration tool.

As defined in the configuration module, pump cellar 2 has a pump direction to either pump cellar 1 or pump cellar 3. Based on this configuration module, additional requirements are needed in the second pump cellar to specify under which conditions the pump direction must be switched. These requirements are given here both in textual form and in event-condition form.

1. The pump direction may only be set to pump cellar 3 when pump cellar 1 is full, i.e. `S5` of pump cellar 1 is on.
2. The pump direction may only be set to pump cellar 1 when pump cellar 1 is sufficiently empty, i.e. `S3` of pump cellar 1 is off.

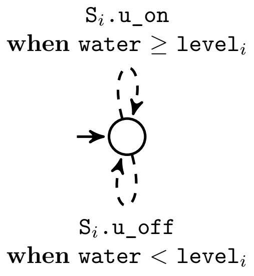
$$\begin{aligned}&\texttt{PumpCellar2.PumpDirection.c\_PC3}\ \textbf{needs}\ \texttt{PumpCellar1.S5.on}\\&\texttt{PumpCellar2.PumpDirection.c\_PC1}\ \textbf{needs}\ \texttt{PumpCellar1.S3.off}\end{aligned} \tag{4.2}$$

The hybrid plant model

Besides the plant model and the requirements model, the hybrid plant model and the visualization file are automatically generated in parameter-based modeling. The differences in the hybrid plant model in the parameter-based modeling method compared to the component-based modeling method are described here.

First, a relation between each sensor and the corresponding water level must be modeled, as shown on the left-hand side of Figure 4.8. Second, the differential equation of the water-level variable is made parameter-dependent in relation to the number of pumps in the pump cellar. As seen on the right-hand side of Figure 4.8, the derivative is increased by 10 when the rain is on, and decreased by 5 for each pump ($P_i,\ i \in I$) that is on.

Sensors



$\texttt{S}_i\texttt{.u\_on}$
when $\texttt{water} \geq \texttt{level}_i$
$\texttt{S}_i\texttt{.u\_off}$
when $\texttt{water} < \texttt{level}_i$

Water-level variable `water`

$$\texttt{water}' = \begin{cases} 10 + \sum_{i=1}^{I}(-5\ \textbf{\texttt{if}}\ \texttt{P}_i\texttt{.On}) & \textbf{\texttt{if}}\ \texttt{Rain.On} \\ \sum_{i=1}^{I}(-5\ \textbf{\texttt{if}}\ \texttt{P}_i\texttt{.On}) & \textbf{\texttt{if}}\ \texttt{Rain.Off} \end{cases}$$

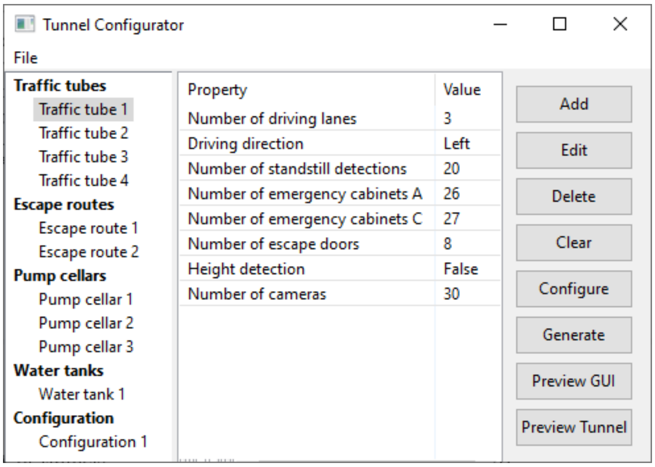
Figure 4.8: Parameter-based hybrid plant model of a pump cellar.

The final file that is generated from the parameters is the SVG visualization and the corresponding declarations. The SVG can be created using the same parameter-based modeling tool as the discrete-event plant model and the hybrid plant model. They can be created using the same parameter list as is used for the plant model with some additional parameters to correctly visualize the system. An example of such a parameter is the driving direction of a traffic tube. The plant behavior of a traffic tube does not depend on its driving direction, but it is relevant for visualization.

4.3 Configuration tool

The parameter-based modeling approach is implemented as a prototype tool to perform the configuration step shown on the left-hand side in Figure 4.3. The tool is developed in the Python programming language using Microsoft Visual Studio, and can be used as a front-end to the CIF toolset that can be used for the subsequent synthesis and simulation steps.

The user interface of the configuration tool is shown in Figure 4.9. It consists of three main panels. On the left-hand side, a list of the modules in the tunnel is shown. In the middle panel, the parameters of the module that is currently selected are shown, e.g., the parameters of a traffic tube as in Figure 4.9. Finally, on the right-hand side, a set of buttons is included for the user to create the tunnel configuration.



Property	Value
Number of driving lanes	3
Driving direction	Left
Number of standstill detections	20
Number of emergency cabinets A	26
Number of emergency cabinets C	27
Number of escape doors	8
Height detection	False
Number of cameras	30

Figure 4.9: Overview of the configuration tool main screen.

The user can create the desired tunnel configuration by using the `Add`, `Delete`, and `Clear` buttons. Through the `Edit` button, the parameters of the currently selected module can be changed. Furthermore, the `Configure` button allows the user to change global settings, such as the output file names and the language of the interface. Once the desired tunnel configuration is constructed, the user can press the `Generate` button to generate the files as shown in Figure 4.3. These files can then be used in the CIF toolset to synthesize the supervisor and perform simulations.

Finally, the configuration tool contains a preview function, using the `Preview` buttons, providing the visualization of the tunnel, as shown in Figure 4.10, and the visualization of the operator interface. Previewing allows the user to quickly validate whether the desired configuration is constructed.

4.3.1 Running example: Pump-cellar system

For the running example, the pump-cellar system is modeled using the parameter-based configuration tool. In this process, first the three modules for the three

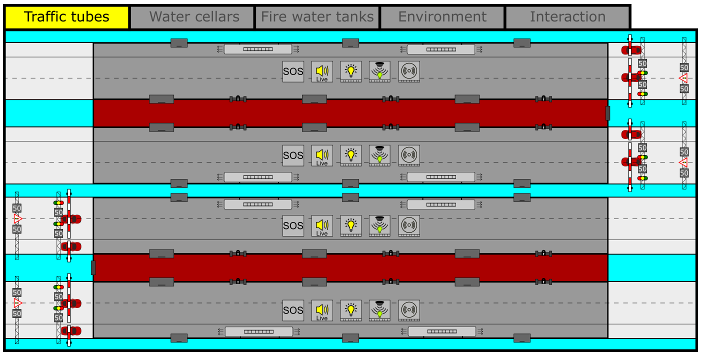
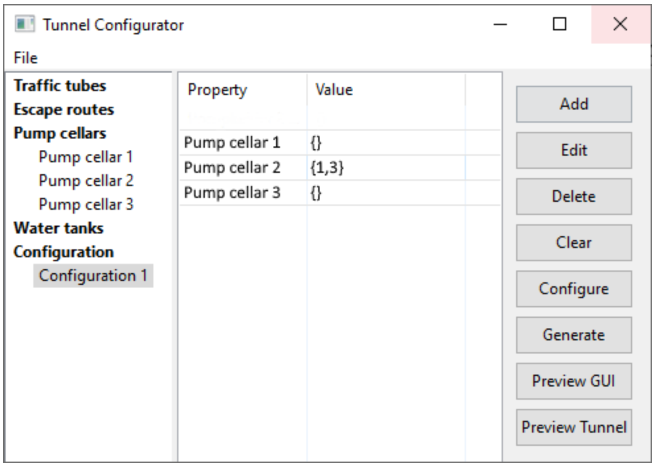


Figure 4.10: Preview functionality in the configuration tool.

pump cellars are added, as can be seen on the left-hand side in Figure 4.11. For each of these modules, the parameters for the number of sensors and the number of pumps in that pump cellar are entered. Furthermore, a configuration module is added to set the dependencies between the pump cellars. The middle panel in Figure 4.11 shows the settings of the configuration module, which indicate that pump cellar 2 is connected to pump cellars 1 and 3.



Property	Value
Pump cellar 1	{}
Pump cellar 2	{1,3}
Pump cellar 3	{}

Figure 4.11: Configuration tool for the pump-cellar system.

Once the complete system has been entered in the configuration tool, the `Generate` button can be used to create all files that are needed for supervisor synthesis and simulation. As explained in Section 4.2, these files include the discrete-event plant model, the requirements model, the hybrid plant model, and the visualization.

4.4 Case study on a family of tunnels

To illustrate the effectiveness of the parameter-based modeling tool, a case study has been performed on a family of tunnels in the Netherlands. This family consists of 22 road tunnels that are managed by RWS. The tunnels vary in configuration, as shown in Table 4.2, since they have different numbers of modules, e.g., traffic tubes and escape routes. Furthermore, the dimensions of the tunnels are different, resulting in varying parameters such as the number of escape doors, the number of emergency cabinets, and the existence of a height detection. The required data of the tunnels was partially retrieved from Juerd (2020), and extended with additional data from RWS.

In this case study, each tunnel was modeled from scratch using the configuration tool, while the same template library is used for each case. As can be seen in Table 4.2, almost all tunnels consist of the same modules, though in varying numbers. Because of the low variation in tunnel configurations, the modeling process using the parameter-based modeling tool shows to be very efficient and intuitive. As long as the required data is available, a tunnel can be modeled and a supervisor can be synthesized in a matter of minutes.

4.5 Concluding remarks

In this chapter, the component-based modeling method is described, and its applicability for the modeling of road tunnel systems is shown. A running example is included for a pump-cellar system that consists of three pump cellars with dependencies between them. Furthermore, an extension of the component-based modeling method, being a parameter-based modeling platform, is proposed for road tunnel supervisory controllers. The parameter-based modeling process is described, where it is shown how a road tunnel system can efficiently be defined and modeled using a template library and a list of parameters. A prototype of this platform has been implemented as a configuration tool, which automatically generates the models and visualizations that are required for supervisor synthesis and simulation. Simulations are performed to validate the correctness of the configuration tool and of the models in the template library. Furthermore, this chapter presents a case study in which the configuration tool is used to model a family of 22 road tunnels in the Netherlands, from which it can be concluded that the parameter-based modeling approach is suitable for such a family of systems.

There are several options for future work to extend the functionality of the parameter-based modeling platform. First, it can be investigated if the files that are required for supervisory controller implementation, such as the hardware configuration and the input and output mapping, can be generated automatically as well. Second, currently all requirements that connect the modules in the tunnel are standardized, though the platform can be extended so that different

Table 4.2: Overview of the modules in the family of tunnels used in the case study.

Tunnel name	Length (m)	Traffic tubes	Driving lanes	Escape routes	Pump cellars	Water tanks	Components	Requirements
Benelux tunnel	800	5	2	2	6	1	1031	701
Botlek tunnel	540	2	3	1	3	1	381	291
Coen tunnel	760	5	2	2	6	1	981	681
Drecht tunnel	570	4	2	2	3	1	671	499
Eerste Heinenoord tunnel	610	2	3	1	3	1	401	299
Gaasperdammer tunnel	3030	5	2	2	6	1	2131	1150
Kethel tunnel	2000	2	4	1	3	1	689	435
Koning Willem-Alexander tunnel	2500	4	2	2	2	1	1230	720
Leidsche Rijn tunnel	1650	5	3/2	2	6	1	1439	885
Noord tunnel	540	2	3	1	3	1	381	291
Roer tunnel	2450	2	2	1	2	1	732	408
Salland Twente tunnel	490	2	1	1	3	1	345	235
Schiphol tunnel	650	4	4/2	2	6	1	788	592
Sijtwende tunnel: Park	300	2	2	1	1	0	278	225
Sijtwende tunnel: Spoor	400	2	2	0	1	1	312	239
Sijtwende tunnel: Vliet	1100	2	2	1	1	1	452	295
Thomassen tunnel	1140	2	3	1	3	1	501	339
Tunnel Swalmen	400	2	2	1	1	1	312	239
Velser tunnel	770	2	2	1	3	1	413	283
Vlake tunnel	330	2	2	1	3	1	333	251
Wijker tunnel	690	2	3	1	3	1	401	299
Zeeburger tunnel	550	2	3	1	3	1	381	291

requirement sets can be selected. Third, as supervisory controllers for road tunnels are often implemented on a set of PLCs, the distribution of the supervisory controller can possibly be included in the functionality of the parameter-based modeling platform as well.

Chapter 5

Model reduction for supervisor synthesis

After creating the plant model and the requirements model, as described in Chapter 4, supervisor synthesis can be used to automatically generate the supervisor. A benefit of using synthesis is that the synthesized supervisor is guaranteed to adhere to the defined requirements, and has several other beneficial properties such as nonblockingness and controllability. Furthermore, synthesis can be performed early in the design process, so validation, e.g. through simulations, can be performed early as well.

Synthesizing a supervisor can be computationally intensive and time-consuming, depending on the number of component models and requirement models. In previous work presented in Goorden et al. (2021), it is shown that component models and requirement models satisfying certain properties can be omitted during synthesis or solved in a separate synthesis procedure, or that synthesis can be skipped entirely. Goorden et al. (2021) uses dependency graphs to analyze the relations between the component models based on the requirement models. In these dependency graphs, vertices represent component models and edges represent requirement models. The vertices that do not exhibit an infinite path in this graph, do not give rise to blocking behavior, and can therefore be omitted during the main synthesis procedure, and solved in a separate synthesis procedure.

The contribution of this chapter is showing how the concept of symmetry can be used to reduce the component models and the requirement models for the purpose of supervisor synthesis. It is an extension on the work shown in Goorden

This chapter is based on: Moormann, L., van de Mortel-Fronczak, J.M., Fokkink, W.J., and Rooda, J.E. Exploiting symmetry in dependency graphs for model reduction in supervisor synthesis. In *2020 IEEE 16th International Conference on Automation Science and Engineering (CASE)*, pages 659–666. IEEE, 2020.

et al. (2021). In this chapter, symmetry is formally defined for component models and requirement models, and this definition is used in the proposed model reduction method. The symmetry in the dependency graph can be observed at two levels, namely symmetry of component models within a single vertex, and symmetry between vertices in the graph. In the first case, certain component models and requirement models in that vertex can be reduced before synthesis. In the second case, the component models and requirement models of entire vertices in the dependency graph can be removed or separated before synthesis.

Symmetry has been used in previous works for the purpose of model reduction. A symmetry-based method to reduce the verification effort in model checking is explained in Norris et al. (1996). A language-based symmetry is used in Eyzell and Cury (2001) to obtain abstracted automata that represent the same behavior as the initial automata, after which synthesis is performed for the abstracted automata. In Rohloff and Lafortune (2004), symmetry in finite state automata is determined for the purpose of reducing the cost in computation time for testing if symmetric systems satisfy propositions in μ-calculus. In Miller et al. (2006), symmetry in Kripke structures is used to reduce the state space for temporal model checking. Finally, the symmetry in state tree structures for discrete-event systems is used in Jiao et al. (2017) to obtain an abstracted supervisor by means of relabeling states and events.

Our concept of symmetry in discrete-event systems for the purpose of model reduction differs from these previous works in various ways. First, symmetry is defined for (extended) finite state automata as well as for requirement models. Second, no abstraction is needed when applying the model reduction steps. Third, no global symmetry is needed where one half of the system is symmetrical with the other half of the system, as symmetry can exist on different levels in the model.

This chapter is structured as follows. In Section 5.1, symmetry in discrete-event systems is formally defined and the concept of dependency graphs is introduced. Section 5.2 describes the proposed model reduction steps, and in Section 5.3 the model restoration process is described. In Section 5.4, the proposed model reduction method is showcased using a set of case studies. Finally, concluding remarks are made in Section 5.5.

5.1 Symmetry in discrete-event systems

In this section, symmetry in discrete-event systems is described. First, symmetry is formally defined for FAs, EFAs, requirements, and control problems. This definition of symmetry is called isomorphism. Second, dependency graphs are introduced as a means to visualize control problems and determine symmetry in such a control problem.

5.1.1 Isomorphism

There are multiple concepts of equivalence of finite-state automata. One of these is the concept of isomorphism, described in Glushkov (1961). Here, isomorphism is formally defined for FAs, and is subsequently extended for EFAs and for sets of automata. Furthermore, isomorphism is defined for requirement models, and combinations of automata and requirements, called control problems.

Finite state automata

Two FAs are said to be isomorphic if there exists a bijective mapping from the first to the second automaton, where the second automaton preserves the transition function, the output, and the initial state. A bijective mapping between two sets pairs each element of one set with exactly one element of the other set, and pairs each element of the other set with exactly one element of the first set.

In Glushkov (1961), isomorphism is formally defined for Mealy machines. A machine is an automaton of an (abstract) process. The main difference between Mealy machines, explained in Mealy (1955), and FAs, as defined in Section 2.2, is how the output is determined. In a Mealy machine, the output is determined based on the input, i.e. the executed event, and the current state using a certain output function. In an FA, the only notion of an output is whether a state is a marked state. In this chapter, the notion of isomorphism for Mealy machines is transposed to isomorphism for FAs. This involves the omission of the mapping between the output alphabets of the two Mealy automata. Furthermore, an additional condition is added for isomorphism of FAs, as is given in Equation (5.5), that defines that marked states of the first automaton must be mapped to marked states of the second automaton. The definition of isomorphism for two FAs is described below.

An isomorphism h between FAs $P_1 = (Q_1 , E_1 , f_1 , q_{0,1} , Q_{m,1})$ and $P_2 = (Q_2 , E_2 , f_2 , q_{0,2} , Q_{m,2})$, with controllable event sets $E_{c,1}$ and $E_{c,2}$, consists of bijective mappings

$$h_1 : E_1 \to E_2, \quad h_2 : Q_1 \to Q_2 \tag{5.1}$$

such that, for every $q \in Q_1$, $e \in E_1$:

$$h_2(f_1(q, e)) = f_2(h_2(q), h_1(e)) \tag{5.2}$$

$$h_2(q_{0,1}) = q_{0,2} \tag{5.3}$$

$$e \in E_{c,1} \Leftrightarrow h_1(e) \in E_{c,2} \tag{5.4}$$

$$q \in Q_{m,1} \Leftrightarrow h_2(q) \in Q_{m,2} \tag{5.5}$$

First, in Equation (5.2), the preservation of the transition function is defined. Here, it is checked if for each transition in P_1 there exists a transition in P_2 while applying mappings h_1 and h_2 to the events and states, respectively. Second, the preservation of the initial state is defined in Equation (5.3). Third, Equation (5.4) provides a check that h_1 maps controllable events of P_1 to controllable events of P_2. Finally, in Equation (5.5), it is checked whether h_2 maps marked states of P_1 to marked states of P_2.

In the case that P_1 and P_2 share events, meaning $E_1 \cap E_2 \neq \emptyset$, an additional condition must be adhered to. This condition is shown in Equation (5.6) that must be met for every $e \in E_1 \cap E_2$.

$$h_1(e) = e \tag{5.6}$$

This additional condition defines that shared events must be mapped onto themselves.

When two models are isomorphic, say two component models P_1 and P_2, then this is denoted by $P_1 \simeq P_2$.

Extended finite state automata

EFAs are introduced in Subsection 2.2.2 as FAs where variables are included. Isomorphism between EFAs therefore also includes preservation of these variables. The definition of isomorphism for EFAs is given here as an extension of the isomorphism definition of FAs given in the previous section. All conditions given in the previous section are maintained, and the additional conditions for isomorphism for EFAs are given here.

An isomorphism h between EFAs $P_1 = (Q_1, V_1, E_1, \rightarrow_1, q_{0,1}, v_{0,1}, Q_{m,1})$ and $P_2 = (Q_2, V_2, E_2, \rightarrow_2, q_{0,2}, v_{0,2}, Q_{m,2})$, consists of bijective mappings h_1, h_2, and

$$h_3 : V_1 \rightarrow V_2 \tag{5.7}$$

such that, for every $v \in V_1$:

$$h_3(v_{0,1}) \quad = \quad v_{0,2} \tag{5.8}$$

Furthermore, any guard or update in the EFA must be preserved as well. First guards are considered. A guard consists of a condition c that is applied to a transition t labeled by an event e. For two EFAs to be isomorphic, for every condition $c_1 \in C_1$ applied to transition $t_1 \in \rightarrow_1$ with start location $q_{s,1}$ and end location $q_{e,1}$ labeled by event $e_1 \in E_1$ the following statements must hold:

1. There must be a transition t_2 in $\rightarrow_2$ labeled by event $h_1(e_1)$ with start location $h_2(q_{s,1})$ and end location $h_2(q_{e,1})$.

2. Transition t_2 must have a condition $c_2 \in C_2$, and condition c_1 must be isomorphic to condition c_2. There are four possible cases:

   (a) In the case that c_1 is of the form **when** q_1, with q_1 a different location in the control problem, this means that c_2 must also be of the form

when q_2, with q_2 a different location in the control problem, where either q_1 and q_2 are the same location:

$$q_1 = q_2, \tag{5.9}$$

or location q_1 is isomorphic to location q_2:

$$q_1 = h_2(q_2). \tag{5.10}$$

(b) In the case that c_1 is of the form **when** $v_1 = x_1$, with x_1 a value of the same type as v_1 (e.g. **when** $v_1 = 5$), this means that c_2 must also be of the form **when** $v_2 = x_2$ where

$$v_1 = v_2, \text{ or} \tag{5.11}$$

$$v_1 = h_3(v_2) \tag{5.12}$$

and

$$x_1 = x_2. \tag{5.13}$$

(c) In the case that c_1 is of the form **when** $v_1 = y_1$, with y_1 a different variable in the control problem, this means that c_2 must also be of the form **when** $v_2 = y_2$ where

$$y_1 = y_2, \text{ or} \tag{5.14}$$

$$y_1 = h_3(y_2). \tag{5.15}$$

(d) In the case that c_1 is a combination of multiple conditions (e.g. **when** q_1 `and` $v_1 = 5$), for each of these conditions the corresponding requisites must hold. Furthermore, all logical operators (e.g. `and`, `or`, `not`) between the conditions must be the same.

Second, preservation of updates is considered. An update u is applied to a transition t and consists of a variable v and a value x to which the variable is updated. The value x can either be another variable in the control problem, or a value of the same type as v. For two EFAs to be isomorphic, for every update $u_1 \in U_1$ applied to transition $t_1 \in \rightarrow_1$ with start location $q_{s,1}$ and end location $q_{e,1}$ labeled by event $e_1 \in E_1$ the following requisites must hold:

1. There must be a transition t_2 in $\rightarrow_2$ labeled by event $h_1(e_1)$ with start location $h_2(q_{s,1})$ and end location $h_2(q_{e,1})$.

2. Transition t_2 must have an update u_2.

3. The variables v_1 and v_2 that are updated in u_1 and u_2, respectively, must be isomorphic, meaning that

$$v_1 = v_2, \text{ or} \tag{5.16}$$
$$v_1 = h_3(v_2). \tag{5.17}$$

4. The value x_1 in update u_1 must be isomorphic to the value x_2 in update u_2, meaning that

$$x_1 = x_2, \text{ or} \tag{5.18}$$
$$x_1 = h_3(x_2). \tag{5.19}$$

Sets of automata

In many practical cases, specifically when component-based or parameter-based modeling is used, sets of models are created that each represent a specific component in the system. In these cases, symmetry can exist between individual components, but also between sets of components. To identify such symmetry between sets of components, isomorphism is defined for sets of automata.

The definition of isomorphism between automata P_1 and P_2 can be extended to isomorphism between sets of automata as follows.

1. Let $\mathcal{P}^1 = \{P_{1,1}, P_{1,2}, .., P_{1,n}\}$ and $\mathcal{P}^2 = \{P_{2,1}, P_{2,2}, .., P_{2,n}\}$ be two sets of (extended) finite automata.
2. Let $\Sigma_{\mathcal{P}^1} = \bigcup_{i=1}^{n} E_{1,i}$ and $\Sigma_{\mathcal{P}^2} = \bigcup_{i=1}^{n} E_{2,i}$ be the event sets of $\mathcal{P}^1$ and $\mathcal{P}^2$.
3. $\mathcal{P}^1$ and $\mathcal{P}^2$ are isomorphic if
   (a) There exists a one-to-one correspondence between $\mathcal{P}^1$ and $\mathcal{P}^2$, such that for each automaton in $\mathcal{P}^1$ there exists an isomorphism $h = (h_1, h_2, h_3)$, as defined in Equations (5.1) and (5.7), with an automaton in $\mathcal{P}^2$. Let $\mathcal{H}$ denote the set of these isomorphisms.
   (b) For every $(h_1, h_2, h_3) \in \mathcal{H}, e \in \Sigma_{\mathcal{P}^1} \cap \Sigma_{\mathcal{P}^2} : h_1(e) = e$.

Requirements

Isomorphism can also be defined for requirement models. In the case of automaton requirements, the same formulations can be used as above.

For state-based requirements, a similar reasoning can be applied. Isomorphism between requirements $R_1 : e_1$ **`needs`** q_1 and $R_2 : e_2$ **`needs`** q_2 can be determined using the following steps:

1. Determine P_1, being the automaton containing location q_1, and P_2, being the automaton containing location q_2.
2. Determine $\mathcal{P}^1$, being the set of automata containing a transition labeled by event e_1, and $\mathcal{P}^2$, being the set of automata containing a transition labeled by event e_2.
3. Determine if $\mathcal{P}^1$ and $\mathcal{P}^2$ are isomorphic, with $\mathcal{H}$ being the set of isomorphisms between $\mathcal{P}^1$ and $\mathcal{P}^2$.
4. If $P_1 \in \mathcal{P}^1 \cup \mathcal{P}^2$ or $P_2 \in \mathcal{P}^1 \cup \mathcal{P}^2$, the additional requirement is that in the one-to-one correspondence between $\mathcal{P}^1$ and $\mathcal{P}^2$, P_1 must be mapped to P_2.
5. Requirements R_1 and R_2 are isomorphic, if for every $(h_1, h_2, h_3) \in \mathcal{H}$ it holds that $h_1(e_1) = e_2$ and $h_2(q_1) = q_2$.

The statements above are based on requirements where q_1 and q_2 consist of one location. They can, however, easily be extended for requirements that refer to multiple locations, such as e **`needs`** q_1 **`and`** q_2. The additional condition is that the logical operators (e.g., `and`, `or`, `not`) are the same for both requirements.

Isomorphism of requirements of the form q_1 **`disables`** e_1 can be defined following the same lines.

Control problems

Let us define a control problem as a two-tuple, $\mathcal{S} = (\mathcal{P}, \mathcal{R})$, where $\mathcal{P} = \{P_1, P_2, .., P_m\}$ is a set of automata, and $\mathcal{R} = \{R_1, R_2, .., R_n\}$ is a set of requirements. Furthermore, let us define the event set, also called the alphabet, of control problem $\mathcal{S}$ as

$$\Sigma_{\mathcal{S}} = \bigcup_{i=1}^{m} E_i \tag{5.20}$$

and the location set of control problem $\mathcal{S}$ as

$$\Phi_{\mathcal{S}} = \bigcup_{i=1}^{m} Q_i. \tag{5.21}$$

Two control problems $\mathcal{S}^1$ and $\mathcal{S}^2$ are isomorphic if

1. the set of automata $\mathcal{P}^1$ is isomorphic to the set of automata $\mathcal{P}^2$ with isomorphism set $\mathcal{H}$, and
2. there exists a one-to-one correspondence between $\mathcal{R}^1$ and $\mathcal{R}^2$, whereby every requirement in $\mathcal{R}^1$ corresponds to an isomorphic requirement in $\mathcal{R}^2$ using $\mathcal{H}$.

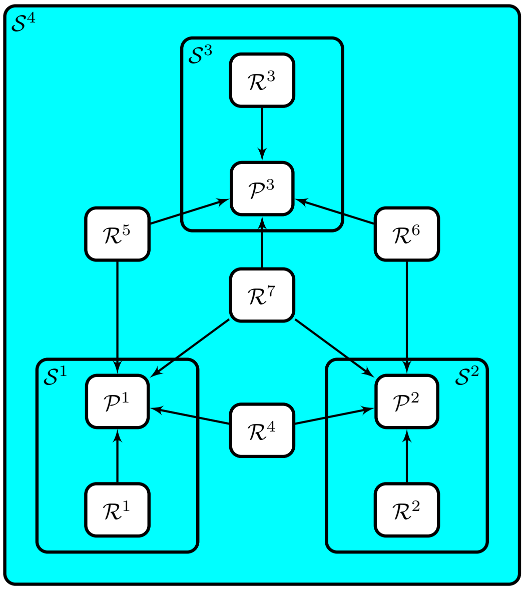


Figure 5.1: Schematic overview of two isomorphic control problems $\mathcal{S}^1$ and $\mathcal{S}^2$ and a third control problem $\mathcal{S}^3$.

Figure 5.1 shows a schematic overview of the general case, where two isomorphic control problems $\mathcal{S}^1$ and $\mathcal{S}^2$ are identified that are connected by requirement set $\mathcal{R}^4$. A third control problem $\mathcal{S}^3$ is connected to $\mathcal{S}^1$ and $\mathcal{S}^2$ via requirement sets $\mathcal{R}^5$, $\mathcal{R}^6$, and $\mathcal{R}^7$. The arrows indicate to which plant sets a certain requirement set refers. Five possible isomorphisms between automata and requirements in these control problems are identified:

1. The automata in $\mathcal{P}^1$ are isomorphic to the automata in $\mathcal{P}^2$.

2. The requirements in $\mathcal{R}^1$ are isomorphic to the requirements in $\mathcal{R}^2$.

3. If $\mathcal{R}^4$ contains the requirements

$$e_1 \ \texttt{needs} \ q_2, \quad e_2 \ \texttt{needs} \ q_1 \tag{5.22}$$

   with $e_1 \in \Sigma_{\mathcal{S}^1}, e_2 \in \Sigma_{\mathcal{S}^2}, q_1 \in \Phi_{\mathcal{S}^1}, q_2 \in \Phi_{\mathcal{S}^2}$, the automata containing e_1 and e_2 are isomorphic, with $h_1(e_1) = e_2$, and the automata containing q_1 and q_2 are isomorphic, with $h_2(q_1) = q_2$, then these requirements are isomorphic.

4. If $\mathcal{R}^5$ and $\mathcal{R}^6$ contain the requirements

$$e_1 \ \texttt{needs} \ q_3, \quad e_2 \ \texttt{needs} \ q_3 \tag{5.23}$$

respectively, with $e_1 \in \Sigma_{\mathcal{S}^1}, e_2 \in \Sigma_{\mathcal{S}^2}, q_3 \in \Phi_{\mathcal{S}^3}$, and the automata containing e_1 and e_2 are isomorphic, with $h_1(e_1) = e_2$, then these requirements are isomorphic.

5. If $\mathcal{R}^7$ contains the requirements

$$e_1 \ \texttt{\textbf{needs}}\ q_2 \ \texttt{and}\ q_3, \quad e_2 \ \texttt{\textbf{needs}}\ q_1 \ \texttt{and}\ q_3 \tag{5.24}$$

with $e_1 \in \Sigma_{\mathcal{S}^1}$, $e_2 \in \Sigma_{\mathcal{S}^2}$, $q_1 \in \Phi_{\mathcal{S}^1}$, $q_2 \in \Phi_{\mathcal{S}^2}$, $q_3 \in \Phi_{\mathcal{S}^3}$, and the automata containing e_1 and e_2 are isomorphic, and the automata containing q_1 and q_2 are isomorphic, then these requirements are isomorphic.

The statements above are based on requirements where q_1, q_2 and q_3 consist of one location. They can, however, easily be extended for requirements that refer to multiple locations. The additional condition is that the logical operators (e.g., `and`, `or`, `not`) are the same for both requirements. Furthermore, isomorphism of requirements of the form q_1 **`disables`** e_1 can be defined following the same lines.

Isomorphic systems are equivalent in behavior, since their uncontrolled behavior is the same (isomorphism between the automata) and their desired controlled behavior is the same (isomorphism between the requirements). If, according to the systems engineer, these systems are required to have the same behavior, model reduction steps can be applied. Isomorphic systems that, according to the systems engineer, should have different behavior may indicate modeling errors in the plant or requirements.

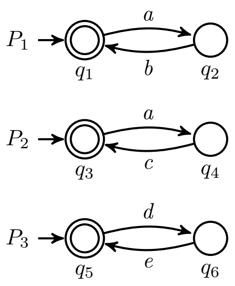
5.1.2 Dependency graphs

Determining if a system contains isomorphic subsystems can both be an intuitive and a difficult task, depending on the state size and complexity of the component models and requirement models in these subsystems. To visualize the structure of a system, based on the automata and the requirements in that system, dependency graphs are used. These graphs allow a design engineer to more easily analyze the model structure, and identify symmetry in (parts of) the system. Below, a short introduction is given to directed graphs, based on Diestel (2017), since dependency graphs are a subset of directed graphs.

A directed graph is a two-tuple $G = (V, E)$ with two maps $init : E \to V$ and $ter : E \to V$. V is the set of vertices (or nodes) and E is the set of edges between these vertices. $init(e)$ assigns to every edge e an initial vertex, while $ter(e)$ assigns to every edge e a terminal vertex. An edge e is said to be directed from vertex $init(e)$ to vertex $ter(e)$. If $init(e) = ter(e)$, the edge e is called a loop. A directed graph is called cyclic if it contains cycles, otherwise it is called acyclic. Note that loops do not count as cycles in directed graphs.

Dependency graphs are directed graphs where the vertices in the graph represent component models and edges between the vertices represent the requirement models. The introduction to dependency graphs given here is based on Goorden et al. (2021). A dependency graph is created from the MRPSR of the modeled system. Each vertex in the dependency graph represents a component model or a set of synchronous component models and the set of requirement models that only refer to component models in that vertex. Synchronous component models are component models that share at least one event. Each vertex can therefore be defined as a two-tuple $V = (\mathcal{P}, \mathcal{R})$, where $\mathcal{P}$ is the set of synchronous component models and $\mathcal{R}$ the set of requirements that only refer to events and states in $\mathcal{P}$. There is an edge from a vertex V_1 to a vertex V_2 if a requirement expresses a condition on a controllable event that occurs in plant V_1, while the requirement requires plant V_2 to be in a certain state. An example is shown in Figure 5.2. It shows a control problem containing three component models (P_1, P_2, and P_3) and two requirement models (R_1 and R_2). Component models P_1 and P_2 are synchronous as they share event a. The dependency graph for this control problem is shown on the right-hand side of Figure 5.2. As P_1 and P_2 are synchronous, they are part of the same vertex, while P_3 is placed in a separate vertex. Requirement R_1 expresses a condition on an event in P_1, and only refers to a state in P_2. Requirement R_1 is therefore a represented by a loop at the vertex of automata P_1 and P_2. Requirement R_2, however, expresses a condition on an event in P_2 and refers to a state in P_3. This requirement is therefore represented as an edge between the two vertices that is labeled by R_2. Note that if two vertices are connected by multiple requirement models, only one edge is created that is labeled by multiple requirement models.

To simplify the notation, a vertex that only contains the component model P_1 is referred to as P_1, and an edge is called R_1 when that edge only contains requirement model R_1.



R_1 : b **needs** $P_2.q_3$

R_2 : c **needs** $P_3.q_5$

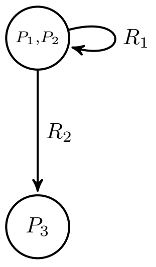


Figure 5.2: Example of a set of component models (left), a set of requirement models (middle), and the corresponding dependency graph (right).

Dependency graphs are visualized in a layered form, where each layers consists of a set of vertices. Furthermore, the layers are automatically chosen in such a way that the directed edges between the vertices are in the downward direction as often as possible. In the case of a cycle in the dependency graph, the vertices that are part of this cycle are placed in the same layer, if possible. The general idea behind this layered structure is that each layer can be considered separately when determining nonblockingness, i.e. bringing all components to a marked state. The components in the lowest layer do not depend on any other layer, so they can be brought to a marked state by only considering this layer. Next, the layer above is considered. For this layer it is determined if all component models can be brought to a marked state without forcing the components in the lower layer out of their marked state. Subsequently, all other layers are considered separately in a similar way. In the case of a cyclic dependency between layers, this approach does not work and traditional synthesis techniques are needed to determine nonblockingness.

5.2 Model reduction

The method described in Goorden et al. (2021) uses dependency graphs to determine if synthesis is required to obtain a supervisory controller for a certain control problem. To this end, concepts of strongly connected components and extended strongly connected components are introduced. A strongly connected component is a set of vertices in the graph that are part of a cycle. Extended strongly connected components consist of the same vertices as the strongly connected components, as well as all vertices from which there exists a path to the strongly connected components. An example of a dependency graph is shown in Figure 5.3. In this example, vertices P_2 and P_3 are part of a cycle, and therefore form a strongly connected component. This strongly connected component is indicated in red in Figure 5.3. As there exists a path from vertex P_1 to the strongly connect component, P_1 is part of the extended strongly connected component. In the figure, the extended strongly connected component is indicated by the combination of the red and orange vertices. Vertices P_4 and P_5 are not part of the extended strongly connected component, as there exists no path from these vertices to the strongly connected component.

It is shown that synthesis is only needed for the entire system when the dependency graph contains extended strongly connected components. When this is not the case, the synthesis procedure can be skipped as the system is inherently controllable, nonblocking, and maximally permissive. When the dependency graph does contain extended strongly connected components, synthesis is needed to determine these properties and whether additional restrictions on the plant are required. It that case, the dependency graph can be used to determine whether the model is eligible for model reduction. In this process, part of the model is removed before the synthesis procedure, while maintaining all synthesis guarantees. Reducing the model reduces the computing effort of the synthesis procedure. Moreover, model reduction can make synthesis procedures solvable that were initially unsolvable. In the following sections, several model reduction steps are discussed and examples are provided.

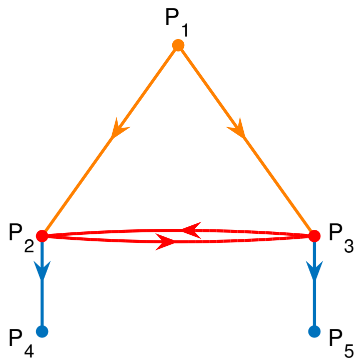


Figure 5.3: Example of a dependency graph with strongly connected component (red), extended strongly connected component (red+orange) and independent vertices (blue).

5.2.1 Independent vertices

The first model reduction step is based on theory described in Goorden et al. (2021), where it is shown that vertices in the dependency graph that have no path to a strongly connected component can be separated from the main synthesis problem and solved as a separate synthesis problem. The dependency graph shown in Figure 5.3 contains such vertices, as indicated in blue. These vertices are called independent vertices, as they do not depend on the extended strongly connected component. Following Goorden et al. (2021), they can be separated from the main synthesis problem.

In practice, removing independent vertices is a procedure of splitting off vertices that have no outgoing edges. This is an iterative procedure, as splitting of vertices might remove the outgoing edges of other vertices, thus making those vertices eligible for removal.

5.2.2 Marked requirements

The second model reduction step is based on requirements that refer to marked states in automata. It specifies that when a requirement refers solely to a marked state or set of marked states, it can be removed before synthesis, and added back after synthesis. Two extra conditions must, however, be met to perform this step. First, the requirement in question may not be part of a cycle in the dependency graph. Second, each marked state that is referred to in the requirement must be the only marked state in that automaton, since it is not guaranteed that a marked state is always reached when there are multiple marked states in an automaton. The reasoning behind the reduction step is that when synthesis is performed for a certain control problem, the nonblocking property guarantees that there always exists a path to reach the marked state. A requirement that solely refers to this marked state will therefore not add any extra restrictions, and can thus be omitted during synthesis. In the sequel, such a requirement is called a *marked requirement.*

Take the example of a control problem shown in Figure 5.4. Requirement R_1 in this example refers to state q_3 of automaton P_2, which is the only marked state in P_2. Performing synthesis for vertex P_2 (and any other vertices connected to this vertex) results in a controller that is nonblocking, and that is thus always able to reach state $P_2.q_3$. Since requirement R_1 only refers to the marked state $P_2.q_3$, and since this is the only marked state in P_2, requirement R_1 does not add any extra restrictions compared to synthesis for vertex P_1. Requirement R_1 can therefore be removed from the control problem, which separates vertices P_1 and P_2. This means that a separate synthesis procedure can be applied to each vertex, which is computationally less intensive than the combined problem.

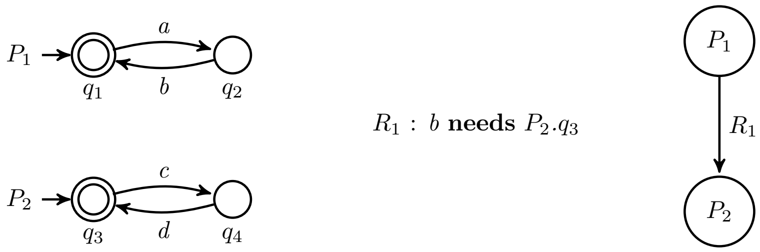


Figure 5.4: Example of a control problem that contains a marked requirement.

There exist two types of marked requirements, as shown in Equations (5.25) and (5.26). The first type of marked requirements are requirements that refer to the marked state of an automaton with an `or` operator between this marked state and the rest of the condition. A requirement of the form

$$e \texttt{ needs } P_1.q_1 \texttt{ or } \ldots \tag{5.25}$$

is a marked requirement when q_1 is the only marked state of automaton P_1, i.e., $Q_{m,1} = \{q_1\}$.

The non-blocking property of synthesis guarantees that there always exists a path to $P_1.q_1$ when q_1 is the only marked state of P_1. Therefore, there always exists a path to a state where this requirement evaluates to `true`. It can thus be removed before performing synthesis, as it will never result in extra restrictions.

The second type of marked requirements are requirements that refer to a set of marked states, with `and` operators between them. A requirement of the form

$$e \texttt{ needs } P_1.q_1 \texttt{ and } P_2.q_2 \texttt{ and } \ldots \texttt{ and } P_k.q_k \tag{5.26}$$

is a marked requirement when for all $i \in [1,k]$, q_i is the only marked state of automaton P_i.

Following a similar reasoning as the first type of marked requirements, there always exists a path to the set of marked states as long as each of those marked states is the only marked state in its automaton. This type of requirements can therefore also be removed before performing synthesis.

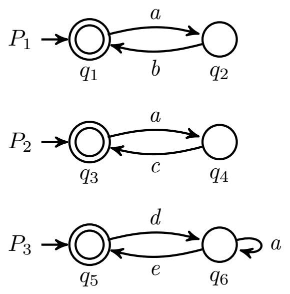
Note that combinations of these types of marked requirements are possible as well.

5.2.3 Internal symmetry

The remaining model reduction steps are based on various forms of symmetry that are identified in the dependency graph of the system. The general idea behind these reduction steps is that when two control problems are symmetrical, i.e. isomorphic, synthesis for these control problems results in the same additional restrictions. It is therefore sufficient to perform synthesis on a reduced form of the control problem, as long as it is correctly restored afterwards. This model restoration process is described in Section 5.3.

The third model reduction step that is proposed is called *internal symmetry*, and is based on symmetry between component models and requirement models within a single vertex. If two synchronous component models are isomorphic and all requirement models that refer to those component models are isomorphic, only one of the component models is needed during synthesis. When synthesis adds no restrictions to the events of this component model, there are no restrictions for the omitted component model either. Conversely, when synthesis does add restrictions to events in the reduced model, the removed part of the model gets the same restrictions in the model restoration process.

An example of internal symmetry is elaborated for the control problem in Figure 5.5. One can see that all component models are synchronous as they share event a. Furthermore, $P_1 \simeq P_2$, and therefore $R_1 \simeq R_2$.



R_1 : b **needs** $P_3.q_5$

R_2 : c **needs** $P_3.q_5$

Figure 5.5: Example of a control problem for internal symmetry.

At the left-hand side in Figure 5.6, the vertex is given that corresponds to the control problem of Figure 5.5, as well as the internal relations between the component models based on the requirement models. The black dots represent the component models and the solid arrows indicate the requirement models that connect the component models. Furthermore, component models P_1, P_2, and P_3 are connected with a solid line to indicate their synchronous connection over a controllable event.

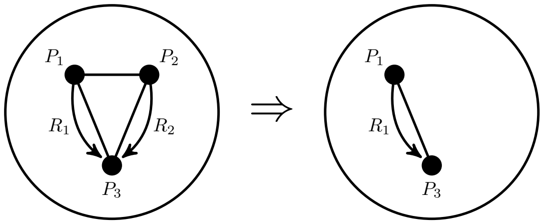


Figure 5.6: Vertex that contains internal symmetry and the corresponding vertex after model reduction.

In this example, from the established isomorphisms it can be determined that the model can be reduced by removing P_2 and R_2. The resulting vertex is shown at the right-hand side of Figure 5.6. The external graph representation of this example is shown in Figure 5.7. Note that when the control problem consists of multiple vertices, requirement models outside the vertex may refer to the isomorphic component models inside the vertex. In this case, these external requirements must be isomorphic as well.

Note the difference in visualization of requirements R_1 and R_2 between Figures 5.5 and 5.6. Both graphs represent the same control problem, though the visualization of Figure 5.5 provides more information on the internal connections. Depending on whether the structure within a vertex or the structure between different vertices is considered, either of the visualization types may be preferred.

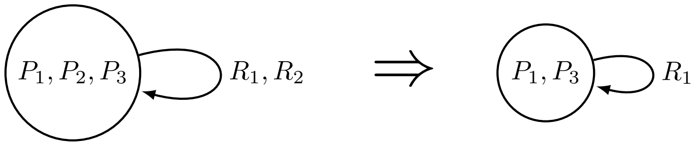


Figure 5.7: Dependency graph that contains V-symmetry and the corresponding graph after model reduction.

An important distinction is made between synchronization over controllable events and synchronization over uncontrollable events. Synchronous connections between components over uncontrollable events are indicated by a dashed line, like in the graphs shown in Figure 5.8. Since a synchronous relation between automata over an uncontrollable event can result in additional restrictions on controllable events in those automata, it is not possible to apply internal symmetry reduction in these cases.

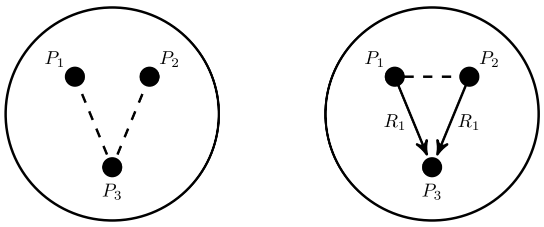


Figure 5.8: Cases with synchronization of uncontrollable events where internal symmetry reduction cannot be applied.

5.2.4 External symmetry

Similar to the internal symmetry within a vertex, symmetry can also exist between vertices and edges, called *external symmetry.* Two or more vertices are symmetric if their sets of component models are isomorphic as well as their requirement models. First, all isomorphisms in the control problem are determined. Second, the dependency graph is used to determine symmetry in the system that can be exploited for model reduction. Three types of external symmetry are described in the following sections. For the sake of simplifying the explanations, all vertices in the examples consist of one automaton, and all edges contain one requirement. The steps are also applicable for sets of automata and sets of requirements.

V-symmetry

The first step of external symmetry is V-symmetry. Here, two isomorphic vertices are both connected to a third vertex, and the requirements of those connections are isomorphic as well. This situation is visualized on the left-hand side of Figure 5.9, where $P_1 \simeq P_2$ and $R_1 \simeq R_2$. In this situation, vertex P_2 and edge R_2 can be removed before synthesis as they are equivalent in behavior to the vertex P_1 and edge R_1. The right side of Figure 5.9 shows the reduced dependency graph for this scenario. When synthesis does not add extra restrictions for component model P_1, there would be no restrictions for P_2 either. Conversely, any additional restrictions added to component model P_1 during synthesis also need to be applied to component model P_2, which is done in the model restoration process.

The dependency graph shown in Figure 5.9 is the simplest form of V-symmetry. It can also be applied when vertices P_1 and P_2 are replaced by multiple vertices and edges, as long as they are disjoint and isomorphic.

U-symmetry

The second type of external symmetry in dependency graphs is U-symmetry. It is similar to V-symmetry, though here it involves a cyclic subgraph. The left side of Figure 5.10 shows the simplest form of U-symmetry.

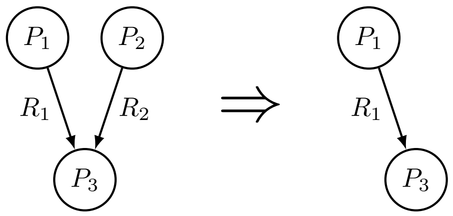


Figure 5.9: Dependency graph that contains V-symmetry and the corresponding graph after model reduction.

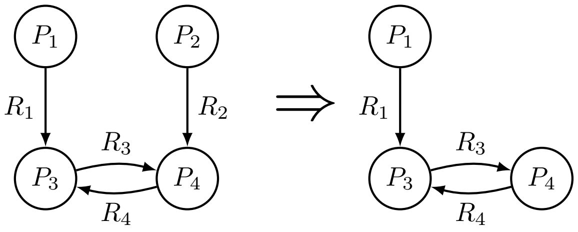


Figure 5.10: Dependency graph that contains U-symmetry and the corresponding graph after model reduction.

In this case, two vertices P_3 and P_4 are connected to each other by requirement R_3 and R_4. Furthermore, there exist two vertices, P_1 and P_2, that are connected to P_3 and P_4, respectively. If

$$P_1 \simeq P_2, \quad P_3 \simeq P_4, \quad R_1 \simeq R_2, \quad R_3 \simeq R_4, \tag{5.27}$$

then component model P_2 and requirement model R_2 can be removed before synthesis. The reduced version of the dependency graph in this scenario is shown at the right-hand side of Figure 5.10. Again, in this explanation the simplest form of U-symmetry is considered. It can be extended for the case where vertices P_1 and P_2 are replaced by sets of vertices and edges, as long as those sets are disjoint and isomorphic.

T-symmetry

The third model reduction step based on external symmetry in dependency graphs is T-symmetry, which stands for triangular symmetry. This involves a subgraph with a cycle between two vertices and a third vertex that is connected to both other vertices, as is visualized on the left-hand side of Figure 5.11. As can be seen, vertex P_1 is connected to both vertices P_2 and P_3 by requirement R_1. This can be the case if R_1 has, for example, the form $P_1.a$ **`needs`** $P_2.q_1$ **`or`** $P_3.q_2$.

Under certain conditions for R_1, P_2, and P_3, the model can be reduced by removing part of R_1 and therefore also removing one of the edges in the dependency graph. Take the dependency graph example shown on the left-hand side in Figure 5.11. The first condition is that $P_2 \simeq P_3$. Secondly, the part of R_1 that refers to P_2 is isomorphic to the part of R_1 that refers to P_3, and there is an `or` operator between them. Under these conditions, it is sufficient to only use one part in the requirement during synthesis. Requirement R_1 is then partly removed, resulting in R_1'. The right-hand side of Figure 5.11 shows the reduced form of the dependency graph.

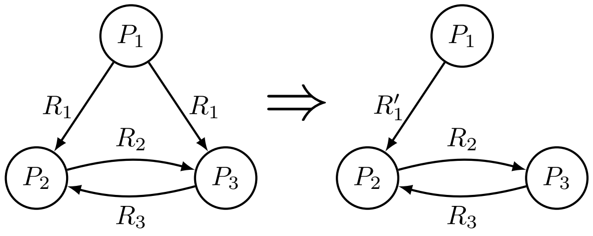


Figure 5.11: Dependency graph that contains T-symmetry and the corresponding graph after model reduction.

The reasoning is that if the part of R_1 that refers to P_2 can become `true` at some point, then the part referring to P_3 can become `true` with equal strictness, i.e. they can become `true` under the same conditions (as long as isomorphisms are applied). Using only one of the parts is therefore exactly as strict as using both, as long as there is an `or` between them. Similarly, if the part that refers to P_2 does never become `true`, then neither will the isomorphic other part, so using only one of the two will result in the same (empty) supervisor.

Note that T-symmetry can also be used when the requirements are marked requirements with **`and`** operators, though these requirements have already been removed in the reduction step that concerns marked requirements as described in Subsection 5.2.2.

5.2.5 Full symmetry

In certain cases, a dependency graph consists of several disconnected subgraphs. This can occur when a system consists of disconnected subsystems, or when part of the dependency graph is removed in the previous reduction steps and thus splits the graph. The final model reduction step, called *full symmetry*, can be applied when disconnected subgraphs are fully symmetrical. This is the case when the control problem of one subgraph is isomorphic to the control problem of the second subgraph.

Take the example shown in Figure 5.12. On the left-hand side, a fully connected dependency graph is shown with 2 cyclic subgraph. It contains a vertex P_3 that can be removed using the independent vertices model reduction step. The resulting dependency graph after applying this reduction step is shown in the middle of Figure 5.12. Here, two disconnected subgraphs are acquired that are fully symmetrical. Using the full symmetry reduction step, one of these subgraphs can be omitted during synthesis, as shown in the dependency graph on the right-hand side of Figure 5.12.

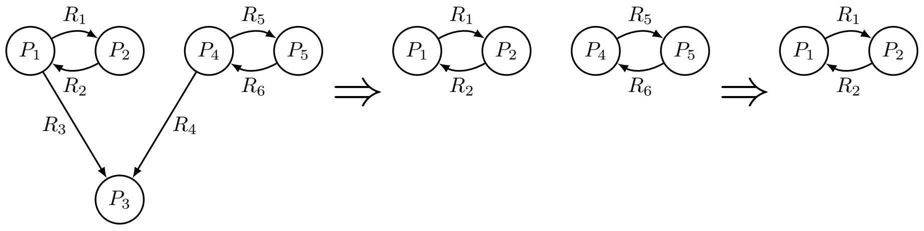


Figure 5.12: Dependency graph (left) that contains full symmetry after removing the independent vertex (middle) and the corresponding graph after model reduction (right).

The reasoning is that synthesis for one of the subgraphs results in the same additional restrictions as in the other subgraph. Performing synthesis for one of subgraphs, and correctly restoring the model afterwards, thus yields a correct supervisor. The model restoration process is described in the next section.

5.3 Model restoration

After each model reduction step has been applied, supervisor synthesis can be used to generate a supervisor. As the number of automata and requirements in the model is lower than before the reduction steps, this synthesis procedure is faster and requires less computational power. The synthesis result must, however, be restored to the original model in order to use it for model simulation and implementation code generation. This model restoration procedure depends on whether any additional restrictions are added to the plant during the synthesis procedure. Both cases are elaborated here.

5.3.1 No additional restrictions

When no additional restrictions are added to the controllable events in the plant during the synthesis procedure, the plant combined with the requirements is already controllable, nonblocking, and maximally permissive. Since the model reduction steps proposed in the previous section are based on the principle that the reduced model is at least equally strict as the original model, i.e. synthesis for the reduced model never results in fewer restrictions, this means that synthesis for the original model would also not add any additional restrictions. Therefore, the model restoration procedure in this case is simply returning to the original plant model and requirements model.

An example case is described where the model reduction steps are applied, synthesis is performed during which no additional restrictions are added, and the model is restored. This example concerns a system consisting of two machines M_1 and M_2, two lights L_1 and L_2, and two sensors S_1 and S_2, as shown in Figure 5.13. The machines and lamps are standard actuator automata, shown in Figure 2.2, and the sensors are standard sensors, shown in Figure 2.3. Figure 5.13 indicates the relations between the components, represented by the dashed arrows that are labeled by the requirements. The requirements define that each machine can only be on when the corresponding sensor is on, but the machines may not be turned on at the same time. Furthermore, each lamp is turned on when the corresponding machine is on.

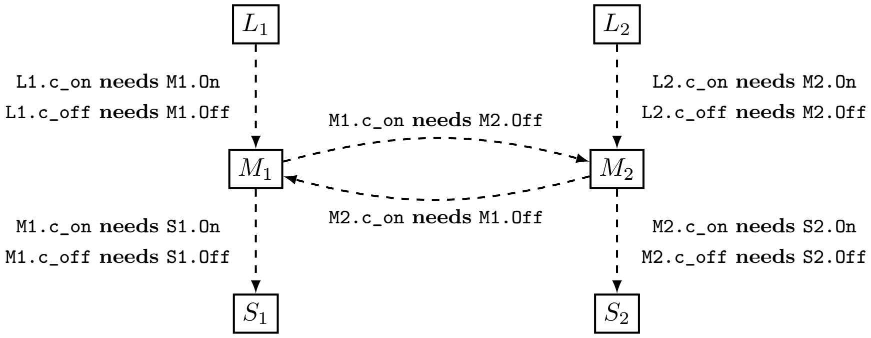


Figure 5.13: Overview of a system consisting of two machines M_1 and M_2, two lights L_1 and L_2, and two sensors S_1 and S_2.

The dependency graph of this system is shown on the left-hand side of Figure 5.14. Using this dependency graph, the applicability of each model reduction step is considered. First, the independent vertices are removed, which are the vertices of sensors S_1 and S_2. The resulting dependency graph can be seen in the middle of Figure 5.14. Second, the requirements in the system are analyzed to determine whether they are marked requirements. In this case, the requirements

$$\texttt{L1.c\_off}\ \textbf{needs}\ \texttt{M1.Off} \quad \text{and} \tag{5.28}$$

$$\texttt{L2.c\_off}\ \textbf{needs}\ \texttt{M2.Off} \tag{5.29}$$

are marked requirements, and can thus be removed from the model. This does, however, not affect the dependency graph as there still remain dependencies between the lights and the machines. Third, internal symmetry is considered for the remaining model. However, since the model contains no synchronous models, internal symmetry is not applicable. Finally, external symmetry is considered. As can be determined by looking at the dependency graph (in the middle of Figure 5.14), there is U-symmetry in the model. Applying the U-symmetry reduction step removes the L_2 vertex and the corresponding edge with M_2 from the model.

The resulting dependency graph is shown on the right-hand side of Figure 5.14. No further reduction steps are applicable for this graph.

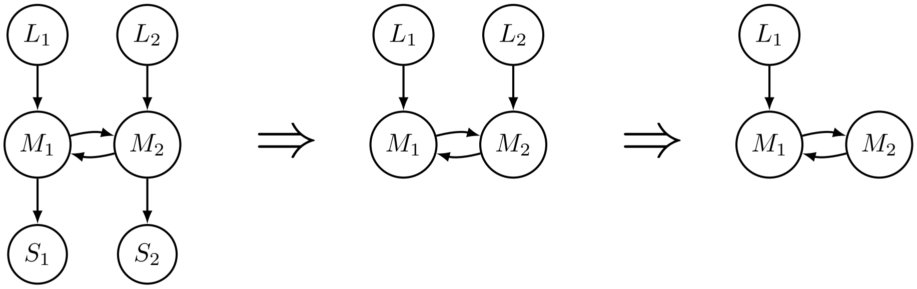


Figure 5.14: Dependency graph of the system shown in Figure 5.13 (left), after removal of independent vertices (middle), and after applying U-symmetry reduction (right).

After all model reduction steps are applied, synthesis is performed on the remaining plant model and requirements model. In this case, synthesis is performed for machines M_1 and M_2 and light L_1. The resulting supervisor is shown in Figure 5.15, which adds no additional restrictions to any of the controllable events in the plant. This means that the model restoration process simply consists of taking the original plant model and requirements model.

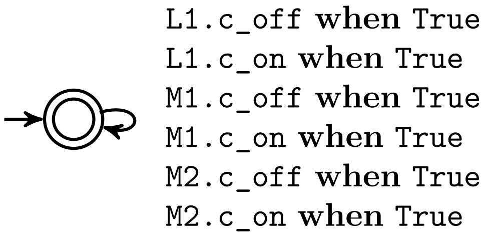

```
L1.c_off when True
L1.c_on when True
M1.c_off when True
M1.c_on when True
M2.c_off when True
M2.c_on when True
```

Figure 5.15: Synthesized supervisor of the reduced model.

5.3.2 Additional restrictions

In the case that additional restrictions are added during the synthesis procedure to one or more controllable events, this means that the plant model combined with the requirements is not inherently nonblocking and controllable. The additional restrictions on the reduced model must therefore also be applied to the original model. Moreover, when additional restrictions are applied to a part of the model that was originally symmetric to a removed part of the model, these restrictions must be applied to the removed part in the original model as well.

The following steps are taken to restore the model after synthesis is performed, and additional restrictions have been added:

1. Take the original plant model and requirements model.

2. Add the supervisor model from the synthesis procedure applied to the reduced model.

3. For every controllable event that is *not* part of an isomorphism, i.e. the event is not part of any $h_1 \in h$ for any $h \in \mathcal{H}$, that has an additional restriction in the supervisor model, apply the following steps:

   (a) Determine whether the additional restriction refers to locations or variables that are part of an isomorphism.

   (b) For each of these parts, extend the restriction by combining the location or variable with its isomorphic counterpart. For instance, in control problem $\mathcal{P} = \{P_1, P_2, P_3\}$ with $P_1 \simeq P_2$ and $h_2(P_1.A) = P_2.B$, an additional restriction
   $$P_3.a \textbf{ when } P_1.A$$
   is extended to
   $$P_3.a \textbf{ when } (P_1.A \ \wedge \ P_2.B).$$

4. Extend the supervisor model with the controllable events that were removed during the reduction steps. For each added event, denoted by e, apply the following steps:

   (a) Use the isomorphism mapping $h_1 \in h \in \mathcal{H}$, which maps the events between symmetrical components, to determine the symmetric counterpart to e, denoted by e_S.

   (b) Look up event e in the synthesized supervisor, and take the guard of this event. Note that this guard can either be `True` or a restriction added during synthesis.

   (c) Apply the same guard to event e_S.

   (d) For each location mentioned in this guard, determine if it is part of an isomorphism, i.e. if the location is part of any $h_2 \in h$ for any $h \in \mathcal{H}$. Depending on the outcome, apply one of the following steps:

      i. If the location is part of an isomorphism, translate this location to its symmetrical counterpart using h_2 of this isomorphism.

      ii. If the location is *not* part of an isomorphism, do nothing. In this case, the location mentioned in the guard remains the same.

To illustrate these model restoration steps, an example case is included where additional restrictions are added during synthesis on the reduced model. The system that is considered in this example is visualized in Figure 5.16, and consists of three machines M_1, M_2, and M_3 and three buffers B_1, B_2, and B_3. The automata that are used to model machines M_1, M_2 and buffers B_1, B_2, and B_3 are shown in Figure 2.11, as well as the requirement that connects each buffer to its corresponding machine.

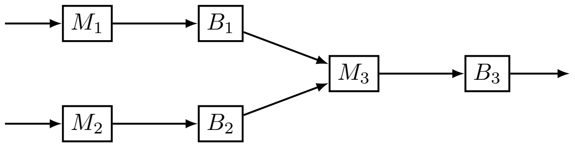


Figure 5.16: Overview of a system consisting of three machines M_1, M_2, and M_3 and three buffers B_1, B_2, and B_3.

As seen in Figure 5.16, machine M_3 is connected to buffers B_1 and B_2. The automaton model for M_3 is shown in Figure 5.17.

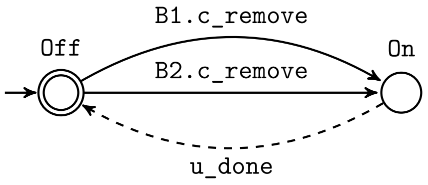


Figure 5.17: Automaton model for machine M_3.

The left-hand side of Figure 5.18 shows the dependency graph corresponding to the system shown in Figure 5.16. This dependency graph consists of a single vertex, as all automata are synchronous. The dependency graph therefore shows the contents of this vertex, in which the synchronous relations between the automata are indicated by dashed lines.

During the model reduction process, internal symmetry is considered as the vertex in the dependency graph consists of multiple components. It is determined that $M_1 \simeq M_2$, with

$$h_1 = (\texttt{M1.c\_start} \to \texttt{M2.c\_start},\ \texttt{M1.u\_done} \to \texttt{M2.u\_done}) \text{ and} \tag{5.30}$$
$$h_2 = (\texttt{M1.Off} \to \texttt{M2.Off},\ \texttt{M1.On} \to \texttt{M2.On}) \tag{5.31}$$

and that $B_1 \simeq B_2$, with

$$h_1 = (\texttt{B1.u\_done} \to \texttt{B2.u\_done},\ \texttt{B1.c\_remove} \to \texttt{B2.c\_remove}) \text{ and} \tag{5.32}$$
$$h_2 = (\texttt{B1.Empty} \to \texttt{B2.Empty},\ \texttt{B1.Full} \to \texttt{B2.Full}). \tag{5.33}$$

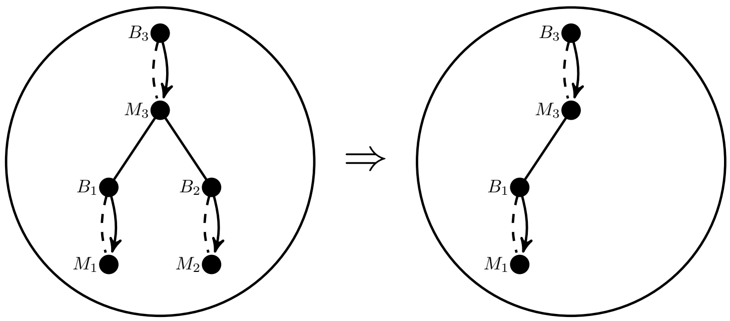


Figure 5.18: Contents of the vertex for the control problem of Figure 5.16 (left) and the corresponding reduced version (right).

Following the internal symmetry reduction step, it is thus determined that components M_2 and B_2 can be removed from the system before performing synthesis. The resulting contents of the vertex of the dependency graph are shown on the right-hand side of Figure 5.18.

Next, as there are no more applicable model reduction steps, synthesis is performed on the remaining model consisting of machines M_1 and M_3 and buffers B_1 and B_3. During this synthesis procedure, additional restrictions are added to events `M1.c_start` and `B1.c_remove` since the system by its own is not controllable and nonblocking. The left-hand side of Figure 5.19 shows the synthesized supervisor, including these additional restrictions.

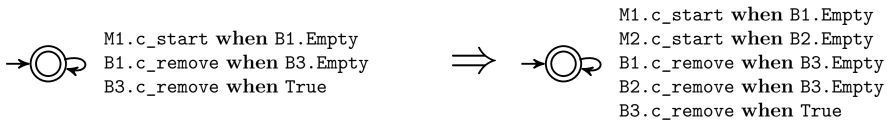

M1.c_start **when** B1.Empty
B1.c_remove **when** B3.Empty
B3.c_remove **when** True
$\Rightarrow$
M1.c_start **when** B1.Empty
M2.c_start **when** B2.Empty
B1.c_remove **when** B3.Empty
B2.c_remove **when** B3.Empty
B3.c_remove **when** True

Figure 5.19: Supervisor with additional restrictions of the reduced model (left) and the supervisor of the restored model (right).

Subsequently, the model is restored using the model restoration steps provided in this section.

1. The original plant model and requirements model are taken.

2. The synthesized supervisor, shown on the left-hand side of Figure 5.19, is added.

3. There are no controllable events that are *not* part of an isomorphism that have an additional restriction in the supervisor model, so this step can be skipped.

4. The supervisor model is extended by the controllable events that were removed during the model reduction steps. In this example, these events include `M2.c_start` and `B2.c_remove`.
   (a) Isomorphism mappings h_1 of Equations (5.30) and (5.32) are used to determine the symmetrical counterparts to these events, which are `M1.c_start` and `B1.c_remove`.
   (b) Events `M1.c_start` and `B1.c_remove` are looked up in the supervisor model, and the corresponding guards **when** `B1.Empty` and **when** `B3.Empty` are obtained.
   (c) The same guards are applied to events `M2.c_start` and `B2.c_remove`.
   (d) Each location mentioned in the added guards is translated (if possible) using isomorphism mappings h_2 of Equations (5.31) and (5.33). For event `M2.c_start`, the guard **when** `B1.Empty` is translated to **when** `B2.Empty`. For event `B2.c_remove`, the guard **when** `B3.Empty` remains the same, as there is no isomorphism mapping for location `B3.Empty`.

The supervisor of the restored model is shown on the right-hand side of Figure 5.19. It shows the added events `M2.c_start` and `B2.c_remove`, with the guards **when** `B2.Empty` and **when** `B3.Empty`, respectively.

5.4 Case studies

In this section, several case studies are described in which the proposed model reduction is applied to synthesize a supervisor. First, an extensive case study for the Eerste Heinenoord tunnel is given. Second, an overview of the model reduction results for a set of case studies is shown, specifically for a road tunnel, a production line, a waterway lock, and a movable bridge.

5.4.1 The Eerste Heinenoord tunnel

In this section, a case study is described in which the proposed method for model reduction is applied for the model of the Eerste Heinenoord tunnel. The Eerste Heinenoord tunnel, shown in Figure 1.1, is a tunnel in the Netherlands to bypass the river the “Oude Maas”. It is a tunnel with two traffic tubes and an escape route in between them.

The complete tunnel has been modeled using parameter-based modeling, resulting in a model that consists of 1163 components and 2015 requirements. Monolithic synthesis proved incapable of solving the synthesis problem since the state space of the uncontrolled plant is too large, with the result that the supervisor cannot be calculated with the available memory. For this reason, the method of model reduction has been applied. For this purpose, a dependency graph has been created for the initial model, which is shown in Figure 5.20. For the sake of simplicity, loops are omitted in this visualization, and multiple dependencies between two vertices are represented by one edge. As is indicated in the figure, the tunnel consists of two traffic tubes, an escape route, and a set of water cellars. One can clearly see the resemblance between the graphs of the two traffic tubes, and how the escape route and the water cellars are connected to both tubes. This level of symmetry is mainly a result of parameter-based modeling, since each traffic tube is instantiated from the same template, only using different parameters.

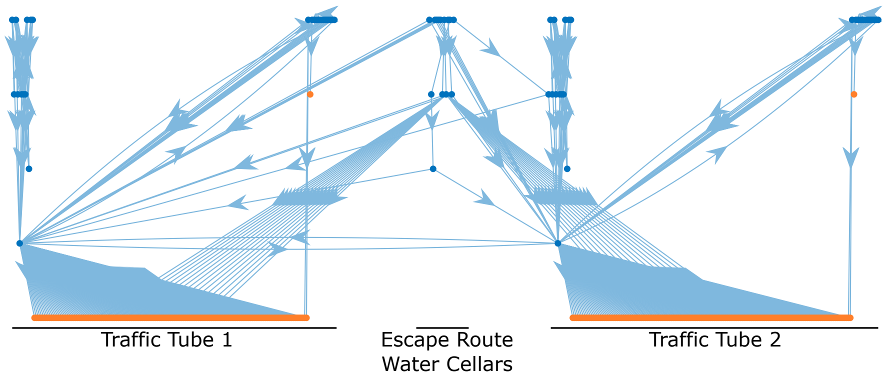


Figure 5.20: Initial graph of the Eerste Heinenoord tunnel model.

The first reduction step that is applied removes the independent vertices in the graph, which are the vertices that have no outgoing edges. In this step, the vertices indicated in orange in Figure 5.20 are removed. These vertices represent the sensor models in the tunnel. Removing the vertices and their incoming edges yields the dependency graph shown in Figure 5.21. In this reduced model, the automata of the sensor models have been removed and the requirement models are adjusted accordingly. For example, a requirement

$$\texttt{Actuator.c\_on}\ \textbf{needs}\ \texttt{Sensor.On} \wedge \texttt{Button.Pressed} \tag{5.34}$$

becomes

$$\texttt{Actuator.c\_on}\ \textbf{needs}\ \texttt{True} \wedge \texttt{Button.Pressed}. \tag{5.35}$$

After this reduction step, the number of component models is reduced to 751 and the number of requirements to 1836. The reason for the low reduction in requirement models is the fact that most requirements are not removed completely, such as the requirement reduction shown in Equations (5.34) and (5.35). In these cases, the requirements are simplified, but the number of requirements remains the same.

The other model reduction steps include the marked requirements, internal symmetry, V-symmetry, U-symmetry, and T-symmetry. Figure 5.21 shows which vertices and edges are removed during each of these steps. The dependency graph of the final Eerste Heinenoord tunnel model is shown in Figure 5.22. It contains 25 vertices and 33 edges, representing the final 277 automata and 384 requirements.

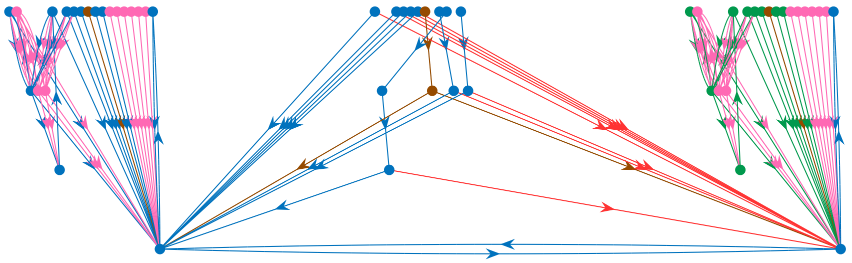


Figure 5.21: Graph of the Eerste Heinenoord tunnel model after applying the independent vertices model reduction step. Vertices and edges that are removed in subsequent steps due to marked requirements (brown), V-symmetry (pink), U-symmetry (green), and T-symmetry (red), are indicated.

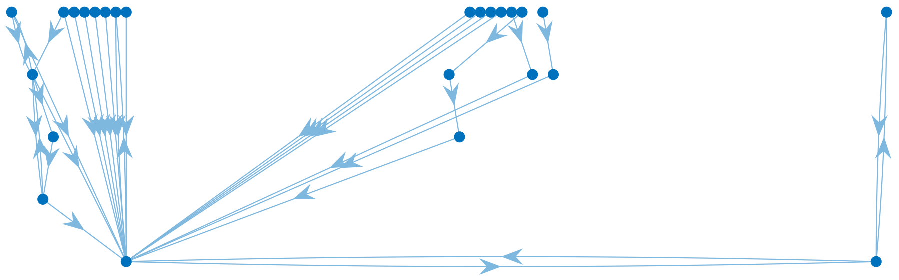


Figure 5.22: Graph of the Eerste Heinenoord tunnel model after all model reduction steps.

In total, six model reduction steps have been applied to reduce the number of component models from 1163 to 277 and reduce the number of requirement models from 2015 to 384. The bar graph in Figure 5.23 gives an overview of the number of component models, requirement models, vertices, and edges after each reduction step.

After applying the model reduction steps to the model of the Eerste Heinenoord tunnel, a supervisor has been synthesized from the remaining component models and requirement models. To give an indication of the effectiveness of the model reduction steps, monolithic synthesis has been attempted after each reduction step. Table 5.1 shows the number of controlled states, i.e. the number of states

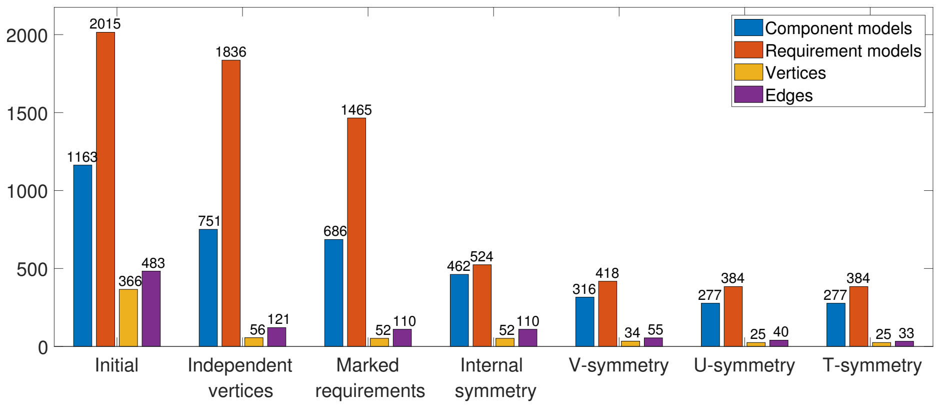


Figure 5.23: The number of component models, requirement models, vertices, and edges after each reduction step.

in the synthesized supervisor. Furthermore, it shows the required computation time of the synthesis procedure, as well as the time reduction factor compared to the synthesis procedure one reduction step earlier.

Note that when determining the order in which the reduction steps are applied, it is only important that the internal symmetry step is applied before the other symmetry steps, as this may affect the final outcome. Interchanging of the other steps is allowed, if so desired.

Table 5.1: The number of states in the controlled state space, the computation time after each reduction step, and the reduction factor compared to the previous synthesis procedure in the Eerste Heinenoord tunnel case study.

	Controlled states	Synthesis time	Reduction factor
Initial	n/a	n/a	n/a
Independent vertices	$2.15 \cdot 10^{220}$	06h30m	–
Marked requirements	$1.03 \cdot 10^{197}$	01h52m	3.5
Internal symmetry	$3.99 \cdot 10^{92}$	15m04s	7.4
V-symmetry	$1.37 \cdot 10^{64}$	03m47s	4.0
U-symmetry	$2.39 \cdot 10^{55}$	02m01s	1.9
T-symmetry	$2.39 \cdot 10^{55}$	01m51s	1.1

The initial model of the Eerste Heinenoord tunnel turned out to be too large for monolithic synthesis, due to insufficient computing power. The number of states in the uncontrolled system is manually calculated to be 10^{257} states.

After the first model reduction step, where independent vertices are removed, a supervisor has been synthesized in around 6.5 hours (23000 seconds) with 10^{220} states. As this is the first successful synthesis procedure, no time reduction factor can be calculated.

Table 5.1 shows that the next model reduction step, where marked requirements are removed, reduces the controlled state space to 10^{197} states, and decreases the required synthesis time by a factor of 3.5.

The other model reduction steps further reduce the synthesis time by a factor of 7.4, 4.0, 1.9 and 1.1, respectively. Based on these factors, it can be concluded that in terms of synthesis time, the internal symmetry model reduction step is the most effective. It should, however, be noted that the order in which the reduction steps are applied may influence the effectiveness of each step.

After the other model reduction steps have been applied, a supervisor was synthesized in 111 seconds with 10^{55} states. Applying the model reduction steps thus resulted in a decrease in required computation time of a factor 210. Even though it is most important that the synthesis problem is solvable at all, it is nonetheless beneficial for the design engineer to be able to synthesize a controller quickly, e.g., to iteratively synthesize, validate, and correct the controller.

The synthesized supervisor added no additional restrictions to the controllable events of the plant. The model restoration process therefore consisted of returning to the initial plant model and requirements model.

5.4.2 Overview

This section provides an overview of a set of case studies to investigate the applicability of the proposed model reduction steps. The case studies include the Eerste Heinenoord tunnel, the FESTO production line (Reijnen et al. (2018)), the Prinses Marijke lock (Reijnen et al. (2019b)), and the Oisterwijksebaan bridge (Reijnen et al. (2021)). For each of these systems, the model reduction steps have been applied, and the number of component models and requirements models is analyzed. Moreover, the synthesis time of the original model is compared with the synthesis time of the reduced model.

Table 5.2 shows the results of the case studies. For each system, the number of component models and requirement models in the initial model is shown. The left number of each entry indicates the number of component models, while the right number indicates the number of requirement models. Furthermore, the table shows the remaining number of component models and requirement models after each applied reduction step. When a reduction step is not applicable for a system, this entry is left empty, such as the marked requirements step for the Prinses Marijke lock. Furthermore, Table 5.2 shows the total model reduction factor for the reduction in the number of component models and requirement models, and the time reduction factor when comparing the initial synthesis time with the reduced synthesis time.

Note that a reduction due to marked requirements also reduces the number of components in the plant model. This is due to the fact that removal of a marked requirement may remove an edge in the dependency graph, which may result in a disconnected vertex. A disconnected vertex can be solved in a separate synthesis procedure, and is therefore also included as a reduction due to removal of marked requirements.

Table 5.2: The model reduction results for a set of case studies pertaining the Eerste Heinenoord tunnel, the FESTO production line, the Prinses Marijke lock, and the Oisterwijksebaan bridge.

	Eerste Heinenoord tunnel		FESTO production line		Prinses Marijke lock		Oisterwijksebaan bridge	
Initial model	1163	2015	105	202	177	300	126	190
Independent vertices	751	1863	21	22	167	153	105	134
Marked requirements	686	1465	20	21				
Internal symmetry	462	524						
External symmetry	277	384					90	119
Full symmetry			18	19	95	81		
Model reduction factor	4.2	5.2	6	10.6	1.9	3.7	1.4	1.6
Time reduction factor	210		248		75		2.5	

From the results shown in Table 5.2 it can be concluded that the proposed model reduction steps are applicable to all case studies considered, though the effectiveness varies per system. The reduction step that considers independent vertices is applicable in all case studies, and is especially effective in sensor-heavy system such as the FESTO production line. Marked requirements can only be considered when there are requirements that are not part of a cycle in the dependency graph. The Prinses Marijke lock and the Oisterwijksebaan bridge do not have such requirements, which makes the reduction step not applicable. The symmetry reduction steps are, naturally, only effective when the system contains symmetrical subsystems. As can be seen in the results, each system is symmetrical to some degree, which makes the symmetry reduction steps applicable in all case studies.

It should be noted that the effectiveness of the model reduction steps also depends on the number of asynchronous components in the system. In systems where all components are synchronous, dependency graphs give no insight in the model structure, as all components are part of the same vertex. In such a case, only the internal symmetry reduction step might be applicable, thus making the complete reduction process much less effective. However, systems that are created using the component-based or parameter-based modeling method are inherently less synchronous, as the behavior of each component is modeled by a separate automaton.

Regarding the model restoration process, in all cases no additional restrictions where added during the synthesis procedure. Following the steps described in

Subsection 5.3.1, restoring the original model in such a case is simply returning to the initial plant model and requirements model. As this requires negligible time, this is not taken into account in the results shown in Table 5.2. In the case that there are additional restrictions added during synthesis, the model restoration process takes slightly more time. However, this is typically still negligible relative to the synthesis time.

5.5 Concluding remarks

In this chapter, a new method is proposed to reduce a plant model and a requirements model for the purpose of supervisor synthesis. Multiple steps are described to remove models and simplify requirements before performing synthesis, without losing the guarantees that synthesis gives. The steps are mainly based on symmetry that can be detected in dependency graphs of the system. Furthermore, the model restoration process is described which depends on the addition of guards during the synthesis procedure.

The new method was applied in several case studies. For the design of a controller for a road tunnel, it removed 82% of the component models and 89% of the requirement models. Note that these percentages only including complete removals of component models and requirement models, not simplifications such as the requirement in Equations (5.34) and (5.35). By reducing the model, the synthesis problem that was initially impossible to solve due to memory issues is solved in two minutes. Furthermore, an overview of a set of case studies is provided that shows the applicability of the model reduction steps to a range of industrial systems. In each of these case studies, the model reduction steps showed to be effective, albeit in varying degrees.

In the case studies considered in this chapter, only event-condition requirements are used. While the proposed definition of isomorphism between EFAs may also be applied to requirements that are modeled as an EFA, it would be interesting how this affects the applicability of all model reduction steps and their effectiveness when this type of requirements is also included in the model.

Further research includes defining an algorithm to automatically detect when model reduction steps are applicable, and to automatically reduce the model accordingly.

Chapter 6

Implementation of supervisory controllers

Automatically synthesizing a supervisor removes the need for verification, as the synthesis algorithms have been proven to deliver correct results with respect to the given input. Validation, however, remains an important step, as the given input may still be erroneous. In SBE, three validation stages are recognized. First, model simulation is performed after supervisory controller synthesis, where the supervisor is combined with the plant, often including a visualization as promoted in Rohrer (2000). This allows the design engineer to quickly and intuitively analyze the controlled behavior. Second, controller code is generated and implemented on the implementation platform, typically a programmable logic controller (PLC). The implemented controller is then again connected to the plant model to perform hardware-in-the-loop (HIL) simulations. In HIL simulations, the actual implemented controller is tested with the actual operating semantics of the PLC in which the real-time aspect plays a role. Furthermore, external subsystems, such as an operator interface, can be connected to the PLC to simulate an operator commanding and monitoring the system. Applications of HIL simulation have been shown before in Li et al. (2009) for wind turbines, in Dai et al. (2016) for a mineral grinding process, and in Reijnen et al. (2019b) for a waterway lock complex. Third, the final testing stage is the integration test, where the implemented controller is tested with the realized plant.

This chapter focuses on the derivation of an implementable controller, and its validation using HIL simulation, specifically for large-scale cyber-physical systems.

This chapter is based on: Moormann, L., Hofkamp, A.T., van de Mortel-Fronczak, J.M., Fokkink, W.J., and Rooda, J.E. Derivation and hardware-in-the-loop testing for a road tunnel controller. In *Proceedings of the 16th IFAC Workshop on Discrete Event Systems (WODES)*, 2022.

As an example, controllers for road tunnels are used. A road tunnel is a large infrastructural system, consisting of many actuators and sensors. Its controller is responsible for maintaining a safe environment for the road users by monitoring for emergencies and correctly handling detected ones.

As introduced in Subsection 2.5.1, PLCs operate in real-time cycles of processing inputs, executing enabled controllable events, and updating the outputs. Generating an implementable controller bridges the gap between the asynchronous, discrete-event, world of synthesized supervisors and the synchronous, real-time, world of PLC controllers. To ensure that the real-time implementation of the controller expresses the same controlled behavior as the synthesized supervisor, certain properties need to be met, being finite response and confluence, and nonblocking under control. Previous work in Malik (2003) extensively describes these properties, and defines sufficient conditions to determine if they are met. In this chapter, especially the confluence property plays an important role, which states that all sequences of events from a state lead to the same end-state.

Besides the properties of an implementable controller, there can be functional discrepancies between a discrete-event supervisor and the real-time implementation of the controller. In certain cases, a discrete-event change in an actuator is not realizable. An example related to road tunnels is the ventilation system. A ventilation unit cannot be set to its maximum speed immediately, as such a sudden change requires too much energy, which could result in a power failure, and the forces generated could result in failing fixtures, and thus breaking of the ventilation unit. To cope with these functional discrepancies between discrete-event supervisors and its real-time implementation, a resource controller is needed that converts the discrete signal, i.e. the instant change in speed, to a correct continuous-time signal.

Another aspect of an implemented controller is its performance, determined by the PLC cycle time. For a given synthesized supervisor, multiple correct implementable controllers are possible that can vary in performance. To obtain an implementable controller with a high performance, event functions can be sequenced to optimize the order in which they are executed. Event functions are functions in the PLC code that determine if an event of a component must be executed, such as turning on an actuator. Existing sequencing procedures are described in Steward (1981), Meier et al. (2007), and Eppinger and Browning (2012).

In this chapter, several extensions to the controller derivation method from Reijnen et al. (2019a) are described, being a relaxation in the sufficient conditions for the confluence property, called *end-state equality*, the design of resource controllers to translate a discrete-event signal to a continuous-time signal, automatic event function sequencing, and automatic code generation that is adaptable for any desired target platform. Furthermore, an industrial-size case study is presented where an implementable controller is derived, PLC code is automatically generated for ABB PLCs, and a HIL setup is created for the Swalmen tunnel, a road tunnel in the Netherlands near Roermond. A road tunnel contains more components than a lock-bridge combination as shown in Reijnen et al. (2019a), which makes supervisor synthesis more challenging, and thus makes the design decisions made during modeling more relevant. For example, using resource controllers to fulfill the local control functionality of a component alleviates the synthesis procedure of the supervisory controller. This makes resource controllers play a larger role in road tunnel controllers.

The remainder of this chapter is organized as follows. First, the Swalmen tunnel is introduced in Section 6.1 as the system that is used as case study throughout this chapter. Second, Section 6.2 describes the controller derivation process in more detail, including the steps of creating resource controllers, verifying the implementable supervisor properties, PLC code generation, and PLC code optimization. In Section 6.3, the validation method of HIL testing is described. Finally, in Section 6.4, conclusions are drawn.

6.1 Case study: The Swalmen tunnel

Throughout this chapter, the Swalmen tunnel is used as a case study to demonstrate the implementation and testing process of supervisory controllers. The Swalmen tunnel is a road tunnel near the town Swalmen in the Netherlands. It consists of two traffic tubes, each with two driving lanes, and is 400 meters long. Between the traffic tubes is an escape route that is used in case of evacuation. In Figure 6.1, the northern entrance of the Swalmen tunnel is shown as well as the exit door of the escape route (on the left side of the traffic tube).

The first purpose of a road tunnel controller is to monitor the situation in the traffic tubes to detect an emergency. Second, after an emergency is detected, the controller handles it by closing the traffic tubes and turning on evacuation systems. Finally, the controller is responsible for correctly carrying out operator commands. The main components in a road tunnel are listed:

- Emergency detection: smoke detection, speed measurements, emergency cabinet sensors, closed circuit television (CCTV);
- Emergency handling: ventilation, lighting, fluid draining, fire extinguishing water supply;
- Tube closure: boom barriers, traffic lights, height detection;
- Evacuation: escape doors, escape route lighting, escape route ventilation, broadcasting system.

The Swalmen tunnel has been modeled using 510 automata and 344 requirements, from which a monolithic supervisor has been synthesized with $1.84 \cdot 10^{129}$ states. The controlled behavior has been validated through model simulation.

Figure 6.1: Entrance of the Swalmen tunnel (adapted from `http://www.jr-consult.nl/projecten/swalmentunnel-a73-32`).

This supervisor is the starting point for the controller derivation process described in Section 6.2. All models for both synthesis and simulation can be found in a repository[1]. Since 2020, a publicly available software toolset is in development called the Eclipse Supervisory Control Engineering Toolset (ESCET™)[2]. The modeling and synthesis steps for the Swalmen tunnel have been performed using this toolset.

6.2 Controller derivation

In this section, several novel aspects of the controller derivation process are described in detail that showed to be relevant in the Swalmen tunnel case study, including the design of resource controllers, a relaxation in the sufficient conditions for the confluence property, PLC code generation, and PLC code optimization.

6.2.1 Resource controllers

In SBE, requirements are specified that define safe and correct behavior of the system. These requirements are typically logic expressions that define that a certain component is allowed or prohibited to do something when another component is in a certain state. A synthesized supervisor is guaranteed to adhere to these requirements.

Furthermore, there are requirements related to the behavior of a specific component. These requirements are typically related to physical or environmental

[1]`https://github.com/LMoormann/Swalmentunnel-WODES22`
[2]`https://projects.eclipse.org/projects/technology.escet`

aspects of the component. It is undesirable to capture these requirements in the synthesized supervisor, as they relate to a specific component, and would therefore make the plant model and requirements model unnecessarily complicated. Instead, additional local controllers are designed that comply with these physical requirements, which are called resource controllers. These resource controllers are implemented between the supervisory controller and the plant, as can be seen in Section 1.1, Figure 1.3.

Using resource controllers brings several advantages for the supervisory controller design process. First, as mentioned, resource controllers contain the requirements related to a specific component. By splitting up the components and requirements related to a specific component and the components and requirements related to supervisory control, both the resource controllers and the supervisory controller become smaller and thus more clear. Second, a benefit of smaller controllers is that the synthesis procedure to create such a controller is less computationally intensive. Note the similarity between the layers in the control system, i.e. the supervisory controller and the resource controllers, and the layers in a dependency graph introduced in Subsection 5.1.2. When determining whether part of the control system can be derived as a separate controller, a similar reasoning can be applied as is done in the independent vertices model reduction step explained in Subsection 5.2.1. This depends on the dependencies between the control layers, as when this is a one-way directed dependency a separate resource controller can be derived. This is the case when the supervisory controller only sends output signals to the resource controller and no input signal is retrieved, or vice versa. Conversely, when there exists a cyclic dependency between the control layers, synthesis techniques must be applied to the combined system to determine global nonblockingness.

Two examples of resource controllers are given here that are designed for the Swalmen tunnel. The first is related to the ventilation units in the tunnel. These units can ventilate in several modes depending on the current air quality, and in case of emergency they must be set to their maximum mode. There are two reasons why this discrete-event change cannot be realized. First, such a sudden change in the mode of all ventilation units requires too much energy, resulting in a power failure. Second, the forces generated by a ventilation unit that switches to its maximum mode result in failing fixtures, and thus in breaking down of the ventilation units. To deal with this, a resource controller is designed that takes the output signal of the supervisory controller, and converts it to a gradual signal for the ventilation unit. This is visualized in Figure 6.2.

A second example of a resource controller for a road tunnel is related to the sound broadcast system. There are several sound broadcast systems in the tunnel that can play various messages for the road users, mainly used when people need to be evacuated. In case of an evacuation, the supervisory controller will output a signal to all broadcast systems to play the evacuation message. However, to prevent overlapping messages or echoing in the traffic tubes, the broadcast

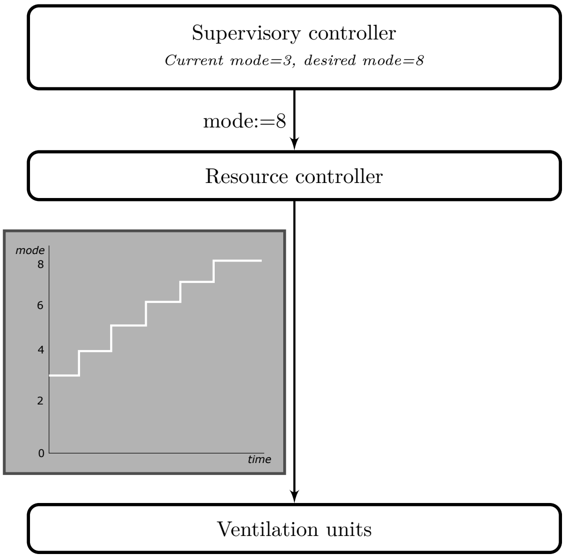


Figure 6.2: Schematic overview of a resource controller.

systems need to play their messages in sequence. A resource controller is therefore designed to realize this physical requirement.

For the Swalmen tunnel, resource controllers are designed for the ventilation, lighting, sound broadcast system, and CCTV.

6.2.2 Implementable supervisor properties

As described in Subsection 2.5.2, it is important to determine that a synthesized controller is implementable by verifying finite response, confluence, and nonblocking under control. For the synthesized controller of the Swalmen tunnel, these properties are checked using the sufficient conditions defined in Reijnen et al. (2019a). The checks work pairwise, meaning a check is performed for each property and for each pair of events in the supervisor, and if all event pairs pass the checks, the supervisor is implementable.

The finite response check and the nonblocking under control check, as described in Subsection 2.5.2, have been applied to the Swalmen tunnel model. These checks indicated that the Swalmen tunnel model has finite response and is nonblocking under control.

When performing the confluence check for the Swalmen tunnel model, certain event pairs failed. In-depth investigation, however, showed that these event pairs are in fact confluent, thus indicating that the sufficient conditions are in this case too strict. An example of a pattern where confluence is not recognized is shown in Figure 6.3. Here, it can be seen that in state 1 multiple controllable events are enabled, being `a` and `b`. The event pair is confluent, as after executing either event, the same end-state is reached, being state 5. The pattern of Figure 6.3, among others, is not recognized as confluent using the implemented algorithms of Reijnen et al. (2019a), so a relaxation is made to the sufficient conditions.

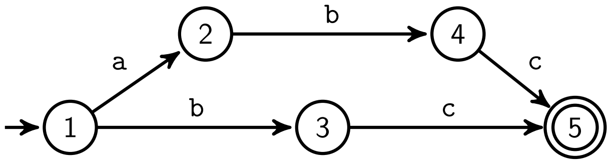


Figure 6.3: Pattern in the controlled behavior where confluence is not recognized.

End-state equality is proposed as a relaxation of the sufficient condition for confluence. End-state equality is an effectively verifiable property, as opposed to the confluence property. This end-state equality condition evaluates for every event pair (e_1,e_2) if the corresponding event sequences s_{e_1} and s_{e_2} eventually result in the same end-state. I.e., the end-states of two event sequences are compared while not requiring the events in the event sequences to be the same. It is relevant for components that have multiple event sequences to the same state, such as a lamp with two possible brightness modes: it does not matter if the lamp is set from mode $0 \rightarrow 2$ directly, or through modes $0 \rightarrow 1 \rightarrow 2$, as long as it happens in the same PLC cycle. In the case of the Swalmen tunnel, these cases occurred for the ventilation systems and the lighting systems, as these systems have 9 possible modes which can sometimes be reached through multiple event sequences.

Definition 1: End-state equality
An event pair e_1, $e_2 \in E_c$, with guards g_1 and g_2 and updates u_1 and u_2, respectively, is said to be end-state equal if there exist event sequences s_1, $s_2 \in E_c^*$ such that $\delta(q, e_1 s_1) = \delta(q, e_2 s_2)$, where at each event in s_1, s_2 the corresponding guard of that event must evaluate to `True`*, and at each event in s_1, s_2 the corresponding update of that event must be applied.*

An algorithm has been implemented that verifies end-state equality for all event pairs that are not recognized as confluent by any of the existing sufficient conditions of Reijnen et al. (2019a). Two notes should be made regarding this implementation. First, end-state equality can only be checked effectively if finite response has been determined beforehand. In case of an infinite sequence of events, the end-state equality algorithm may not terminate. Second, the algorithm still works in the case where there are multiple event pairs in the pattern, as each event pair is checked individually.

The proposed algorithm to check end-state equality is defined in Algorithm 1. For this algorithm controllable event pair (e_1, e_2) is supplied. First, a list of *enabledEvents* is created in line 1 that consists of event e_1. Then a while loop is started in line 2 that runs as long as *enabledEvents* is not empty. In each iteration of the while loop, the first event in the list of *enabledEvents* is chosen in line 3, and the valuation is updated using the update of that event in line 5. Subsequently, a new list of *enabledEvents* is created by iterating through all controllable events in line 6, and if its guard is satisfied (line 8) it is added to the list of *enabledEvents*. As long as at least one event is enabled, i.e. the list *enabledEvents* is not empty, the while loop in line 2 continues. Once the while loop is terminated, *evalE1* denotes the end-state evaluation of event e_1.

Subsequently, in lines 13 through 24, the same process is performed for controllable event e_2 to determine its end-state evaluation. Finally, in line 25, the end-state evaluations of events e_1 and e_2 are compared to return whether they are end-state equal.

The algorithm for the proposed end-state equality check is included in the existing confluence check described in Reijnen et al. (2019a). The adapted form of the algorithm of Reijnen et al. (2019a) is shown in Algorithm 2. Here, the end-state equality check is performed in line 8 after the existing confluence checks have been performed. The end-state equality check is performed last, as this check is needed the least and is the most computationally intensive.

6.2.3 PLC code generation

To deploy the synthesized supervisor as a controller for a physical system, the model has to be converted to a PLC program. Conversion is too costly and too error-prone to do by hand, so an automatic conversion has been developed instead. In Reijnen et al. (2019a), an implementation of a PLC code generator is proposed that used the normalization of EFAs as described in Subsection 2.5.3. A new PLC code generation algorithm has been developed and implemented, inspired by Reijnen et al. (2019a), that is more easily adaptable for multiple target platforms. Several requirements for this conversion are defined.

The first requirement is that the program must be semantically equivalent to the supervisor model. This means that the program is complete, i.e. it implements everything that is possible in the supervisor model, and that the program is correct, i.e. anything that is disallowed in the supervisor model is also not possible in the program. The concept of semantic equivalence is extensively discussed in Forssell et al. (2020).

A second requirement is that the resulting PLC code is readable to a PLC field engineer diagnosing a problem in the physical system. To get such readable code at global level, the structure of the model is reflected in the global structure of the generated code. Location pointer variables are added to represent the current active state of the model, and events are heuristically ‘assigned’ to an automaton, as is also suggested in Reijnen et al. (2019a), thus making it more likely that closely related events are grouped close to each other. For generating event

Algorithm 1: Code of the *end-state equal()* function

```
Input: Event pair ($e_1$, $e_2$), controllable event set $E_c$, valuation set $V$,
       update set $U$, guard set $G$
Output: True indicates that event pair ($e_1$, $e_2$) is end-state equivalent
// Find event sequence and end-state for event $e_1$
$enabledEvents$ := list($e_1$)
while !$enabledEvents.isEmpty$ do
    $enabledEvent$ := $enabledEvents(0)$
    $enabledEvents$ := list()
    $evalE1$ := updateValuation($enabledEvent$)
    for $eventNext \in E_c$ do
        $guardEventNext$ := getGuard($eventNext$)
        if guardSatisfied($evalE1, guardEventNext$) then
            $enabledEvents$.add($eventNext$)
        end
    end
end
// Find event sequence and end-state for event $e_2$
$enabledEvents$ := list($e_2$)
while !$enabledEvents.isEmpty$ do
    $enabledEvent$ := $enabledEvents(0)$
    $enabledEvents$ := list()
    $evalE2$ := updateValuation($enabledEvent$)
    for $eventNext \in E_c$ do
        $guardEventNext$ := getGuard($eventNext$)
        if guardSatisfied($evalE2, guardEventNext$) then
            $enabledEvents$.add($eventNext$)
        end
    end
end
// Return whether the end-state valuation of $e_1$ and $e_2$ are
   equal
return $evalE1$.equals($evalE2$)
```

Algorithm 2: Adapted confluence check

```
Input: EFA system $\mathcal{E}$, variable set $X_{\mathcal{E}}$, valuation set $V$, and controllable
       set $E_c$
Output: True indicates that $\mathcal{E}$ is confluent
1  Transform $\mathcal{E}$ into its NLEFA representation, to obtain for each $e \in E_c$ its
     global guard $g_e$ and global update $u_e$.
2  for $e_1, e_2 \in E_c : e_1 \neq e_2$ do
3      for $v \in V : v \models g_1$ and $v \models g_2$ do
4          if $\neg$ update equivalent$(e_1, e_2, v)$ $\wedge$
5            $\neg$ independent$(e_1, e_2, v)$ $\wedge$
6            $\neg$ (skippable$(e_1, e_2, v)$ $\vee$ skippable$(e_2, e_1, v)$) $\wedge$
7            $\neg$ (reversible$(e_1, e_2, v)$ $\vee$ reversible$(e_2, e_1, v)$) then
8              if $\neg$ end-state equal$(e_1, e_2, v)$ then
9                  return False
10             end
11         end
12     end
13 end
14 return True
```

functions in PLC code that implement model transitions, the model specification structure is followed, and an event function is generated for each event associated with an automaton in the model. Under the assumption that the model designer keeps related automata close to each other in the model, this further enhances grouping of related events.

An event function first tries to find an enabled edge for the event in every participating automaton (one edge at a time as locations are not merged). If that succeeds, the edge is taken in the event function. The updates associated with the enabled edges are performed and location pointer variables are updated, thus updating the variables of the controller. Finally, the function reports whether or not an event was performed.

The global control program that represents the executing model is performed in each Execute step of the PLC, as seen in Subsection 2.5.1, Figure 2.13. It starts by storing the new sensor information from the preceding Input step in model variables and updating continuous variables for the passed time since the previous cycle. It then repeatedly calls all event functions until none of them reports a performed event. Note that this is semantically sound due to the confluence and finite response properties previously proven for the model. The final step in the execution is to write the new program state to the actuator outputs in the Output step of the PLC cycle that immediately follows the Execute step. The actuator outputs are then forwarded to the physical system.

A third requirement is to support multiple PLC types. While IEC 61131-3, see International Electrotechnical Commission (2013), is an international standard for PLC programming, commercially available PLC systems do not always follow it in the same way. As a result, generated code is slightly different for different PLC platforms.

A fourth requirement is good performance. Semantically, performing an event means making a new state with a full copy of the current state, selecting an enabled edge in each participating automaton, performing all updates of all selected edges, writing the results in the new state, and finally copying the new state back to the original state. The CIF modeling language ensures there is no overlap in written data, and since reading and writing happens in disjunct data, original values are available until all updates are performed. A practically better performing solution is implemented here, where no full copies are made of the state, but instead the state is directly updated by writing computed updates back to it. This trivially works if read data and written data are disjoint. In the case of overlap, problems are solved either by ordering updates such that the original value is destroyed after its last read, or by creating a temporary copy to break the dependency cycle. In practice, the need to create a temporary copy is rare.

Last but not least, the implementation of a PLC code generator of Reijnen et al. (2019a) uses a template-like approach to generate code. One of the remarks was that while the resulting code is semantically correct, it is not good enough for readability. PLC code is not optimal, for example code like `BOOLEAN b := TRUE; IF b THEN ... END_IF` is generated where `b` never becomes `FALSE`. The conclusion was that template-based code generation is too crude to get better quality, as it cannot easily handle all the subtle different cases that may happen. Instead, an approach with an intermediate model of the generated code, which is a metamodel in the code generation algorithm, is used to allow additional analysis and fine-tuning of the result. Using model transformations, non-optimal code is caught and fixed, much like how advanced code generation for a compiler works, as described in Cooper and Torczon (2011). This approach enables splitting the code generation process in multiple independent stages, where one can perform model transformations not only for catching and fixing non-optimal code, but also for transforming code towards a specific PLC target, and fixing clashes in read and write patterns of variables.

PLC code of the Swalmen tunnel is generated for both the supervisory controller and the resource controllers, consisting of 18853 and 16496 lines of code, respectively. Both PLC code generation processes take around one second.

6.2.4 PLC code optimization

The performance of a controller that is implemented on a PLC is measured by the PLC cycle time, which mostly depends on the time it takes to execute the PLC code (Subsection 2.5.1, Figure 2.13). To improve the performance and readability of the PLC code, the order in which events are considered and executed in the PLC code can be optimized. This optimal order is determined using event order sequencing. Take for example a lamp that needs to be turned on when a button is pushed. The PLC determines, based on the input signal of the button, if the button is pushed, and it determines if the lamp should be turned on or off. The optimal order is to first evaluate the button and then the lamp, as the state of the button influences the evaluation of the lamp.

When (part of) the PLC code is repeated in a PLC cycle, this is called rework. This often occurs when the event order is not optimized, but it is sometimes unavoidable even in the optimized code.

To optimize the order of events, the component relations are gathered in a process dependency structure matrix (DSM), which was introduced in Eppinger and Browning (2012). An example of a process DSM is shown in Figure 6.4. The axes of the DSM are labeled by the components (A-E in this example). The order of these components indicates the order in which the components are evaluated in a PLC cycle. A dependency between two components is indicated in the DSM by a gray box. The dependencies are based on the requirements in the model, following the method shown in Goorden et al. (2019a). For instance, the requirement

$$\texttt{Boombarrier.c\_close}\ \textbf{needs}\ \texttt{TrafficLight.red} \tag{6.1}$$

indicates that the boom barrier component depends on the traffic light component. In the DSM, this dependency is noted by a gray box in the row of the boom barrier and the column of the traffic light following the IR/FAD (Input shown in Rows/Feedback is Above Diagonal) notation convention.

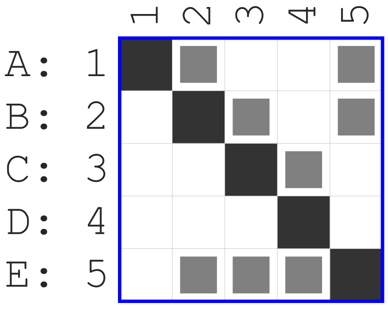


Figure 6.4: Directed DSM

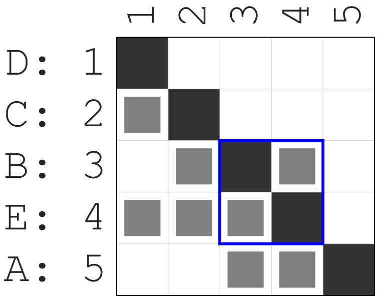


Figure 6.5: Sequenced DSM

A dependency in the top right triangle is a feedback element, since information about the state of a component later in the order is required. In Figure 6.4, there exists a feedback element between components A and E, meaning that in the worst-case scenario the entire PLC code must be repeated. This repeat loop is indicated by the blue box.

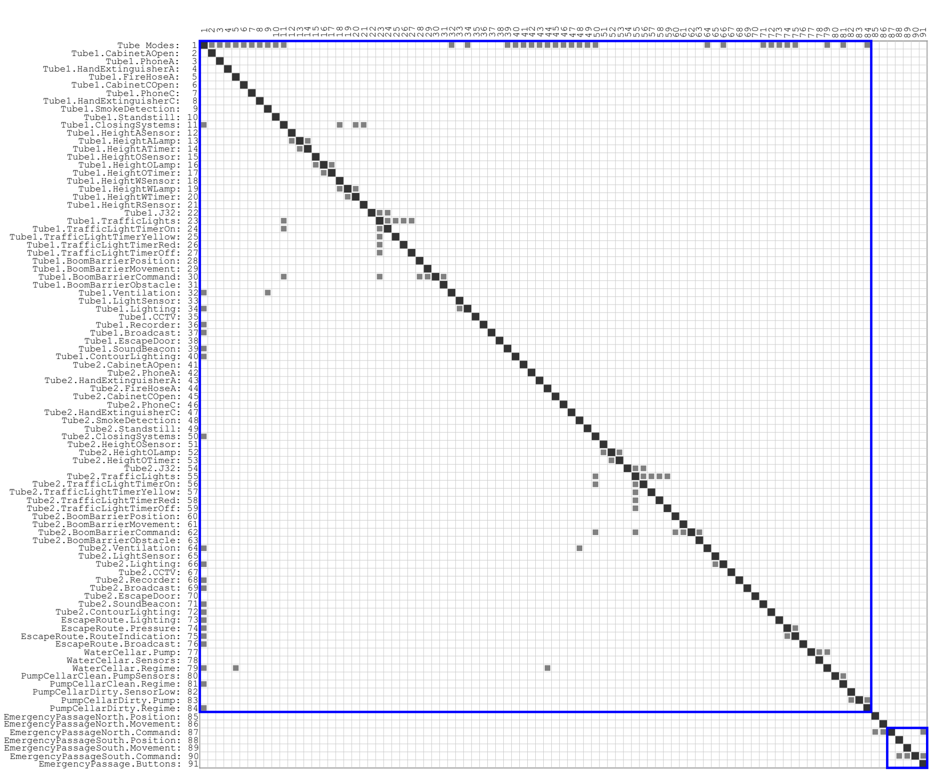


Figure 6.6: Unsequenced DSM of the Swalmen tunnel.

To optimize the order of events, the number of feedback elements is minimized. This is done by sequencing the DSM, e.g. using the algorithm from Meier et al. (2007), which is an automatic process where the components on the axes are reordered to try to put as many dependencies as possible in the lower left triangle. The sequenced DSM can be seen in Figure 6.5. As can be seen by the smaller blue box, the worst-case rework loop now only contains the PLC code of 2 components. Note that only the order of components is sequenced. Ordering of events in the same component is not considered here.

Event sequencing is currently being applied in the automatic PLC code generation program. In this program, the optimal event order is automatically determined using sequencing algorithms, and this order is implemented in the PLC code manually. In the case of feedback loops, such as the blue box in Figure 6.5, a rework loop is added to ensure all enabled events are executed.

In the case study for the Swalmen tunnel, a DSM has been automatically generated from the supervisory controller. This DSM is shown in Figure 6.6 and consists of 91 components that are connected through 412 dependencies. The

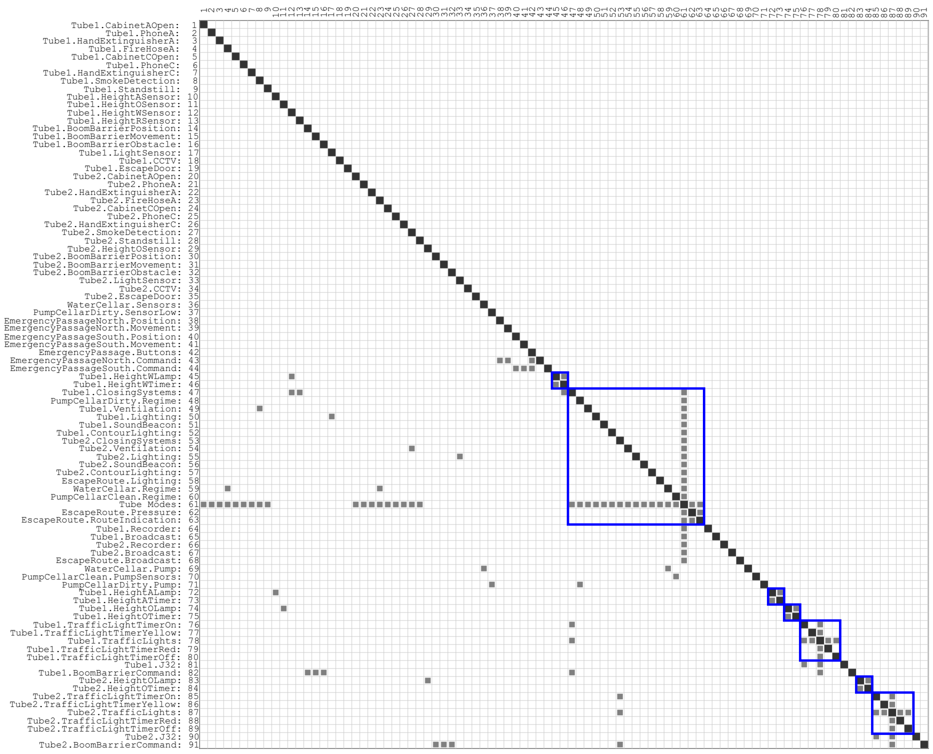


Figure 6.7: Sequenced DSM of the Swalmen tunnel.

component names are shown on the left-hand side. As is indicated by the blue boxes in the figure, the unsequenced DSM contains two feedback loops of 84 and 5 components, respectively, meaning that in certain cases almost the entire PLC code needs to be rerun.

This DSM is automatically sequenced using the algorithm described in Meier et al. (2007) to optimize the order of events in the PLC code. Figure 6.7 shows the sequenced version of the DSM. In the sequenced DSM, 7 feedback loops exist of 17 (once), 5 (twice), and 2 (four times) components. The amount of rework is thus decreased, as the size of the feedback loops in the sequenced DSM is much smaller than the size of the feedback loops in the original DSM.

Feedback loops are unavoidable in the sequenced DSM, as there are components with cyclic dependencies. For instance, the two feedback loops that consist of 5 components in Figure 6.7 contain the components of the timers and the lamps of the traffic lights. There exists a cyclic dependency between these components, since the timers must turn on when the traffic light shows a certain color, and the traffic light is allowed to switch to the next color once the timers has run out.

The feedback loop is thus unavoidable, but capturing it in a dedicated `WHILE` loop ensures that only the PLC code related to these traffic lights is rerun.

6.3 Hardware-in-the-loop testing

As described in Section 2.1, the second testing stage in SBE is HIL simulation, where the controller is implemented on the PLC, and combined with the hybrid plant model to perform simulations with visualized interfaces. The first benefit of HIL simulation, compared to model simulation, is that the PLC operating semantics is used. Second, external subsystems like the operator interface can be connected to the PLC. Third, the actual input and output signals of the PLC are connected to the system model and the operator interface. Finally, PLC cycle times can be measured to determine if the PLC performance is sufficiently high. These combined benefits allow the design engineer to perform HIL simulations as if he were operating the real system.

A typical HIL setup consists of three layers, as shown on the right-hand side of Figure 6.8. The controller, implemented on a PLC, is connected to two PCs. PC 1 contains the operator interface, which is used to send commands to the PLC, and read the status of the PLC and the system. PC 2 contains the system model, which receives actuations from the PLC and returns sensor measurements. A third PC, the programming PC, is used to implement the code on the PLC, as is shown on the left-hand side of Figure 6.8.

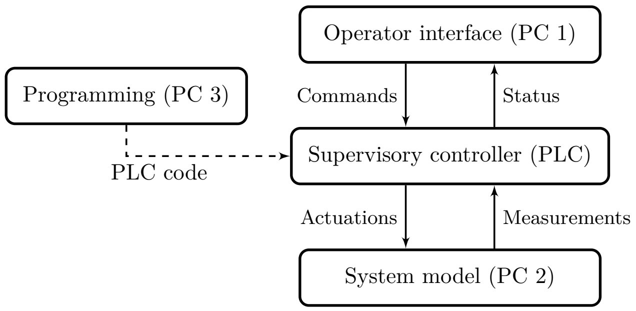


Figure 6.8: Overview of a HIL test setup, adapted from Reijnen et al. (2019a).

The inputs and outputs of the operator interface and the system model need to be connected to the PLC controller, for which a hardware mapping is created. A hardware mapping is an enriched version of the plant model where guards are added to sensor events to connect them to the input signals of the PLC, and updates are added to actuator events to connect them to the output signals of the PLC. The actuator shown on the left-hand side in Figure 6.9 includes the hardware mapping with the output signal of that actuator. As can be seen, output variable Q is set to **True** or **False** when the actuator is turned on or off, respectively. The sensor on the right-hand side of Figure 6.9 also includes a hardware mapping. Here, an input Boolean I is declared, and the hardware mapping defines that event u_on or u_off is enabled only when I is **True** or **False**.

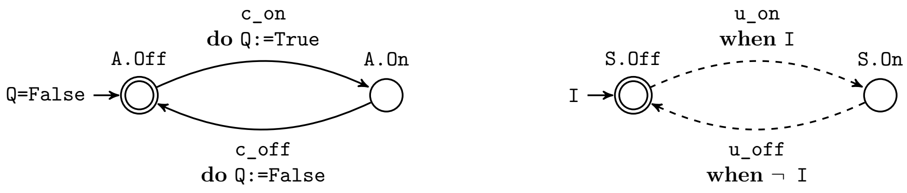


Figure 6.9: Automata for an actuator A (left) and a sensor S (right) including their hardware mappings.

Both the operator interface and the system model are implemented on a supervisory control and data acquisition (SCADA) system, described in Boyer (2009). Here, Ignition SCADA software is used. In this SCADA system, visualizations of the operator interface and of the system model are created, similar to the SVG visualizations used in model simulation. Furthermore, the input and output signals of the PLC controller are connected to the object properties in the SCADA models, e.g. an output signal of the PLC to turn on a lamp is connected to the visualization in the SCADA model of that lamp to show when it is turned on by the PLC.

HIL simulation is used to validate the behavior of the implemented controller of the Swalmen tunnel. To incorporate the resource controllers described in Subsection 6.2.1 in this setup, the schematic overview shown in Figure 6.8 is extended by an additional PLC, as depicted in Figure 6.10.

In Figure 6.11, the physical HIL setup is shown containing the same components as shown in the schematic overview. The PLCs in the HIL setup are of type ABB AC800M PM866A, which are the same PLCs as are used in the Swalmen tunnel.

The visualizations of the operator interface and of the system model can be seen in Figure 6.11. These are created using the Ignition SCADA software and can communicate with the I/O of the PLCs through an OPC UA server. Furthermore, all system behavior that is relevant for the purpose of controller validation is programmed in the behavior scripts of the system model. Examples in the Swalmen tunnel case study are the movement of boom barriers, the water level in the pump cellars, and the smoke and light level in the traffic tubes.

Using the HIL simulations, the controlled behavior of the implemented controller of the Swalmen tunnel has been validated by running through test scenarios provided by RWS and observing the system model and the operator interface.

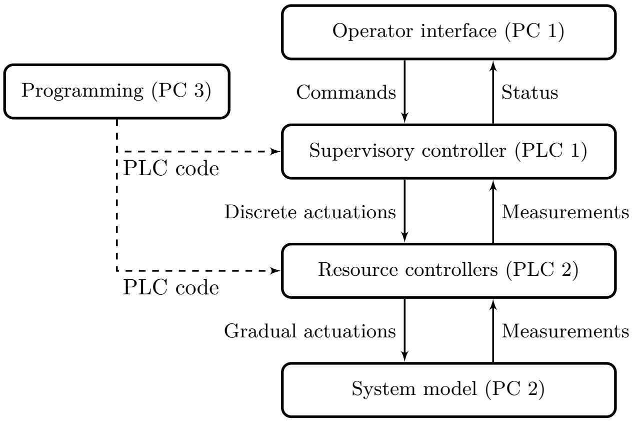


Figure 6.10: Schematic overview of a HIL test setup, extended with resource controllers.

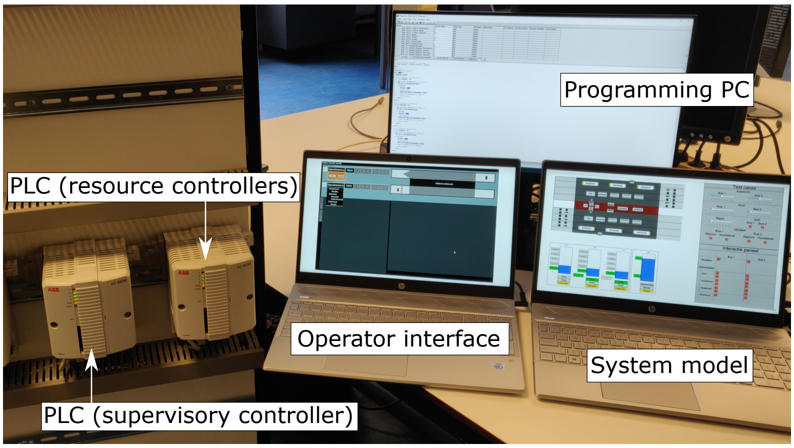


Figure 6.11: Overview of the physical HIL setup.

In all cases, the controlled behavior was immediately as expected, i.e. in accordance with the behavior defined in the test scenarios, thus meaning that the implemented controller works as desired.

6.4 Concluding remarks

In this chapter, the process of deriving and implementing a supervisory controller is described and applied in a real-life case study for the Swalmen tunnel.

A road tunnel supervisory controller is needed to maintain a safe environment in the tunnel. While this is mainly a logic controller that specifies which output signals must be high or low depending on the input signals, certain controller requirements are related to the physical or environmental behavior of a specific component. To comply with these requirements, resource controllers are designed. In the Swalmen tunnel case, 4 resource controllers are designed.

In the process of deriving an implementable controller, the required properties of such a controller are verified. In the case of the supervisory controller of the Swalmen tunnel, although present, confluence could not be determined using existing confluence checks. Therefore, a relaxation in the sufficient conditions for confluence is proposed called end-state equality, which is used to determine confluence of the Swalmen tunnel supervisory controller. An algorithm is defined and implemented that can be used to check end-state equality, and it is incorporated in the existing confluence check of Reijnen et al. (2019a).

From the supervisory controller, PLC code is automatically generated using a new PLC code generator. This generator is set up to be easily adaptable for new hardware platforms by only having to adapt its back-end. For the Swalmen tunnel, PLC code is generated for PLCs of the ABB brand.

Subsequently, the generated PLC code is improved in terms of performance and readability by optimizing the event execution order. For this purpose, DSMs are used and automatically sequenced to obtain the optimal order. In the case study, the possible amount of rework is reduced from 81 components to 17 components.

The behavior of the implemented supervisory controller is validated using HIL simulations, which allow for validation while using the actual operating semantics, external subsystems, and operator interface. The Swalmen tunnel controller is validated by running through scenarios, and in all cases the controlled behavior was immediately as expected.

Future work includes the implementation of the sequencing algorithm in the PLC code generator to automate applying the obtained event order to the generated PLC code.

Furthermore, steps towards realization can be taken in the HIL setup by replacing part of the virtual system model by the physical components that will be put in the realized tunnel. This way, the implemented controller can be tested with each physical component before realization of the complete tunnel.

Chapter 7

Distributed supervisory controllers

As described in the previous chapter, synthesized supervisors can be implemented on a physical platform such as a PLC. However, there are multiple advantages to implementing a supervisory controller on multiple PLCs. A system is then not controlled by a single global controller, but by multiple local controllers that are able to communicate with one another. Such a distributed implementation can reduce PLC cycle times, as the computing power of the combined control system is increased and the size of each individual controller can be reduced. Furthermore, distributing a supervisory controller can increase the maintainability, reliability, and availability of the control system, and can decrease the required length of cables to connect to the plant. In a distributed implementation, however, communication is required, which can introduce delays and increase the required memory and computation power compared to a centralized implementation. The main challenge in dealing with communication delays is ensuring correct controlled behavior when the local controllers are not synchronized, meaning that the state of a local controller can change before a message arrives.

In this chapter, a method is described to obtain a distributed supervisor that ensures correct behavior regardless of communication delays. This method is based on several existing techniques previously presented in literature and a new technique is proposed to determine whether a controller is robust to communication delays. Furthermore, extensions are made to existing techniques

This chapter is based on: Moormann, L., Schouten, R.H.J., van de Mortel-Fronczak, J.M., Fokkink, W.J., and Rooda, J.E. Synthesis and implementation of distributed supervisory controllers with communication delays. *Transactions on Automation Science and Engineering.* Submitted.

in several places to make the method applicable in an industrial setting. The method consists of the following steps. First, the plant is modeled using automata and the desired controlled behavior is defined in requirements, as described in Cassandras and Lafortune (2008). Second, a centralized supervisor is synthesized using any algorithm of choice, e.g. the algorithm of Ouedraogo et al. (2011). Third, the distribution of the system is performed using DSMs (Eppinger and Browning (2012)) to create clusters of components that have minimal communication between them. Fourth, for each cluster a local supervisor is derived, by localizing the global supervisor using the theory of Cai and Wonham (2010). Fifth, a new property, called delay robustness, is proposed allowing to estimate the effects of communication delays, and to determine in which cases these effects negatively impact the controlled behavior. In a delay robust system, communication delays do ultimately not affect the system's behavior. Finally, a mutual exclusion algorithm is used to counter the negative effects in the case that a system is not delay robust. A typical tunnel in the Netherlands is controlled by up to 10 PLCs, which underscores the industrial relevance of a distributed supervisory controller.

Besides describing the complete method, this chapter provides additional results regarding certain steps in the method. First, in the distribution step of the method, a parameter study is performed to optimize the clustering of DSMs. Second, the theorem for delay robustness of a distributed supervisor is formally introduced and substantiated, and a formal proof of the theorem is included. Third, elaborate implementation tests are performed to analyze the performance of the implemented controller, both in a monolithic and in a distributed setting. Fourth, the implementation of the mutual exclusion algorithm is studied to improve the performance in the distributed setting. Finally, a large case study is provided, where a supervisor is synthesized, distributed, and implemented for the Swalmen tunnel, a road tunnel in the Netherlands.

Over the passed years, distributed supervisory control has been researched extensively. In Wang et al. (2020), distributed supervisory control is considered where multiple controllers observe the same plant, but each of them can only observe a subset of transitions. In this setting, there is no communication between the controllers. Distributed control is also investigated in Zhang and Cai (2016) and Cai and Wonham (2010), where supervisor localization is used to obtain local controllers that communicate. We use the same approach of localization, and add communication delays between the controllers. In Kalyon et al. (2013), distributed supervisory control is investigated using an online state estimation approach, though no communication delays are included and no global nonblockingness is guaranteed. Su (2013) and Wong et al. (2000) provide research on conflict handling between local supervisors, using a global coordinator approach in Su (2013) and a priority-based conflict resolution approach in Wong et al. (2000), respectively.

The subject of communication delays in supervisory control has mainly been investigated in the context of networked systems, where delays occur in the communication between the controller and the plant. This differs from the distributed control setting that is investigated in this project, where delays occur in the communication between different controllers. Similar approaches in handling communication delay can, however, be applied. In Shu and Lin (2014), a supervisor is synthesized for a networked system with bounded communication delays where the language of the supervisor is extended with all possible event orders due to the delays. Such a supervisor does not always exist, so necessary and sufficient conditions are provided. Zhu et al. (2019) also considers the control of networked systems with delays in the communication between the controller and the plant. This communication is modeled using channels, which are used in our approach as well. Furthermore, Zhu et al. (2020) is an extension of Zhu et al. (2019) where non-FIFO (First In First Out) communication channels are considered. Alves et al. (2020) and Liu et al. (2021) also investigate delayed communication between the controller and plant, though their approaches differ from ours as they use assumptions on the maximal communication delay and state estimation, respectively, to predict the effect of control delays, respectively. To the best of our knowledge, none of the existing works on distributed supervisory control cover the system distribution process for the purpose of supervisor implementation. Furthermore, no existing literature applies their method to industrial systems of the size considered in this project.

This chapter is structured as follows. Relevant theory from literature on the topic of distributed supervisors is discussed in Section 7.1. Next, a running example is introduced in Section 7.2. An overview of the method proposed in this chapter is given in Section 7.3, and the two main steps are then discussed in Sections 7.4 and 7.5. Section 7.6 provides the results of a large case study on synthesizing and implementing a distributed supervisor for a road tunnel. Final remarks and conclusions are given in Section 7.7.

7.1 Existing theory on distributed supervisors

In this section, relevant theory related to distributed supervisors is discussed. In Subsection 7.1.1, existing work on obtaining distributed supervisors through synthesis is described. Subsection 7.1.2 introduces the process of supervisor localization, which is used to obtain a set of local supervisors from a global synthesized supervisor. Finally, in Subsection 7.1.3, mutual exclusion algorithms are described, which are used in the setting of distributed supervisors to prevent multiple supervisors from writing some critical data at the same time.

7.1.1 Synthesis of distributed supervisors

In literature, several contributions towards acquiring distributed supervisors through synthesis have been made. Su et al. (2010) introduces an aggregative method for synthesizing a distributed supervisor. There is no need for synthesizing a global supervisor (i.e. a supervisor for the entire plant and all its requirements) in this method, as local supervisors are synthesized right away. Computation times can therefore be relatively low. This is a great advantage in the scalability of the method. In order to achieve global nonblockingness, abstractions of other local supervisors are used during the synthesis of a local supervisor. Therefore, the order in which local supervisors are synthesized is important for both the computational complexity of the supervisor and the existence of a supervisor. Su et al. (2010) gives some guidelines to choose an efficient ordering, however, it is unclear how to find an optimal one. Moreover, the computations that are needed to obtain an efficient ordering are relatively complex.

In Komenda et al. (2016), an alternative algorithm called multilevel synthesis is introduced that can handle more complex synthesis problems. It uses a MultiLevel Discrete Event System (MLDES) which consists of a tree-based structure. For each node in the tree, a local supervisor is synthesized that influences a subset of the plant models and satisfies a subset of the requirements model. The set of local supervisors is safe and controllable, however, maximal permissiveness is only guaranteed under sufficient conditions, as described in Komenda et al. (2020), and a global nonblocking check is required. An MLDES consists of a number of supervisors that control a subset of components, which is beneficial for localization and distribution, as explained later in this chapter.

In contrast to the bottom-up approach of aggregate synthesis, supervisor localization proposed in Cai and Wonham (2010) is a top-down approach to acquiring local supervisors. First, a global supervisor for the entire plant is synthesized. A localized version of this global supervisor is then implemented for each component or component group. This method is a relatively simple way of obtaining a distributed supervisor. The behavior of this distributed supervisor is equal to that of the global supervisor. The concept of localization is further discussed in Subsection 7.1.2.

None of the aforementioned methods discuss the effects of communication delays between local supervisors, while these delays are unavoidable in a distributed implementation. Rashidinejad et al. (2018) introduces a synthesis method for (non-distributed) supervisors with a known delay on its inputs and outputs. The effects of communication delays can be similar for distributed supervisors, therefore Rashidinejad et al. (2018) gives some useful insights into this subject. In Kalyon et al. (2011), a distributed supervisor synthesis method is proposed which takes into account the effects of communication delays. The framework in Kalyon et al. (2011), however, assumes a fixed communication architecture, uses state estimates and abstractions, and requires direct communication between the local plants. This framework is therefore not suited for this project. Zhang et al. (2016) gives further insight into the effects of communication delays for distributed supervisors created by localization. This topic is further discussed in Subsection 7.5.2.

7.1.2 Localization

Supervisor localization describes the process of creating a set of local supervisors from a monolithic global one. Supervisor localization is introduced in Cai and Wonham (2010) for FA. It takes a global supervisor $\mathbf{SUP} = (X, E, \xi, x_0, X_m)$, which is defined for a global plant $G = (Y, E, \eta, y_0, Y_m)$. Plant G consists of component agents G_k defined over disjoint alphabets $E^k, k \in K$ with K an index set, where $E = \bigcup \left\{E^k \mid k \in K\right\}$. The global supervisor **SUP** is localized into local supervisors $\mathbf{LOC}_k$ for component agents G_k.

Each local supervisor $\mathbf{LOC}_k$ is a copy of the global supervisor **SUP**, where the controllable event set is adjusted to the events that can be controlled by that local supervisor $E_c^k = E_c \cap E^k$. All other events are uncontrollable in that local supervisor. Each controllable event is controllable in exactly one local supervisor. This is not a significant restriction, as controllable events are typically related to output signals of the controller, i.e. actuator signals, and it is undesirable that multiple local supervisors drive the same actuator. A local supervisor can observe events that are controlled by another local supervisor, called shared events. The uncontrollable events are, as usual, observed directly from the plant. In Cai and Wonham (2010), it is assumed no communication delays occur.

Using a monolithic global supervisor as a starting point for localization is often undesirable as localization has to be done for a relatively large supervisor. Instead, a multilevel supervisor can be used. A supervisor obtained by multilevel synthesis as described in Goorden et al. (2019a), consists of a number of supervisors that disable events in subsets of components. As an example, let plant G consist of 4 components G$_k$ ($k = 1, 2, 3, 4$), with a requirement model consisting of 8 requirements R$_i$ ($i = 1, 2, ..., 8$), for which multilevel synthesis is performed. The resulting MLDES tree is given in Figure 7.1. The global multilevel supervisor is given by $S = ||_i \ Supi$, for $1 \leq i \leq 7$. Note that $Sup6$ does not contain any requirements, as any requirements that refer to component G2 are already automatically placed in $Sup1$ and $Sup5$.

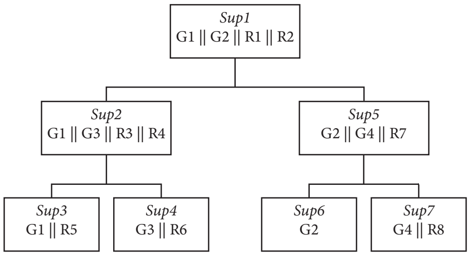


Figure 7.1: Multilevel supervisor example

The multilevel supervisor can now be localized for a certain distribution of component agents. If, for example, a distribution of two groups is chosen, where group 1 contains G1 and G3 and group 2 contains G2 and G4, it is not necessary to localize the entire multilevel supervisor. As group 1 is only controlled by supervisors 1 through 4, only these supervisors need to be localized. Local supervisor $\mathbf{LOC}_1$ is the set of localized supervisors $Sup1$, $Sup2$, $Sup3$ and $Sup4$. Similarly, local supervisor $\mathbf{LOC}_2$ is the set of localized supervisors $Sup1$, $Sup5$, $Sup6$ and $Sup7$. $\mathbf{LOC}_1$ now controls agents G1 and G3 while observing G2, whereas $\mathbf{LOC}_2$ controls agents G2 and G4 while observing G1.

7.1.3 Mutual exclusion algorithms

Mutual exclusion (mutex) algorithms pose a solution to the problem in concurrent programming where multiple processes try to enter a, so-called, 'critical section' simultaneously, while this is not allowed. In most applications, mutual exclusion algorithms aim for several properties, as discussed in Kanrar et al. (2018):

Safety No two processes can enter their critical section simultaneously.

Liveness A process that requests access to the critical section is eventually allowed to do so.

Fairness The first process to request access is granted access the first.

Many solutions have been proposed since the presentation of Dijkstra's algorithm in Dijkstra (1965). Some algorithms are able to prioritize one process over another, and are thus not fair. Most algorithms aim to reduce the amount of communication, i.e. the number of messages required to get access to a critical section.

Distributed mutex algorithms often follow a token-based approach. Token-based algorithms, as defined in Raynal (2013), use a mobile object, referred to as the token, which travels from process to process. The process in possession of the token is allowed to enter its critical section. As there is only one token, safety is ensured automatically.

Different network structures have been proposed, as discussed in Kanrar et al. (2018), in efforts to reduce the amount of communication. In permission-based algorithms, a process requests permission to enter its critical section from all other processes or from a subgroup of other processes called a quorum or coterie. Kanrar et al. (2018) states that generally token-based algorithms need less communication than permission-based algorithms. Permission-based algorithms are generally more suited for implementation of fairness or prioritization.

A simple implementation for a mutex algorithm that is safe, live, and fair, is a home-based token algorithm as discussed in Raynal (2013), for which an example is shown in Figure 7.2. Here, there exist one home process (1) and a number of other processes (2 and 3). The token normally resides at the home process, and any process can request the token (blue arrow). The process returns the token (red arrow) after it leaves the critical section. If a process requests the token while it is already in use by some other process, such as during the request of process 2 in the example, the requesting process will be added to a FIFO queue. The home process will always send the token to the first process in the queue. This algorithm requires little communication as each process only communicates with the home process, though it becomes inefficient for large numbers of processes and when the token is used for short amounts of time. One disadvantage is the single point of failure of the home process. This disadvantage, however, does not apply in this chapter, as hardware and communication failures are not considered. If, in future work, this assumption is dropped, other algorithms should be explored as well.

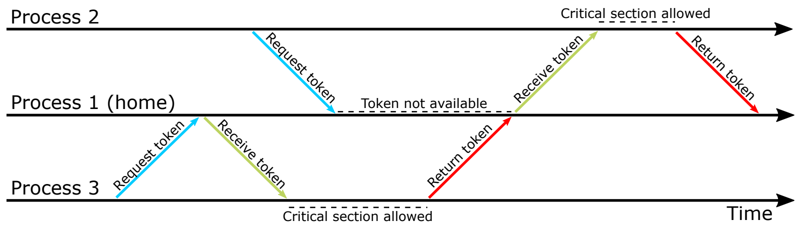


Figure 7.2: Home-based token passing mutual exclusion

7.2 Running example: Pump-cellar system

Throughout the upcoming sections of this chapter, a running example is used to show an application of the method discussed in Sections 7.4 and 7.5. The running example concerns a pump-cellar system which is used in road tunnels to collect rain water and leaked fluids. It is part of a running study on industrial applications, as described in Goorden et al. (2020). In this case, the system contains 3 cellars, each equipped with 2 pumps and 5 sensors, as depicted in Section 3.1, Figure 3.6. The sensors indicate the fluid level and the pumps are used to drain the cellar. There are 2 head pump cellars, located at the two entrances of the tunnel, and a middle pump cellar that can pump fluids to either of the 2 head pump cellars.

Each pump cellar can operate in one of the three modes: `Store`, `Drain` and `Off`. Based on the mode, the pumps should be on or off at certain fluid levels. The mode of the pump cellar depends on the mode of the tunnel, which can be `Operational`, `Emergency`, and `Recovery`. For example, during an emergency the pump-cellar mode should be `Store` so no dangerous leaked fluids are pumped into nearby rivers. The uncontrolled behavior of the mode of the pump cellars and the mode of the tunnel are modeled using FAs, as shown in Figures 7.3 and 7.4.

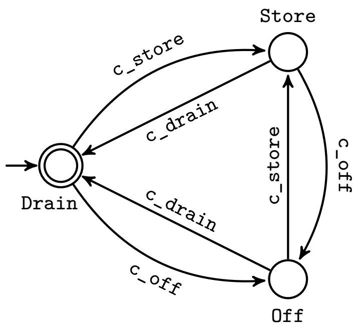


Figure 7.3: Mode automaton

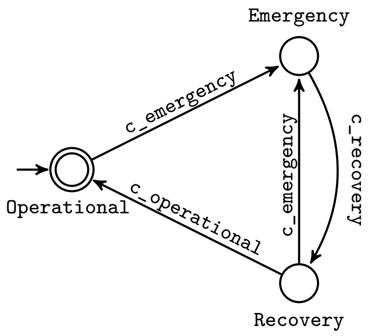


Figure 7.4: Traffic tube automaton

The uncontrolled behavior of the pumps (Figure 7.5), the sensors (Figure 7.6), the pump direction (Figure 7.7), and the buttons (Figure 7.8) are also modeled using FAs. The complete plant model is available in a repository[1].

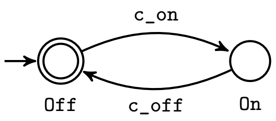


Figure 7.5: Pump automaton.

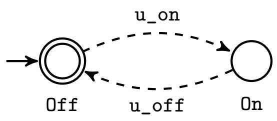


Figure 7.6: Sensor automaton.

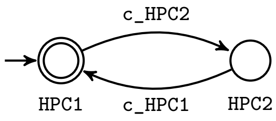


Figure 7.7: Direction automaton.

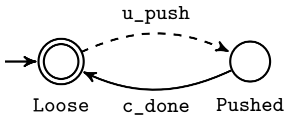


Figure 7.8: Button automaton.

A set of requirements is created to describe what behavior the controller should allow. For each pump cellar, a number of requirements is defined to state when the pumps are allowed to turn on or off based its mode. When the mode is `Drain`, the pumps are turned on when sensors `S1` and `S2` are on, while in mode `Store` they are turned on when sensor `S5` is on. In the `Off` mode, the pumps are off.

The requirements are stated as event conditions. Some examples of such requirements are shown in Equations (7.1)-(7.3). Note that these requirements

[1]`https://github.com/LMoormann/Distributed_Pumpcellars`

can also be formulated as EFAs.

$$\textbf{requirement}\ \texttt{Pump1.c\_on}\ \textbf{needs not}\ \texttt{Mode.Off} \tag{7.1}$$
$$\textbf{requirement}\ \texttt{Mode.c\_store}\ \textbf{needs}\ \texttt{ButtonStore.Pushed} \tag{7.2}$$
$$\textbf{requirement}\ \texttt{Direction.c\_HPC2}\ \textbf{needs}\ \texttt{HPC1\_S5.On} \tag{7.3}$$

Requirement (7.1) states that event `P1.c_on` is only enabled if the `Mode` automaton in not in the state `Off`. The complete collection of the requirements is available in the repository[1].

7.3 Method - Obtaining distributed supervisors

In this chapter, a method is described for obtaining a distributed supervisor, based on several existing techniques, while taking into account communication delays. This method is an extension on the SBE process, as is shown in Figure 7.9. The additional steps are shown in blue.

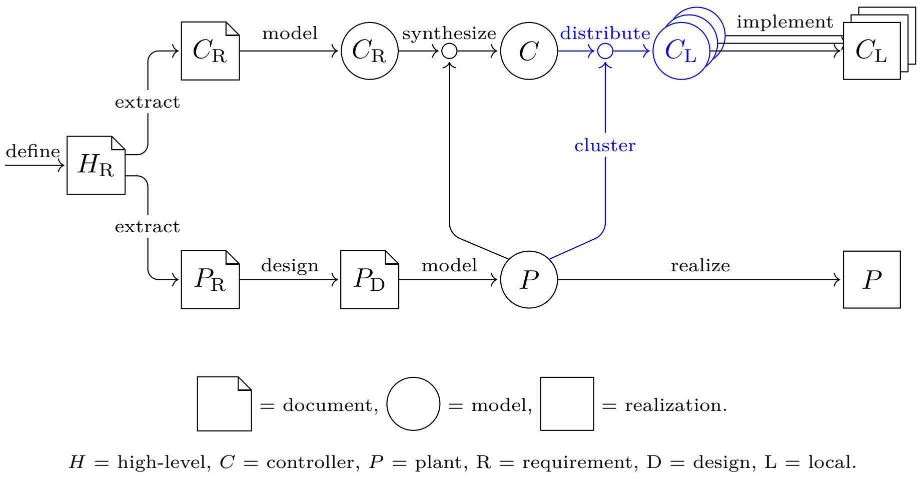


Figure 7.9: Schematic overview of the distributed SBE process.

The first additional step is obtaining a distribution of the system, which is performed by clustering the plant components P in Figure 7.9. Here, dependencies between the plant components are analyzed using DSM techniques (Eppinger and Browning (2012)) to obtain a system distribution where the number of dependencies between clusters is minimized. This step is explained in more detail in Section 7.4.

The second additional step is distributing the synthesized supervisor. This is done by localizing the global supervisor to obtain a set of local supervisors using theory from Cai and Wonham (2010). Then, the shared events between the local supervisors are analyzed to determine whether they are delay robust. For events that are not delay robust, called delay critical, mutual exclusion algorithms are implemented to enforce delay robustness. Section 7.5 describes the supervisor distribution step in more detail.

7.4 Step 1: Distributing the system

The first step in acquiring a distributed supervisor is distributing the system by mapping the control dependencies between the components, and creating clusters of these components. For each cluster a local supervisor can then be derived. We assume full observability, meaning that every local supervisor can obtain all information from other local supervisors and plant components. Full observability is guaranteed in the type of systems considered in this chapter. If this assumption is disregarded, the distribution process is limited by the observability of the system. In that case, certain components are required to be part of the same local supervisor, as their events are not observable when they are part of separate supervisors. The method itself still works in this case.

7.4.1 Dependency mapping and component clustering

Dependency Stucture Matrices (DSM), as introduced by Eppinger and Browning (2012), provide a means to give insight into the dependencies within a process or system. In Subsection 6.2.4, DSMs are introduced for the purpose of PLC code optimization, in which the DSM is sequenced to obtain an optimal ordering of components. In the method discussed here, DSMs are used for the purpose of clustering. The initial DSM is the same, where the components of the system are the elements on both the horizontal and the vertical axis, and each entry in the matrix denotes a dependency between two components. Figure 7.10 shows an example of such a DSM.

The type of dependency that is used when creating the DSM varies per system and decomposition purpose. Some examples of dependencies that are used in DSMs given in Pimmler and Eppinger (1994) are physical connections, information flow, material flow, or electrical flow. As mentioned in Eppinger and Browning (2012), physical connections, material flow, and electrical flow are typically used when decomposing hardware products, while information flow is often used in the case of software products. Depending on the type of dependency, the DSM can be symmetric or asymmetric. Physical dependencies will result in a symmetric DSM, since when A is connected to B, B is also connected to A, whereas information flow might result in an asymmetric DSM, since when A

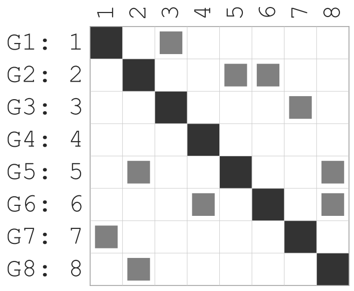


Figure 7.10: Example of a DSM.

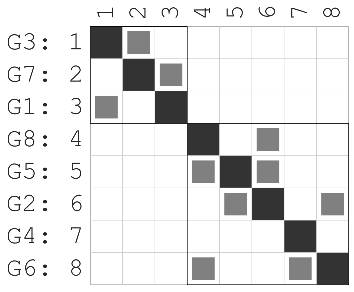


Figure 7.11: The clustered DSM.

provides information to B, B does not necessarily have to provide information to A. A physical relationship is therefore called an undirected relationship, while an information flow relationship would be called a directed relationship. The DSM shown in Figure 7.10 is a directed, asymmetric, DSM.

For the purpose of analyzing control relations in a DES, a DSM is created using the control requirements as dependencies, as shown in Goorden et al. (2019a). A plant model and a requirement model are needed for the creation of such a DSM. The components of the plant model are listed on the axes of the DSM. Through requirements, different components depend on each other. These dependencies are shown in the DSM. As these requirements are directional, the resulting DSM is a directed DSM.

The applied analysis method that is used here is clustering, which is used to obtain groups of closely related components. When clustering components in a DSM, the order in which the components are placed on the axes is changed and components are grouped. The aim is to identify clusters in such a way that the number of dependencies between components within a cluster is maximized and the number of dependencies between components of different clusters is minimized. This way clusters are obtained of closely connected components, and insight in the interaction between clusters is gained. In Figure 7.11, the clustered version of the DSM shown in Figure 7.10 is shown. Here, two clusters of respectively 3 and 5 components are shown.

In Wilschut (2018), Markov clustering is introduced, which is a relatively complex random walker algorithm that uses four clustering parameters α, β, μ and γ. α determines the number of jumps the random walker takes. β and μ are both used to tune the cluster size and the number of hierarchical levels. Parameter γ is a threshold value for the number of detected bus elements, increasing γ decreases the number of bus elements in the clustering. Markov clustering can create multiple hierarchical clusters and can detect bus elements. Bus elements are elements in the DSM that have a large number of dependencies. Wilschut (2018) shows that Markov clustering is a scalable and versatile solution for clustering

DSMs.

In the process of obtaining a distributed supervisory controller, a DSM is created for the MRPS by following the procedure provided in De Queiroz and Cury (2000). As the goal of the DSM is to decompose a software system, information flow is used to determine the dependencies between components, as proposed in Eppinger and Browning (2012). Following the approach of Goorden et al. (2019a), a DSM is created based on control relations, as this minimizes the amount of communication needed between local supervisors. Note that the dependency type used in the DSM is up to the control engineer, and that the proposed method works for any type of DSM based on any type of dependency.

To create a DSM based on control relations, first Domain Mapping Matrices (DMM) $P1$ and $P2$ are created, which are rectangular matrices, see Wilschut (2018). These DMMs show relations between plant components and requirements in the system. For example, the requirement in Equation (7.1) refers to the event `c_on` in component `Pump1` with a condition that depends on the `Off` location of component `Mode`. $P1_{i,j} = 1$ when requirement j refers to an event of component i and $P1_{i,j} = 0$ otherwise. $P2_{i,j} = 1$ when the condition of requirement j refers to an automaton of component i and $P2_{i,j} = 0$ otherwise. The DSM P can then be computed using $P = P1 \cdot P2^T$.

When clustering the DSM, control over a number of characteristics of the clustering is desired. Firstly, control over the number of created clusters is needed, as for each cluster a local supervisor is synthesized. Secondly, the size of the created clusters needs to be controlled as this determines the number of components that are controlled by each local supervisor. Lastly, the algorithm must be able to cluster systems with a large number of components and dependencies. Markov clustering is chosen, as it has been shown to be scalable. Furthermore, tuning of the clustering parameters gives control over the clustering characteristics as is elaborated in the next section.

7.4.2 Clustering parameter study

As explained in the previous section, 4 clustering parameters are used in Markov clustering to control the clustering characteristics. In this section, a parameter study is described to give insight in the relation between the clustering parameters and the desired properties of a distributed supervisor.

In the context of distributed supervisors, three aspects are analyzed when clustering the DSM. The first aspect is the number of required hardware platforms (PLCs) on which the controllers are implemented, which is determined by the number of clusters and the existence of a bus. For instance, a clustering with a bus and 2 clusters requires 3 PLCs, 1 for the bus components, and 1 for each cluster. The desired value is typically determined by the number of available PLCs.

The second aspect that is analyzed is the maximum cluster size, as this affects the maximum size of a local supervisor, and thus its performance. This aspect is therefore preferably minimized.

The third aspect is the number of dependencies outside of the clusters, as this determines the amount of communication that is needed between the controllers. Communication, and thus dependencies outside of clusters, should be minimized, as communication delays can introduce problems, and thus decrease performance.

Note that the clustering is only a suggestion for a good distribution that is obtained in a methodical way. If, for any reason, the control engineer desires to deviate from this clustering, this is possible. This includes combining clusters, splitting clusters, or moving components between clusters.

Coefficients α, β, μ and γ can be tuned to control these aspects. In this parameter study, the following values are used, as inspired by Wilschut (2018). The value of α has little influence on the clustering results, as noted in Van Dongen (2008), and is therefore kept constant.

$$\begin{aligned} \alpha &= 2 & \beta &= \{1.1, 2, 2.5, 3, 3.5\} \\ \mu &= \{1.5, 2, 2.5, 3, 3.5\} & \gamma &= \{1.5, 30\} \end{aligned} \tag{7.4}$$

For each combination, the aspects described above are analyzed, and the conclusions drawn are given here.

The combination of β and μ largely determines the number of clusters and dependencies between them. The best results are observed when μ is kept low (1.5), and β starts low (1.01). β should then slowly be increased until the desired number of clusters is obtained.

The γ value determines the existence of a bus. γ should thus be chosen low (~ 1.5) if a bus is desired, or high (> 25) when no bus is desired. Whether a bus is desired depends on the system. In the cases observed in this project, a bus is undesirable as it results in a large amount of communication.

It is observed that a clustering typically either shows few larger clusters with a low number of dependencies outside the clusters, or many smaller clusters with a high number of dependencies outside the clusters. It is application dependent which case is more desirable. When communication occurs often, the number of dependencies outside clusters should be minimized. Vice versa, when communication occurs rarely, cluster size should be minimized to improve the individual controller performance.

7.4.3 Application to pump-cellar system

A DSM is created for the pump-cellar system introduced in Section 7.2. First, the MRPS of the pump-cellar system is derived, resulting in a system with 21 components and 34 requirements. The resulting DSM is depicted on the left-hand side of Figure 7.12. The 21 components are listed at both axes as components G_i.

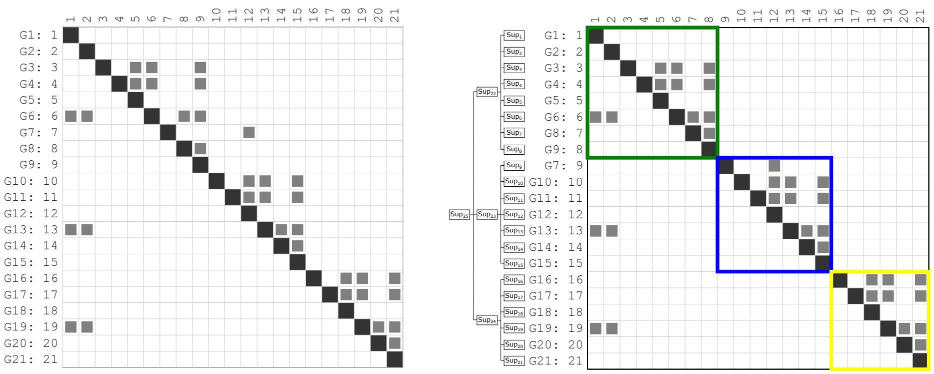


Figure 7.12: Unclustered pump-cellar DSM (left) and the clustered version (right)

Clustering parameters are chosen such that three clusters are obtained, as three PLCs are available in the test setup, and dependencies outside of clusters are minimized. The chosen clustering parameters are $\alpha = 2$, $\beta = 1.2$, $\mu = 1.5$, and $\gamma = 30$. The final clustering is depicted on the right-hand side of Figure 7.12. The maximum cluster size in this clustering is 8, and 4 dependencies are outside of the clusters.

The first cluster (green) contains the two traffic tube modes and the components of the middle pump cellars. The second (blue) and third (yellow) cluster contain the components of head pump cellar 1 and 2, respectively.

7.5 Step 2: Distributing the supervisor

After distributing the system, the second step is distributing the supervisor. First, multilevel synthesis procedure is performed, using a plant model, a requirement model and the acquired distribution of the system. Distributing the multilevel supervisor consists of three parts: the supervisor is localized, a delay-robustness check is performed and, if needed, a mutex algorithm is implemented in the model.

7.5.1 Supervisor localization

The acquired multilevel supervisor consists of a set of supervisors similar to the example in Figure 7.1. For each main cluster k, the set of relevant supervisors is taken as local supervisor $\mathbf{LOC}_k$. This local supervisor can only disable events in the controllable alphabet E_c^k of the components in its cluster. $\mathbf{LOC}_k$ is adjusted such that all other events are always enabled, which is done as follows using the theory of Cai and Wonham (2010).

$\mathbf{LOC}_k$ generally consists of a number of plant automata, requirements and one or more supervisor automata. First, all requirements and guards in the supervisor that disable events outside of its controllable alphabet are removed from the supervisor. All components outside of cluster k, that $\mathbf{LOC}_k$ no longer refers to in its requirements or guards, are removed from $\mathbf{LOC}_k$. If, after the adjustments, the supervisor automata or requirements in $\mathbf{LOC}_k$ refer to automata that are not part of components in cluster k, then those automata are not removed, but are referred to as observers. The events of observers in $\mathbf{LOC}_k$ are not controlled by $\mathbf{LOC}_k$.

For example, a requirement of $\mathbf{LOC}_k$ refers to automaton A that is part of cluster l. As A is not in cluster k, A is an observer in $\mathbf{LOC}_k$. The events of automaton A are observed by $\mathbf{LOC}_k$ in $\mathbf{LOC}_l$, such that every time an event of A happens in $\mathbf{LOC}_l$, it is communicated to $\mathbf{LOC}_k$. Events of A are referred to as shared events.

7.5.2 Delay robustness

The local supervisors, derived from the global supervisor, only impose globally correct behavior in the absence of communication delays. Communication delays might change the order in which events are observed in local supervisors. If, for example, in the local supervisor of Figure 7.13, event `b` is delayed, the order in which events `a` and `b` are observed might be different (`ab` or `ba`), resulting in a different end-state. Hence, communication delays might cause the localized supervisors to be globally unsafe, blocking, or uncontrollable.

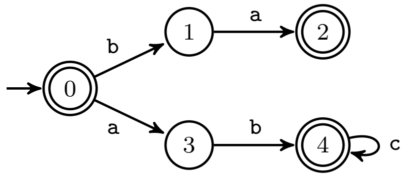


Figure 7.13: Example local supervisor

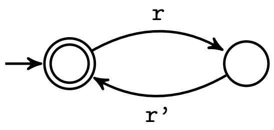


Figure 7.14: Channel $\mathbf{CH}(1, r, 2)$

When the local supervisors are implemented on separate PLCs, communication delays are unavoidable. It is up to the control engineer to decide whether it is better to implement multiple local supervisors on the same PLC to remove the communication delays between these local supervisors. This decision heavily depends on the application and which aspects of an implemented controller the control engineer deems relevant. This section provides a process for adjusting the local supervisors to deal with communication delays.

Zhang et al. (2016) proposes a method to check if the distributed supervisor is delay robust with respect to events that are shared among local supervisors. A number of requirements are stated for a distributed supervisor to be delay robust, i.e., a supervisor in which the end-state is not altered by communication delays. Within these requirements, equality of the language and marked language between a supervisor with zero delay and a supervisor with finite delay is guaranteed. As guards and requirements often refer to the state of an automaton, the requirements for delay robustness are defined in that paper such that local supervisors eventually reach the same state after delayed events. Definitions from Zhang et al. (2016), stated below in Equations (7.5) through (7.8) and the automaton of Figure 7.14, are used to define these requirements.

A global supervisor **SUP** is defined by n local supervisors: $\mathbf{SUP} = ||_i \; \mathbf{LOC}_i$, where $i = 1, 2, ..., n$. To check delay robustness for a shared event, first, this shared event is renamed. Say event `r` is shared between local supervisors $\mathbf{LOC}_1$ and $\mathbf{LOC}_2$, where `r` occurs in $\mathbf{LOC}_1$ and is observed by $\mathbf{LOC}_2$. `r` is now renamed to `r'` in $\mathbf{LOC}_2$ creating $\mathbf{LOC}_2'$. `r`, in $\mathbf{LOC}_1$, is referred to as a channeled event and `r'` is referred to as a delayed event. E_{ch} represents the set of channeled events. Next, a channel is defined is shown as in Figure 7.14. Since `r` is "communicated" from $\mathbf{LOC}_1$ to $\mathbf{LOC}_2$, the channel is named $\mathbf{CH}(1, \mathtt{r}, 2)$.

Using the channel and local supervisors, **SUP** and $\mathbf{SUP}'$ can be defined as in equations (7.5) and (7.6). **SUP** can be seen as the distributed supervisor, whereas $\mathbf{SUP}'$ is the distributed supervisor where communication delays are modeled.

$$\mathbf{SUP} = || \; (\mathbf{LOC}_1, \mathbf{LOC}_2) \tag{7.5}$$

$$\mathbf{SUP}' = || \; (\mathbf{LOC}_1, \mathbf{CH}(1, \mathtt{r}, 2), \mathbf{LOC}_2') \tag{7.6}$$

Equations (7.5) and (7.6) show the minimal example with 2 local supervisors and 1 communicated event. For n local supervisors with multiple shared events, the definitions in Equations (7.5) and (7.6) are adjusted. For each shared event r, a channel $\mathbf{CH}(i, r, j)$ is defined, where i is the index of the local supervisor that controls this event, and j the index of the local supervisor observing it. If multiple channels exist for one event, each channel has its own delayed event. This results in the following definitions.

$$\mathbf{SUP} = || \; (\mathbf{LOC}_j | j \in N) \tag{7.7}$$

$$\mathbf{SUP}' = || \; (\mathbf{LOC}_j', \mathbf{CH}(i, r, j) | r \in E_{ch}(i, j), i \in I_j, j \in N) \tag{7.8}$$

Above, $N = \{1, 2, ..., n\}$, and I_j is the set of indexes of all supervisors from which $\mathbf{LOC}_j$ observes events. $E_{ch}(i, j)$ is the set of all events which $\mathbf{LOC}_j$ observes from $\mathbf{LOC}_i$ and $\mathbf{LOC}_j'$ is $\mathbf{LOC}_j$ with all events in $E_{ch}(i, j)$ renamed to $E_{ch}'(i, j)$. Note that if $I_j = \emptyset$ for $\mathbf{LOC}_j$, i.e., if $\mathbf{LOC}_j$ does not observe any events from other local supervisors, $\mathbf{LOC}_j' = \mathbf{LOC}_j$.

Take supervisor **SUP** with event set E and channeled events E_{ch}. Let E_{delay} be the set of new (delayed) events introduced by the communication channels, like shown in Figure 7.14, in which each element r' is the delayed event of an event r in E_{ch}, i.e.

$$E_{delay} = \{r' | r \in E_{ch}, r' \text{ is the signal event of } r\}. \tag{7.9}$$

Then the event set of $\mathbf{SUP'}$ is $E' = E \cup E_{delay}$. Let $P : E'^* \to E^*$ be the natural projection.

Definition 2: Delay robustness
A supervisor **SUP** *with event set E is said to be delay robust with respect to its channeled events E_{ch} if the following 5 conditions, defined by Zhang et al. (2016), hold:*

$$P(L\left(\mathbf{SUP'}\right)) \subseteq L(\mathbf{SUP}) \tag{7.10}$$

$$P(L_m\left(\mathbf{SUP'}\right)) \subseteq L_m(\mathbf{SUP}) \tag{7.11}$$

$$P(L\left(\mathbf{SUP'}\right)) \supseteq L(\mathbf{SUP}) \tag{7.12}$$

$$P(L_m\left(\mathbf{SUP'}\right)) \supseteq L_m(\mathbf{SUP}) \tag{7.13}$$

$$\begin{aligned} &(\forall s \in E'^*)(\forall w \in E^*) \quad s \in L(\mathbf{SUP'}) \wedge P(s)w \in L_m(\mathbf{SUP}) \\ &\Rightarrow (\exists v \in E'^*) P(v) = w \wedge sv \in L_m(\mathbf{SUP'}) \end{aligned} \tag{7.14}$$

Intuitively, Equations (7.10) through (7.13) define that anything **SUP** can do can also be done by $\mathbf{SUP'}$ ($\mathbf{SUP'}$ is 'complete'), and anything that is disallowed by **SUP** is not possible in $\mathbf{SUP'}$ ($\mathbf{SUP'}$ is 'correct'). Furthermore, Equation (7.14) provides the observer property, which defines that the nonblocking property of **SUP** is preserved in $\mathbf{SUP'}$.

In this section, sufficient conditions are defined to check if a distributed supervisor is delay robust with respect to its shared events. First, the definitions of independence and mutual exclusiveness from Malik (2003) are given.

Definition 3: Independence
Two different events $r_1, r_2 \in E$, with guards g_1 and g_2 and updates u_1 and u_2, respectively, that share starting location q, are said to be independent with respect to a valuation v, if execution of one event does not disable the other event, and if the order in which the updates are applied does not affect the outcome.

Formally,

$$\begin{aligned} &independent(r_1, g_1, u_1, r_2, g_2, u_2, v) = \\ &u_1(v) \models g_2 \wedge u_2(v) \models g_1 \wedge u_1(u_2(v)) = u_2(u_1(v)). \end{aligned} \tag{7.15}$$

Two simultaneously enabled events are called independent when after the execution of either event, the other event is still enabled and either order reaches the same state. An example is shown in Figure 7.15, where events `a` and `b` are independent.

Definition 4: Mutual exlusiveness
Given an EFA G with event set E, guard set C, and update set U, two different events $r_1, r_2 \in E$ with guards g_1 and g_2, respectively, are said to be mutually exclusive with respect to a valuation v if guards g_1 and g_2 never evaluate to `True` *at the same time, or when event r_1 and r_2 do not share a starting location q.*
Formally,

$$\begin{aligned}
&mutually\_exclusive(r_1, g_1, r_2, g_2, v) = \\
&\neg(v \models g_1 \wedge v \models g_2) \\
&\vee((\forall r \in E)(\forall g \in C)(\forall u \in U) \rightarrow (q, r, g, u)! \Rightarrow r \neq r_1) \\
&\vee((\forall r \in E)(\forall g \in C)(\forall u \in U) \rightarrow (q, r, g, u)! \Rightarrow r \neq r_2).
\end{aligned} \tag{7.16}$$

When two events are never enabled simultaneously, they are called mutually exclusive. This is the case when they share no starting location, or when their guard conditions never evaluate to `True` at the same time. An example of two mutually exclusive events is shown in Figure 7.16, where the guards of events `a` and `b` never evaluate to `True` at the same time.

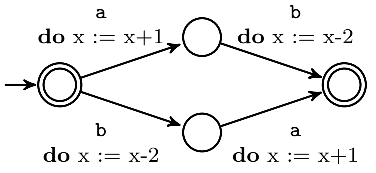


Figure 7.15: Independence

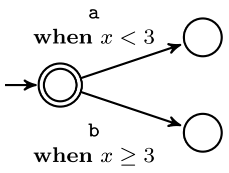


Figure 7.16: Mutual exclusiveness

A number of assumptions are made in this project, before defining the sufficient conditions for delay robustness. First, it is assumed that communication occurs with a finite delay and that a series of communicated events between two supervisors can be modeled by a FIFO queue. This is no issue in practice, as FIFO communication can be guaranteed by the communication protocol, and the timescale between communication occurrences is much higher than the timescale of the communication delay. Note that this does not mean that communication delays have no effect on the controlled behavior, as the timescale of the communication delay and the time-scale of the PLC cycle are the same. Second, it is assumed that an event r is always followed by its delayed event r' before a second occurrence of event r, as is modeled by the channels. This is a reasonable assumption, as in practice the timescale of the communication delay is much lower than the time between communication occurrences.

Theorem 1:
*If for every location $q \in Q'$ where a delayed event $r' \in E_{delay}$ is enabled, r' is either independent or mutually exclusive with respect to all other events $r \in E'$ with associated guard g and update u, then **SUP** is delay robust.*

More formally,

$$\begin{aligned}&((\forall q \in Q')(\forall r' \in E_{delay})(\forall r \in E') \to (q, r', \mathtt{true}, -)! \wedge (\exists g \in G)(\exists u \in U) \to (q, r, g, u)! \Rightarrow \\ &\mathit{independent}(r', \mathtt{True}, -, r, g, u, v) \vee \mathit{mutually\_exclusive}(r', \mathtt{True}, r, g, v)) \qquad (7.17)\\ &\quad \Longrightarrow \text{SUP is delay robust.}\end{aligned}$$

A proof of Theorem 1 is provided in the next section.

In Reijnen et al. (2019b), an algorithm is provided to check if a supervisor is confluent. Confluence is proven if, among other options, event combinations are independent or mutually exclusive. These checks for independence and mutual exclusiveness, that are part of the algorithm presented in Reijnen et al. (2019b), are used in this project to check the delayed events in **SUP**$'$. If all delayed events are independent or mutually exclusive, the distributed supervisor **SUP** is delay robust.

The proposed delay-robustness check works pairwise over all event combinations consisting of a local event and a communicated event. This means that the worst-case time complexity of the check is $\mathcal{O}(n^2)$ with n the number of controllable events in the plant.

It should be noted that event combinations of two delayed events that originate from the same local supervisor need not be checked. This is because the communication channels are assumed to be FIFO queues, so the event order of these two delayed events is preserved.

7.5.3 Proof of Theorem 1

In this section, a proof of Theorem 1 is provided. First, some intermediate results are provided in Lemmas 1 through 3.

Lemma 1:
By definition of **SUP**$'$ and by definition of channels, in a location reached by channeled event r, the corresponding delayed event r' is always enabled.

Lemma 2:
An event r is only enabled after a string $s \in L(\mathbf{SUP}')$, if for every occurrence of event r in s, s contains an event r', due to the definition of the channels.

Definition 5: Delay-free string
A string s is defined as a delay-free string if and only if every channeled

event r in s is directly followed by the corresponding delayed event r', essentially meaning that no delay has occurred. Note that for every delay-free string $s \in L(\mathbf{SUP'})$, $P(s) \in L(\mathbf{SUP})$. Similarly, for every delay-free string $s \in L_m(\mathbf{SUP'})$, $P(s) \in L_m(\mathbf{SUP})$.

Lemma 3:
For any string $s \in L(\mathbf{SUP'})$, where any channeled event is at some point followed by its delayed event, there exists a delay-free string s' such that $P(s) = P(s')$ and $(q_0, v_0) \overset{s}{\mapsto} = (q_0, v_0) \overset{s'}{\mapsto}$. This is true as any delayed event is independent or mutually exclusive with all other events, following Lemma 1. For example, consider a string $s_1 r s_2 r' s_3 \in L(\mathbf{SUP'})$ such that $s_1 \in E^*, s_2 \in E^*, s_3 \in E^*$. Note that $P(s_1 r s_2 r' s_3) = s_1 r s_2 s_3 = P(s_1 r r' s_2 s_3)$, moreover, as r' is independent with its simultaneously enabled events, $(q_0, v_0) \xmapsto{s_1 r s_2 r' s_3} = (q_0, v_0) \xmapsto{s_1 r r' s_2 s_3}$. Here, $s_1 r r' s_2 s_3$ is a delay-free string as every channeled event, in this case r, is directly followed by its delayed event, in this case r'.

Now a proof is provided for every condition in Equations (7.10) through (7.14).

Condition 1: Equation 7.10
It is proven by induction on the length of s that:
$(\forall s \in L(\mathbf{SUP'}))P(s) \in L(\mathbf{SUP})$.
Base step: $\epsilon \in L(\mathbf{SUP'})$ and $\epsilon \in L(\mathbf{SUP})$, trivially.
Inductive step: suppose $t \in L(\mathbf{SUP'})$, $P(t) \in L(\mathbf{SUP})$ and $ta \in L(\mathbf{SUP'})$, we must prove that $P(ta) \in L(\mathbf{SUP})$.
Following Lemma 2, two possible cases are identified:

1. t contains a delayed event for every occurrence of a channeled event, (i.e. no delayed event is enabled in $(q_0, v_0) \overset{t}{\mapsto}$).
2. t does not contain a delayed event for every occurrence of a channeled event, (i.e. at least one delayed event is enabled in $(q_0, v_0) \overset{t}{\mapsto}$).

In case 1), following Lemma 3, $(\exists t' \in L(\mathbf{SUP'}))$ $(q_0, v_0) \overset{t'}{\mapsto} = (q_0, v_0) \overset{t}{\mapsto}$, where t' is a delay-free string. In this case, a cannot be a delayed event, as every delayed event is disabled by its channel, i.e. $ta \notin L(\mathbf{SUP'})$. If a is a channeled event, $t'aa' \in L(\mathbf{SUP'})$ is again a delay-free string as every channeled event is directly followed by its delayed event. Now, $P(ta) = P(t'a) = P(t'aa') \in L(\mathbf{SUP})$. If a is not a channeled event, $t'a$ is a delay-free string. Therefore, $P(ta) = P(t'a) \in L(\mathbf{SUP})$.

In case 2), ta can be extended with a string of delayed events d, such that the string $tad \in L(\mathbf{SUP'})$ contains a delayed event for every occurrence of a channeled event. Therefore, $P(tad) = P(ta)$ and following Lemma 3 there exists a delay-free string $t' \in L(\mathbf{SUP'})$ such that $(q_0, v_0) \overset{t'}{\longmapsto}= (q_0, v_0) \overset{tad}{\longmapsto})$. Now, $P(ta) = P(tad) = P(t') \in L(\mathbf{SUP})$.

In both the cases, $(\forall a \in E')\ ta \in L(\mathbf{SUP'})$ it is proven that $P(ta) \in L(\mathbf{SUP})$, therefore condition 1 holds.

Condition 2: Equation 7.11
Condition 2 holds if: $(\forall s \in L_m(\mathbf{SUP'}))P(s) \in L_m(\mathbf{SUP})$.

Any string $s \in L_m(\mathbf{SUP'})$ contains a delayed event for every occurrence of a channeled event, otherwise s does not reach a marked state by definition of channels. Therefore, following Lemma 3, for every string $s \in L_m(\mathbf{SUP'})$ there exists a delay-free string s' such that $(q_0, v_0) \overset{s'}{\longmapsto}= (q_0, v_0) \overset{s}{\mapsto}$ and $P(s) = P(s')$. Hence, $P(s) \in L_m(\mathbf{SUP})$ by definition of delay-free strings.

Condition 3: Equation 7.12
Condition 3 holds if: $(\forall s \in L(\mathbf{SUP}))\quad s \in P(L(\mathbf{SUP'}))$.

Following Lemma 3, for any string s, a delay-free string $s' \in L(\mathbf{SUP'})$ can be created by replacing any channeled event in s by the channeled event directly followed by its delayed event. For example, if s contains a channeled event r, create s' by replacing every occurrence of r by rr'. By definition of the channels, $\mathbf{SUP'}$ must allow such a delay-free string. As $s' \in L(\mathbf{SUP'})$, it must hold that $s = P(s') \in P(L(\mathbf{SUP'}))$.

Condition 4: Equation 7.13
Condition 4 holds if: $(\forall s \in L_m(\mathbf{SUP}))\quad s \in P(L_m(\mathbf{SUP'}))$.

Following Lemma 3, for any string s, a delay-free string $s' \in L_m(\mathbf{SUP'})$ can be created by replacing any channeled event in s by the channeled event directly followed by its delayed event, as is done for Condition 3. Following Lemma 1, $\mathbf{SUP'}$ must allow such a delay-free string. As $s' \in L_m(\mathbf{SUP'})$, it must hold that $s = P(s') \in P(L_m(\mathbf{SUP'}))$.

Condition 5: Equation 7.14
Following Lemma 2, two possible cases are identified:

- s contains a delayed event for every occurrence of a channeled event, (i.e. no delayed event is enabled in $(q_0, v_0) \overset{t}{\mapsto}$.)
- s does not contain a delayed event for every occurrence of a channeled event, (i.e. at least one delayed event is enabled in $(q_0, v_0) \overset{t}{\mapsto}$.)

In case 1), due to the independence of delayed events with their simultaneously enabled events, following Lemma 3, $(\exists s' \in L(\mathbf{SUP'}))\ (q_0, v_0) \overset{s'}{\longmapsto}= (q_0, v_0) \overset{s}{\mapsto}$, where s' is a delay-free string. Next, for any string w a delay-free string w' can be created as is done for Condition 3. Let $v = w'$, then $P(v) = w$. Moreover, as $s'v$ is a delay-free string and $P(s'v) = P(s)w \in L_m(\mathbf{SUP})$, it must hold that $s'v \in L_m(\mathbf{SUP'}) \Rightarrow sv \in L_m(\mathbf{SUP'})$.

In case 2), s can be extended with a string of delayed events y, such that the string $sy \in L(\mathbf{SUP'})$ contains a delayed event for every occurrence of a channeled event. For sy it must hold that $(\exists s' \in L(\mathbf{SUP'}))\ (q_0, v_0) \xmapsto{s'} = (q_0, v_0) \xmapsto{sy}$, due to the independence of delayed events with their simultaneously enabled events. Next, for any string w a delay-free string w' can be created following Lemma 3 as is done for Condition 3. Let $v = yw'$, then $P(v) = w$. Due to the independence of delayed events with their simultaneously enabled events, following Lemma 1, $(\exists z' \in L(\mathbf{SUP'}))\ (q_0, v_0) \xmapsto{f'} = (q_0, v_0) \xmapsto{s'w'} = (q_0, v_0) \xmapsto{syw'}$. As z' is a delay-free string and $P(z') = P(s)w \in L_m(\mathbf{SUP})$, it must hold that $z' \in L_m(\mathbf{SUP'}) \Rightarrow sv \in L_m(\mathbf{SUP'})$.

This completes the proof of Theorem 1.

7.5.4 Mutual exclusion

If a supervisor is not delay robust with respect to a delayed event, this event is denoted a delay-critical event. Each event combination with such an event is called a delay-critical event combination. When a distributed supervisor contains such combinations, nonblockingness, safety, and controllability cannot be guaranteed. Therefore, the supervisor needs to be adjusted. The first option is to change the model of the system such that the supervisor no longer contains delay-critical event combinations. This is only possible in specific situations, and currently no guidelines exist to identify such situations. The second option is less desirable, but it is always possible. Using mutex algorithms, it is possible to enforce the mutual exclusiveness of the delay-critical event combinations in the global supervisor **SUP**. The event combinations are, therefore, no longer delay-critical.

The undesirable part of implementing mutex algorithms is the fact that delay-critical events must always first be enabled by the mutex algorithm before the event can be executed. In practice, this means that the execution of a delay-critical event in a distributed settings may take more time compared to a monolithic setting. As the timescale of these delays is much smaller than the timescale at which these events are executed, this is not a problem in the type of systems considered in this thesis.

A mutex implementation process is described here based on theory of Raynal (2013). An implementation of a mutex algorithm in the model is referred to as a mutex lock. A delay-critical event combination of events r' and a, can be made mutually exclusive by implementing a mutex lock for events r and a that ensures that events r and a are always disabled by the corresponding local supervisor, unless it has entered the critical section. Note that this only holds under the

FIFO queue assumption. A home-based token passing algorithm is used, as this is a simple and efficient algorithm for low numbers of processes.

For each mutex lock, a home process and an away process are defined. The away process is modeled by an automaton (Figure 7.17) that models the request procedure, and three requirements (Table 7.1) that contain a request and a return condition. Depending on the event (`crit_event`) that is to be made mutually exclusive, these conditions state when the event is available and when it has occurred, respectively.

Table 7.1: Requirements for the away process

1	requirement	`request`	needs	`RequestCondition`
2	requirement	`return`	needs	`ReturnCondition`
3	requirement	`crit_event`	needs	`Requester.Received`

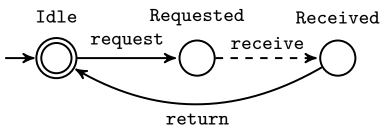


Figure 7.17: Request procedure

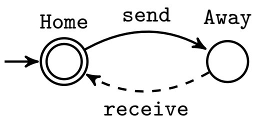


Figure 7.18: Token tracker

The home process consists of an automaton for the token tracker (Figure 7.18) and two requirements (Table 7.2). An input `R`, modeled as a sensor similar to the automaton shown in Figure 7.16, is on if the away process is requesting the token, i.e., `R` turns on if the away process `Requester` automaton is in the state `Requested`. Similar to the away process, the home process has a return condition to ensure the token is not sent before the critical event is taken.

Table 7.2: Requirements for the home process

1	requirement	`send`	needs	`ReturnCondition` $\wedge$ `R.on`
2	requirement	`crit_event`	needs	`Tracker.Home`

As a mutex lock is implemented for each delay-critical event combination, it is possible that an event is the critical event in multiple mutex locks. Deadlock can occur if there exists an overlap in critical sections, as each process might need the token of the other, called a cyclic dependency. To prevent this, the mutex locks are ordered. The order in which tokens are acquired is controlled by adjusting the `RequestCondition`, such that a lock can only be acquired if locks earlier in the order are already acquired, thus breaking the cyclic dependency.

7.5.5 Mutex implementation decision

Implementing mutex locks increases the state space of the local supervisors, as they introduce additional automata to the plant. Each mutex that is implemented increases the state space by a factor 4 for the home process, and a factor 3 for the away process. One of the goals of obtaining a distributed controller is the smaller state space sizes of the local supervisors. Therefore, the increase in state space size due to mutexes should be kept to a minimum.

In order to do this, it is possible to implement mutex locks for two sets of events instead of for two events. The `crit_event` in each process of the mutex is then replaced by a critical event set. All events of the home process set are then mutually exclusive with all events of the away process set. This means that fewer mutexes need to be implemented for a certain number of delay-critical events, at the cost that events can be unnecessarily disabled.

To minimize the negative effect of mutex locks, the mutex event sets are chosen in a smart way. To do this, first it is determined which delay-critical event pairs are eligible to be combined in one mutex lock. This is the case when they are communicated between the same two PLCs, and have the same home process. To minimize the number of mutex locks, delay-critical event pairs are combined when they have the same request condition in the away process. This is typically the case when an event in the away process is delay-critical with multiple events in the home process.

7.5.6 Application to pump-cellar system

In Subsection 7.4.3, a clustering for the pump-cellar system was determined. Based on this clustering, the multilevel synthesis procedure is performed. The multilevel tree is depicted on the left of the clustered DSM in Figure 7.12. The tree contains a supervisor for each cluster, as well as a supervisor for each individual component. The resulting multilevel supervisor consists of a set of 25 supervisors. The advantage of using multilevel synthesis as a basis for localization, is that only supervisor $Sup25$ refers to components of the three main clusters, and is thus the only supervisor that needs to be localized.

The first step in distributing this multilevel supervisor is localization. As the distributed supervisor is implemented on three PLCs, three local supervisors, $\mathbf{LOC}_1$, $\mathbf{LOC}_2$, and $\mathbf{LOC}_3$, are created. $\mathbf{LOC}_1$ is created using supervisors 1-8, 22, and 25, $\mathbf{LOC}_2$ is created using supervisors 9-15, 23, and 25, and $\mathbf{LOC}_3$ is created using supervisors 16-21 and 24-25. In all three cases $Sup25$ is the only supervisor that needs to be adjusted. $Sup25$ adjusted for $\mathbf{LOC}_1$, $\mathbf{LOC}_2$, and $\mathbf{LOC}_3$ is referred to as $Sup25_a$, $Sup25_b$, and $Sup25_c$, respectively.

When $Sup25$ is adjusted to $Sup25_a$, the requirements and guards that disable events in components $G7 - G21$ are removed, so it does not disable any events from those components. The remaining requirements in $Sup25_a$ do not refer to components $G7 - G21$, so the localization of $Sup25_a$ is finished.

To create $Sup25_b$, all requirements and guards in $Sup25$ that disable events in components $G1 - G8$ and $G16 - G21$ are removed. The remaining requirements still refer to components $G1$ and $G2$, so observer automata are added to $Sup25_b$ for these components.

Finally, $Sup25_c$ is created similarly to $Sup25_b$, where observer automata are added for components $G1$ and $G2$.

The next step is to check for delay robustness. As $\mathbf{LOC}_2$ and $\mathbf{LOC}_3$ contain observer components $G1$ and $G2$, the events of these components are delayed events in $\mathbf{LOC}_2$ and $\mathbf{LOC}_3$. The delay-robustness check discussed in Subsection 7.5.2 is used to check if the distributed supervisor is delay robust with respect to these events. The check indicates a set of 18 event combinations for both $\mathbf{LOC}_2$ and $\mathbf{LOC}_3$ that are neither independent nor mutually exclusive, resulting in a total of 36 delay-critical event combinations.

The final step is setting up mutexes for these 36 event combinations. Following the guidelines set in Subsection 7.5.5, event sets are chosen such that the 36 event combinations are made mutually exclusive using 4 mutex locks. An example of a delay-critical event combination is event `c_operational`$'$ in the first traffic tube component ($G1$) and event `c_store` in the `Mode` automaton of head pump cellar 1 ($G13$). Event `c_operational` is in the critical event set of the home process of the mutex lock, and `c_store` is in the critical event set of the away process. Therefore, these events are now mutually exclusive. The resulting distributed supervisor is nonblocking, safe, and controllable, following the theory of Ouedraogo et al. (2011) and Cai and Wonham (2010), and it is delay robust following the theory of Zhang et al. (2016) and Proposition 1.

7.6 Case study: The Swalmen tunnel

In this section, a real-life case study is described in which a distributed supervisor is obtained and implemented for a road tunnel. The Swalmen tunnel is a road tunnel in the Netherlands that is previously introduced in Section 6.1.

A road tunnel supervisor has two main purposes: monitoring the situation in the traffic tubes to detect an emergency, and subsequently handling the emergency by closing the traffic tubes and turning on evacuation systems.

In previous work the Swalmen tunnel was modeled using 180 automata and 414 requirements, from which a supervisor is synthesized with approximately 10^{71} states. The complete model can be found in a repository[2].

Deriving a distributed supervisory controller for a road tunnel is of industrial importance, since a typical tunnel in the Netherlands is controlled by up to 10 PLCs.

[2]`https://github.com/LMoormann/Swalmentunnel`

7.6.1 Distributing the system

To obtain a distribution of the Swalmen tunnel, a DSM is created using the MRPS of the Swalmen tunnel and the requirements. This DSM is clustered using the conclusions of the parameter study in Subsection 7.4.2. The clustering parameters that are used are $\alpha = 2$, $\beta = 1.05$, $\mu = 1.5$, $\gamma = 30$. These parameters result in a small number of dependencies outside of the clusters, but the generated number of clusters exceeds the available number of PLCs. Therefore, in the end we manually combine some generated clusters, whereby smaller clusters that have dependencies outside the cluster are combined to reduce the number of dependencies outside the clusters, and more evenly distribute the components over the clusters. The final clustering is shown in Figure 7.19, which contains three clusters, as the goal is to distribute the controller over three PLCs, and four dependencies lie outside of the clusters, meaning that communication is required. The text in Figure 7.19 indicates which tunnel components are part of which cluster.

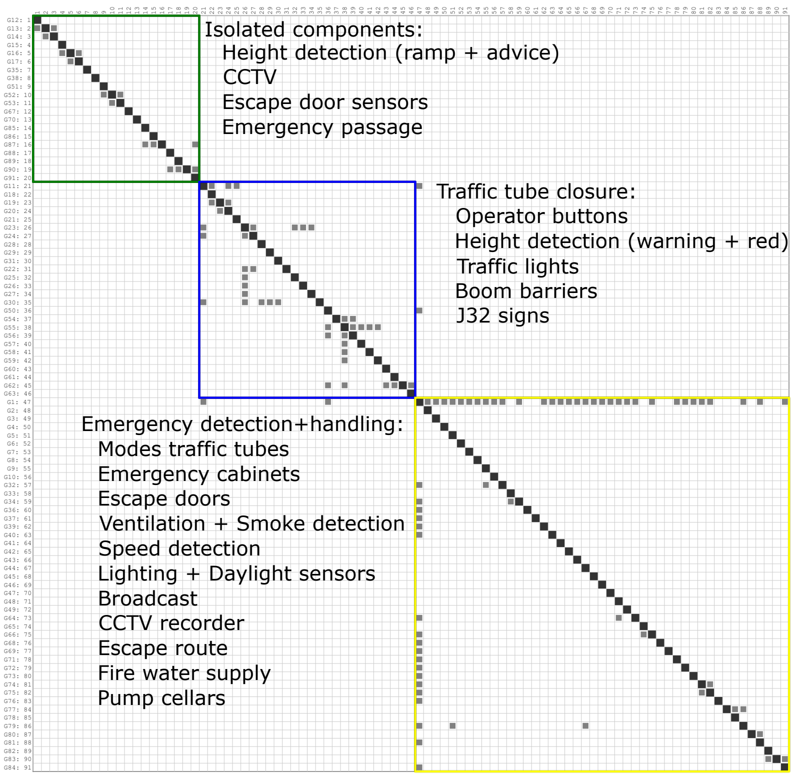


Figure 7.19: Clustered Swalmen tunnel DSM

7.6.2 Distributing the supervisor

The global supervisor is localized into three local supervisors following the distribution of the clustered DSM in Figure 7.19. Observer automata are added to local supervisors 2 and 3, as they require information of each other.

Secondly, delay-critical event pairs are identified by checking for delay robustness, using the algorithm described in Subsection 7.5.2. 28 events are shared, resulting in 20440 event pairs that are analyzed. Of these pairs, 454 are delay critical.

For the delay-critical event pairs, mutex locks are implemented to enforce mutual exclusion. Following the guidelines of Subsection 7.5.5, 28 mutex locks are implemented for the 454 delay-critical event pairs. Local supervisor 2 is chosen as the home process of 20 of the mutex locks, and local supervisor 3 is the home process of the remaining 8, as this results in the lowest number of required locks.

7.6.3 Implemented performance analysis

The local supervisors are implemented on a HIL setup to validate the controlled behavior using the actual control hardware. The HIL setup consists of three PLCs of type ABB AC800M PM866A that can communicate over an Ethernet connection, and two laptops that simulate and visualize the operator interface and the virtual tunnel, respectively. The implemented PLC code is automatically generated from the local supervisors. Subsequently, communication is set up between the PLCs using the ABB control builder software. Time delays based on Kostenko (2019) are simulated for reading inputs and writing outputs.

To analyze the results of the distributed controller, two tests are performed: one where the global supervisor is implemented on a single PLC and one where the local supervisors are implemented on three PLCs. The results are shown in Table 7.3.

Table 7.3: Results of the HIL tests

	Single PLC	**Local 1**	**Local 2**	**Local 3**
File size	673kb	65kb	463kb	595kb
Cycle time	17ms	10ms	10ms	12ms
Lines of code	17620	1727	7485	13507
Inputs	260	35	61	188
Outputs	46	11	13	46

As can be seen in Table 7.3, the file size of each of the local controllers is smaller than that of the global controller, meaning that distributing the controller has decreased the needed PLC memory. When looking at the cycle times, it is seen that the single PLC is slower than the individual local PLCs. Local PLC

1 and 2 achieve the optimal cycle time, which cannot be lower than 10ms. To give a further indication of the improvement in performance of the distributed controllers, the number of lines of code that is run each cycle is compared. Here, it is seen that each of the local controllers runs more quickly than the global controller. Furthermore, comparing the inputs and outputs of the global controller and the local controllers also indicates that the read/write process is faster in the distributed setting.

7.7 Concluding remarks

In this chapter, a method for developing distributed supervisors is presented, based on supervisor synthesis, localization theory, DSMs, and delay-robustness theory to take into account communication delays. The resulting distributed supervisor is delay robust, nonblocking, controllable and safe. The method is demonstrated by developing a distributed supervisor for a pump-cellar system.

The method consists of two main steps, being the distribution of the system and the distribution of the supervisor. The system is distributed using DSMs, and a parameter study is performed to gain insight in the relation between the clustering parameters and the desired properties of a distributed supervisor.

In the second step the supervisor is distributed, which introduces communication between the local supervisors. As communication can have delays, a delay-robustness check is proposed and implemented to determine whether a combination of events is delay robust by checking for independence and mutual exclusiveness. In the case of delay-critical event combinations, mutual exclusiveness is enforced by implementing mutex locks. To counteract the increase in state space due to the introduction of mutex locks, guidelines are proposed to determine sets of events that are covered by the same mutex lock.

Finally, two HIL tests are performed to compare the global controller implementation to the distributed controller implementation. These tests show that the distributed controller requires less memory and achieves higher performance results than the global controller.

The method has been applied in a real-life case study, where a supervisor is synthesized, distributed, and implemented for the Swalmen tunnel, a road tunnel in the Netherlands.

The proposed method uses the top-down approach of localization, for which synthesis of a global supervisor is required. For large-scale systems this might prove to be impossible, so more research is needed to investigate bottom-up methods, as for example have been proposed by Su et al. (2010).

Chapter 8

Digital twins for road tunnels

Over the recent years, a digitization trend can be seen in the design process of road tunnel systems. Where traditionally the design process was mostly document-based, nowadays many types of digital models are used. Existing studies, such as Lidström (1998), Min et al. (2008), and Borg et al. (2014), have shown the applicability of digital models in the design of road tunnels. The concept of creating such digital models, explained in Kensek (2014) and Hardin and McCool (2015), is called building information modeling (BIM) in which multiple facets of infrastructural design are integrated in a single model, called a digital twin. In Pires et al. (2019), a digital twin is most generally described as the digital copy of a physical object or system, that is connected and shares functional and/or operational data. Digital twins are seen as an important part of the Industry 4.0 initiative, as is reported in Pires et al. (2019), which represents an industrial shift towards digitization. Digital twins are expected to play a major role in this shift, by enabling performance boosts with the use of high-end simulations in development and maintenance of systems.

The application field of digital twins is very broad. Some examples include connected and automated mobility of smart vehicles Schranz et al. (2020), road traffic control to improve traffic flows Kumar et al. (2018), development of algorithms for autonomous driving Atorf and Roßmann (2018), and digital twin simulation for train operation and control Meng et al. (2020).

This chapter is based on: Moormann, L., van Hegelsom, J., van de Mortel-Fronczak, J.M., Maessen, P., Fokkink, W.J., and Rooda, J.E. Digital twins for the validation of road tunnel controllers. In *ITA-AITES World Tunnel Congress, WTC2022 and 47th General Assembly*, 2022.

In the field of road tunnels, digital twins are mostly used for designing underground constructions. Lidström (1998), Min et al. (2008), and Borg et al. (2014) showcase the use of digital models in the design and construction of road tunnels, though controller design is often omitted. The Center for Underground Construction in the Netherlands is a network organization that gathers, creates and spreads knowledge about usage of underground spaces such as tunnels. This organization has written a document in which possible advantages and disadvantages of using digital twins in the field of underground construction are discussed Centrum Ondergronds Bouwen (2021). The document mentions several exploratory studies, though no conclusions are drawn.

In Nooijens (2020), a study is performed on the use of digital twins in the exploitation phase of infrastructural systems, focused on management and maintenance of information for a tunnel and a waterway lock. They conclude that a digital twin can increase the maintenance performance through overall improved information provision. Another study related to using digital twins in the maintenance and operation of tunnels is found in Tijs (2020), where interviews with end users are held that identify that possible advantages lie in the domain of monitoring, predicting, and controlling.

Road tunnels are controlled by a supervisory controller to ensure safe operation of the tunnel by guaranteeing proper cooperation between its components. In the previous chapters, it is shown how a supervisory controller for a road tunnel is automatically synthesized from a model of the plant and a model of the controller requirements. Advantages of synthesizing the supervisory controller include a much shorter time-to-market and the guarantee that the controller always adheres to the defined requirements. An important step, however, remains the validation of the controller. While the synthesized controller is guaranteed to adhere to the requirements, it is still possible to define incorrect or incomplete requirements.

Validation is traditionally performed using tests on the realized systems, but with the digitization trend in the design process, as well as advances in model-based engineering, the use of simulation for the purpose of controller validation is becoming more popular. In simulations, the designed controller is connected to a model of the system to run through possible scenarios while observing the response of the controller.

In this chapter, the advantages of combining digital twin simulation and synthesis-based design are discussed. This combination provides advantages for several purposes. First, a digital twin is more easily created when an unambiguous description of the plant is available, as is the case when the system is modeled using automata. Second, a controller can more intuitively and extensively be validated when it is connected to a digital twin. Third, when a supervisory controller is connected to a digital twin, it can be used for the purpose of operator training, which is an important aspect in the case of road tunnel systems.

This chapter is structured as follows. Section 8.1 describes the design process of a digital twin for a road tunnel. Next, Section 8.2 discusses the benefits of using digital twins in the validation process of supervisory controllers. In Section 8.3, the use of digital twins in combination with supervisory controllers for the purpose of operator training is discussed. Finally, in Section 8.4, concluding remarks are given.

8.1 Development of a digital twin

In this section, the development of a digital twin for a road tunnel is described. Specifically, examples related to the digital twin for the Swalmen tunnel are given. First, the process of designing the digital twin is described in Subsection 8.1.1. Next, Subsection 8.1.2 covers the process of connecting the digital twin to the HIL setup.

8.1.1 Designing the digital twin

In the design process of the supervisory controller and the digital twin, various software tools are used. Figure 8.1 gives an overview of these tools. The tools shown in the top row are used for the purpose of controller design and implementation, as described in Chapters 4 and 6, and the tools shown in the bottom row are used for the design of the digital twin.

Most work on designing a digital twin is done in *Unity*, which is a 3D game engine that is described in Nicoll and Keogh (2019). In this engine, 3D models can be created of all the system components. In many cases, existing 3D models can be imported from *SketchUp*, introduced in Chopra (2012). Besides visualization of the 3D models, behavior of the components is also modeled in Unity using behavior scrips. This behavior includes how components can move, how components act when they collide with other components, and how they can transform, e.g. by changing color. As the goal of this chapter is to investigate the possibilities of combining digital twins with supervisory controller design, the digital twin is connected to a PLC on which the supervisory controller is implemented. The *Prespective* software, a plug-in for Unity that is described in Prespective (2021), is used to allow the digital twin to be connected to the I/O of the PLC. By incorporating the I/O signals of the PLC in the behavior scripts of the digital twin, the actuator components can move according to the PLC outputs, and the sensor components can send input signals to the PLC. Listing 8.1 shows an example of the behavior script of a boom barrier in which actuator and sensor signals are connected to the I/O of the PLC. The connection between the digital twin and the PLC is described in more detail in Subsection 8.1.2. Some digital twin design examples are elaborated here for components in the Swalmen tunnel. A complete overview of the digital twin can be found in van Hegelsom (2021).

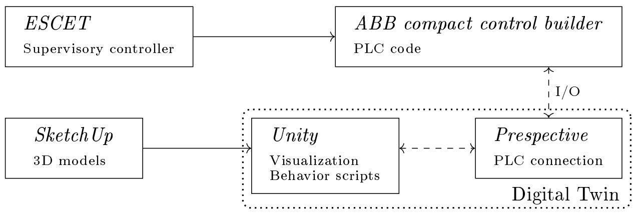


Figure 8.1: Overview of the tools used in the controller design process and the digital twin design process.

Static environment and vehicles

The first part in designing a digital twin for a road tunnel is designing the static environment of the tunnel. This includes the tunnel structure, the highway lanes, and the surrounding environment, which are the largest 3D models in the digital twin and act as a basis for the other components. The model of the static environment is made in a modular way, as can be seen on the left in Figure 8.2. The tunnel parts shown here are a generic road section, a road section with room for an emergency passage, a tunnel entrance section, and a tunnel middle section. These parts are then used to build the environment in Unity. The main advantage of this way of assembling the tunnel is that the road before and after the tunnel and the tunnel itself can be made as long as desired, without having to change the 3D models. Also, the number of models that need to be designed is smaller, since the entrance is used twice for example.

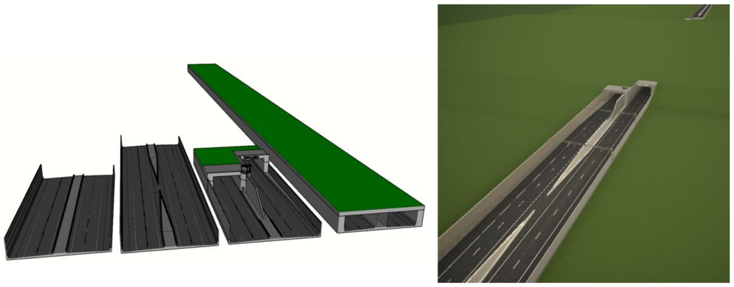


Figure 8.2: Modeled tunnel parts in SketchUp (left) and the tunnel environment built from these parts in Unity (right).

Since the Swalmen tunnel is a tunnel for road traffic, the traffic stream is the most important non-controllable entity that has a lot of interaction with the controlled components in the tunnel. The traffic stream in the 3D model is composed of a collection of cars and trucks shown in Figure 8.3. The vehicles are described in detail below the figure, from left to right.

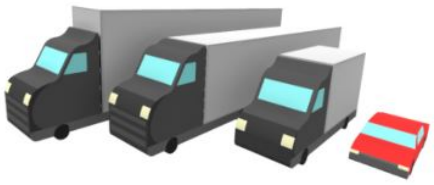


Figure 8.3: The four vehicles modeled in the digital twin.

High large truck This vehicle should not enter the tunnel, as it is too high (> 4.1 [m]) and can damage equipment mounted on the ceiling of the tunnel or the ceiling itself. The model is 16.6 [m] long and 4.65 [m] high.

Low large truck This long vehicle can safely enter the tunnel. Its dimensions are 16.6 [m] long and 3.65 [m] high.

Small truck This vehicle is also allowed to enter the tunnel and is mainly there to get more variety in the traffic stream. The small truck is 6.6 [m] long and 3.25 [m] high.

Car Most of the vehicles on the highway are standard cars. The cars in the digital twin can have different colors. They are 4.2 [m] long and 1.4 [m] high.

Boom barriers

The boom barriers in the tunnel are an example of a controllable entity that must be able to respond to an output signal of the controller, and send back input signals to the controller depending on the current position.

First, a 3D model of a boom barrier is obtained in SketchUp and imported to Unity. This 3D model is shown in Figure 8.4. It consists of the base on the left-hand side and the actual barrier indicated by the blue box.

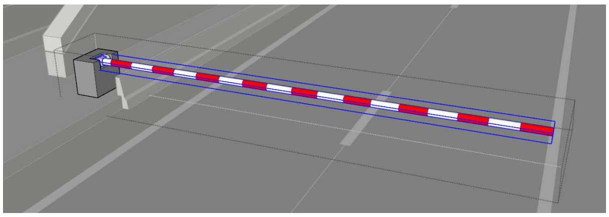


Figure 8.4: The boom barrier entity in SketchUp. The whole group is represented with the dotted box and the barrier itself is its own group, shown by the blue box.

To create a moving boom barrier in the digital twin, a behavior script for a rotational joint is added in the model, shown in Listing 8.1, which has two functions. First, it assigns a variable to the rotational position and direction of the boom barrier, `Motor.Position` and `Motor.Direction`, respectively. Second, it links variables to the actuators and sensors of the boom barrier. The variable `Motor.Direction` is linked to `ActuatorUp` and `ActuatorDown` to rotate the boom barrier when the actuators are on. Furthermore, the sensor values are linked to the variable `Motor.Position` to turn the sensors on and off at the correct position.

```
// Enable events based on the actuator states
if (IO_dict["ActuatorUp"].Boolean) { Motor.Direction = 1; }
else if (IO_dict["ActuatorDown"].Boolean) { Motor.Direction = -1; }
else if (IO_dict["ActuatorStop"].Boolean) { Motor.Direction = 0; }
else { Motor.Direction = 0; }

// Set the sensor values based on position
IO_dict["SensorOpen"].Boolean =
Mathf.Abs(Motor.Open - Motor.Position) < SensorOffset;
IO_dict["SensorOpening"].Boolean =
Mathf.Abs(Motor.Closed - Motor.Position) < SensorOffset;
IO_dict["SensorStopped"].Boolean = Motor.Direction == 0;
IO_dict["SensorClosing"].Boolean = Motor.Direction == 1;
IO_dict["SensorClosed"].Boolean = Motor.Direction == -1;
```

Listing 8.1: Behavior script of the rotational joint.

The final step is to connect the variables that are declared in the behavior script to the I/O of the PLC, which is done using the Prespective plug-in.

Aid cabinets

Aid cabinets in the traffic tubes are a type of component that provides input signals for the controller. An aid cabinet can be opened and one of the tools in the aid cabinet can be taken out and used. Each of these actions is registered by a sensor and communicated to the controller. Modeling an aid cabinet in the digital twin, as shown in Figure 8.5, provides an interactive model where the user can click on a component to open or use it.

CCTV system

The CCTV system adds an element to the digital twin that can be modeled much more realistically in Unity compared to a 2D visualization. This is because Unity supports cameras that can render to a texture, which can be projected onto a plane. This means that camera GameObjects can be positioned in the scene and the views can be displayed on TV screens in a control room in the digital twin.

This implementation is shown in Figure 8.6 below, where four camera views are shown as can be seen from the control room. On the right-hand side of the

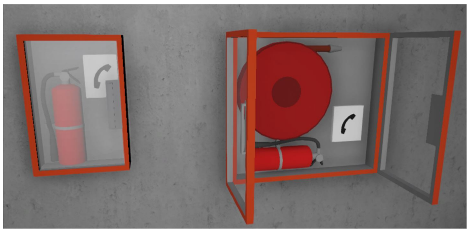


Figure 8.5: Digital twin model of an aid cabinets C (left) and an aid cabinet A (right).

figure, a camera is shown that renders the view on the top right screen. The white thin lines coming from the camera object on the right represent the viewpoint of the CCTV camera.

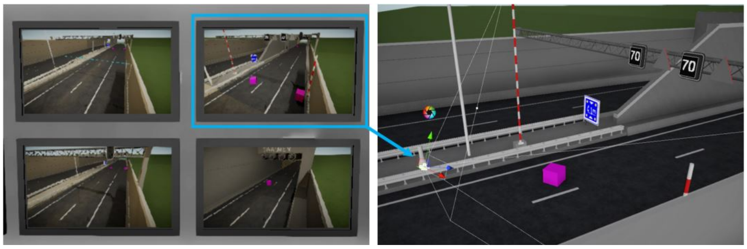


Figure 8.6: Four TV screen views rendered from CCTV cameras in the digital twin (left), and one of the corresponding camera positions in the digital twin (right).

Swalmen tunnel

To give an indication of the fidelity of the digital twin, a comparison between a photo of the real Swalmen tunnel and the same view in the digital twin is shown in Figure 8.7.

8.1.2 Connection to the HIL setup

To use a digital twin in combination with a supervisory controller, the signals of the digital twin must be connected to signals of the controller. In this section, the process of connecting a digital twin to a PLC in a HIL setup is described. Note that, when talking about signals between the digital twin and the PLC, inputs and outputs are defined from the perspective of the PLC.

Figure 8.8 gives an overview of the variable communication in a HIL setup where a digital twin is incorporated. The digital twin is shown in the lowest layer, Layer 1, and contains a set of variables that represent the actuator and sensor signals in the virtual plant. As mentioned before, the Prespective plug-in of Unity

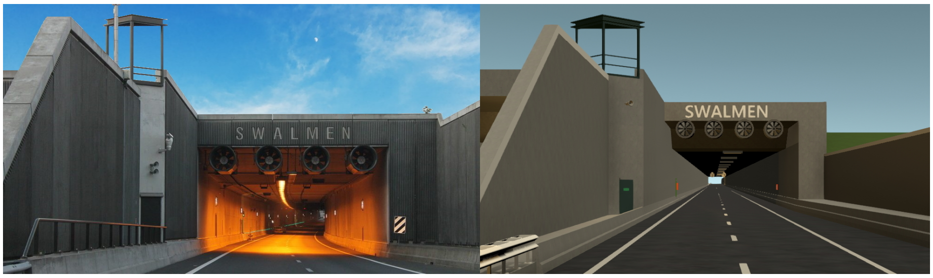


Figure 8.7: Comparison of the entrance of the real Swalmen tunnel (adapted from `http://www.jr-consult.nl/projecten/swalmentunnel-a73-32`) and the entrance in the digital twin.

is used to set up the variable connection. Listing 8.1 contains some examples of these variables related to the actuators and sensors of the boom barrier. In order to set up communication with the PLC, an Ignition OPC tag is created for each variable that is stored on an OPC-UA server, as indicated in Layer 2 in Figure 8.8. The Ignition OPC tags are, in turn, connected to the variables in the PLC, shown in Layer 3. In the case of an ABB PLC, these variables are stored on an OPC-DA server. Layer 4 is similar to Layer 2 as it again uses Ignition OPC tags stored on an OPC-UA server, though in this layer the variables represent data that is important for the GUI. This data includes commands that are sent from the GUI, or information that must be visualized in the GUI. The GUI itself, consisting of buttons and visualizations, is shown in Layer 5, and is created in the Ignition software as explained before in Section 6.3.

8.2 Digital twins for controller validation

One of the goals of combining the possibilities of digital twin simulation with the design process of supervisory controllers is using the digital twin for the purpose of controller validation. For this purpose, the synthesis-based design process shown in Figure 2.1 is adapted to include the digital twin, as can be seen in Figure 8.9. The advantages of this incorporation work both ways: a digital twin is more easily created when an unambiguous description of the plant is available, and a controller can more intuitively and more extensively be validated when a digital twin is available. Both types of advantages are elaborated here.

The first advantage in digital twin creation is that all possible plant behavior is unambiguously defined in models P and P_{H}. Furthermore, this information on the plant behavior can intuitively be obtained by observing 2D model simulations, e.g. how components should move or interact.

The second advantage in digital twin creation is that the I/O connection between the controller and the digital twin can be automated. As the controller

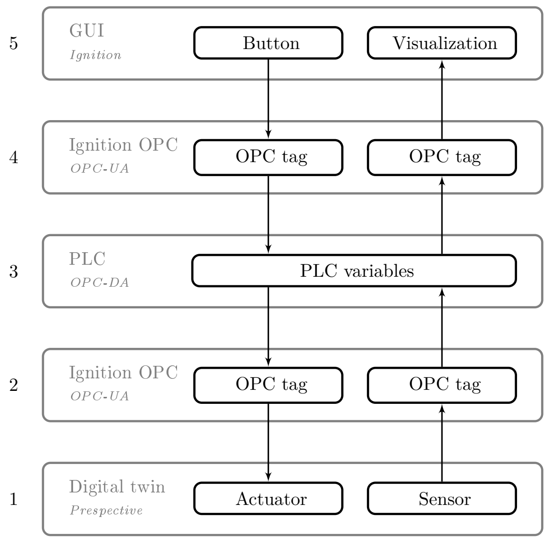


Figure 8.8: Structure of the communication in a HIL setup including a digital twin.

model C is a formal model, event and location names can be imported in the digital twin, thus making the connection process faster and less error-prone.

The first advantage in controller validation is the more intuitive interface for the test engineer. During the controller validation, many different scenarios are inspected in which road user interactions are simulated by the test engineer. Examples of these interactions are cars that trigger speed detection and height detection systems, and people that use emergency cabinets and escape doors. For the test engineer, opening an emergency cabinet by clicking on the door in the 3D visualization is more intuitive than pressing a button in a 2D simulation.

The second advantage in controller validation is the high-fidelity tunnel environment. A test engineer can recognize aspects such as smoke formation and light levels more easily and quickly as they can be simulated more realistically in the digital twin. Furthermore, as moving cars can respond to a closed traffic tube, the closing procedure is more realistic and shows at what point cars stop entering the traffic tube.

The third advantage in controller validation is the possibility to test components in the tunnel that cannot easily be visualized in a 2D simulation. Examples include the sound systems and the cameras in the tunnel. Sound systems need to be tuned correctly to avoid overlapping messages and echoes. In the digital twin, sounds can be played and the sound tuning can therefore be validated more extensively compared to 2D simulations. Cameras are used by the operator to monitor the tunnel. In the 3D digital twin, cameras can be simulated and their recordings can be streamed to a video wall at the operator interface. This way,

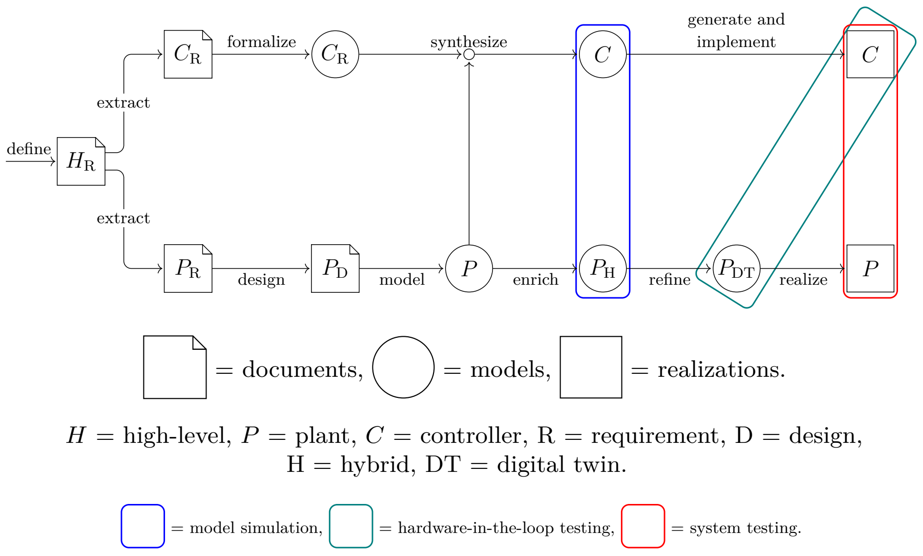

define
H_R
extract
extract
C_R
formalize
C_R
synthesize
C
generate and implement
C
P_R
design
P_D
model
P
enrich
P_H
refine
P_{DT}
realize
P
= documents, = models, = realizations.
H = high-level, P = plant, C = controller, R = requirement, D = design, H = hybrid, DT = digital twin.
= model simulation, = hardware-in-the-loop testing, = system testing.

Figure 8.9: Synthesis-based design process extended with a digital twin.

the test engineer sees the same images that would be visible in the real-world tunnel.

There are also two disadvantages to incorporating digital twins in the synthesis-based design method. The first disadvantage is the data required to create a digital twin. To create an intuitive tunnel environment with high fidelity, the digital twin should closely resemble the (to be) realized tunnel. This means that data of the plant design of the tunnel is required at an early stage in the design process. This is, however, often not a problem, as only a basic design of the plant is needed to create an adequate digital twin, and the digital twin can be updated when a more detailed plant design is available.

A second disadvantage is the additional design effort that is needed to create the digital twin. As the digital twin is more realistic than a 2D simulation model, more effort is involved with creating 3D models and writing behavior scripts for the components. In the category of large systems with complex controllers, to which tunnels belong, the gained advantages outweigh the additional effort that is needed.

8.3 Digital twins for operator training

A second goal of combining digital twin simulation and the design process of supervisory controllers is involving the road tunnel operators. As mentioned in Chapter 3, road tunnel systems are at all times monitored by road tunnel operators that can intervene when deemed necessary. Monitoring happens using the GUI of the road tunnel and through CCTV images. When a road tunnel operator deems it necessary to intervene, the GUI is used to send commands to systems in the tunnel, e.g. to turn up the ventilation or the lighting. Note that the supervisory controller is always active, and will not execute a given command if it contradicts the controller requirements.

As road tunnel operators are the only users of the road tunnel systems, traffic users excluded, it is beneficial to involve them in the supervisory controller design process at an early stage. Practice shows that the road tunnel operators are the most knowledgeable people in terms of supervisory controllers for road tunnels, so incorporating their professional views during the validation process of the supervisory controller increases the quality of the controller. For this purpose, the availability of a digital twin is important, as it provides a high-fidelity simulation model in which the GUI, control room, and CCTV images are as realistic as possible.

Conversely, having a digital twin of a road tunnel also provides benefits in the training process of road tunnel operators. Operators have to be well-trained to understand the functionality of a road tunnel controller, and to be able to execute safety procedures in the case of an emergency. Operator training is thus of high importance, and the availability of a road tunnel digital twin that is connected to an executable supervisory controller facilitates this training process. A first benefit of training using a digital twin is the increased safety, as errors made in the digital twin do not affect the real world tunnel or its traffic users. Second, training in a digital twin provides more and cheaper possibilities for test scenarios. For example, performing a test where fire arises from a car crash and smoke forms in the tunnel is difficult and costly to execute in a real-life tunnel. A similar test in a digital twin is, however, much simpler, safer, and cheaper. A third benefit of training operators using a digital twin is the fact that the real tunnel does not need to be shut down, which increases the availability of the road tunnel system.

8.4 Concluding remarks

This chapter combines synthesis-based design and digital twin development. More specifically, it describes the advantages of creating a digital twin when a plant model is available, and the advantages of using a digital twin in the validation process of the synthesized controller and in the process of operator training.

In the context of validating the synthesized controller, the advantages include the unambiguously defined plant behavior, the automated I/O connection, the intuitive simulation interface, the high-fidelity tunnel environment, and the possibility to test components that cannot easily be visualized in a 2D simulation.

On the other hand, disadvantages are discussed that are mainly related to the additional work that is required to create the digital twin.

This chapter also describes the possibilities of combining a digital twin with an executable supervisory controller for the purpose of operator training. First, these possibilities include the involvement of road tunnel operators in an early design stage of the supervisory controller, which increases the controller quality as their professional views can be applied to the controller design. Second, the availability of a digital twin that is connected to a supervisory controller allows road tunnel operators in training to practice on a digital version of the system. This is safer, as errors made do not affect the realized tunnel, provides more possibilities for testing scenarios, and increases the availability, as the realized tunnel does not need to be shut down for training purposes.

The contributions of this chapter are illustrated in the Swalmen tunnel case study. Future work related to this study includes the steps towards realization, in which components in the digital twin are one by one replaced by physical components to perform the final system tests.

Chapter 9

Concluding remarks

This chapter concludes this thesis. First, the research questions that are posed in Section 1.3 are answered in Section 9.1. Subsequently, several possibilities for future work are discussed in Section 9.2.

9.1 Answers to research questions

In Section 1.3, five research questions are posed related to supervisory controller design for road tunnel systems. These questions are answered in this section.

Research question 1

> *What is a suitable way to model road tunnels and their requirements for the purpose of supervisor synthesis?*

Practice has shown that the input and output interfaces of the supervisory controller, both at the side of the plant and the side of the operator interface, give a good indication of the abstraction level that should be used in the modeling of the components and the requirements. In this abstraction level, a separate automaton model is created for each input signal, i.e. sensor, and each output signal, i.e. actuator. This modeling approach is called component-based modeling, and is described in Section 4.1. One of the benefits of component-based modeling is that many of the modeled component become loosely coupled, since each automaton represents an individual signal. This makes the modeling of the components and the requirements more intuitive, and is therefore faster and requires less effort. The loosely coupled components also allow for multiple other advantages in the supervisory controller design process, such as parameter-based modeling described in Section 4.2, model reduction described in Chapter 5, and implementation of distributed supervisory controllers described in Chapter 7.

Each of these advantages is covered in more detail in the answers to the subsequent research questions.

Several case studies have been performed and described throughout this thesis to show the modeling process for the purpose of synthesizing a road tunnel supervisor. These tunnels include the Koning Willem-Alexander tunnel in Maastricht, the Eerste Heinenoord tunnel near Rotterdam, and the Swalmen tunnel near Swalmen.

Research question 2

> *How can the characteristics of road tunnels be exploited in the synthesis-based engineering process?*

The road tunnel systems that are the considered type of system in this thesis have certain characteristics compared to other types of infrastructural systems. First, they are large infrastructural system with numerous sensors and actuators. Second, they have a high degree of symmetry, both in the repeating components throughout a traffic tube and in the resemblance between traffic tubes. These characteristics are exploited in two steps in the synthesis-based design process.

The first step where the characteristics of road tunnels are exploited is in the modeling step of the components and the requirements. In Section 4.2, the parameter-based modeling method is introduced. This method allows the design engineer to specify the tunnel characteristics in terms of parameters, and subsequently automatically generate the files that are required for synthesis and model simulation. These parameters describe the number of modules and components in the tunnel, such as the number of traffic tubes and the number of escape doors. A case study pertaining a family of 22 tunnels in the Netherlands shows the applicability of the parameter-based modeling method.

The second step where the characteristics are exploited is in the synthesis step. Chapter 5 introduces the model reduction process for supervisor synthesis, where symmetry in the plant model and the requirements model is used to remove part of the control problem before performing synthesis. Dependency graphs are described as a means to visualize the components in a control problem and the dependencies between them. These dependency graphs are subsequently used to identify symmetric components that are eligible for model reduction. The model reduction process is showcased in a set of case studies described in Section 5.4.

Research question 3

> *Is it possible to synthesize a supervisor for large-scale systems like road tunnels?*

As mentioned before, road tunnels are systems with a large number of components. Moreover, there are many, possibly complicated, dependencies between these components. Case studies, such as the Eerste Heinenoord tunnel described in Subsection 5.4.1, show that it can take hours to solve the synthesis problem for a road tunnel, and that sometimes it is even impossible to solve due to computational memory constraints. The model reduction process introduced in Chapter 5 solves this problem by removing part of the plant model and the requirements model, which alleviates the synthesis problem. Five different model reduction steps are described in Section 5.2 that exploit a (symmetrical) characteristic in the dependency graph. In the case of the Eerste Heinenoord tunnel, the synthesis problem is simplified from being unsolvable to being solvable within 2 minutes.

While the model reduction steps are inspired by the symmetric characteristics of road tunnel systems, several other case studies are performed for a range of industrial systems to investigate the applicability of the model reduction steps. Specifically, the model reduction process has been applied to a production line, a waterway lock, and a movable bridge. In each of these case studies, the symmetry reduction steps are applicable to a certain degree, thus simplifying the synthesis problem.

Research question 4

How can a supervisory controller correctly be synthesized for the purpose of implementation on multiple PLCs?

As road tunnels are typically controlled by multiple PLCs, a method is needed to synthesize a set of distributed supervisory controllers without losing the properties that are guaranteed by supervisor synthesis. The main challenge in this method follows from the communication delays that occur when information is shared between PLCs. In the case that PLCs are not synchronized, i.e. the read-calculate-write cycles of the different PLCs are not synchronized, one local controller can calculate what actions its should perform while important information in a communicated message from a different PLC is still underway. In Chapter 7, a method is described for obtaining and implementing a distributed supervisory controller.

The first step in this method is distributing the system to determine which components in the system are controlled by which PLC. This is determined using DSMs, in which dependencies are mapped between components. By clustering the DSM, groups of components are identified in such a way that the number of dependencies between components within a group is maximized and the number of dependencies between components of different groups is minimized. This approach minimizes the required communication between the PLCs.

After the system distribution has been determined, local supervisors are created for each group in the distribution. This is done by synthesizing a global supervisor and subsequently localizing it for each local supervisor. This way the safety, controllability, and nonblockingness properties are upheld.

In the third step, the effects of communication delays are analyzed. A delay-robustness property and check are defined that can be used to determine if a pair of events is unaffected by communication delays. An algorithm for this check has been implemented, and is used to determine which event pairs are not delay robust. For these event pairs, mutual exclusion algorithms are implemented to enforce delay robustness.

The final step in the method is generating PLC code for each of the local supervisors, and implementing the code of each local supervisor on the intended PLC.

In a case study for the Swalmen tunnel, a distributed supervisory controller has been obtained using the proposed method. The supervisory controllers are implemented on a HIL setup consisting of three PLCs to validate the controlled behavior of each controller and the communication between them.

Research question 5

How can a supervisory controller for a road tunnel be synthesized, implemented, and tested?

The complete process of designing a supervisory controller for a road tunnel consists of the steps of synthesis, implementation, and testing. The synthesis step is extensively covered in Research question 2. In Chapter 6, the process of deriving an implementable controller and testing it in a HIL setup is described.

Several aspects of deriving an implementable controller for a road tunnel are highlighted in Section 6.2. First, the concept of resource controllers is introduced to deploy part of the local control for a specific component on a separate PLC. This results in smaller and clearer controllers, thus making the synthesis procedure to create these controllers less computationally intensive. Second, the existing checks for the properties of an implementable controller are too strict for a road tunnel controller. Therefore, a relaxation has been proposed for the confluence property called end-state equality. Third, a new automatic PLC code generation algorithm has been developed and implemented that is more easily adaptable for new target platforms. Fourth, DSM sequencing is used for the purpose of PLC code optimization, where the order in which events are handled during a PLC cycle is optimized to minimize rework.

To test whether the controlled behavior of an implemented controller is as intended, a HIL setup is used. This setup consists of one or more PLCs on which the generated PLC code is implemented. The PLCs are connected to two PCs that contain the operator interface and virtual plant, respectively. This way, simulations can be performed where the PLC operating semantics is used, where external subsystems like the operator interface are connected, and where the PLC performance can be measured. Moreover, as explained in Chapter 8, a digital twin can be used as the virtual plant to provide a more intuitive and realistic simulation environment.

The applicability of the controller derivation, implementation, and testing process is shown in a case study for the Swalmen tunnel.

9.2 Future work

In this thesis, the process of supervisory controller design with an application to road tunnels is described, including the steps of modeling, synthesis, simulation, implementation, and HIL testing. While the applicability of synthesis-based engineering for road tunnels supervisory controllers has been demonstrated throughout the various case studies in this thesis, there are several ways to extend this research. Some possible extensions are described shortly below.

Extension of parameter-based modeling

In Chapter 4, the process of parameter-based modeling is introduced and a prototype is implemented as a configuration tool to automatically generate the files required for synthesis and simulation. A possible extension to parameter-based modeling is to include generation of files for HIL testing, digital twins, and supervisor distribution.

Algorithms for automatic model reduction

Model reduction steps for supervisor synthesis and the process of subsequently restoring the model are extensively described in Chapter 5. To increase the applicability of the model reduction and restoration processes, algorithms can be defined and implemented to automatically detect when model reduction steps are applicable, and reduce and restore the model accordingly.

Implementation of automatic PLC code optimization

Subsection 6.2.4 describes how PLC code is optimized through DSM sequencing. An algorithm is implemented to automatically sequence a DSM, though the resulting event order is applied to the generated PLC code manually. By incorporating the DSM sequencing algorithm in the PLC code generation algorithm, this process can be fully automated.

Bottom-up methods for distributed supervisory controllers

The main disadvantage of the localization approach used in the process of synthesizing a distributed supervisor, as explained in Chapter 7, is the need to first synthesize a global supervisor. For large-scale systems, this can be computationally intensive or even unfeasible. Future research can look into using bottom-up methods for synthesizing local supervisors. In such an approach, the system is distributed before the synthesis procedure, and local supervisors are synthesized according to this distribution. A drawback of such an approach is the difficulty to give global guarantees such as global nonblockingness.

Integration and system testing

The final step in the supervisory controller design process is to integrate the implemented controller in the realized system. While all errors related to the supervisory controller and its interfaces should have been found during the simulation and HIL testing phases, a final system test can be performed to validate whether the integrated controller works as intended. This final testing stage has not been performed for a road tunnel system due to the unavailability of a vacant tunnel.

Fault-tolerant control

In the supervisory controller design process described in this thesis, the nominal behavior of the plant is modeled, i.e. the assumption is made that the plant components never fail or break down. While the supervisory controller cannot prevent a component from failing, it is able to change its controlled behavior when a fault has been detected by adapting the controller requirements based on detected faults. Such a controller, called a fault-tolerant controller, adapts to a detected fault in a specified way. For instance, when one ventilation unit in a traffic tube breaks down, another unit might be set to a higher mode to compensate for it. Synthesis of fault-tolerant controllers for infrastructural systems has been investigated in previous works such as Reijnen et al. (2021), and future research can investigate its applicability to road tunnel systems.

Industrial PC-based control

The implementation process described in Chapter 6 focuses on implementing a synthesized supervisor on a PLC. As described in Subsection 2.5.1, PLCs work in cycles of reading inputs, executing enabled events, and writing output. Based on this cycle-based semantics, properties for an implementable controller have been defined, described in Subsection 2.5.2. Besides implementation on a PLC, a supervisory controller can also be implemented on an industrial personal computer (IPC). The semantics of an IPC differs from the semantics of a PLC, in the sense that IPCs work interrupt-based. In future work, implementation of synthesized supervisors on IPCs can be investigated, such as described in Brandin (1996), and a different set of properties for an implementable supervisor can be proposed.

Bibliography

A2Maastricht. Deze tunnel gaat ons overleven [This tunnel is going to outlive us] (in Dutch), 2018. URL `https://a2maastricht.nl/application/files/7715/2162/4105/VTTI-1.pdf`.

Alves, M. V., Carvalho, L. K., and Basilio, J. C. Supervisory control of networked discrete event systems with timing structure. *IEEE Transactions on Automatic Control*, 66(5):2206–2218, 2020.

Atorf, L. and Roßmann, J. Interactive analysis and visualization of digital twins in high-dimensional state spaces. In *2018 15th International Conference on Control, Automation, Robotics and Vision (ICARCV)*, pages 241–246. IEEE, 2018.

Baeten, J., Mortel-Fronczak, J. M., and Rooda, J. E. Integration of supervisory control synthesis in model-based systems engineering. In *Complex systems*, pages 39–58. Springer, 2016.

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

Borg, A., Bjelland, H., and Njå, O. Reflections on Bayesian network models for road tunnel safety design: A case study from Norway. *Tunnelling and Underground Space Technology*, 43:300–314, 2014.

Boyer, S. A. *SCADA: supervisory control and data acquisition.* International Society of Automation, 2009.

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

Bullock, D., Johnson, B., Wells, R. B., Kyte, M., and Li, Z. Hardware-in-the-loop simulation. *Transportation Research Part C: Emerging Technologies*, 12(1): 73–89, 2004.

Cai, K. and Wonham, W. M. Supervisor localization: a top-down approach to distributed control of discrete-event systems. *IEEE Transactions on Automatic Control*, 55(3):605–618, 2010.

Cassandras, C. G. and Lafortune, S. *Introduction to discrete event systems.* Springer, 2008.

Centrum Ondergronds Bouwen. Groeiboek - Digitaal aantonen [Improvement book - Digital demonstration] (in Dutch), 2021. Available at `https://www.cob.nl/wat-doet-het-cob/groeiboek/digitaal-aantonen/`.

Chen, Y. L. and Lin, F. Hierarchical modeling and abstraction of discrete event systems using finite state machines with parameters. In *40th International Conference on Decision and Control*, pages 4110–4115. IEEE, 2001.

Cheng, K.-T. and Krishnakumar, A. S. Automatic functional test generation using the extended finite state machine model. In *30th ACM/IEEE Design Automation Conference*, pages 86–91. IEEE, 1993.

Chopra, A. *Introduction to Google Sketchup.* John Wiley & Sons, 2012.

Cooper, K. D. and Torczon, L. *Engineering a compiler.* Elsevier, 2011.

da Cunha, A. E. C. and Cury, J. E. R. Hierarchical supervisory control based on discrete event systems with flexible marking. *IEEE Transactions on Automatic Control*, 52(12):2242–2253, 2007.

Dai, W., Zhou, P., Zhao, D., Lu, S., and Chai, T. Hardware-in-the-loop simulation platform for supervisory control of mineral grinding process. *Powder technology*, 288:422–434, 2016.

De Queiroz, M. H. and Cury, J. E. R. Modular control of composed systems. In *Proceedings of the 2000 American Control Conference (ACC)*, volume 6, pages 4051–4055. IEEE, 2000.

de Queiroz, M. H. and Cury, J. E. R. Modular supervisory control of large scale discrete event systems. In *Discrete Event Systems*, pages 103–110. Springer, 2000.

Diestel, R. *Graph Theory (Graduate Texts in Mathematics).* Springer, 2017.

Dijkstra, E. W. Solution of a Problem in Concurrent Programming Control. *Communications of the ACM*, 8:569, 1965. doi: 10.1007/978-3-642-59412-0_20.

Eppinger, S. D. and Browning, T. R. *Design structure matrix methods and applications.* MIT press, 2012.

Estefan, J. A. Survey of model-based systems engineering (MBSE) methodologies. *Incose MBSE Focus Group*, 25(8):1–12, 2007.

European Commission. Road Tunnels, 2021. URL `https://ec.europa.eu/transport/road_safety/eu-road-safety-policy/priorities/infrastructure/road-tunnels_en`.

Eyzell, J. M. and Cury, J. E. R. Exploiting symmetry in the synthesis of supervisors for discrete event systems. *IEEE Transactions on Automatic Control*, 46(9):1500–1505, 2001.

Fabian, M. and Hellgren, A. PLC-based implementation of supervisory control for discrete event systems. In *Proceedings of the 37th IEEE Conference on Decision and Control*, volume 3, pages 3305–3310. IEEE, 1998.

Fabian, M., Fei, Z., Miremadi, S., Lennartson, B., and Åkesson, K. Supervisory control of manufacturing systems using extended finite automata. In *Formal Methods in Manufacturing*, pages 295–314. CRC Press, 2018.

Forschelen, S. T. J., van de Mortel-Fronczak, J. M., Su, R., and Rooda, J. E. Application of supervisory control theory to theme park vehicles. *Discrete Event Dynamic Systems*, 22(4):511–540, 2012.

Forssell, H., Kharlamov, E., and Thorstensen, E. On equivalence and cores for incomplete databases in open and closed worlds. *arXiv preprint arXiv:2001.04757*, 2020.

Glushkov, V. M. The abstract theory of automata. *Russian Mathematical Surveys*, 16(5):1–53, 1961.

Goorden, M. A. *Supervisory control synthesis for large-scale infrastructural systems*. PhD thesis, Eindhoven University of Technology, 2019.

Goorden, M. A. and Fabian, M. No synthesis needed, we are alright already. In *2019 IEEE 15th International Conference on Automation Science and Engineering (CASE)*, pages 195–202. IEEE, 2019.

Goorden, M. A., van de Mortel-Fronczak, J. M., Reniers, M. A., Fokkink, W. J., and Rooda, J. E. Structuring multilevel discrete-event systems with dependency structure matrices. *IEEE Transactions on Automatic Control*, 65(4):1625–1639, 2019a.

Goorden, M. A., van de Mortel-Fronczak, J. M., Reniers, M. A., Fokkink, W. J., and Rooda, J. E. Modeling guidelines for component-based supervisory control synthesis. In *International Conference on Formal Aspects of Component Software*, pages 3–24. Springer, 2019b.

Goorden, M. A., Moormann, L., Reijnen, F. F. H., Verbakel, J. J., van Beek, D. A., Hofkamp, A. T., van de Mortel-Fronczak, J. M., Reniers, M. A., Fokkink, W. J., Rooda, J. E., and Etman, L. F. P. The road ahead for supervisor synthesis. In *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*, pages 1–16. Springer, 2020.

Goorden, M. A., van de Mortel-Fronczak, J. M., Reniers, M. A., Fabian, M., Fokkink, W. J., and Rooda, J. E. Model properties for efficient synthesis of nonblocking modular supervisors. *Control Engineering Practice*, 112:104830, 2021.

Gössler, G. and Sifakis, J. Composition for component-based modeling. *Science of Computer Programming*, 55(1-3):161–183, 2005.

Grigorov, L. and Rudie, K. Techniques for the parametrization of discrete-event system templates. *IFAC Proceedings Volumes*, 43(12):370–375, 2010.

Grigorov, L., Butler, B. E., Cury, J. E. R., and Rudie, K. Conceptual design of discrete-event systems using templates. *Discrete Event Dynamic Systems: Theory and Applications*, 21(2):257–303, 2011.

Hardin, B. and McCool, D. *BIM and construction management: proven tools, methods, and workflows.* John Wiley & Sons, 2015.

Harland, P. E., Uddin, Z., and Laudien, S. Product platforms as a lever of competitive advantage on a company-wide level: a resource management perspective. *Review of Managerial Science*, 14(1):137–158, 2020.

Henzinger, T. A. The theory of hybrid automata. In *Verification of digital and hybrid systems*, pages 265–292. Springer, 2000.

Huang, Y., Seck, M. D., and Verbraeck, A. Component-based light-rail modeling in discrete event systems specification (DEVS). *Simulation*, 91(12):1027–1051, 2015.

International Electrotechnical Commission. *IEC 61131-3: Programmable Controllers – Part 3: Programming Languages (3rd).* Standard, 2013.

Jiao, T., Gan, Y., Xiao, G., and Wonham, W. M. Exploiting symmetry of state tree structures for discrete-event systems with parallel components. *International Journal of Control*, 90(8):1639–1651, 2017.

Juerd. Tunnels in Nederland [Tunnels in the Netherlands] (in Dutch), 2020. `https://www.wegenwiki.nl/Categorie:Tunnels_in_Nederland`.

Kalyon, G., Le Gall, T., Marchand, H., and Massart, T. Synthesis of communicating controllers for distributed systems. In *2011 50th IEEE Conference on Decision and Control and European Control Conference*, pages 1803–1810. IEEE, 2011.

Kalyon, G., Le Gall, T., Marchand, H., and Massart, T. Symbolic supervisory control of distributed systems with communications. *IEEE Transactions on Automatic Control*, 59(2):396–408, 2013.

Kanrar, S., Chaki, N., and Chattopadhyay, S. *Concurrency Control in Distributed System Using Mutual Exclusion.* Springer, 2018.

Kensek, K. M. *Building information modeling.* Routledge, 2014.

Komenda, J., Masopust, T., and van Schuppen, J. H. Control of an engineering-structured multilevel discrete-event system. In *2016 13th International Workshop on Discrete Event Systems (WODES)*, pages 103–108. IEEE, 2016.

Komenda, J., Masopust, T., and van Schuppen, J. H. Maximal permissiveness of modular supervisory control via multilevel structuring. *IFAC-PapersOnLine*, 53(2):2116–2121, 2020.

Korssen, T., Dolk, V. S., van de Mortel-Fronczak, J. M., Reniers, M. A., and Heemels, W. P. M. H. Systematic model-based design and implementation of supervisors for advanced driver assistance systems. *IEEE Transactions on Intelligent Transportation Systems*, 19(2):533–544, 2017.

Kostenko, K. V. Real-time communication between robot PLC and PC over Ethernet-based protocols. *arXiv preprint arXiv:1902.01924*, 2019.

Kovács, G., Piétrac, L., and Bálint, K. A component-based approach for supervisory control. In *2012 20th Mediterranean Conference on Control & Automation (MED)*, pages 800–805. IEEE, 2012.

Kumar, S. A. P., Madhumathi, R., Chelliah, P. R., Tao, L., and Wang, S. A novel digital twin-centric approach for driver intention prediction and traffic congestion avoidance. *Journal of Reliable Intelligent Environments*, 4(4):199–209, 2018.

Lafortune, S. On decentralized and distributed control of partially-observed discrete event systems. *Advances in control theory and applications*, pages 171–184, 2007.

Lee, E. A. Cyber physical systems: Design challenges. In *2008 11th IEEE international symposium on object and component-oriented real-time distributed computing (ISORC)*, pages 363–369. IEEE, 2008.

Li, W., Joós, G., and Bélanger, J. Real-time simulation of a wind turbine generator coupled with a battery supercapacitor energy storage system. *IEEE Transactions on Industrial Electronics*, 57(4):1137–1145, 2009.

Lidström, M. Using advanced driving simulator as design tool in road tunnel design. *Transportation research record*, 1615(1):51–55, 1998.

Lin, F. and Wonham, W. M. Decentralized control and coordination of discrete-event systems with partial observation. *IEEE Transactions on Automatic Control*, 35(12):1330–1337, 1990.

Liu, Z., Yin, X., Shu, S., Lin, F., and Li, S. Online supervisory control of networked discrete-event systems with control delays. *IEEE Transactions on Automatic Control*, 2021.

Majdara, A. and Wakabayashi, T. Component-based modeling of systems for automated fault tree generation. *Reliability Engineering & System Safety*, 94 (6):1076–1086, 2009.

Malik, P. From supervisory control to nonblocking controllers for discrete event systems, 2003. PhD thesis. Universität Kaiserslautern.

Malik, R., Fabian, M., and Åkesson, K. Modelling large-scale discrete-event systems using modules, aliases, and extended finite-state automata. *IFAC-PapersOnline*, 44(1):7000–7005, 2011.

Markovski, J., van Beek, D. A., Theunissen, R. J. M., Jacobs, K. G. M., and Rooda, J. E. A state-based framework for supervisory control synthesis and verification. In *49th Conference on Decision and Control*, pages 3481–3486. IEEE, 2010.

Mealy, G. H. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, 1955.

Meier, C., Yassine, A. A., and Browning, T. R. Design process sequencing with competent genetic algorithms. 2007.

Meng, Z., Tang, T., Wei, G., and Yuan, L. Digital twin based comfort scenario modeling of ATO controlled train. In *Journal of Physics: Conference Series*, volume 1654, page 012071. IOP Publishing, 2020.

Meyer, M. H. and Lehnerd, A. P. *The power of product platforms.* Simon and Schuster, 1997.

Miller, A., Donaldson, A., and Calder, M. Symmetry in temporal logic model checking. *ACM Computing Surveys (CSUR)*, 38(3):8–es, 2006.

Min, S. Y., Kim, T. K., Lee, J. S., and Einstein, H. H. Design and construction of a road tunnel in Korea including application of the decision aids for tunneling - a case study. *Tunnelling and Underground Space Technology*, 23(2):91–102, 2008.

Minch, R. P. and Burns, J. R. Conceptual design of decision support systems utilizing management science models. *IEEE Transactions on Systems, Man, and Cybernetics*, (4):549–557, 1983.

Miremadi, S., Åkesson, K., and Lennartson, B. Symbolic computation of reduced guards in supervisory control. *IEEE Transactions on Automation Science and Engineering*, 8(4):754–765, 2011.

Murata, T. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

Nicoll, B. and Keogh, B. *The Unity game engine and the circuits of cultural software*. Springer, 2019.

Nooijens, C. Potential application of the digital twin in the exploitation phase of infrastructural projects, 2020. BSc Thesis, Available at `https://www.cob.nl/document/potential-digital-twin-in-exploitation-phase/`.

Norris, I. C., Dill, D. L., et al. Better verification through symmetry. *Formal methods in system design*, 9(1):41–75, 1996.

Ouedraogo, L., Kumar, R., Malik, R., and Åkesson, K. Nonblocking and safe control of discrete-event systems modeled as extended finite automata. *IEEE Transactions on Automation Science and Engineering*, 8(3):560–569, 2011.

Overstreet, C. M. and Nance, R. E. A specification language to assist in analysis of discrete event simulation models. *Communications of the ACM*, 28(2):190–201, 1985.

Pimmler, T. U. and Eppinger, S. D. Integration analysis of product decompositions. In *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, volume 12822, pages 343–351. American Society of Mechanical Engineers, 1994.

Pires, F., Cachada, A., Barbosa, J., Moreira, A. P., and Leitão, P. Digital twin in industry 4.0: Technologies, applications and challenges. In *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*, volume 1, pages 721–726. IEEE, 2019.

Prenzel, L. and Provost, J. PLC implementation of symbolic, modular supervisory controllers. *IFAC-PapersOnLine*, 51(7):304–309, 2018.

Prespective. Prespective 2020.1 - Digital Twin Software for Unity, 2021. Available at `https://prespective-software.com/`.

Quint, A. Scalable vector graphics. *IEEE MultiMedia*, 10(3):99–102, 2003.

Ramadge, P. J. and Wonham, W. M. Supervisory control of a class of discrete event processes. *SIAM journal on control and optimization*, 25(1):206–230, 1987.

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

Ramos, A. L., Ferreira, J. V., and Barceló, J. Model-based systems engineering: An emerging approach for modern systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(1):101–111, 2011.

Rashidinejad, A., Reniers, M. A., and Feng, L. Supervisory control of timed discrete-event systems subject to communication delays and non-FIFO observations. *IFAC-PapersOnLine*, 51(7):456–463, 2018.

Raynal, M. *Distributed algorithms for message-passing systems*, volume 500. Springer, 2013.

Reijnen, F. F. H. *Putting supervisor synthesis to work.* PhD thesis, Eindhoven University of Technology, 2020.

Reijnen, F. F. H., Leliveld, E.-B. M. L., van de Mortel-Fronczak, J. M., van Dinther, J., Rooda, J. E., and Fokkink, W. J. Synthesized fault-tolerant supervisory controllers, with an application to a rotating bridge. *Computers in Industry*, 130:103473, 2021.

Reijnen, F. F. H., Goorden, M. A., van de Mortel-Fronczak, J. M., Reniers, M. A., and Rooda, J. E. Application of dependency structure matrices and multilevel synthesis to a production line. In *2018 IEEE Conference on Control Technology and Applications (CCTA)*, pages 458–464. IEEE, 2018.

Reijnen, F. F. H., Hofkamp, A. T., van de Mortel-Fronczak, J. M., Reniers, M. A., and Rooda, J. E. Finite response and confluence of state-based supervisory controllers. In *2019 IEEE 15th International Conference on Automation Science and Engineering (CASE)*, pages 509–516. IEEE, 2019a.

Reijnen, F. F. H., Verbakel, J. J., van de Mortel-Fronczak, J. M., and Rooda, J. E. Hardware-in-the-loop set-up for supervisory controllers with an application: the Prinses Marijke complex. In *2019 IEEE Conference on Control Technology and Applications (CCTA)*, pages 843–850. IEEE, 2019b.

Reijnen, F. F. H., Goorden, M. A., van de Mortel-Fronczak, J. M., and Rooda, J. E. Modeling for supervisor synthesis–a lock-bridge combination case study. *Discrete Event Dynamic Systems*, 30(3):499–532, 2020a.

Reijnen, F. F. H., van de Mortel-Fronczak, J. M., Reniers, M. A., and Rooda, J. E. Design of a supervisor platform for movable bridges. In *2020 IEEE 16th International Conference on Automation Science and Engineering (CASE)*, pages 1300–1306. IEEE, 2020b.

Rijkswaterstaat. De tunnel van de toekomst [The tunnel of the future] (in Dutch), 2020. URL `https://www.magazinesrijkswaterstaat.nl/zakelijkeninnovatie/2020/04/renovatie-heinenoordtunnel`.

Rijkswaterstaat. Landelijke Tunnelstandaard [National Tunnel Standard] (in Dutch), 2021a. URL `https://www.rijkswaterstaat.nl/zakelijk/werken-aan-infrastructuur/bouwrichtlijnen-infrastructuur/aanleg-tunnels/landelijke-tunnelstandaard`.

Rijkswaterstaat. Vernieuwen van bruggen, tunnels, sluizen en viaducten [Renewal of bridges, tunnels, locks and overpasses] (in Dutch), 2021b. URL `https://www.rijkswaterstaat.nl/over-ons/onze-organisatie/vervanging-en-renovatie`.

Rijkswaterstaat. Tunnels in beheer Rijkswaterstaat [Tunnels managed by Rijkswaterstaat] (in Dutch), 2022. URL `https://www.rijkswaterstaat.nl/wegen/wegbeheer/tunnels#tunnels-in-beheer-rijkswaterstaat`.

Rohloff, K. and Lafortune, S. Symmetry reductions for a class of discrete-event systems. In *2004 43rd IEEE Conference on Decision and Control (CDC)(IEEE Cat. No. 04CH37601)*, volume 1, pages 38–44. IEEE, 2004.

Rohrer, M. W. Seeing is believing: the importance of visualization in manufacturing simulation. In *2000 Winter Simulation Conference Proceedings*, volume 2, pages 1211–1216. IEEE, 2000.

Rudie, K. and Wonham, W. M. Think globally, act locally: Decentralized supervisory control. In *1991 American Control Conference*, pages 898–903. IEEE, 1991.

Schranz, C., Strohmeier, F., and Damjanovic-Behrendt, V. A digital twin prototype for product lifecycle data management. In *2020 IEEE/ACS 17th International Conference on Computer Systems and Applications (AICCSA)*, pages 1–6. IEEE, 2020.

Shu, S. and Lin, F. Supervisor synthesis for networked discrete event systems with communication delays. *IEEE Transactions on Automatic Control*, 60(8): 2183–2188, 2014.

Sköldstam, M., Åkesson, K., and Fabian, M. Modeling of discrete event systems using finite automata with variables. In *46th International Conference on Decision and Control*, pages 3387–3392. IEEE, 2007.

Steward, D. V. The design structure system: A method for managing the design of complex systems. *IEEE Transactions on Engineering Management*, (3): 71–74, 1981.

Su, R. Coordinated distributed time optimal supervisory control. In *2013 American Control Conference (ACC)*, pages 905–910. IEEE, 2013.

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

Swartjes, L., van Beek, D. A., and Reniers, M. A. Towards the removal of synchronous behavior of events in automata. *IFAC Proceedings Volumes*, 47 (2):188–194, 2014.

Taipale, O., Kasurinen, J., Karhu, K., and Smolander, K. Trade-off between automated and manual software testing. *International Journal of System Assurance Engineering and Management*, 2(2):114–125, 2011.

Tijs, K. Digital tunnel twin: Enriching the maintenance & operation of Dutch tunnels, 2020. MSc Thesis, Available at `https://www.cob.nl/document/digital-tunnel-twin/`.

Van Dongen, S. Graph clustering via a discrete uncoupling process. *SIAM Journal on Matrix Analysis and Applications*, 30(1):121–141, 2008.

van Hegelsom, J. Development of a 3D digital twin of the Swalmen Tunnel in the Rijkswaterstaat project, 2021. BSc Thesis, Available at `https://arxiv.org/abs/2107.12108`.

Verbakel, J. J., Vos de Wael, M. E. W., van de Mortel-Fronczak, J. M., Fokkink, W. J., and Rooda, J. E. A configurator for supervisory controllers of roadside systems. In *2021 IEEE 17th International Conference on Automation Science and Engineering (CASE)*, pages 784–791. IEEE, 2021.

Vieira, A. D., Santos, E. A. P., de Queiroz, M.-H., Leal, A. B., de Paula Neto, A. D., and Cury, J. E. R. A method for PLC implementation of supervisory control of discrete event systems. *IEEE Transactions on Control Systems Technology*, 25(1):175–191, 2016.

Wang, W., Zhang, T., Yan, Z., Yu, M., and Gong, C. Dynamic event observation and partial state-change perception for distributed supervisory control. In *2020 European Control Conference (ECC)*, pages 644–649. IEEE, 2020.

Weber, M. and Weisbrod, J. Requirements engineering in automotive development-experiences and challenges. In *Proceedings IEEE Joint International Conference on Requirements Engineering*, pages 331–340. IEEE, 2002.

Wilschut, T. *System specification and design structuring methods for a lock product platform*. PhD thesis, Eindhoven University of Technology, 2018.

Wong, K. C., Thistle, J. G., Malhamé, R. P., and Hoang, H.-H. Supervisory control of distributed systems: Conflict resolution. *Discrete Event Dynamic Systems*, 10(1):131–186, 2000.

Wonham, W. M., Cai, K., and Rudie, K. Supervisory control of discrete-event systems: A brief history. *Annual Reviews in Control*, 45:250–256, 2018.

Wonham, W. M. and Ramadge, P. J. Modular supervisory control of discrete-event systems. *Mathematics of control, signals and systems*, 1(1):13–30, 1988.

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

Zaytoon, J. and Riera, B. Synthesis and implementation of logic controllers – A review. *Annual Reviews in Control*, 43(1):152–168, 2017.

Zhang, R. and Cai, K. On supervisor localization based distributed control of discrete-event systems under partial observation. In *2016 American Control Conference (ACC)*, pages 764–769. IEEE, 2016.

Zhang, R., Cai, K., Gan, Y., and Wonham, W. M. Distributed supervisory control of discrete-event systems with communication delay. *Discrete Event Dynamic Systems*, 26(2):263–293, 2016.

Zhong, H. and Wonham, W. M. On the consistency of hierarchical supervision in discrete-event systems. *IEEE Transactions on Automatic Control*, 35(10): 1125–1134, 1990.

Zhu, Y., Lin, L., Ware, S., and Su, R. Supervisor synthesis for networked discrete event systems with communication delays and lossy channels. In *2019 IEEE 58th Conference on Decision and Control (CDC)*, pages 6730–6735. IEEE, 2019.

Zhu, Y., Lin, L., Tai, R., and Su, R. Supervisor synthesis for networked discrete event systems with delays against non-fifo communication channels. In *2020 16th International Conference on Control, Automation, Robotics and Vision (ICARCV)*, pages 1027–1032. IEEE, 2020.