Putting supervisor synthesis to work

Putting supervisor synthesis to work

Controller software generation for infrastructural systems

Ferdie Reijnen

Department of Mechanical Engineering
EINDHOVEN UNIVERSITY OF TECHNOLOGY
Eindhoven, The Netherlands 2020

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: 978-90-386-5113-2

Typeset using LaTeX.
Printed by: Gildeprint
Cover photo: Algera complex [beeldbank.rws.nl, Rijkswaterstaat / Joop van Houdt]

Putting supervisor synthesis to work

Controller software generation for infrastructural systems

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 12 november 2020 om 13:30 uur

door

Ferdie Ferdinand Henricus Reijnen

geboren te Venlo

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

voorzitter:	prof.dr. H. Nijmeijer
1e promotor:	prof.dr. W.J. Fokkink
copromotoren:	dr.ir. J.M. van de Mortel-Fronczak
	dr.ir. M.A. Reniers
leden:	prof.dr. S. Lafortune (University of Michigan)
	dr. E. Rutten (INRIA Grenoble - Rhône-Alpes)
	prof.dr.ir. J.P.M. Voeten
adviseur:	ir. W.H.M. van der Vegt (Rijkswaterstaat)

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

Summary

In the coming decades, numerous locks and bridges in the Netherlands have to be renovated or replaced, as they reach their end-of-life or have capacity problems. In the past, these infrastructural systems have been engineered, built, and maintained on a project basis, resulting in a variety of unique solutions to almost the same engineering problem. This uniqueness has proven to be disadvantageous for quality, evolvability, and life-cycle costs. For future infrastructural systems, Rijkswaterstaat, part of the Dutch Ministry of Infrastructure and Water Management, is seeking methods for modularization and standardization to increase quality, increase evolvability, and decrease life-cycle costs. The supervisory controller of an infrastructural object, which controls and monitors its components, is identified as one of the subsystems having a significant impact on these three aspects.

The aim of this thesis is to investigate the applicability of supervisory control theory (SCT) for the design of supervisory controllers for locks and bridges. SCT is concerned with systems that are discrete in time and state space, and where the behavior is driven by instantaneous events. SCT provides techniques to automatically derive supervisors, called supervisor synthesis. For supervisor synthesis, a model of the behavior of the system's components and a model of the desired behavior (the requirements) are required. The synthesis result is correct-by-construction with respect to the modeled requirements. From the synthesized supervisor, a supervisory controller can be derived. SCT may contribute to the aim of Rijkswaterstaat to increase the quality and the evolvability of the supervisory controller.

While supervisor synthesis is an active research topic, only a few studies exist on industrial applications. One of the reasons is the lack of acquaintance of control engineers with modeling in the framework of automata. In addition to this, there are no clear guidelines for obtaining the necessary models for synthesis. As the *first* contribution of this thesis, a way of modeling the plant and the requirements is proposed. The models result in a supervisor from which a supervisory controller can be derived that can straightforwardly be implemented on a programmable logic controller (PLC). This way of modeling is illustrated with an industrial case study in which a supervisor has been synthesized for the Algera complex.

A major incentive to use supervisor synthesis is that from the resulting supervisor a supervisory controller can be derived, which in turn can be used to generate controller code for a PLC. As a *second* contribution of this thesis, an overview of the steps to go from a supervisor model to controller code is given, as well as the potential problems that can be encountered during these steps. For each problem, it is shown how to verify that the supervisory controller is not affected by it. Furthermore, optimization

techniques are applied to structure the generated code in such a way that the execution time and its variability can be reduced.

In industry, supervisory controllers have to adhere to strict safety standards. To achieve these standards, safety PLCs (SPLCs) are used. For an SPLC implementation, the supervisory controller has to be split into a regular part and a safety part. The *third* contribution described in this thesis is a method to automatically split a supervisor into a supervisor for the regular part and a supervisor for the safety part. These two resulting supervisors can be used to derive supervisory controllers for the two parts.

Even though the supervisory controller satisfies the requirements by construction, it is not always known whether the requirements are complete and correct. Therefore, the supervisory controller has to be validated. For this, model simulation is commonly used. While model simulation is a powerful tool for validation, it offers only a partial analysis. Aspects related to the execution of the supervisory controller on a hardware platform like a PLC and communication with subsystems, such as a graphical user interface, cannot be validated with model simulation. To bridge the gap between model simulation and realization, hardware-in-the-loop (HIL) validation can be performed after model simulation and before implementation on the system. The *fourth* contribution described in this thesis is a method for HIL validation for synthesized supervisory controllers.

The previously described contributions have been demonstrated in an industrial application. A supervisory controller for a swing bridge has been synthesized, validated, implemented, and tested on the real system. The *fifth* contribution described in this thesis is the demonstration of all the necessary steps to go from a specification to an implementation of a supervisory controller for an infrastructural system. This case study shows that synthesis techniques have matured to a point where they are powerful enough to be applied to industrial-size problems.

Finally, supervisor synthesis for a family of similar systems is investigated. To this end, a graphical modeling method based on standardized modules is presented as the *sixth* contribution. In this method, the subsystems in the plant are modeled by instantiating modules from a library. A prototype tool has been developed that illustrates the proposed method in the development of supervisory controllers for movable bridges. Using this method, supervisory controllers for a family of seventeen bridges have been developed.

Samenvatting

In de komende decennia worden verschillende sluizen en bruggen in Nederland gerenoveerd of vervangen, omdat deze het einde van hun levenscyclus naderen of capaciteitsproblemen hebben. In het verleden zijn deze infrastructurele systemen ontworpen, gebouwd en onderhouden op projectbasis, wat heeft geleid tot een grote verscheidenheid aan oplossingen voor bijna dezelfde ontwerpproblemen. Deze verscheidenheid is nadelig gebleken voor de kwaliteit, de herbruikbaarheid en de levenscycluskosten. Voor toekomstige infrastructuur zoekt Rijkswaterstaat, onderdeel van het Nederlandse Ministerie van Infrastructuur en Waterstaat, naar methoden voor modularisatie en standaardisatie om de kwaliteit te verhogen, de herbruikbaarheid te vergroten en de levenscycluskosten te verlagen. Het is gebleken dat het besturingssysteem van deze infrastructurele systemen een onderdeel is dat een significante invloed heeft op deze drie aspecten.

Het doel van dit proefschrift is om de geschiktheid van supervisory control theory (SCT) te onderzoeken voor het ontwerpen van besturingssystemen voor sluizen en bruggen. SCT houdt zich bezig met het besturen van systemen die discreet zijn in tijd en in toestandsruimte en waarvan het gedrag wordt gestuurd door externe gebeurtenissen. SCT biedt technieken om automatisch deze besturingen te berekenen. Deze wijze van berekening wordt besturingssynthese genoemd. Voor besturingssynthese zijn een model van het gedrag van de componenten in het systeem en een model van het gewenste gedrag (de eisen) nodig. Het resultaat van synthese voldoet per definitie aan de gemodelleerde eisen. Uit dit resultaat kan vervolgens een besturing worden verkregen. SCT kan bijdragen aan de doelstelling van Rijkswaterstaat om de kwaliteit en de herbruikbaarheid van het besturingssysteem te verhogen.

Hoewel besturingssynthese een actief onderzoeksgebied is, bestaan er maar enkele studies over industriële toepassingen. Een van de redenen hiervan is het gebrek aan ervaring bij ingenieurs om te modelleren met automaten. Daarnaast zijn er ook geen duidelijke richtlijnen voor het verkrijgen van de benodigde modellen voor synthese. De *eerste* bijdrage van dit proefschrift is een manier om het systeem met zijn eisen te modelleren. Deze modellen resulteren in een syntheseresultaat waaruit een besturing kan worden verkregen die vervolgens eenvoudig op een programmeerbare logische besturing (Eng. programmable logic controller, PLC) kan worden geïnstalleerd. De manier van modelleren is geïllustreerd met het onderzoek van een industriële casus waarin een besturing voor het Algera complex is gesynthetiseerd.

Een belangrijke stimulans om synthese te gebruiken, is dat uit het resultaat een besturing kan worden afgeleid die op zijn beurt gebruikt kan worden om besturingscode te genereren voor een PLC. De *tweede* bijdrage van dit proefschrift geeft een overzicht

om vanuit het syntheseresultaat tot de besturingscode te komen. Daarbij worden de mogelijke problemen geanalyseerd. Voor elk probleem wordt inzichtelijk gemaakt hoe geverifieerd kan worden dat de besturing hier niet gevoelig voor is. Tevens worden optimaliseringstechnieken gebruikt om de code zo te structureren dat de uitvoeringstijd en de variabiliteit van een programmacyclus worden gereduceerd.

In de industrie moeten besturingen aan strikte veiligheidsstandaarden voldoen. Om aan deze standaarden te voldoen, worden veiligheids-PLCs (Eng. Safety-PLCs, SPLCs) gebruikt. Voor de implementatie op een SPLC moet de besturing in een regulier deel en een veiligheidsdeel worden gesplitst. De *derde* bijdrage beschreven in dit proefschrift is een methode die een besturing automatisch splitst in een regulier deel en in een veiligheidsdeel. Vanuit deze twee resulterende delen kan besturingscode voor de gehele SPLC worden verkregen.

Ondanks dat de besturing per definitie aan de eisen voldoet, is het niet altijd bekend of de eisen compleet en correct zijn. Daarom dient de besturing te worden gevalideerd. Hiervoor wordt vaak modelsimulatie gebruikt. Hoewel modelsimulatie een krachtig hulpmiddel is voor validatie, biedt het slechts een gedeeltelijke analyse. Aspecten die verband houden met de uitvoering van de besturing op een hardwareplatform, zoals een PLC, en communicatie met subsystemen, zoals een mens-machine-interface, kunnen niet worden gevalideerd met modelsimulatie. Om de kloof tussen modelsimulatie en realisatie te overbruggen, kan hardware-in-the-loop (HIL)-validatie worden uitgevoerd na modelsimulatie en vóór implementatie op het systeem. De *vierde* bijdrage die in dit proefschrift wordt beschreven, is een methode voor HIL-validatie van gesynthetiseerde besturingen.

De voorgaande bijdragen zijn gedemonstreerd in een industriële toepassing. Een besturing voor een draaibrug is gesynthetiseerd, gevalideerd, geïmplementeerd en getest op het echte systeem. De *vijfde* bijdrage die in dit proefschrift wordt beschreven, is het demonstreren van alle stappen die nodig zijn om van een specificatie naar een implementatie van een besturing voor een infrastructureel systeem te gaan. Deze casus laat zien dat synthesetechnieken een punt hebben bereikt waarop ze krachtig genoeg zijn om te worden toegepast op industriële schaal.

Tenslotte wordt besturingssynthese voor een familie van vergelijkbare systemen onderzocht. Een grafische modelleermethode op basis van gestandaardiseerde modules wordt hier als *zesde* bijdrage gepresenteerd. Bij deze methode worden de subsystemen gemodelleerd door middel van het instantiëren van modules uit een bibliotheek. Er is een prototype van een computerprogramma ontwikkeld dat de voorgestelde methode illustreert voor de ontwikkeling van besturingen van beweegbare bruggen. Met deze methode zijn de besturingen voor een familie van een zeventiental bruggen ontwikkeld.

Dankwoord

Dit proefschrift is het resultaat van de intensieve samenwerking tussen Rijkswaterstaat (RWS) en de Technische Universiteit Eindhoven (TU/e) in het MultiWaterWerk (MWW)-project. Het voordeel van een samenwerking met de industrie is dat ontwikkelde theorieën direct op echte systemen getoetst kunnen worden, iets wat ik heb ervaren als uiterst waardevol. De combinatie van academie en industrie heeft ervoor gezorgd dat het een uitdagend en interessant project is geworden. Hiervoor wil ik iedereen bij RWS en de TU/e enorm bedanken.

In het bijzonder wil ik bij RWS de volgende personen bedanken. Han Vogel en Maria Angenent wil ik bedanken voor het mogelijk maken van het MWW-project. Het is inspirerend om te zien hoe zij zoveel mensen binnen RWS enthousiast hebben gemaakt voor ons onderzoek. Bert van der Vegt bedank ik voor het mij wegwijs maken in de Landelijke Bruggen- en Sluizenstandaard. Zijn deelname aan de promotiecommissie stel ik zeer op prijs. John van Dinther ben ik dankbaar voor zijn enorme inspanning die het mogelijk heeft gemaakt om de praktijktesten op de Oisterwijksebaan brug uit te voeren. De eerste keer dat we samen de brug aan de praat kregen, is een van de hoogtepunten uit mijn promotieonderzoek. De hulp hierbij van Paul Bruinsma, Roel Driessens, Arnoud de Kruijf en Joep Schatorjé wil ik hier tevens noemen. Gerrit Bruggink bedank ik voor het mij bijbrengen van de technische kennis over bruggen en sluizen en voor het mede ontwikkelen van de experimenteeropstelling. Robert de Roos, projectleider van het MWW-onderzoek namens RWS, wil ik bedanken voor al zijn tijd die hij aan het onderzoek heeft gegeven. Verder wil ik alle overige RWS-medewerkers bedanken die op enige manier hebben bijgedragen aan dit onderzoek.

De deelnemers vanuit de TU/e aan het MWW-project wil ik hierbij graag bedanken. Asia van de Mortel-Fronczak wil ik bedanken voor haar begeleiding die al ver voor mijn promotie begon, namelijk tijdens mijn bachelor-, stage- en masterproject. Bedankt voor alles wat je mij in die bijna zeven jaar hebt geleerd, te veel om hier te noemen. Koos Rooda bedank ik voor de grote hoeveelheid tijd en inspanning die hij in mijn project heeft gestoken. Ik heb dat altijd erg gewaardeerd. Na iedere bespreking kwamen er weer nieuwe inzichten hoe ik een probleem anders kon benaderen of hoe ik een artikel nog beter kon structureren. Michel Reniers bedank ik voor het bijschaven van mijn 'werktuigbouwkunde-wiskunde' en voor het bedenken van oplossingen waar ik die soms niet zag. Wan Fokkink, mijn promotor, bedank ik voor de bemoedigende woorden die hij elke keer weer uitsprak tijdens onze voortgangsbesprekingen. Pascal Etman, projectleider van het MWW-onderzoek namens de TU/e, wil ik bedanken dat hij mij heeft overtuigd om aan een promotie te beginnen. Albert Hofkamp bedank ik voor het implementeren van de algoritmen. Ook heeft hij mij vertrouwd gemaakt

met het ontwikkelen van professionele software. Henk van Rooy bedank ik voor alle praktische zaken met betrekking tot de labopstellingen.

I thank the other members of the doctorate committee, prof. Stéphane Lafortune, dr. Eric Rutten, and prof. Jeroen Voeten, for reviewing this thesis, providing constructive feedback, and for taking part in the thesis defense.

Alle studenten die tijdens hun bachelor-, stage-, of masterproject hebben meegewerkt aan dit onderzoek wil ik bedanken voor hun inzet. In het bijzonder wil ik Bart, Erwin, Eva-Britt en Toby hiervoor bedanken.

Mijn collega's Aida, Jeroen, Joshua, Karlijn, Lars, Lennart, Martijn, Nick, Sander, Sjoerd, Thomas en Tim wil ik bedanken voor het creëren van de leuke werkomgeving in het SE-lab!

In Peel en Maas wil ik iedereen bedanken die ervoor heeft gezorgd dat ik op vrijdag- en zaterdagavond niet thuis op de bank hoefde te zitten. Met name de Boyzz van de Kiët: Dennis, Janick, Mick en Thomas. Ik wil mijn huisgenoten Arjan, Nicky en Sanne bedanken voor de gezelligheid tijdens mijn studie en het grootste gedeelte van mijn promotieonderzoek.

Pap, mam en Maartje wil ik bedanken voor alles wat ze voor mij hebben gedaan. Zij hebben mij altijd gemotiveerd en er alles aan gedaan zodat ik het beste uit mezelf zou halen, zonder hen was dit onderzoek waarschijnlijk nooit gelukt. En pap, bedankt voor de kleine tweehonderd keer dat je mij de afgelopen jaren naar het station hebt gebracht. Als laatste, Shirley bedankt voor alle liefde en de leuke momenten en avonturen die we samen hebben beleefd! Ik had niemand liever naast me gehad tijdens deze jaren.

Ferdie Reijnen
September 2020

List of publications

Peer-reviewed journal contributions

Reijnen, F.F.H., Goorden, M.A., van de Mortel-Fronczak, J.M., and Rooda, J.E. (2020). "Modeling for supervisor synthesis – A lock-bridge combination case study". In: *Discrete Event Dynamic Systems* vol. 30, no. 3, pp 499-532.

Reijnen, F.F.H., Leliveld, E.-B.M.L., van de Mortel-Fronczak, J.M., van Dinther M.J.T., Rooda, J.E., and Fokkink, W.J. (2020) "A synthesized fault-tolerant supervisory controller for a swing bridge". Submitted.

Peer-reviewed conference contributions

Reijnen, F.F.H., Goorden, M.A., van de Mortel-Fronczak, J.M., and Rooda, J.E. (2017). "Supervisory control synthesis for a waterway lock". In: *Proceedings of the 1st Conference on Control Technology and Applications*. IEEE, pp. 1562-1568.

Reijnen, F.F.H., Goorden, M.A., van de Mortel-Fronczak, J.M., Reniers, M.A., and Rooda, J.E. (2018). Application of dependency structure matrices and multilevel synthesis to a production line". In: *Proceedings of the 2nd Conference on Control Technology and Applications*. IEEE, pp. 458-464.

Reijnen, F.F.H., Reniers, M.A., van de Mortel-Fronczak, J.M., and Rooda, J.E. (2018). "Structured synthesis of fault-tolerant supervisory controllers". In: *IFAC-PapersOnLine* vol. 51, no. 24, pp. 894-901.

Reijnen, F.F.H., Verbakel, J.J., van de Mortel-Fronczak, J.M., and Rooda, J.E. (2019). "Hardware-in-the-loop set-up for supervisory controllers with an application: The Prinses Marijke complex". In: *Proceedings of the 3rd Conference on Control Technology and Applications*. IEEE, pp. 843-850.

Reijnen, F. F. H., Hofkamp, A.T., van de Mortel-Fronczak, J.M., Reniers, M.A., and Rooda, J.E. (2019). "Termination and confluence for state-based controllers". In: *Proceedings of the 15th Conference on Automation Science and Engineering*. IEEE, pp. 509-516.

Reijnen, F.F.H., Erens, T.R., van de Mortel-Fronczak, J.M., and Rooda, J.E. (2020). "Supervisory control synthesis for safety PLCs". In: *Proceedings of the 15th Workshop on Discrete Event Systems*. IFAC. In press.

Reijnen, F.F.H., van de Mortel-Fronczak, J.M., Reniers, M.A., and Rooda, J.E. (2020). "Design of a supervisor platform for movable bridges". In: *Proceedings of the 16th Conference on Automation Science and Engineering*. IEEE, pp. 1298-1304.

Reijnen, F.F.H., van de Mortel-Fronczak, J.M., and Rooda, J.E. (2020). "Data logging and reconstruction of discrete-event system behavior". In: *Proceedings of the 16th Conference on Control, Automation, Robotics and Vision*. IEEE. In press.

Non peer-reviewed conference contributions

Reijnen, F.F.H., van de Mortel-Fronczak, J.M., and Rooda, J.E. (2018). "Supervisory control synthesis for a lock-bridge combination". In: *Proceedings of the 37th Benelux Meeting on Systems and Control*. pp. 154.

Reijnen, F.F.H., van de Mortel-Fronczak, J.M., and Rooda, J.E. (2018). "Supervisory controllers for cyber-physical systems". In: *The European EduNet Conference*, Oral presentation.

Reijnen, F.F.H., van de Mortel-Fronczak, J.M., Rooda, J.E., and van Dinther, M.J.T. (2019). "Synthesis and implementation of supervisory control for infrastructural systems". In: *Proceedings of the 38th Benelux Meeting on Systems and Control*. pp. 92.

Reijnen, F.F.H., van de Mortel-Fronczak, J.M., Reniers, M.A., and Rooda, J.E.(2020). "Graphical modeling for supervisor synthesis". In: *Proceedings of the 39th Benelux Meeting on Systems and Control*. pp. 81.

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. (2020). "The road ahead for supervisor synthesis". In: *Proceedings of the 6th Symposium on Dependable Software Engineering*, In press.

Contents

Chapter 1

Introduction

In this chapter, the research context and infrastructural systems, like waterway locks and movable bridges, are introduced. Subsequently, the problem of designing supervisory controllers for these systems is discussed. Then, the synthesis-based engineering method is described. This chapter continues with providing the research questions and main contributions. Finally, the outline of this thesis is provided.

1.1 Research context

The Dutch waterway network is the densest in Europe. It consists of more than 6,000 kilometers of rivers and canals, forming a network serving all parts of the country. The main waterway network, consisting of 3,500 kilometers of waterways (Rijkswaterstaat, 2020), is state-owned, operated, and maintained by Rijkswaterstaat (RWS), the executive branch of the Dutch Ministry of Infrastructure and Waterway Management. Smaller waterways are managed by different stakeholders, like provinces and municipalities.

In the main waterway network, a total of 129 locks and 344 bridges is present to facilitate water and land traffic. The main function of a lock is to maintain the difference in water level on each of its sides while at the same time allowing vessels to pass from one side to the other. Other lock functions are separating salt and fresh water, protecting against floods, and protecting against drought. Figure 1.1 shows the waterway lock, consisting of three chambers, located in Maasbracht, the Netherlands. The main function of a bridge is to allow land traffic to cross a river or a canal. A movable bridge is a bridge that contains a section that can be opened to give clearance for large vessels to pass the bridge. Figure 1.2 shows the Prins Bernhard bridge, a movable bridge, located in Zaandam, the Netherlands.

In the coming decades, numerous locks and bridges have to be renovated or replaced, as they have reached their end-of-life or have capacity problems. In the past, these infrastructural systems have been engineered, built, and maintained on a project basis. This resulted in a large variety of unique solutions to almost the same engineering problems. This uniqueness has a negative impact on the quality, the evolvability, and the life-cycle costs. The negative impact is primarily due to the need for specialized knowledge to operate and maintain these systems. The supervisory controller of an

Figure 1.1: A waterway lock consisting of three chambers, located in Maasbracht, the Netherlands [Source: https://sluizenenstuwen.nl, Stichting Historische Sluizen en Stuwen Nederland].

Figure 1.2: A movable bridge, located in Zaandam, the Netherlands [Source: Politie, Landelijke Eenheid, Dienst Infrastructuur, Afdeling Luchtvaart].

infrastructural system, which controls and monitors its components, is identified as one of the subsystems having a significant impact on these three aspects. RWS is seeking methods for modularization and standardization to improve these three aspects for future infrastructural systems.

To this end, RWS initiated the MultiWaterWerk (MWW) project. In this project, RWS collaborates with Eindhoven University of Technology to establish a shift from an Engineering-to-Order to a Configure-to-Order production method, see Wilschut (2018). With the Engineering-to-Order method, each infrastructural system is uniquely

designed. With the Configure-to-Order method, (partially) standardized components and modules are combined to configure the design of a specific infrastructural system. Within the area of supervisory control, the MWW project seeks new methods for designing supervisory controllers for infrastructural systems. In Goorden (2019), formal model-based methods are described that can be used for the development of these supervisory controllers. There, the focus is on improving the performance of the so-called synthesis techniques for supervisory controllers such that they can be used for large-scale infrastructural systems. Goorden et al. (2020a) reports on recent research advances in supervisor synthesis, as well as industrial applications and future research challenges, in the context of the MWW project.

1.2 Control systems of infrastructural systems

The control structure of an infrastructural system, shown in Figure 1.3, is similar to that of most computer-controlled mechanical systems. The control structure consists of five layers. The bottom three layers, 1 through 3, constitute the physical part of the system (depicted by the dashed box). This physical part consists of the mechanical components, such as boom barriers, traffic lights, and lock gates. The state of these components can be influenced by actuators and can be measured by sensors. A resource controller acts as a low-level controller (e.g., for the control of a frequency converter controlling the speed of a bridge deck). The supervisory controller, layer 4, is responsible for the safe coordination between the various components. Based on the signals received from the other layers, it decides when actuators should be switched on or off. Typically, for infrastructural systems, the supervisory controller is implemented on a programmable logic controller (PLC), located close to the system. A graphical user interface (GUI), layer 5, is connected to the supervisory controller to facilitate the interaction between a (human) operator and the system. This GUI visualizes information about the system's state (such as the water level in a lock chamber or the position of a boom barrier at a bridge) to the operator and contains buttons allowing the operator to send commands to the supervisory controller. The operator controls the system from a control center located elsewhere. There, the system can be monitored via camera images. The operator is responsible for communicating with arriving vessels, giving commands to the supervisory controller, and monitoring the system.

Infrastructural systems are becoming increasingly complex due to the high demands in terms of safety, quality, and functionality. As a result, supervisory controllers for these systems are getting more complex as well. Traditionally, the supervisory controllers are programmed in PLC code by hand and then manually tested. Designing error-free control systems with this way of working is challenging, as illustrated by the recent discovery of urgent problems in the control systems of locks and bridges at the Afsluitdijk, see van Nieuwenhuizen (2019). Moreover, there exist concerns about the safety of control systems of movable bridges, as reported in Dijsselbloem et al. (2019). There, it was specifically noted that the lack of uniformity in the control system can lead to serious accidents.

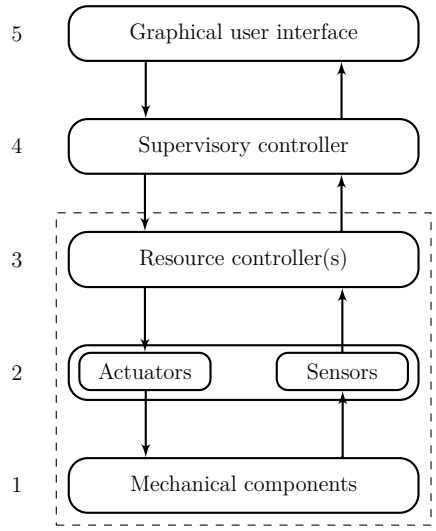


Figure 1.3: Control structure of an infrastructural system.

At the same time, for the development process, it is desired to decrease time-to-market and costs. Model-based development methods have been promoted as a solution to handle the difficulties arising in control-system development by raising the abstraction level (away from implementation details) and automating error-prone and repetitive tasks, see Ramos et al. (2011), Vyatkin (2013), Vogel-Heuser et al. (2014), and Thramboulidis (2015). Combining model-based methods with formal methods can further enhance the design of control systems, by mathematically proving that certain properties are guaranteed.

Within the MWW project, RWS aims to develop methods for the specification, design, realization, implementation, and maintenance of supervisory control systems to cope with the above mentioned problems and to increase quality, decrease life-cycle costs, and increase evolvability.

1.3 Problem description

Supervisory control theory (SCT), as initiated by Ramadge and Wonham (1987), is a research area focused on developing techniques for the automatic synthesis of supervisors. SCT is developed for systems for which the uncontrolled behavior of the system and a set of requirements that the controlled system must adhere to are given. From these, a supervisor for the system can be calculated that always satisfies the requirements. This calculation step is called supervisor synthesis. A more in-depth introduction to supervisor synthesis is provided in Chapter 2. From this supervisor, a supervisory controller can be derived.

Applying a formal method like SCT in the design of supervisory controllers provides several advantages. SCT guarantees that the derived supervisory controller controls the system such that all formulated requirements are satisfied. In other words, the supervisory controller is correct-by-construction. As safety is of great concern, this advantage is essential for the design of supervisory controllers for infrastructural

systems. Moreover, the use of formal models allows for a more consistent and less ambiguous description of the system and its desired behavior in comparison to textual documents. Other advantages include the possibility for early validation and automatic implementation code generation. In Baeten et al. (2016), the integration of SCT in the engineering process for supervisory controllers is discussed. This method is called synthesis-based engineering (SBE). In the following section, SBE is explained in more detail.

While synthesis techniques have been under development for over 30 years, their use in industry has not been widespread (Wonham et al., 2018; Laing et al., 2020). One of the main reasons for this is that until recently synthesis techniques were unable to deal with industrial-size problems. Advances in algorithms, such as Vahidi et al. (2006), Miremadi et al. (2011), Ouedraogo et al. (2011), and Goorden (2019) facilitated the use of synthesis techniques for industrial-size problems.

The aim of this thesis is to investigate whether SBE is suitable for the design of supervisory controllers for infrastructural systems. In doing so, this (formal) model-based approach can contribute to an increase in quality, a decrease in the cost, and an increase in evolvability when building and renovating infrastructural systems.

1.4 Synthesis-based engineering method

This section provides an overview of the synthesis-based engineering method. The mathematics behind the modeling and the synthesis procedure are given in Chapter 2. Figure 1.4 schematically shows the synthesis-based engineering process, adapted from Baeten et al. (2016), of designing and implementing a supervisory controller for a system. First, a set of high-level requirements for the overall system H_{R} is defined. After this, several parallel design tracks are initiated. To illustrate SBE, only two parts are shown: the plant and the supervisory controller. For the supervisory controller, a set of control requirements is defined and formalized, this yields C_{R}. For the plant, a set of requirements P_{R} is defined, and from P_{R} a plant design P_{D}, and a plant model P are made. Alternatively, if the plant realization already exists, for example in case of a renovation, the realization can be modeled. From C_{R} and P, a model of the supervisor S can be synthesized. From this supervisor, a model of the supervisory controller C is derived. There exist subtle differences between a supervisor and a supervisory controller, which are explained in Chapter 4. The synthesis and derivation step together are sometimes called supervisory-controller synthesis. The behavior of the supervisory controller might not be as desired, as it is not always known beforehand whether the requirements are complete and correct. Hence, the resulting supervisory controller has to be validated. For validation, the supervisory controller is simulated together with the plant model. To match the simulated behavior to the observed behavior in reality as closely as possible, this plant model is often enriched with continuous-time behavior (called a hybrid plant model) and an interactive visualization. When the observed behavior is consistent with the intended behavior, the PLC controller code C can be generated automatically and implemented on a PLC. To validate the behavior of the supervisory controller on the PLC and its interaction with all subsystems, hardware-in-the-loop (HIL) simulation can be used. Here, the real PLC is connected

to a model of the system. Finally, the plant is built and the PLC can be connected to the real plant.

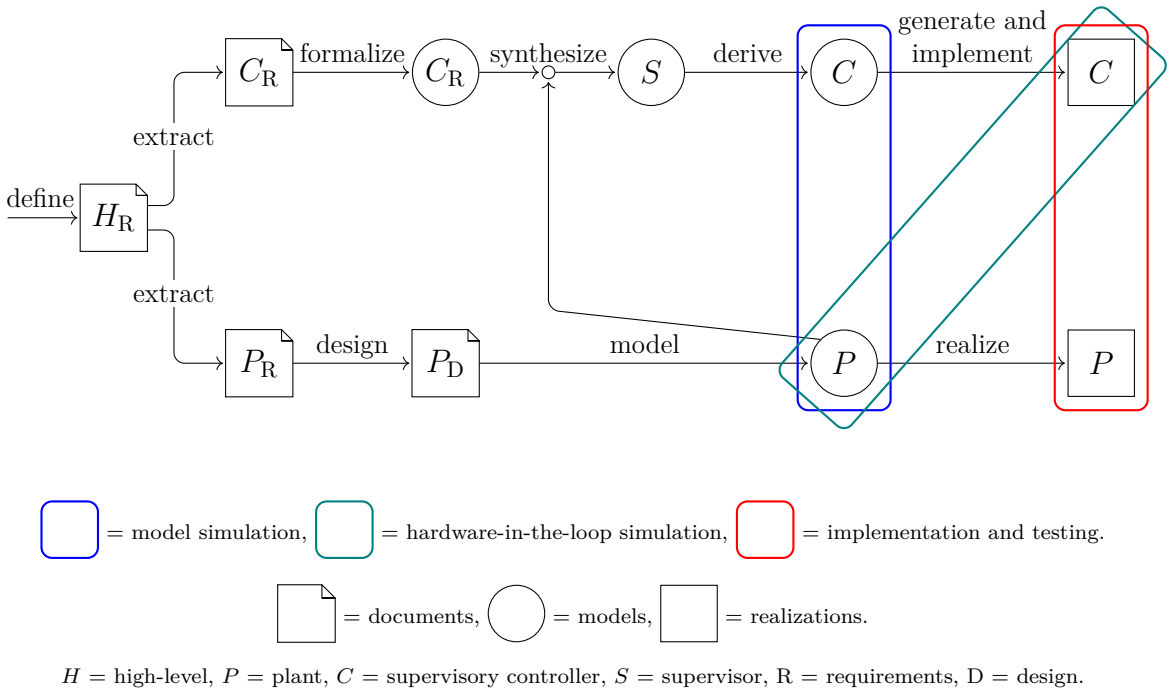


Figure 1.4: A schematic overview of the synthesis-based engineering method, adapted from Baeten et al. (2016).

Compared to traditional engineering methods and other model-based engineering methods (see, e.g., Frey and Litz (2000) and Swartjes et al. (2017)), the major difference is that no supervisory controller is programmed (or modeled) by hand when using synthesis-based engineering. Instead, much effort is put into defining and formalizing the requirements, such that the supervisory controller can be derived from those. Hence, the focus shifts from developing and debugging the implementation code to designing and improving the requirements.

For infrastructural systems, the necessary documents are already available, as RWS has developed several specifications. The design of waterway locks is described in 'Basisspecificatie Schutsluis' (Nieman, 2016) and the design of movable bridges in 'Basisspecificatie Beweegbare Brug' (van der Heide, 2019), these can be used as a basis for the plant model. The requirements for the control systems for waterway lock and movable bridges are defined in 'Landelijke Bruggen- en Sluizenstandaard' (Rijkswaterstaat, 2019).

1.5 Research questions

This thesis investigates the applicability of supervisor synthesis to the design of supervisory controllers for infrastructural systems. Specifically, this thesis tries to find an answer to the following question.

How suitable is the synthesis-based engineering method for the design, validation, and implementation of supervisory controllers for infrastructural systems?

To answer this question, five research questions have been defined that need to be answered first. These research questions (RQs) are given below.

Research question 1

What is a suitable way to model infrastructural systems and their requirements for the purpose of supervisor synthesis?

For supervisor synthesis, it is crucial to have meaningful and correct models of the system and its control requirements. So far, in the literature, no modeling guidelines are provided that can help in obtaining these models. In Wonham et al. (2018), it was argued that the lack of applications for supervisor synthesis is partly due to the difficulty of obtaining the necessary models for supervisor synthesis. To increase the applicability of supervisor synthesis, a suitable way of modeling is necessary.

Research question 2

Which steps are necessary to derive a supervisory controller, and subsequently, implementation code, from a synthesized supervisor model?

A major incentive to use supervisor synthesis is that from the resulting supervisor a supervisory controller can be derived, which in turn, can be used to generate implementation code. In the literature, it has been shown that some difficulties exist in the derivation and implementation steps, and not every supervisor is suitable for implementation as a supervisory controller. It has to be determined which additional properties a supervisor needs to satisfy such that it can be used as a supervisory controller, and how these properties can be verified for large supervisors.

Research question 3

Which additional steps have to be performed to use a synthesized supervisor as a safety PLC controller?

For infrastructural systems, the supervisory controller has to adhere to strict safety standards as defined in the machinery directive (European Commission, 2006). To comply with the hardware standards of this directive, safety PLCs are used. Safety PLCs require a split between the 'safety' part of the supervisory controller and the 'regular' part of the supervisory controller. To comply with these standards, it is necessary to have a method to automatically split the supervisory controller.

Research question 4

> *How can a supervisory controller for a real infrastructural system be synthesized, implemented, and tested?*

To assess whether synthesis-based engineering is a valid option for the development of the supervisory controllers, case studies have to be performed. Aspects related to the scalability of the methods can only be evaluated with real cases. In these case studies, all the necessary steps have to be performed for a real infrastructural system. Special attention should be given to the failure of components in a real system and the integration of GUIs.

Research question 5

> *In what way can the similarities between similar systems be exploited to efficiently develop the plant and the requirements models?*

As mentioned, Rijkswaterstaat has to renovate and replace numerous waterway locks and movable bridges in the foreseeable future. These systems show lots of similarities. Designing the necessary models for each case is laborious and error-prone. It should be determined whether the similarities can be exploited such that the models can be obtained more efficient.

1.6 Main contributions

The thesis has the following six main contributions. While the presented research focuses on waterway locks and movable bridges, these contributions are also applicable and useful in other application domains that have similar system characteristics, as for example: theme park attractions (Forschelen et al., 2012), manufacturing lines (Reijnen et al., 2018a), tunnels (Moormann et al., 2020a), and automotive systems (Korssen et al., 2018).

Contribution 1

Various case studies on infrastructural systems have led to a method for the design of the plant model and the requirements model. The case study of Lock III in Reijnen et al. (2017), a waterway lock in Tilburg, showed that it is possible to model the system and the control requirements and subsequently synthesize a supervisor. Similar modeling has been done for lock Empel and lock Hintham (Verbakel, 2017). Another study has been conducted on the Algera complex (Reijnen et al., 2020a), where a waterway lock and a bridge are controlled together. Another bridge that has been modeled is the Oisterwijksebaan bridge in Tilburg (Reijnen et al., 2020e). The lessons learned from these case studies have led to a component-based modeling method that can be used to model infrastructural systems. This contribution relates to RQ 1.

Contribution 2

When a supervisor has been synthesized successfully, the next step is to derive a supervisory controller from it. For this, the supervisor needs to satisfy three additional properties, which are confluence, finite response, and nonblocking under control, as shown in Malik (2003). This thesis proposes sufficient conditions for confluence and finite response that are useful if the modeling method as proposed in Contribution 1 is applied. It is also shown how nonblocking under control can be verified. When the supervisory-controller model is used to generate PLC code, sequencing algorithms, see Steward (1981) and Eppinger and Browning (2012), are used to optimize the execution time of the generated code. This contribution allows a synthesized supervisor to be used as a supervisory controller for an infrastructural system. This contribution relates to RQ 2.

Contribution 3

If safety PLCs are used as the implementation platform for the supervisory controller, the model first has to be split into two parts. A method has been developed that automatically splits a model for this purpose. To validate the method, a case study on the Oisterwijksebaan bridge has been performed. The result is compared to a manual splitting result made by experts. Furthermore, the supervisory controller has been implemented to control the real bridge. This contribution allows the supervisory controller to be used in combination with safety PLCs. This contribution relates to RQ 3.

Contribution 4

Going from a model of the supervisory controller to a realization on a real system remains a large step. Model simulation is often used to validate that the behavior of the supervised system is as intended. Even though simulation is a valuable tool for early validation, it provides only a partial analysis. One of the main shortcomings is that the models are used, and not the actual implementation code and the implementation hardware. To bridge the gap between model simulation and implementation on the real system, hardware-in-the-loop (HIL) simulation can be used (Bullock et al., 2004). For HIL simulation, the real implementation code and the implementation hardware are connected to a model of the system. In this thesis, HIL simulation has been integrated in the engineering method by automatically generating the necessary models. With this way of working, almost no additional effort is required for HIL simulation. This is illustrated with a case study on the Prinses Marijke complex. This contribution relates to RQ 4.

Contribution 5

To demonstrate the applicability of the synthesis-based engineering method, a case is described where a supervisory controller for the Oisterwijksebaan bridge, a swing bridge, has been developed. The case study illustrates all the necessary steps, i.e., modeling, synthesis, validation, code generation, and implementation of the supervisory

controller. Moreover, identification of failures and handling of failures, so-called fault-diagnosis and fault-tolerant control, are included as well. This case study, performed on the *real system*, shows that synthesis techniques have matured to a point where they are powerful enough to be applied to industrial-size problems. This contribution relates to RQ 4.

Contribution 6

In the performed case studies, it has been observed that different infrastructural systems consist of similar models. Inspired by the work of Grigorov et al. (2011), the use of standardized modules for the design of the necessary models for synthesis and simulation has been investigated. A graphical modeling method based on standardized modules is presented. A prototype tool has been developed that illustrates the proposed method for the development of supervisors for movable bridges. Using this method, the models for a family of seventeen bridges have been developed. This contribution relates to RQs 1 and 5.

1.7 Thesis outline

This thesis is structured as follows. Chapter 2 describes the preliminaries of supervisory control theory. In Chapter 3, a way of modeling for supervisor synthesis is proposed. In this chapter, a case study on a lock-bridge combination is described as an example. Chapter 4 describes the steps needed to go from a synthesized supervisor to an implemented (PLC) supervisory controller. Sufficient conditions are described that can be used to verify that a supervisor can be implemented as a supervisory controller. Moreover, it is shown how code can be generated from the model. Chapter 5 is an extension to the previous chapter. In this chapter, a method is described that splits a supervisory controller into two parts that can be used for safety PLCs. Validation of the implemented supervisory controller with the use of hardware-in-the-loop simulation is described in Chapter 6. To illustrate this method, a case study on the Prinses Marijke locks is described. In Chapter 7, a case study on the Oisterwijksebaan bridge is described, where the concepts developed in this thesis are applied on the real system. In this case study, a fault-tolerant supervisory controller has been synthesized, validated, implemented, and tested on the actual bridge. In Chapter 8, a graphical modeling method is proposed for the synthesis of supervisors for a family of systems. Finally, answers to the research questions and recommendations for future work are provided in Chapter 9.

Chapter 2

Supervisory control theory

This chapter provides a brief introduction to modeling of discrete-event systems and supervisory control theory. This introduction is based on the work of Ramadge and Wonham (1987), Sköldstam et al. (2007), Cassandras and Lafortune (2009), Ouedraogo et al. (2011), and Wonham and Cai (2019). First, modeling of discrete-event systems is discussed. Second, it is shown how requirements are modeled. Third, supervisor synthesis is explained. Then, normalization of models is discussed. Finally, the mathematics behind the synthesis algorithm are provided.

2.1 Modeling of discrete-event systems

In the context of supervisor synthesis, systems are usually modeled as (extended) finite-state automata (FAs) or Petri nets. Both formalisms are used to represent event-driven behavior. In this thesis, extended finite-state automata (EFAs) are used as the modeling formalism. This allows for more compact and elegant models compared to FAs, as demonstrated in Miremadi et al. (2010). As EFAs are an extension to FAs, first FAs are introduced, followed by EFAs.

2.1.1 Finite-state automata

An FA is formally defined as a 5-tuple, $A = (L, \Sigma, \delta, l^0, L^{\mathrm{m}})$, where L is a finite set of locations, Σ a finite set of events, $\delta \subseteq L \times \Sigma \times L$ a transition relation, $l^0 \in L$ the initial location, and $L^{\mathrm{m}} \subseteq L$ a set of marked locations. A location is marked (by the modeler) when this location represents a 'safe' situation. The event set can be partitioned into controllable events Σ_{c} and uncontrollable events Σ_{u}, which denote actions that can and cannot be disabled by the supervisor, respectively. With Σ^* the set of all finite strings of events in Σ is denoted and $\Sigma^+ = \Sigma^* \setminus \{\epsilon\}$, where ϵ is the empty string.

An FA is called deterministic if for each location $l \in L$ and event $\sigma \in \Sigma$ there exists at most one location $l' \in L$ such that $(l, \sigma, l') \in \delta$; otherwise, it is called nondeterministic. Given the nature of the infrastructural systems, only deterministic FAs are considered in this thesis.

For most systems, it is not feasible to model their behavior with a single FA, as the state space of the model is often too large. Instead, a system can be modeled as a set of several interacting FAs A_i (referred to as component models). The behavior of the combined set of FAs is given by the synchronous product $A = A_1 \parallel \ldots \parallel A_m$ (Cassandras and Lafortune, 2009), which requires simultaneous execution of transitions labeled with the same event. This means that an event can only be executed when all FAs that contain this event can execute this event. Let $A_k = (L_k, \Sigma_k, \delta_k, l_k^0, L_k^{\mathrm{m}})$, $k = 1, 2$ be FAs. The synchronous product of A_1 and A_2 is

$$A_1 \parallel A_2 = (L_1 \times L_2, \Sigma_1 \cup \Sigma_2, \delta, (l_1^0, l_2^0), L_1^{\mathrm{m}} \times L_2^{\mathrm{m}})$$

where the transition relation δ is defined as:

- if $\sigma \in \Sigma_1 \cap \Sigma_2$, then $((l_1, l_2), \sigma, (l_1', l_2')) \in \delta$, for all $(l_1, \sigma, l_1') \in \delta_1$ and $(l_2, \sigma, l_2') \in \delta_2$.
- if $\sigma \in \Sigma_1 \setminus \Sigma_2$, then $((l_1, l_2), \sigma, (l_1', l_2)) \in \delta$, for all $(l_1, \sigma, l_1') \in \delta_1$ and $l_2 \in L_2$.
- if $\sigma \in \Sigma_2 \setminus \Sigma_1$, then $((l_1, l_2), \sigma, (l_1, l_2')) \in \delta$, for all $(l_2, \sigma, l_2') \in \delta_2$ and $l_1 \in L_1$.

FAs can be displayed graphically as well. Here, a (labeled) circle denotes a location, an unconnected incoming arrow indicates the initial location, and a filled circle indicates a marked location. Controllable and uncontrollable events are visualized by (labeled) solid and dashed arrows, respectively. Controllable-event names start with `c_` and uncontrollable-event names start with `u_`. It is assumed that the event set consists only of the events displayed on a transition. An example is shown in Figure 2.1. The right-hand side FA is the synchronous product of the other two FAs. The FAs synchronize over the `u_transfer` event.

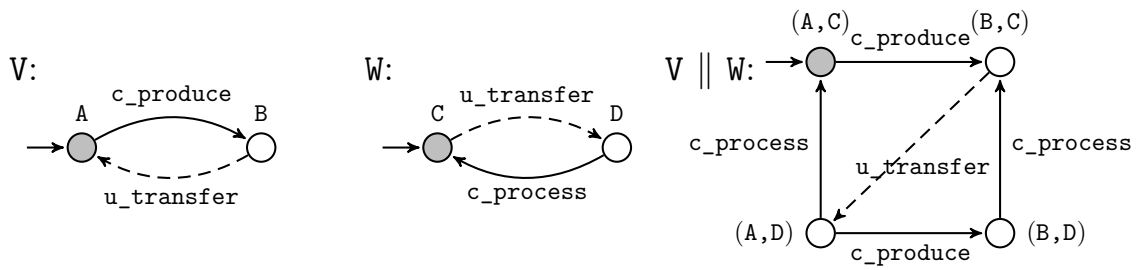


Figure 2.1: Graphical representation of FAs (left and middle) and their synchronous product (right).

2.1.2 Extended finite-state automata

In Cheng and Krishnakumar (1996), Chen and Lin (2000), and Sköldstam et al. (2007), EFAs are used for modeling systems. EFAs are FAs parameterized by bounded discrete variables. A model consists of several interacting EFAs, called an EFA system, $\mathcal{E} = \{E_1, \ldots, E_m\}$ together with a set of variables $X_{\mathcal{E}} = \{x_1, \ldots, x_n\}$. Transitions in an EFA may contain guards (i.e., logical conditions) over the variables, and updates (i.e., assignments) to the variables.

With each variable x, a finite discrete domain is associated, $\mathrm{dom}(x)$. A valuation $v \in V$ is a function $v : X_{\mathcal{E}} \to \bigcup_{x \in X_{\mathcal{E}}} \mathrm{dom}(x)$, for which $v(x) \in \mathrm{dom}(x)$ for every $x \in X_{\mathcal{E}}$. The set of all valuations is V, the initial valuation is $v^0 \in V$, and the set of marked valuations is $V^{\mathrm{m}} \subseteq V$.

Formally, an EFA is defined as $E = (L, X, \Sigma, \to, l^0, L^{\mathrm{m}})$, where $X \subseteq X_{\mathcal{E}}$ is a set of local variables, $\to$ is the extended transition relation, and the other elements are equal to those for FAs. The transition relation is defined as $\to \subseteq L \times G \times \Sigma \times U \times L$, with G the set of guards and U the set of updates. We assume that each variable in the EFA system is local to exactly one EFA in the EFA system.

A guard is a function, $g : V \to \{\mathbf{T}, \mathbf{F}\}$, where $\mathbf{T}$ and $\mathbf{F}$ are the Boolean literals. For clarity, $v \models g$ is used instead of $g(v)$. In the models, guards are written as propositional formulas over the variables in the EFA system. It is assumed that each EFA contains a variable that represents the current location, called the location pointer `LP`, with $\mathrm{dom}(\texttt{LP}) = L$. As a result, locations of other EFAs can be used in the guards. An example of a guard is $x > 5 \wedge \texttt{E.On}$, where `E.On` means that EFA `E` is in location `On`. This guard evaluates to $\mathbf{T}$ for some $v \in V$ if $v(x) > 5$ and $v(\texttt{LP}_{\texttt{E}}) = \texttt{On}$.

An update is a function $u : V \to V$. Only the values of the local variables of an EFA can be updated, i.e., $v(x) = u(v)(x)$ for all $x \in X_{\mathcal{E}} \setminus X$. In the models, updates are written as (conditional) assignments. An example of an update is $x := x + 3$, which increases the value of x by 3. An example of a conditional update is $x := \mathrm{if}(x < 5) : x + 1 \text{ else} : x$, which increases the value of x by 1 as long as the value of x is less than 5. Only deterministic updates are considered, e.g., $x + y := 7$ is not considered. An assignment of a value outside of the variable's domain is undefined.

A state of an EFA is a combination of the current location and the current valuation. The set of states is $Q = L \times V$, the initial state is $q^0 = (l^0, v^0)$, and the marked states are $Q^{\mathrm{m}} = L^{\mathrm{m}} \times V^{\mathrm{m}}$. A transition $(l, g, \sigma, u, l') \in \to$ is enabled in a state (l, v) if $v \models g$. This transition updates the location to l' and the valuation to $u(v)$. An event is enabled in a state if a transition labeled with this event is enabled in this state. An EFA is deterministic if in each state there is at most one transition enabled for each event. Given the nature of the infrastructural systems, only deterministic EFAs are considered in this thesis.

The transition function on states, $\delta : Q \times \Sigma \to Q$, is defined as follows. $\delta(q, \sigma) = q'$ if and only if there exists a $(l, g, \sigma, u, l') \in \to$ that is enabled in $q = (l, v)$ and the state is updated to $q' = (l', u(v))$. With $\delta(q, \sigma)!$ it is denoted that an enabled transition exists from state q labeled with event σ. The set of eligible events in state q is defined as $\mathrm{Elig}(q) = \{\sigma \in \Sigma \mid \delta(q, \sigma)!\}$.

The transition function can be extended in a natural way to strings. For string $s \in \Sigma^*$ and event $\sigma \in \Sigma$, $\delta(q, s\sigma) = \delta(\delta(q, s), \sigma)$, and $\delta(q, s\sigma)!$ if and only if $\delta(q, s)!$ and $\delta(\delta(q, s), \sigma)!$. For the empty string ϵ, $\delta(q, \epsilon) = q$ and $\delta(q, \epsilon)! = \mathbf{T}$.

State q is called reachable if there exists a string $s \in \Sigma^*$ such that $\delta(q^0, s) = q$. An EFA is called nonblocking if for every reachable state q, there exists string $s \in \Sigma^*$ such that $\delta(q, s) \in Q^{\mathrm{m}}$. EFA E_1 is called controllable with respect to EFA E_2 (with set of uncontrollable events Σ_{u}) if for every $s \in \Sigma_{12}^*$ and $\sigma \in \Sigma_{\mathrm{u}}$ such that $\delta_{12}(q_{12}^0, s)!$ and $\delta_2(q_2^0, s\sigma)!$, then $\delta_{12}(q_{12}^0, s\sigma)!$, where subscripts 2 and 12 refer to elements belonging to E_2 and $E_1 \parallel E_2$, respectively.

Two EFAs in an EFA system can be combined by using the synchronous product. Let $E_k = (L_k, X_k, \Sigma_k, \rightarrow_k, l_k^0, L_k^{\mathrm{m}})$ be EFAs, $k = 1, 2$. The synchronous product of E_1 and E_2 is

$$E_1 \parallel E_2 = (L_1 \times L_2, X_1 \cup X_2, \Sigma_1 \cup \Sigma_2, \rightarrow, (l_1^0, l_2^0), L_1^{\mathrm{m}} \times L_2^{\mathrm{m}})$$

where the transition relation $\rightarrow$ is defined as:

- if $\sigma \in \Sigma_1 \cap \Sigma_2$, then $((l_1, l_2), g_1 \wedge g_2, \sigma, u_1 \oplus u_2, (l_1', l_2')) \in \rightarrow$, for all $(l_1, g_1, \sigma, u_1, l_1') \in \rightarrow_1$ and $(l_2, g_2, \sigma, u_2, l_2') \in \rightarrow_2$.
- if $\sigma \in \Sigma_1 \setminus \Sigma_2$, then $((l_1, l_2), g_1, \sigma, u_1, (l_1', l_2)) \in \rightarrow$, for all $(l_1, g_1, \sigma, u_1, l_1') \in \rightarrow_1$ and $l_2 \in L_2$.
- if $\sigma \in \Sigma_2 \setminus \Sigma_1$, then $((l_1, l_2), g_2, \sigma, u_2, (l_1, l_2')) \in \rightarrow$, for all $(l_2, g_2, \sigma, u_2, l_2') \in \rightarrow_2$ and $l_1 \in L_1$.

In the definition above, $g_1 \wedge g_2$ evaluates to $\mathbf{T}$ for some $v \in V$ if $v \models g_1$ and $v \models g_2$. 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. This is well-defined, as X_1 and X_2 are disjoint. If a guard contains a location $l \in L_1 \cup L_2$, the location is substituted by $\mathbf{T}$ if for that transition $l = l_1$ or $l = l_2$, and by $\mathbf{F}$ otherwise.

$\mathcal{E}_{\parallel} = E_1 \parallel \ldots \parallel E_m$ denotes the synchronous product of all the component models in EFA system $\mathcal{E}$.

An example of synchronizing EFAs is shown in Figure 2.2. The EFA system is $\mathcal{E} = \{\texttt{Button}, \texttt{Lamp}\}$, and $X_{\mathcal{E}} = \{\texttt{Q}\}$, where `Q` is a Boolean variable. In the graphical representation, the keywords **when** and **do** are used to denote guards and updates, respectively. For the local variables, the initial value is denoted near the initial-location arrow. In this example, the events `c_on` and `c_off` are only enabled when EFA `Button` is in location `Pushed` and in location `Released`, respectively. When `c_on` or `c_off` occur, the value of `Q` is negated. The synchronous product is shown in Figure 2.3. Transitions with a guard that always evaluates to $\mathbf{F}$ are not displayed.

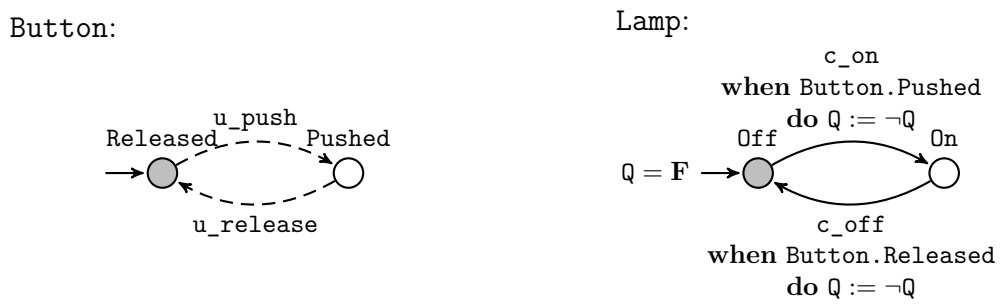


Figure 2.2: Graphical representation of EFAs.

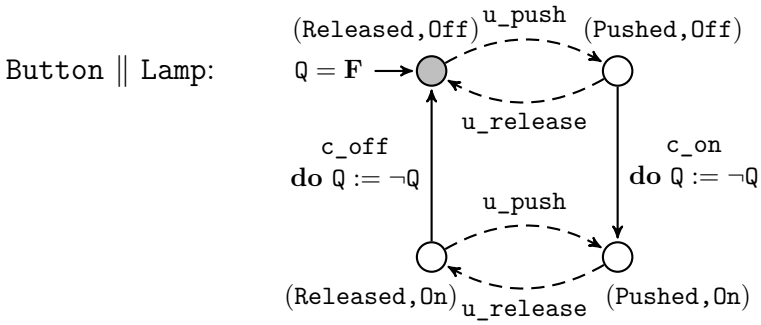


Figure 2.3: Synchronous product of EFAs `Button` and `Lamp`.

2.2 Modeling of requirements

For supervisor synthesis, the desired behavior of a system is given by a requirements model. A requirements model can be defined, just like the plant model, using a collection of EFAs. EFAs are especially useful when the order of events is important. Typically, other requirements, such as safety requirements, can be formulated more concisely using state-based requirements, introduced in Ma and Wonham (2006) and extended in Markovski et al. (2010). State-based requirements come in two types, event conditions and state invariants.

Event-condition requirements provide conditions for an event to be enabled. For event e and condition c, e **needs** c defines that e may only occur whenever c evaluates to **T**. Opposite, c **disables** e defines that e may only occur whenever c evaluates to **F**. Conditions are defined similar to guards used in the extended transition relation. An event-condition requirement can also be represented as an EFA, such that the synchronous product with another EFA can be computed. The EFA representation of an event-needs-condition requirement is as shown in Figure 2.4. A condition-disables-event requirement can be modeled similarly, by replacing c with $\neg c$.

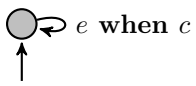


Figure 2.4: EFA representation of an event-condition requirement e **needs** c.

State-invariant requirements restrict the behavior of the plant by prohibiting combinations of states. For condition Y over the location variables of the plant, all locations where Y evaluates to **F** are prohibited. The synchronous product of an EFA and a state-invariant requirement can be computed by removing in the EFA all transitions to locations where Y evaluates to **F**.

2.3 Supervisor synthesis

Supervisory control theory, initiated by Ramadge and Wonham (1987), provides a method to derive a supervisor for a system. Given a model of the plant and a model

of the requirements, a supervisor can be synthesized automatically. The supervisor restricts the behavior of the system (together called the supervised system), such that the following properties are always satisfied:

Safety The supervised system cannot reach states or enable events that are forbidden by the requirements.

Controllability The supervisor is controllable with respect to the plant.

Nonblockingness The supervised system is nonblocking.

Maximal permissiveness The supervisor imposes the minimal restriction on the system to satisfy safety, controllability, and nonblockingness.

It should be noted, that 'safety' as used here, differs from the definition used by Rijkswaterstaat. Rijkswaterstaat uses the term 'safety requirement' when the requirement is necessary to prevent human injuries or damage to the system.

By applying monolithic supervisor synthesis (e.g., with the algorithm of Ouedraogo et al. (2011)), a single supervisor is synthesized to control the plant. In that case, a single EFA is returned that represents this supervisor.

As an example, consider EFAs `V` and `W` from Figure 2.1 and requirement `R`: '`c_process` **needs** `V.A`'. The synchronous product of `V`, `W`, and `R` is shown on the left-hand side of Figure 2.5. As can be seen, location (`B`, `D`) is blocking. Applying supervisor synthesis on `V` ‖ `W` and `R` results in supervisor `S`, shown the right-hand side EFA in Figure 2.5. To resolve the blocking issue, event `c_produce` is disabled in location (`A`, `D`).

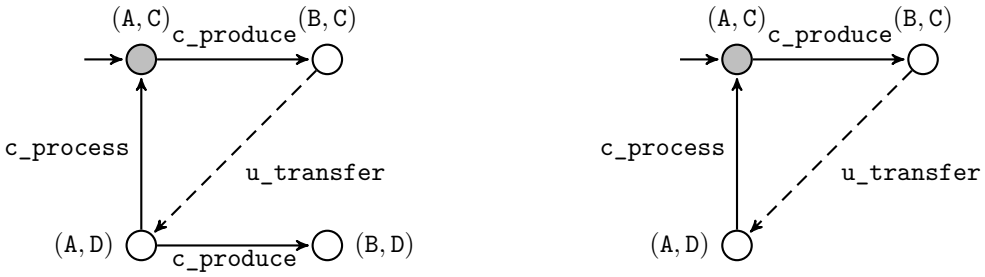


Figure 2.5: Synchronous product of `V`, `W`, and `R` (left) and supervisor `S` synthesized for `V` ‖ `W` and `R` (right).

For large state spaces, returning a supervisor represented by a single EFA becomes infeasible. The method of Miremadi et al. (2011) allows for a compact representation of the synthesis result. It characterizes the restrictions of the supervisor as guards extracted during the synthesis procedure. The result is an EFA with a single location and for each controllable event in the plant a selfloop with the extracted guard. The extracted guards can further help modelers to understand why some events become disabled after synthesis. The supervisor is then represented by the original collection of component models, the original collection of requirement models, and the extracted guards.

As an example, consider supervisor S from Figure 2.5. Instead of returning this EFA, the method of Miremadi et al. (2011) returns EFA G, shown in Figure 2.6. The supervisor is now represented by V ‖ W ‖ R ‖ G.

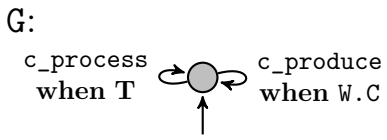


Figure 2.6: Synthesis result G for V ‖ W and R, using the method of Miremadi et al. (2011).

As already noted in Fabian et al. (2014), representing the synthesis result as guards has further advantages when analyzing the supervisor. Often, for many events, synthesis does not introduce additional guards (Goorden and Fabian, 2019). This implies that the supervisor does not have to impose extra restrictions on these events to satisfy nonblockingness and controllability. Similarly, sometimes events have guards that always evaluate to **F**, which indicates that these events are never enabled. This is useful information when the behavior of the supervisor has to be validated. This information is not trivially obtained for a supervisor represented as an EFA.

2.4 Normalized models

For mathematical analysis of an EFA system, it can be useful to transform the EFA system to a normalized EFA (NEFA). In an NEFA, all state information is captured by the variables and for all events a single guard function and a single update function are defined. The NEFA representation is used in Chapter 4.

The first step of the transformation is to make the location pointer of each EFA explicit. For this, the algorithm defined in Swartjes et al. (2014) can be used. In short, for each EFA $E_i \in \mathcal{E}$ a variable $\mathtt{LP}_i$ is added to $X_{\mathcal{E}}$. This variable is local to EFA E_i. The domain of $\mathtt{LP}_i$ is L_i, the initial value $v^0(\mathtt{LP}_i) = l_i^0$, and the marked valuations are defined such that they correspond to the marked location. Each transition in the EFA is adjusted, $(l, g, \sigma, u, l') \in \rightarrow$ is changed to $(l, g \wedge \mathtt{LP}_i = l, \sigma, u \oplus \mathtt{LP}_i := l', l')$, to explicitly relate the guard and update to the location pointer.

The second step of the transformation is to transform the EFA system with location pointers to a locationless EFA (LEFA) system, as introduced in Alenljung et al. (2007). An LEFA is defined as $E^{\mathrm{L}} = (X, \Sigma, \hat{\rightarrow})$, with $\hat{\rightarrow} \subseteq G \times \Sigma \times U$.

Definition 1 (LEFA transformation)**.** *Let $E = (L, X, \Sigma, \rightarrow, l^0, L^m)$ be an EFA, with location pointer $\mathtt{LP} \in X$. The corresponding LEFA is $E^L = (X, \Sigma, \hat{\rightarrow})$, with $\hat{\rightarrow} = \{(g, \sigma, u) \mid (l, g, \sigma, u, l') \in \rightarrow\}$.*

The third step of the transformation is to synchronize the LEFAs in the LEFA system. Two LEFAs can be synchronized, similar to EFA synchronization, as follows.

Definition 2 (LEFA synchronization)**.** *Let $E_i^L = (X_i, \Sigma_i, \hat{\rightarrow}_i)$, with $i = 1, 2$ be LEFAs, the synchronized LEFA is $E_1^L \parallel E_2^L = (X, \Sigma, \hat{\rightarrow})$, where $X = X_1 \cup X_2$, $\Sigma = \Sigma_1 \cup \Sigma_2$, and $\hat{\rightarrow}$ is defined as follows.*

- *if* $\sigma \in \Sigma_1 \cap \Sigma_2$*, then* $(g_1 \wedge g_2, \sigma, u_1 \oplus u_2) \in \hat{\rightarrow}$ *for every* $(g_1, \sigma, u_1) \in \hat{\rightarrow}_1$ *and* $(g_2, \sigma, u_2) \in \hat{\rightarrow}_2$
- *if* $\sigma \in \Sigma_1 \setminus \Sigma_2$*, then* $(g_1, \sigma, u_1) \in \hat{\rightarrow}$ *for every* $(g_1, \sigma, u_1) \in \hat{\rightarrow}_1$
- *if* $\sigma \in \Sigma_2 \setminus \Sigma_1$*, then* $(g_2, \sigma, u_2) \in \hat{\rightarrow}$ *for every* $(g_2, \sigma, u_2) \in \hat{\rightarrow}_2$

The final step of the transformation is to combine all the transitions for the same event. In this way, for each event exactly one transition is defined. The result is an NLEFA.

Definition 3 (NLEFA transformation)**.** *Let* $E^L = (X, \Sigma, \rightarrow)$ *be an LEFA. The corresponding NLEFA is* $E^N = (X, \Sigma, \hat{\rightarrow})$*, with* $\hat{\rightarrow} = \{(g_\sigma, \sigma, u_\sigma) \mid \sigma \in \Sigma\}$*, with* $g_\sigma = \bigvee_{(g,\sigma,u)\in\rightarrow} g$ *and* $u_\sigma = \bigoplus_{(g,\sigma,u)\in\rightarrow} g \Rightarrow u$*.*

In the above definition, $\bigoplus$ denotes the extension of the update function, where $g \Rightarrow u$ indicates that if g evaluates to $\mathbf{T}$, the valuation is updated according to u (a conditional update). This is well-defined, as for deterministic NLEFAs there is at most one condition that holds.

In an NLEFA, for every event σ, a *global guard expression* g_σ is defined, such that σ is enabled if and only if g_σ evaluates to $\mathbf{T}$. Additionally, u_σ denotes the *global update function* for σ.

As an example, consider EFA system $\mathcal{E} = \{\mathtt{V}, \mathtt{W}, \mathtt{G}, \mathtt{R}\}$, from the previous section. The NLEFA representation of $\mathcal{E}$ is shown in Figure 2.7. The global guard and the global update for each event are given below the events, in the figure. The marked state (not shown in the figure) is the state where $\mathtt{LP\_V} = \mathtt{A}$ and $\mathtt{LP\_W} = \mathtt{C}$.

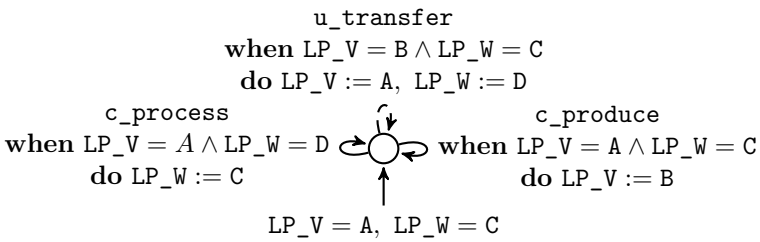


Figure 2.7: NLEFA representation of $\mathcal{E}$.

It can be shown that the behavior of a deterministic EFA system with location pointers is equal to its corresponding NLEFA. This means that every event sequence that is enabled in the EFA system is also enabled in the NLEFA, and vice versa, see, e.g., Swartjes et al. (2014).

2.5 Synthesis algorithm

The synthesis algorithm used in this thesis is based on the algorithm presented in Ouedraogo et al. (2011). The algorithm has been implemented in CIF (van Beek et al., 2014) in a previous project. The implementation uses binary decision diagrams (BDDs) for efficient calculations of the predicates used in the algorithm.

The algorithm consists of three steps 1) for each location, calculate the good-state predicate, 2) for each location, calculate the bad-state predicate, and 3) for each transition labeled with a controllable event, adapt the guard. The good-state predicate indicates for a location, for which valuations a marked state can be reached. The bad-state predicate indicates for a location, for which valuations no marked state can be reached, or (uncontrollably) a state can be reached, from where no marked state can be reached. Since bad-states must be avoided, in step 3, the guards on edges labeled with controllable events are updated such that bad-state predicates cannot evaluate to **T**.

The good-state predicate is calculated as follows.

1. Initially, the good-state predicate for a location l consists of marked-location predicate $M_l = l \in L^{\mathrm{m}}$ that is **T** for marked locations and **F** otherwise and marked valuation predicate $M_v = v \in V_{\mathrm{m}}$ that is **T** for marked valuations and **F** otherwise.
$$N_l := M_l \wedge M_v$$

2. For each location, compute a new good-state predicate as follows. The new good-state predicate of a location is the old good-state predicate N_l and, for each outgoing transition (l, g, σ, u, l') of the location, a disjunction of the form $g \wedge N_{l'}[u]$. $N_{l'}[u]$ denotes the old good-state predicate of the target location, where all occurrences of the variables are replaced by the expression they are assigned in the update of the transition.
$$N_l = N_l \vee \bigvee_{\substack{(l,g,\sigma,u,l') \\ \sigma \in \Sigma}} (g \wedge N_{l'}[u])$$

3. Repeat step 2 until the good-state predicates do not change anymore.

The bad-state predicate is calculated as follows.

1. Initially, the bad-state predicated for a location l is the negation of the good-state predicate of the location.
$$B_l := \neg N_l$$

2. For each location, compute a new bad-state predicate as follows. The new bad-state predicate of a location is the old-bad state predicate and, for each outgoing transition (l, g, σ, u, l') of the location that is labeled with an uncontrollable event, a disjunction of the form $g \wedge B_{l'}[u]$. $B_{l'}[u]$ denotes the old bad-state predicate of the target location, where all occurrences of the variables are replaced by the expression they are assigned in the update of the transition.
$$B_l := B_l \vee \bigvee_{\substack{(l,g,\sigma,u,l') \\ \sigma \in \Sigma_{\mathrm{u}}}} (g \wedge B_{l'}[u])$$

3. Repeat step 2 until the bad-state predicates do not change anymore.

The guard on a transition (l, g, σ, u, l') labeled with a controllable event is adapted as follows.

$$g := g \wedge \neg B_{l'}[u]$$

Whenever a guard has been adapted, all three steps are repeated. When no guard has been adapted, the synthesis process terminates. In case that for the initial valuation, the bad-state predicate of the initial location evaluates to $\mathbf{T}$, no supervisor can be synthesized and the algorithm also terminates.

In CIF, the EFA system is first transformed into an NLEFA, see Section 2.4. In that case, there is exactly one good-state predicate, one bad-state predicate, and one updated guard per controllable event. For a compact representation of the synthesis result, an EFA with one location is constructed that contains for each controllable event a self-loop transition with the guard created in step 3, as shown in Section 2.3.

Chapter 3

Modeling of infrastructural systems

Even though supervisory control theory has been a subject of research since the mid 80s, reports on realistic industrial applications are few in numbers. As is stated in Wonham et al. (2018), this is partly due to the lack of acquaintance of control engineers with modeling and specifying in the framework of automata, the lack of adequate tooling, and the computational complexity when synthesizing supervisors for industrial systems. Moreover, in Grigorov et al. (2011) and Zaytoon and Riera (2017) it is noted that obtaining the necessary models for supervisor synthesis is difficult, as there exist no clear guidelines on how to develop them.

In the literature, there are a few reports on applications of supervisor synthesis. The first application was the rapid thermal multiprocessor, described in Balemi et al. (1993). By far, most cases described in the literature focus on the domain of manufacturing systems; e.g., Leduc and Wonham (1995), Brandin (1996), Lauzon et al. (1996), Kim et al. (2001), de Queiroz and Cury (2002), Chandra et al. (2003), Nourelfath and Niel (2004), Ljungkrantz et al. (2007), Pétin et al. (2007), Hasdemir et al. (2008), Moor et al. (2010), Silva et al. (2011), van der Sanden et al. (2015), and Pena et al. (2016). Other applications are theme park vehicles (Forschelen et al., 2012), chemical process control (Rawlings et al., 2014), patient support table for an MRI scanner (Theunissen et al., 2013), smart homes dedicated to disabled people (Guillet et al., 2014), mobile robots (Lopes et al., 2016), computer science (Liao et al., 2013; Auer et al., 2014; Atampore et al., 2019), and driver assistance systems (von Bochmann et al., 2017; Korssen et al., 2018). In Reijnen et al. (2017), we reported on an application related to a lock control system in Tilburg. With a few exceptions, many of these case studies were based on experimental set-ups to show the feasibility of using supervisor synthesis.

A small industrial application is the control of a patient support table for an MRI scanner (Theunissen et al., 2013). For this application, three actuators are controlled based on observations of eight sensors, resulting in a plant state space of 5×10^4 states.

This chapter is based on: Reijnen, F.F.H., Goorden, M.A., van de Mortel-Fronczak, J.M., and Rooda, J.E. (2020). "Modeling for supervisor synthesis – A lock-bridge combination case study". In: *Discrete Event Dynamic Systems* vol. 30, no. 3, pp 499-532.

Here, finite-state automata are used to model the plant and the requirements. A second small industrial application is the control of the oxide growth process on a silicon wafer (Balemi et al., 1993). The plant model, consisting of eight components, has a state space of 10^6 states. The plant and requirements are modeled with finite-state automata. The driver assistance system, considered in Korssen et al. (2018), consists of 28 components, modeled by finite-state automata, leading to a plant state space of 3.4×10^9 states. Differently from the previous examples, event conditions are used to represent the requirements. Another application is the control of a theme park vehicle, described in Forschelen et al. (2012). Here, six actuators are controlled by a supervisor based on the observations of eleven sensors. The plant state space of the theme park vehicle is 1.7×10^{10} states. The requirements model is represented by a combination of finite-state automata and event conditions. While for most of these applications the necessary synthesis models are shown, they do not provide guidelines on how these models should be obtained. Also, the number of components involved in the cases is relatively low compared to systems encountered in industrial practice.

Modeling of the plant is discussed in Balemi et al. (1993), Chandra and Kumar (2002), and Grigorov et al. (2011). In Balemi et al. (1993), an input-output perspective is proposed. They show how the plant components can be modeled based on the inputs and outputs of the control unit. In Chandra and Kumar (2002), a modeling formalism for the plant is proposed. Here, the models also follow the input-output perspective of Balemi et al. (1993). Furthermore, they provide a method that derives conditions on the occurrence of events in the plant. These conditions represent the interactions between components in the system. In Grigorov et al. (2011), the use of templates is introduced. Templates allow to model a plant consisting of many similar components in a relatively straightforward way, greatly decreasing the modeling time and effort.

Modeling of the requirements is discussed in Markovski et al. (2010), Theunissen (2015), and Göbe et al. (2016). Originally, for supervisor synthesis, requirements are modeled with finite-state automata. In Ma and Wonham (2006) and Markovski et al. (2010), this is expanded with the introduction of event-condition requirements, which are stated to be more intuitive and compact. Event-condition requirements specify conditions for an event to be enabled, based on propositional logic. The advantages of using event-condition requirements is further shown in Göbe et al. (2016), where the authors reported improvements in terms of modeling time and the clarity of the resulting models. In Theunissen (2015), a supervisor has been synthesized for the control of a patient support table for an MRI scanner based on automata models of the requirements and based on event-condition models of the requirements. When comparing the models, it was concluded that the event-condition requirements are more concise and more intuitive to understand. These papers do not provide guidelines on how a requirements model can be derived.

The contribution of this chapter is twofold. Firstly, it proposes guidelines to obtain the plant model and the requirements model, necessary for supervisor synthesis. Secondly, it reports on a real infrastructural system, the Algera complex, for which a supervisor has been synthesized. This case study illustrates the proposed guidelines and shows the feasibility of using supervisor synthesis for industrial system.

For modeling the plant, we choose the abstraction level of inputs and outputs of the control unit. This has two advantages. Firstly, the models can be used for the generation of implementation code. Secondly, this leads to many small loosely-coupled (component) models for the sensors, actuators, and operator commands in the system. This way of modeling has a close resemblance to component-based modeling, frequently applied in software engineering, see e.g., Gössler and Sifakis (2005). The similarity between many of these component models can be exploited such that they can be modeled using templates. Furthermore, we show that this abstraction level, along with event-condition models, leads to a requirements model that relates closely to the specifications control engineers are acquainted with in practice. Finally, we show how the plant model can be augmented with continuous behavior such that it can be used for simulation-based validation, further aiding the supervisor design process.

To demonstrate the way of modeling, we report on an industrial application for which a supervisor has been synthesized: the Algera complex located in the Netherlands. The system consists of a waterway lock together with a movable bascule bridge over the lock. The supervisor has to control 80 actuators based on the observations from 96 sensors, and in response to 63 operator commands. Even though the plant state space consists of 2.3×10^{57} states and the supervisor has to adhere to 491 requirements, it is shown that a monolithic synthesis algorithm is able to solve the synthesis problem. The design description and the models for the Algera complex are available in a repository[1].

This chapter is structured as follows. Section 3.1 describes the Algera complex. In Section 3.2, guidelines for obtaining the necessary models for synthesis are given. The models developed for the Algera complex and the synthesized supervisor are discussed in Section 3.3. Section 3.4 discusses how the synthesized supervisor is validated. Finally, Section 3.5 concludes this chapter.

3.1 Case study: the Algera complex

The Algera complex, shown in Figure 3.1, is located in the Hollandse IJssel, a river in the Netherlands. The system is part of the Delta Works: a series of locks, storm surge barriers, levees, and dams that protect the Netherlands from the sea. The building of these works was initiated after the North Sea flood of 1953.

The Algera complex consists of a lock, a bascule bridge, and two storm surge barriers. In case of an extremely high sea-water level, the storm surge barriers (80 m x 12 m) are closed to protect the inland. The complex is located close to Rotterdam, between Krimpen aan den IJssel and Capelle aan den IJssel. Because of its location close to Rotterdam, it is a part of an important shipping route. Whenever the storm surge barriers are closed, the adjacent lock is used to raise or lower vessels (up to 24 m in width) between the different water heights. The lock gates are strong enough to withstand the extremely high water level. Additionally, in case the storm surge barriers are open, the lock in combination with the bascule bridge is used as a route

[1] www.github.com/ffhreijnen/AlgeraComplex

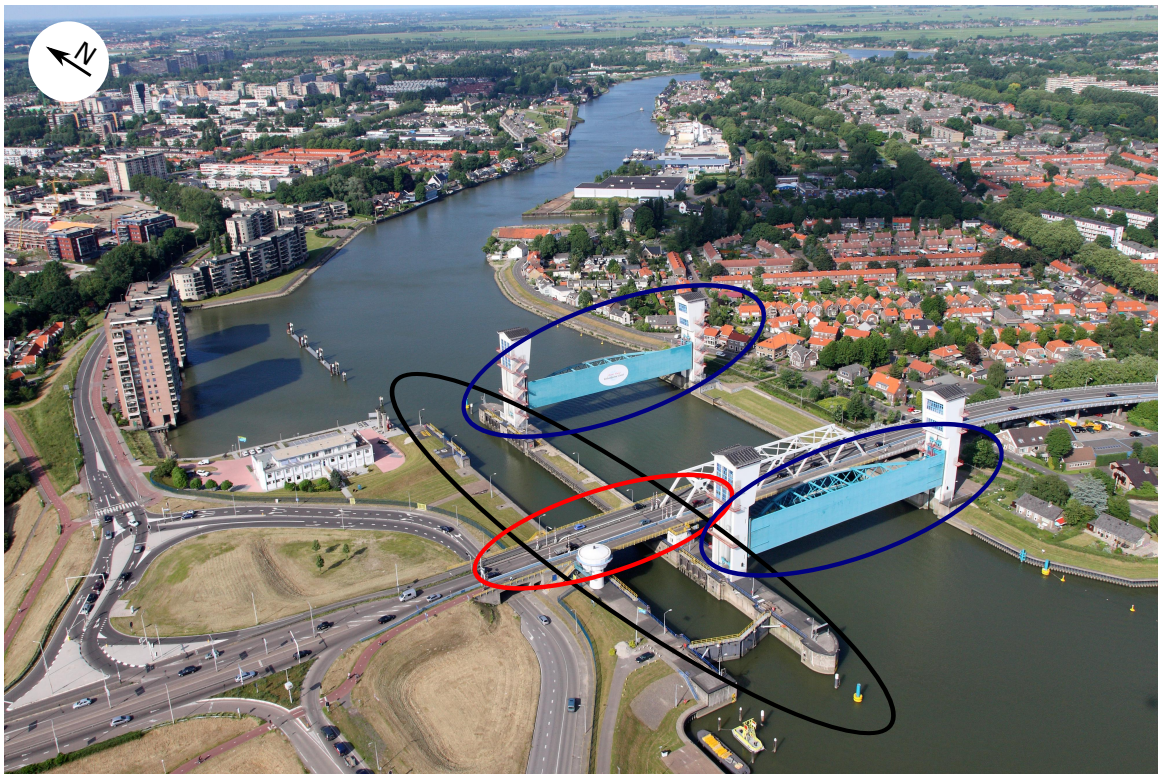


Figure 3.1: The Algera complex consisting of a lock (encircled in black), a bascule bridge (encircled in red), and two storm surge barriers (encircled in blue) [https://beeldbank.rws.nl, Rijkswaterstaat / Joop van Houdt].

for vessels that are too high to pass under the storm surge barriers. When the bridge is open, tall sailing ships can pass the Algera complex.

A human operator controls the Algera complex from a nearby control center, where the complex can be viewed via camera images. The operator is responsible for communicating with arriving vessels, giving commands via a control panel to the system, and monitoring the system.

In the remainder of this section, the functions of the lock, of the bascule bridge, and of the control panel are described in more detail. The storm surge barriers operate independently of the lock and of the bridge, and are not considered further.

3.1.1 Description and functionality of the Algera lock

The Algera lock, schematically depicted in Figure 3.2, is used to facilitate raising and lowering of vessels between different water heights. To this end, a chamber is used that can be separated from the rest of the river by watertight mitre gates. The water level inside the chamber can be varied by opening paddles in the gates.

Because of the tide, the water height outside the lock varies. As a result, the water level at the sea side is sometimes higher and sometimes lower than the water level at the river side. Because the gates are kept closed by the force generated from the difference in water height, at least two types of gate sets are used at each side, flood gates and ebb gates. When the sea-side water level is higher than the river-side water

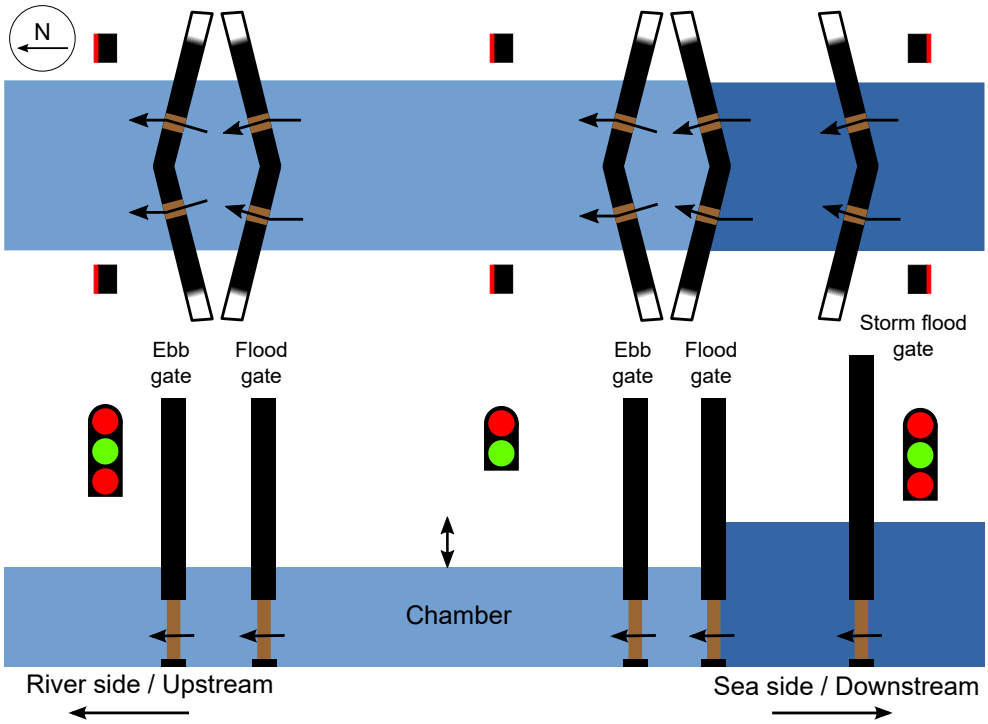


Figure 3.2: A schematic representation of the Algera lock when all gates are closed, all paddles are open, and the sea side is the high-water side. View from above (top) and view from the side (bottom).

level, the flood gates are used. Opposite, when the river-side water level is higher, the ebb gates are used. At the sea side, additional gates are used for safety in case of a storm flood.

The water level inside the chamber can be regulated. Each gate is equipped with a paddle that covers a hole in the gate. By opening this paddle, water can flow into or out of the lock chamber, filling or emptying the chamber, respectively.

To communicate with vessels outside the lock, two lock traffic lights (red-green-red) are used per side (i.e., river side and sea side). A lock traffic light can display four different aspects, shown in Figure 3.3 on the left-hand side. The double-red aspect indicates that the lock is out-of-service. The red, red-green, and green aspects indicate that entering the lock is not allowed, almost allowed, and allowed, respectively.

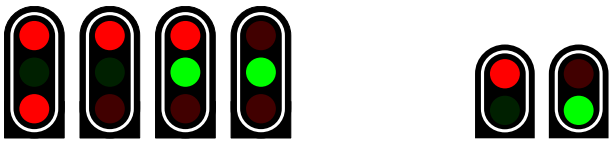


Figure 3.3: The aspects of the lock traffic light: double-red, red, red-green, and green (left), and the aspects of the bridge traffic light: red and green (right).

Inside the lock, two bridge traffic lights (red-green) are used to communicate with vessels. These traffic lights are positioned in front of the bridge at the sea side of the lock. They have two functions: to communicate whether it is safe to pass under the bridge and to communicate whether it is safe to exit the lock. At the river side, no

bridge traffic light is present. A bridge traffic light can display a red or green aspect, shown on the right-hand side of Figure 3.3, having a similar meaning as for the lock traffic light.

To safely open the gates, the water height is measured at three different locations: at the river side, inside the lock, and at the sea side. This information is combined to determine if there is an (almost) equal water level over a set of gates. When water heights are equal, it is safe to open a set of gates.

Desired controlled behavior

The desired behavior of the system is as follows. Consider a vessel intending to pass from the sea side to the river side of the lock, while the flood gates at both sides are closed. The current tide is flood, meaning the sea side is the high-water side (as shown in Figure 3.2). First, the chamber is filled by opening the paddles in the flood gates at the sea side of the lock. Subsequently, when there is equal water, the flood gates are opened. During opening, the lock traffic lights are set to the red-green aspect. When the gates reach the open position, the lock traffic lights are set to the green aspect, allowing the vessel to enter. Once the vessel has entered the lock, the lock traffic lights are set to the red aspect and the gates and paddles are closed. The water level is then lowered by opening the paddles in the flood gate at the river side of the lock. Finally, when there is equal water at the river sea, the flood gates are opened and the vessel can leave. Whenever the vessel is too high to pass under the bridge, the bridge has to be open before the vessel can enter the lock. For vessels traveling in the opposite direction, the process is similar.

3.1.2 Description and functionality of the Algera bridge

The Algera bridge, schematically depicted in Figure 3.4, is used by land traffic, e.g., motorized traffic, cyclists, and pedestrians, to cross the Hollandse IJssel river. It consists of four lanes: a slow-traffic lane (for cyclists and pedestrians), two lanes for motorized traffic, and an additional rush-hour lane for motorized traffic. The rush-hour lane reverses traffic directions during the evening rush-hours. Whenever high vessels have to pass the bridge, the bridge deck is swung upwards to provide clearance. To safely open the bridge, land traffic has to be warned and stopped first.

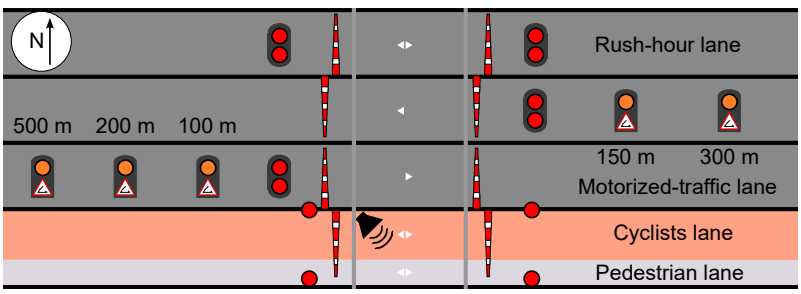


Figure 3.4: A schematic representation of the Algera bridge. Orange lights represent approach signs and red lights represent stop signs. White triangles display the traffic direction.

To warn motorized traffic in advance, approach signs are positioned before the bridge, visualized by orange circles in Figure 3.4. At the west side, approach signs are positioned at 100 m, 200 m, and 500 m before the bridge. At the east side, two approach signs are located at 150 m and 300 m before the bridge. At both sides, approach signs are shared between the standard lane and the rush-hour lane. Additionally, at both sides, stop signs are located close to the bridge, visualized by red circles in Figure 3.4. Each traffic lane has a set of two stop signs. The rush-hour lane and slow-traffic lane both have a set of two stop signs at each side of the bridge. Eight boom barriers are used to close the bridge for land traffic. The boom barriers are opened and closed by electric motors. To warn the slow traffic, an extra buzzer is installed near the boom barriers. Another, more powerful, motor is used to open and close the bridge deck. An additional feature of the bridge is its connection to the emergency service center. The center can request to keep the bridge closed if there is an emergency.

Desired controlled behavior

The desired behavior of the system is as follows. Consider a sailing ship intending to pass under the bridge. First, on the bridge, the approach signs are activated, and after 15 seconds the stop signs are activated. The buzzer to warn slow traffic activates 20 seconds after the approach signs. When traffic is safely stopped, the operator gives a command to close the boom barriers for the motorized traffic. This is done in two steps, first the entering boom barriers (i.e., the boom barriers that block the motorized traffic from entering the bridge) are closed. Subsequently, the leaving boom barriers (i.e., the boom barriers that block the motorized traffic from leaving the bridge) are closed. Once these boom barriers are closed, both slow-traffic boom barriers are closed by the operator. At the same time, the boom barriers of the rush-hour lane are closed. The order depends on the direction of traffic at that moment. When the land traffic has safely been stopped, the bridge deck can be opened. The bridge is closed in the reversed order.

3.1.3 Description and functionality of the control panel

The Algera complex is operated from a control center nearby, where human operators monitor the complex using camera images. For communication with vessels and bridge users, marine radios and loudspeakers are available, respectively. An operator controls the lock and the bridge from a graphical user interface (GUI) implemented on a PC. The PC is connected via an optical fiber connection to the controller at the Algera complex. The important part of the GUI for the Algera complex is shown in Figure 3.5. Clickable buttons (e.g., start leveling and open gate) are used to give commands to the supervisor. In total, there are 63 commands available to the operator. Not all of these commands are visualized in Figure 3.5, some windows will only show when that specific component is clicked (e.g., clicking on a gate or barrier). The state of the system is also visually displayed as feedback for the operators. For example, the position of the gates, the position of the barriers, the aspect shown to the vessels, and the water heights are visualized.

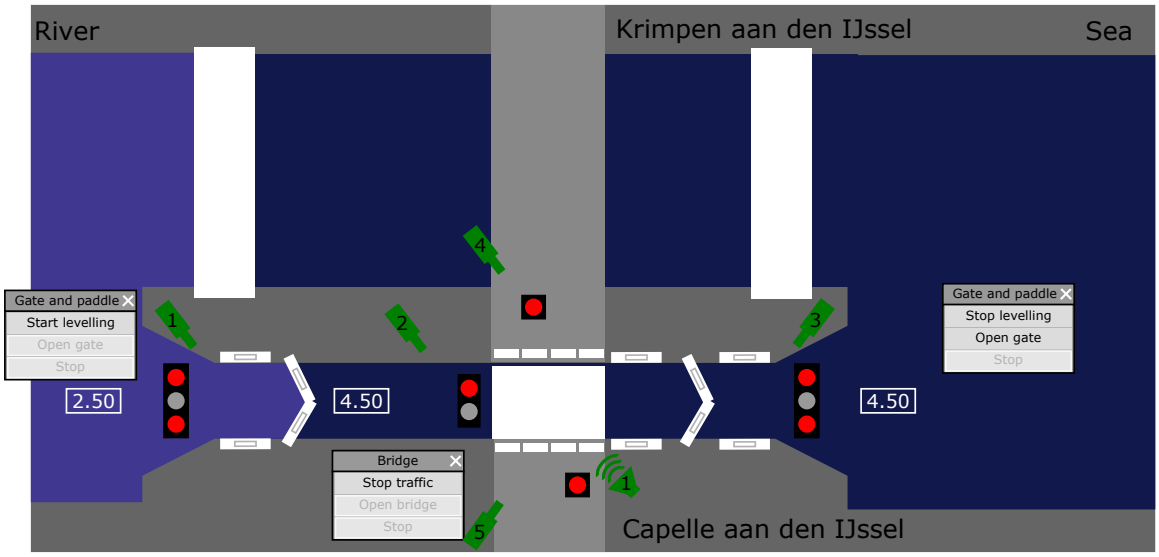


Figure 3.5: Part of the graphical user interface of the Algera complex.

3.2 Modeling method

In this section, a method to obtain the necessary models for supervisor synthesis is described. The focus is on obtaining models that can be used to synthesize a supervisor. Based on this supervisor, controller code can be generated. The task of the supervisor is to achieve the specified system's behavior by turning on or off actuators, based on the value of the sensors.

For modeling the plant, we apply ideas from component-based modeling. More specifically, we use (small) models for the components and glue the models by interaction models, to obtain a model of the plant. For the requirements, we identify textual formats that can straightforwardly be translated into models. This method produces models that show similarities to the models of a theme park vehicle (Forschelen et al., 2012), a patient support table for an MRI scanner (Theunissen et al., 2013), a FESTO production line (Reijnen et al., 2018a), and a driver assistance system (Korssen et al., 2018). While these papers all present their models in detail, none of them discusses a method to obtain those models. In this section, firstly component-based modeling is discussed. Secondly, a method for obtaining a plant model is given. Finally, different types of textual formats are discussed that can straightforwardly be modeled.

3.2.1 Component-based modeling

Component-based modeling is a modeling paradigm that uses the fact that large systems can be obtained by assembling smaller components, i.e., building blocks. Each component can be modeled separately. These components can be reused in different parts of the system. This way of modeling has proven to be successful for software-engineering applications (Crnkovic, 2001). The advantages of component-based modeling are observed to be useful for modeling for supervisor synthesis as well, as discussed in Kovács and Piétrac (2009), Kovács et al. (2012), and Huang et al.

(2015). However, they do not discuss how these component models can be obtained for other applications.

In Gössler and Sifakis (2005), composition of component-based models is discussed. There, behavioral models and interaction models are used. Behavioral models describe the dynamics of components. Interaction models describe the constraints on the behavior of components caused by other components. In this way, large systems can be composed. In the rest of this thesis, we refer to physical relation models instead of interaction models.

In supervisory control papers, e.g., Balemi et al. (1993) and Roussel and Giua (2005), it is shown that it is advantageous to model the plant based on the inputs and outputs (IOs) of the control unit of the system. As the interface is already present in the model, the implementation of the supervisor is straightforward. The inputs correspond to sensors and (digital) commands, and the outputs correspond to actuators. These models also fit the component-based modeling framework: each sensor, command, and actuator can be modeled as a separate (reusable) component, and the physical relations between the components can be modeled as interaction models. Subsequently, these models can be composed to obtain the plant model.

This way of modeling produces loosely-coupled component models for all sensors, commands, and actuators in the plant. This is advantageous, as there is a lot of similarity between these component models, allowing for the re-use of models via templates. In Grigorov et al. (2011), it is shown that the use of templates greatly reduces modeling time and effort for the plant. This reduced effort is even more noticeable when component templates are reused in different projects.

3.2.2 The plant model

When modeling the plant, the first step is to make component models for all sensors, commands, and actuators in the plant. The IO signals can be divided in four groups: Boolean input and output signals, and integer input and output signals (originating from analog signals).

Boolean input and output signals

For Boolean signals, a component model can be obtained by modeling the value of the signal as a location and the change of the value as an event. Uncontrollable and controllable events are used to represent changes in the values of inputs (i.e., sensors) and outputs (i.e., actuators), respectively. Secondly, the initial location and marked locations should be chosen. Typically, the marked locations are all 'safe' locations. In Figure 3.6, examples of models for Boolean signals are shown.

Figure 3.6: Component models for a Boolean input (left) and a Boolean output (right).

Sometimes, it is desired to have one event that changes two signals at once. For example, for a traffic light, an event can be used to represent a switch between aspects, instead of separate events for switching individual lamps. In that case, multiple signals can be modeled as one component. An example of this is provided in Section 3.3.1.

Integer input and output signals

For integer sensor signals, the values of the signals are mapped to a set of discrete states. Which states to choose depends on the requirements that are modeled later. Events relate to changes between the states. For example, consider a water tank with an integer signal from a water-height sensor. The signal value ranges between 0 and 100. Assume that two requirements are defined. The first requirement states that a pump may only start when the value drops below a lower threshold (< 20). The second states that a pump may only stop when the value exceeds an upper threshold (> 80). Consequently, three states can be identified, below the lower threshold, above the upper threshold, and nominal. The mapping between the signal value and the state set is as shown on the left-hand side of Figure 3.7. The component model is as shown on the right-hand side of Figure 3.7.

Value	State	Abbreviation
0 - 19	Below	B
20 - 80	Nominal	N
81 - 100	Above	A

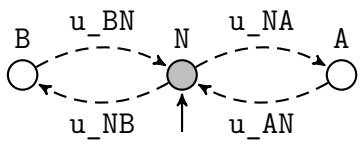


Figure 3.7: Mapping between signal value and states (left) and component model for the sensor (right).

For integer actuator signals, the states of the component model are mapped to signal values. For example, consider an integer signal for a filling pump. The signal value ranges between 0 and 100. For the requirements, it is necessary to distinguish between no flow (0), low flow rate (30), and high flow rate (100). The mapping between the states and the signal value is then as shown on the left-hand side of Figure 3.8. The component model is as shown on the right-hand side of Figure 3.8.

State	Value	Abbreviation
No	0	N
Low	30	L
High	100	H

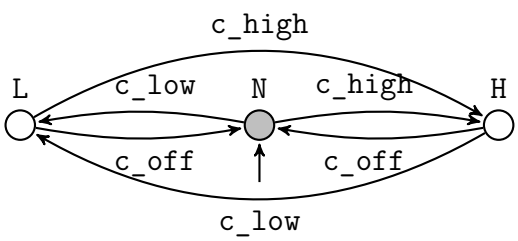


Figure 3.8: Mapping between states and signal value (left) and component model for the actuator (right).

Physical relation models

Aside from modeling the behavior of the individual components, the physical relations (or interaction models in Gössler and Sifakis (2005)) between the different components need to be modeled as well. In Zaytoon and Carré-Ménéatrier (2001), it has been shown that not including these physical relations may lead to deadlocks in real systems that have been proven to be deadlock-free in the model. That is because the modeled behavior contains actions that cannot occur in the real system. These relations are present if a sensor measures the behavior of a certain actuator, or between two sensors that measure the same actuator.

As an example, consider the filling pump and the water-height sensor from Section 3.2.2. For this example, it is only possible to measure an increase in water height whenever the filling pump is on. This physical relation has to be modeled explicitly in order to correctly capture the behavior of the plant. In this method, we choose to model the physical relations as guards. In Figure 3.9, this physical relation model is shown. Here, `S` and `A` refer to the previous sensor and actuator model, respectively.

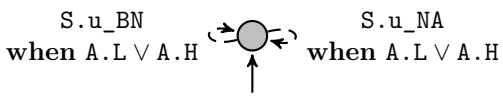


Figure 3.9: The physical relation model.

In many cases, deriving the physical relations between components is straightforward, as they are often simple. Alternatively, the physical relations between components can be derived via the method of Chandra and Kumar (2002). There, they derive the relations from a hybrid model, similar to the hybrid model we use for simulation, see Section 3.4.

3.2.3 The requirements model

To model the requirements, the notions used in the textual requirements should relate to events and locations in the plant model. Because of the component-based modeling, this means that the requirements should relate to sensors, commands, or actuators. While this might seem restrictive, this is also how control engineers program PLCs in practice. Furthermore, observers can be used to do state reconstruction, such that information that is not directly available from the sensors can be used. For example, in Sampath et al. (1995), observers are used to diagnose whether a fault has occurred.

To ease the requirement modeling process, we identify textual requirements formats that can straightforwardly be modeled. The requirements model can then be obtained by reformulating requirements in design documents to these formats. We consider four forms: event-condition requirements, event-order requirements, timer-based requirements, and state-invariant requirements. Experience with case studies, for example, in Markovski et al. (2010), Forschelen et al. (2012), and Reijnen et al. (2018a), has shown that many requirements can be reformulated in this way. In the following subsections, for each requirement type it is discussed which textual requirement it

represents and how it can be modeled. The textual requirement formats, directly leading to formal models, are given in the Backus-Naur Form (BNF) notation.

Event-condition requirements

Event-condition requirements are expressions that specify when an event is allowed to occur, based on a condition in the form of propositional logic. The textual form of these requirements in BNF is:

<component> *may only* | *may not* <event> *when* <condition>

where <component> refers to a component model, <event> to an event in this component model, and <condition> to a propositional logic formula over the variables and locations in the plant. This textual requirement can be modeled with event-condition requirements as defined in Section 2.2. A *may only* requirement is modeled as in (3.1), whereas a *may not* requirement is modeled as in (3.2).

$$\texttt{component.event}\ \textbf{needs}\ \texttt{condition} \tag{3.1}$$

$$\texttt{component.event}\ \textbf{needs}\ \neg\texttt{condition} \tag{3.2}$$

An example of a textual requirement in this form is: *The gate may only close when the traffic light shows a red aspect.* Here, *the gate* is the component, *close* is the event, and *the traffic light shows a red aspect* is the condition. This condition can be expressed by variables from the plant models: the current location of the red and green sensor should be on and off, respectively.

Event-order requirements

Event-order requirements specify in which order events are allowed to occur. The textual forms of these requirements in BNF is:

First, <component> *may* <event> [*when* <condition>]{, *then*, <component> *may* <event> [*when* <condition>]}

Here, [ ] and { } denote an optional argument and a (zero or more) repeating argument, respectively. This textual requirement can be modeled with an EFA requirement, where each step is a transition with an (optional) condition. An example of a textual requirement in this form is: *First, the traffic light may show a red-green aspect when the gate is open, then the traffic light may show a green sign aspect, then the traffic light may show a red sign aspect.* Modeling of this event-order requirement is as shown in Figure 3.10.

Timer-based requirements

Timer-based requirements are expressions specifying that an event may only occur after a certain condition holds for a minimum time interval. The textual form of this requirement in BNF is:

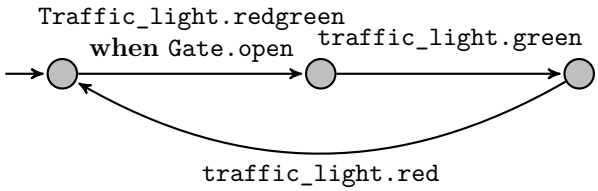


Figure 3.10: Example of an event-order requirement.

<component> *may only* <event> *x time units after* <condition>

Above, the definitions of <component>, <event>, and <condition> are similar to the event-condition requirements.

To model a timer-based requirement, a timer component is introduced. A timer measures how long a condition `condition` holds. The timer can start when the condition is satisfied and can stop when the condition is no longer satisfied. If the timer is running, a timeout event represents that the condition holds long enough. A model of this timer is shown in Figure 3.11. The requirement is modeled as shown in (3.3). Note that in this model, the time is not explicitly modeled. This is included later, see Section 3.4.1.

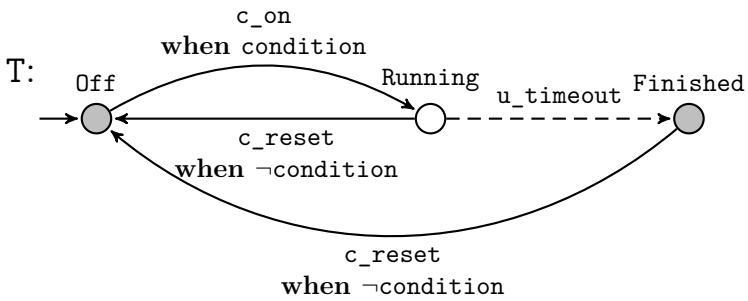


Figure 3.11: Model of a timer for condition `condition`.

$$\texttt{component.event}\ \textbf{needs}\ \texttt{condition} \wedge \texttt{T.Finished} \tag{3.3}$$

An example of a textual requirement in this form is: *The boom barrier may only close 10 seconds after the warning signs are enabled.* Here, *The boom barrier* is the component, *close* is the event, and *the warning signs are enabled* is the condition.

State-invariant requirements

State-invariant requirements are expressions that specify conditions that must always hold. The textual form of this requirement in BNF is:

<condition>

An example of a textual requirement in this form is *Gate 1 and gate 2 may not be open simultaneously.* This condition can be expressed in terms of elements of a plant model like: 'not (`gate1.sensor.open` and `gate2.sensor.open`)'.

3.3 Model development

To synthesize a supervisor for the Algera complex, a model of the plant and a model of the control requirements are required. For the plant model, a set of component templates is used, as recommended in Section 3.2. In Section 3.3.1, the templates for the plant models are introduced. The plant and requirements models for the Algera lock and the Algera bridge are described in Sections 3.3.2 through 3.3.6. In Section 3.3.7, the synthesis result is discussed.

3.3.1 Plant component templates

The modeling of the plant is based on the inputs and the outputs of the Algera (PLC) control unit. The full list of control inputs and outputs, on which the templates are based, can be found in the repository. Based on this list, a set of templates has been defined. These templates are re-usable for different components in the system. For the Algera complex, in total 16 templates are used to represent the behavior of 176 actuators and sensors and 63 commands. The templates are provided in the subsequent subsections.

Single input - single output template

An often encountered combination is a single actuator (output) with a single sensor (input) for feedback, for example, an approach sign. Both the actuator and the sensor are modeled by an automaton consisting of two locations, `On` and `Off`, shown in the upper left and upper right of Figure 3.12, respectively. As is usual, the actuator events are controllable (denoted by `c_`) and the sensor events are uncontrollable (denoted by `u_`). The physical relation between those components is that the sensor can only switch on (or off) after the actuator has been activated (or deactivated). These physical relations are shown as the bottom EFA in Figure 3.12.

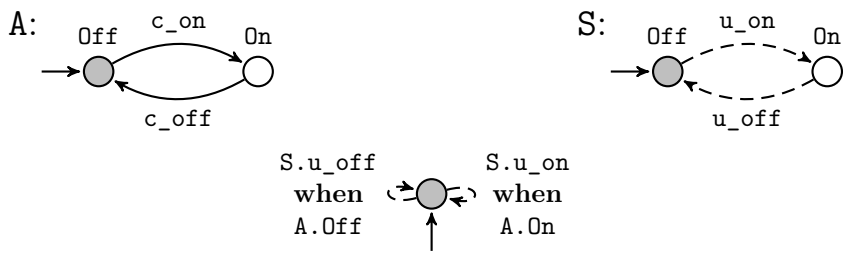


Figure 3.12: Template of single output actuator `A` (left), single input sensor `S` (right), and the actuator-sensor physical relations (bottom).

Double input - double output template

Another often encountered combination is an actuator that can move in two directions (two outputs) together with two end-position sensors (two inputs), for example, an electric cylinder actuating a lock gate. Since it is never desired to actuate in both

directions, this behavior is blocked by a low-level controller. Instead, only the `Closing`, `Idle`, and `Opening` behavior is included in the actuator template, shown in the upper left of Figure 3.13. There are two stop events: `c_emrgStop` and `c_endStop`, to distinguish between an emergency stop and a regular end-position stop. The two end-position sensors (`S_Closed` and `S_Open`) are modeled as the sensor from Figure 3.12. There is a physical restriction that both sensors cannot be on simultaneously; this is modeled as the upper-right automaton in Figure 3.13. The relation between the actuator and the sensors is that the sensors can only switch on or switch off when the actuator is moving in a certain direction, represented by the bottom automaton in Figure 3.13.

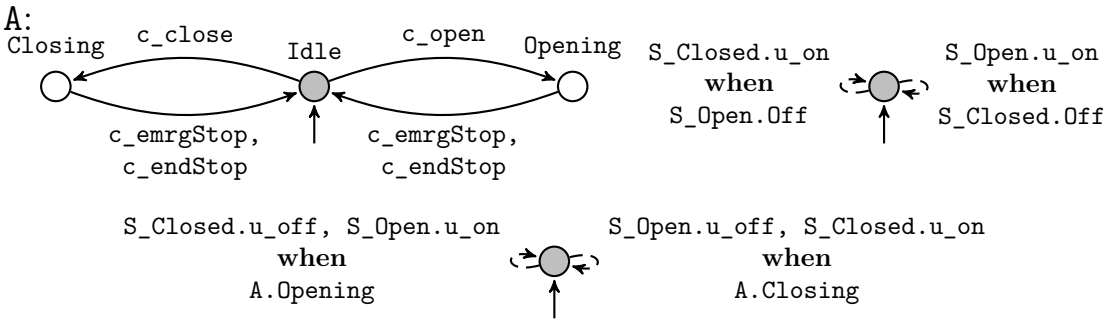


Figure 3.13: Template of double output actuator `A` (left), the sensor-sensor physical relations (right), and the actuator-sensor physical relations (bottom).

Traffic light template

The traffic light templates are used to represent the behavior of the lock traffic light and the behavior of the bridge traffic light. The lock traffic light consists of three outputs (an output for each individual lamp), whereas the bridge traffic light consists of two outputs. Each lamp is equipped with a sensor for feedback. Only a few output combinations are allowed, such that only legal aspects can be displayed. For the lock traffic light these are: `RedRed`, `Red`, `RedGreen`, and `Green`. For the bridge traffic light these are: `Red` and `Green`. Furthermore, some transitions are not allowed, such as switching from the `Red` aspect directly to the `Green` aspect for the lock traffic light.

A low-level controller makes sure that only legal aspects can be displayed. For the events, it is chosen to model an aspect switch as an event (instead of switching a lamp on or off). This is advantageous when specifying requirements as they also refer to aspect switches. Still, aspect switches can directly be related to control outputs. An additional event is used to represent the switch to the red aspect, in case of an emergency. The templates for the lock traffic light actuator and the bridge traffic light are shown in Figure 3.14 and Figure 3.15, respectively. The template for the sensors is similar to the sensor template in Figure 3.12 (one for each lamp); the differences are in the initial and marked location for the red lamp sensor (which is the `On` location). The interaction between the sensor and the actuator is that the sensor can only switch on or off when the lamp is activated or deactivated in the current aspect, respectively. For

the lock traffic light sensors (`S_Red`, `S_Green`, and `S_Red2`) and the bridge traffic light sensors (`S_Red` and `S_Green`), the physical relation models are shown in Figure 3.14 and Figure 3.15, respectively.

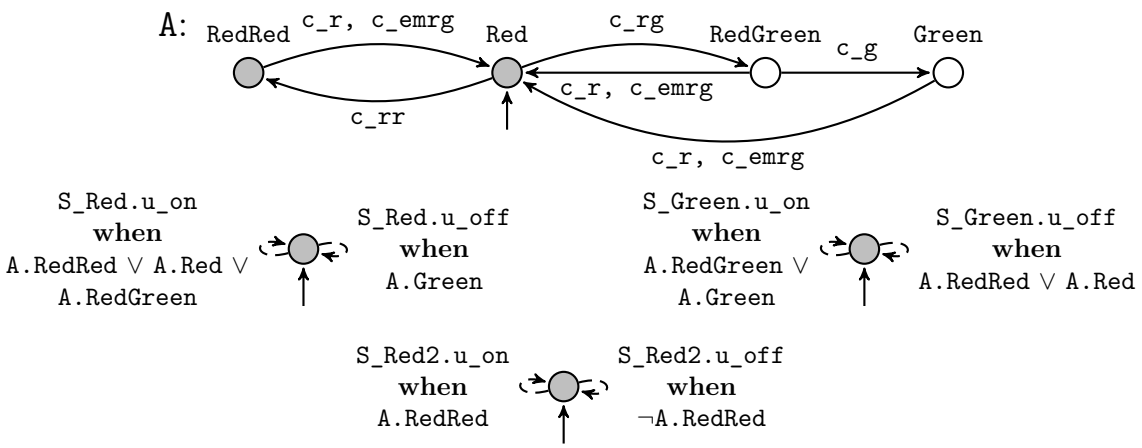


Figure 3.14: Template of the lock traffic light actuator `A` (top), the top red lamp sensor-actuator physical relations (middle left), the green lamp sensor-actuator physical relations (middle right), and the bottom red lamp sensor-actuator physical relations (bottom).

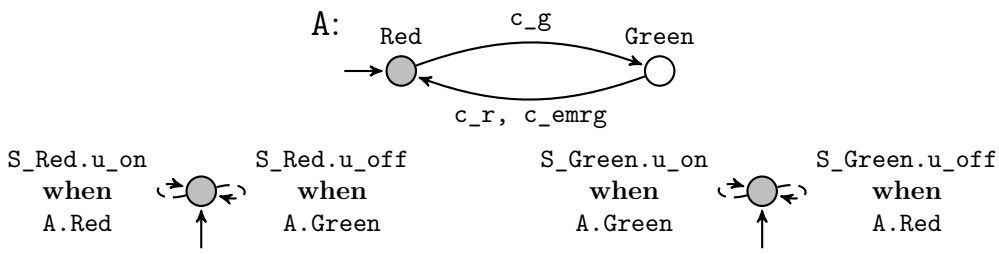


Figure 3.15: Template of the bridge traffic light actuator `A` (top), the red lamp sensor-actuator physical relations (bottom left), and the green lamp sensor-actuator physical relations (bottom right).

User-interface template

Commands from an operator are given via buttons, and are implemented in the graphical user interface. A variety of commands is available, for example, opening a boom barrier, changing traffic light aspects, or activating the emergency stop. There are different types of commands per component. Moving components (e.g., gates, paddles, and boom barriers) can be opened, closed, and stopped, whereas others are more specific, e.g., the lock and bridge traffic lights. The commands available for the moving components are modeled as the automaton on the upper left-hand side of Figure 3.16. Here, the behavior is such that different commands can never be active simultaneously. Instead, a new command overrules the old command, which is how the GUI is implemented. The emergency stop is modeled as the automaton on the upper right-hand side of Figure 3.16. The commands for the lock and bridge traffic light

are modeled on the bottom left-hand side and bottom right-hand side of Figure 3.16, respectively.

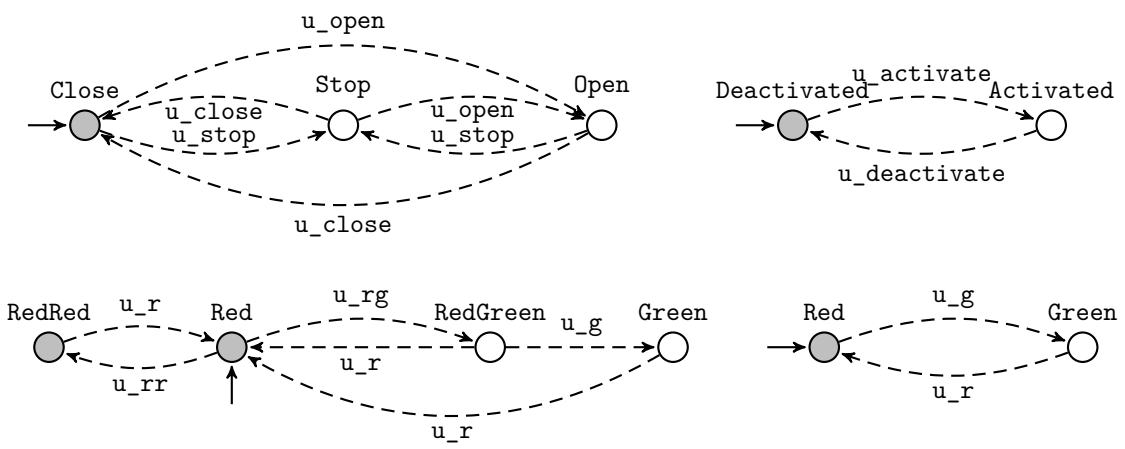


Figure 3.16: Template for the movable components commands (top left), for the emergency stop (top right), for the lock traffic light commands (bottom left), and for the bridge traffic light commands (bottom right).

3.3.2 Plant model of the Algera lock

The plant model for the Algera lock is based on the IO of the control unit that controls the lock, the commands available from the GUI, and the functional description of the components. The components of the lock can be divided into five distinct types: gates, paddles, lock traffic lights, bridge traffic lights, and equal water sensors. The behavior of the gates and paddles is modeled as the double input - double output template. In total, there are ten gates and ten paddles, all controlled individually. At both sides, there are two lock traffic lights. There are two bridge traffic lights inside the lock. The three analog water-height sensors are modeled as two discrete equal-water sensors. If the analog value of two water-height sensors differs by at most a specified margin, the equal-water sensor is on, otherwise it is off. The commands available to the operator relate to opening and closing a set of gates or paddles, or switching aspects. In Table 3.1, for each component, the model template, the number of instantiations, and the number of states are given.

3.3.3 Requirements model of the Algera lock

For the lock to function in a safe and desired manner, a set of textual requirements has been specified by Rijkswaterstaat. These requirements, as given in the design documents (available in the repository), are listed below. In the requirements, downstream and upstream refer to the sea side and the river side of the lock, respectively.

1. The lock traffic lights may only display a green aspect when:
   (a) the gates at that side are open, and
   (b) the bridge traffic lights at that side display a red aspect (downstream only).

Table 3.1: Component models for the Algera lock.

Component type	Component template	Number	States
Gate	Double input - double output	10	9
Paddle	Double input - double output	10	9
Lock traffic light	Lock traffic light	4	32
Bridge traffic light	Bridge traffic light	2	8
Equal water sensor	Single input sensor	2	2
Gate command	Movable component command	5	3
Paddle command	Movable component command	5	3
Lock traffic light command	Lock traffic light command	2	4
Bridge traffic light command	Bridge traffic light command	1	2
Emergency stop	Emergency stop	1	2

2. The bridge traffic lights may only display a green aspect when:
   (a) the gates at the downstream side are open, and
   (b) the lock traffic lights at that side display a red or a double-red aspect.
3. The gates may only close when:
   (a) the lock traffic lights at that side display a red or double-red aspect, and
   (b) the bridge traffic lights at that side display a red aspect (downstream only).
4. The gates may only open when:
   (a) at least one set of gates and its paddles at the other side is closed, and
   (b) there is equal water at that side.
5. The paddles may only open when at least one set of gates and its paddles at the other side is closed.
6. Whenever a gate is not closed, its paddles are open.
7. When the emergency stop is active:
   (a) the red aspect has to be displayed, and
   (b) no other aspect can be displayed.
8. When the emergency stop is active:
   (a) the moving components have to stop via the emergency stop, and
   (b) the moving components cannot start opening or closing.
9. Actuators have to stop when they reach their end position.
10. Actuators may only start when the operator gives the corresponding command.
11. Aspects may only be displayed when the operator gives the corresponding command.

Requirements 1-5, and 7-11 are modeled as event-condition requirements, which are given in Table 3.2. The events listed in the left column are only enabled when the condition in the right column is satisfied. Some requirements are listed twice, as they are imposed on both sides of the lock (e.g., Requirement 1a.). For brevity, abbreviations are used, these are listed in the table's caption. Note that Requirements 7 and 11 are imposed on every traffic light and Requirements 8, 9, and 10 are imposed on every gate and paddle. Requirement 6 is modeled as a state-invariant requirement, where `Gate.Closed` $\vee$ `Paddle.Open` should always be satisfied, for all ten sets of

gates and paddles. As can be seen, almost all requirements are of one of the forms defined in Section 3.2.3. Exceptions are Requirements 7a, 8a, and 9 that state that something should happen. For this, each component model contains an emergency event, which is only enabled when the emergency stop is activated.

Table 3.2: Event-condition requirements for the Algera lock. Abbreviations: `A`: Actuator, `S`: Sensor, `LTL`: Lock traffic light, `BTL`: Bridge traffic light, `.U`/`.D`: every gate, paddle, or traffic light at the upstream or downstream side, respectively, `E`/`F`/`SF`: every gate/paddle from the ebb, flood, or storm flood gate type, respectively. State abbreviations: `Open`: `S_Open.On` ∧ `A.Rest`, `Closed`: `S_Closed.On` ∧ `A.Rest`, `Red`: `S_Red.On` ∧ `S_Green.Off` ∧ `S_Red2.Off` ∧ `A.Red`, `RedRed`: `S_Red.On` ∧ `S_Green.Off` ∧ `S_Red2.On` ∧ `A.RedRed`.

Req.	Event(s)	Condition
1a	`LTL.U.A.c_g`	`Gates.U.Open`
1a	`LTL.D.A.c_g`	`Gates.D.Open`
1b	`LTL.D.A.c_g`	`BTL.D.Red`
2a	`BTL.D.A.c_g`	`Gates.D.Open`
2b	`BTL.D.A.c_g`	`LTL.D.Red` ∨ `LTL.D.RedRed`
3a	`Gates.D.A.c_close`	`LTL.D.Red` ∨ `LTL.D.RedRed`
3b	`Gates.D.A.c_close`	`BTL.D.Red`
3a	`Gates.U.A.c_close`	`LTL.U.Red` ∨ `LTL.U.RedRed`
4a	`Gates.D.A.c_open`	(`Gates.UE.Closed` ∧ `Paddles.UE.Closed`) ∨ (`Gates.UF.Closed` ∧ `Paddles.UF.Closed`)
4b	`Gates.D.A.c_open`	`EqualWater.D.On`
4a	`Gates.U.A.c_open`	(`Gates.DE.Closed` ∧ `Paddles.DE.Closed`) ∨ (`Gates.DF.Closed` ∧ `Paddles.DF.Closed`) ∨ (`Gates.DSF.Closed` ∧ `Paddles.DSF.Closed`)
4b	`Gates.U.A.c_open`	`EqualWater.U.On`
5	`Paddles.D.A.c_open`	(`Gates.UE.Closed` ∧ `Paddles.UE.Closed`) ∨ (`Gates.UF.Closed` ∧ `Paddles.UF.Closed`)
5	`Paddles.U.A.c_open`	(`Gates.DE.Closed` ∧ `Paddles.DE.Closed`) ∨ (`Gates.DF.Closed` ∧ `Paddles.DF.Closed`) ∨ (`Gates.DSF.Closed` ∧ `Paddles.DSF.Closed`)
7a	`A.c_emrg`	`EmrgStop.Activated`
7b	{`A.c_rr`, `A.c_rg`, `A.c_g`}	`EmrgStop.Deactivated`
8a	`A.c_emrgStop`	`EmrgStop.Activated` ∨ `Command.Stop`
8b	{`A.c_close`, `A.c_open`}	`EmrgStop.Deactivated`
9	`A.c_endStop`	(`A.Opening` ∧ `S_Open.On`) ∨ (`A.Closing` ∧ `S_Closed.On`)
10	`A.c_open`	`Command.Open`
10	`A.c_close`	`Command.Close`
11	`A.c_rr`	`Command.RedRed`
11	`A.c_r`	`Command.Red`
11	`A.c_rg`	`Command.RedGreen`
11	`A.c_g`	`Command.Green`

There are two advantages of using event-condition requirements instead of automata-based requirement models. The first advantage is size. All the event-condition requirements can also be modeled using FAs. This is done by taking the synchronous product of all the FAs related to a condition, and adding a selfloop of the event in the locations where the condition evaluates to **T**. For Requirement 5 this would result in an FA with 2.8×10^{11} locations. The second advantage is the similarity to concepts used by PLC control engineers, such as ladder diagrams and function block diagrams.

3.3.4 Plant model of the Algera bridge

The components of the Algera bridge can be divided into approach signs, stop signs, boom barriers, bridge deck, sound signals, and light signals. There are two control outputs to switch on the five approach signs: one control output for the two outer most, and one control output for the remaining three. Each approach sign is equipped with a sensor for feedback. The stop signs are controlled by two outputs: one control output for the rush-hour lane stop signs, and a second one for the other stop signs. Each stop sign is again equipped with a sensor for feedback. All the boom barriers are actuated with an electric motor that is controlled by two outputs, for moving upwards and for moving downwards. Each boom barrier has two end-position sensors. The bridge deck is actuated in a similar way, and also contains two end-position sensors. Furthermore, there is a buzzer close to the cyclists lane that can be activated and there are light signals on the boom barriers. Finally, sometimes emergency services request the bridge to be kept closed, which is an additional control input. For each component, Table 3.3 lists the model template, the number of instantiations, and the number of states.

Table 3.3: Plant models for the Algera bridge.

Component type	Component template	Number	States
Approach sign actuator	Single output	2	2
Approach sign sensor	Single input	5	2
Stop sign actuator	Single output	2	2
Stop sign sensor	Single input	12	2
Boom barrier	Double input - double output	8	9
Bridge deck	Double input - double output	1	9
Sound signal	Single output	1	2
Light signal	Single output	1	2
Close request	Single input	1	2
Land traffic stop command	Movable component command	1	3
Barrier command	Movable component command	5	3
Bridge deck command	Movable component command	1	3
Emergency stop	Emergency stop	1	2
Timer	Timer	8	3

3.3.5 Requirements model of the Algera bridge

Similar to the Algera lock, a set of textual requirements for the bridge has been specified by Rijkswaterstaat. These requirements, as given in the design documents (available in the repository), are listed below. Furthermore, Requirements 8-10 from Section 3.3.3 are also imposed on the bridge, but are not repeated here, for brevity.

1. The stop signs may only turn on 15 s after the approach signs are on.
2. The sound signal may only turn on 20 s after the approach signs are on.
3. The entering barriers may only close 15 s after the stop signs are on.
4. The leaving barriers may only close 1 s after the entering barriers are closed.
5. The rush-hour and slow-traffic barriers may only close when the leaving barriers are closed.
6. The slow-traffic barriers may only close 6 s after the sound signal is on.
7. The bridge may only open when all barriers are closed.
8. The barriers may only open when the bridge is closed.
9. The entering barriers may only open 1 s after the leaving barriers are open.
10. The stop signs may only turn off when the barriers are open.
11. The near approach signs may only turn off 60 s after the stop signs are off.
12. The far approach signs may only turn off 60 s after the near approach signs are off.
13. The barriers may not close and the bridge may not open when the close request has been given.

Requirements 1-13 are modeled as event-condition requirements, shown in Table 3.4. The events listed in the left column are only enabled when the condition in the right column is satisfied. The index i is used to distinguish between the different boom barriers. Entering boom barriers, leaving boom barriers, slow-traffic boom barriers, and rush-hour boom barriers are denoted by 3 and 6, 2 and 7, 4 and 8, and 1 and 5, respectively. As can be seen, all requirements can be expressed as defined in Section 3.2.3.

3.3.6 Requirements model of the Algera lock-bridge combination

For the Algera lock and Algera bridge combination to function properly, there are four requirements that express interaction between the two subsystems. These requirements are as follows:

1. The bridge may only move when:
   (a) the gates are not moving, and
   (b) the bridge traffic lights display a red aspect, and
   (c) the lock traffic lights display a red or double-red aspect.
2. The gates may only open or close when the bridge is not moving.
3. The bridge traffic lights may only display a green aspect when the bridge is fully open or fully closed.

Table 3.4: Event-condition requirements for the Algera bridge. Abbreviations: `A`: Actuator, `S`: Sensor, `LTAS`: land traffic approach signs, `LTSS`: land traffic stop signs. State abbreviations: `LTAS.On`: every approach sign sensor and actuator in state `On`, `LTSS.Off/On`: every stop sign sensor and actuator in state `Off/On`, `Barriers.Closed/ Open`: every barrier sensor in state `S_Closed.On/S_Open.On` and every actuator in state `Idle`.

Req.	Event(s)	Condition
1	{`LTSS.MainLane.A.c_on`, `LTSS.SwitchLane.A.c_on`, `Barriers.Light.A.c_on`}	`LTAS.On` ∧ `LTAS.On15Timer.Finished`
2	`Barriers.Sound.c_on`	`LTAS.On` ∧ `LTAS.On20Timer.Finished`
3	`Barriers.B`i`.A.c_close` $i \in \{3, 6\}$	`LTSS.On` ∧ `LTSS.On15Timer.Finished`
4	`Barriers.B`i`.A.c_close` $i \in \{2, 7\}$	`Barriers.B3.Closed` ∧ `Barriers.B6.Closed` ∧ `Barriers.B3B6Closed1Timer.Finished`
5	`Barriers.B`i`.A.c_close` $i \in \{1, 4, 5, 8\}$	`Barriers.B2.Closed` ∧ `Barriers.B7.Closed`
6	`Barriers.B`i`.A.c_close` $i \in \{4, 8\}$	`Barriers.Sound.On` ∧ `Barriers.SoundOn6Timer.Finished`
7	`Deck.A.c_open`	`Barriers.Closed`
8	`Barriers.B`i`.A.c_open` $i \in \{1, 2, 3, 4, 5, 6, 7, 8\}$	`Deck.Closed`
9	`Barriers.B`i`.A.c_open` $i \in \{3, 6\}$	`Barriers.B2.open` ∧ `Barriers.B7.open` ∧ `Barriers.B2B7Open1Timer.Finished`
10	{`LTSS.MainLane.A.c_off` `LTSS.SwitchLane.A.c_off`, `Barriers.Light.c_off`}	`Barriers.Open`
11	`LTAS.Near.A.c_off`	`LTSS.Off` ∧ `LTSS.Off60Timer.Finished`
12	`LTAS.Far.A.c_off`	`LTAS.Near.Off` ∧ `LTAS.NearOff60Timer.Finished`
13	`Barriers.B`i`.A.c_close` $i \in \{1, 2, 3, 4, 5, 6, 7, 8\}$	¬`CloseRequest.On`
13	`Deck.A.c_open`	¬`CloseRequest.On`

4. The lock traffic lights may only display a green aspect when the bridge is fully open or fully closed.

In Table 3.5, these requirements are defined formally.

3.3.7 Supervisor synthesis

A supervisor has been synthesized from the plant and requirements models. For synthesis, the CIF toolset (van Beek et al., 2014) has been used. The synthesis

Table 3.5: Event-condition requirements for the lock-bridge combination. Requirement 2 for the gates is imposed on every gate (ten in total).

Req.	Event(s)	Condition
1a	{`Deck.A.c_open`, `Deck.A.c_close`}	`Gates.D.Rest` ∧ `Gates.U.Rest`
1b	{`Deck.A.c_open`, `Deck.A.c_close`}	`BTL.D.Red`
1c	{`Deck.A.c_open`, `Deck.A.c_close`}	`LTL.U.Red` ∨ `LTL.RedRed`
2	{`Gate.A.c_open`, `Gate.A.c_close`}	`Deck.A.Rest`
3	`BTL.D.A.c_g`	`Deck.Closed` ∨ `Deck.Open`
4	`LTL.U.A.c_g`	`Deck.Closed` ∨ `Deck.Open`

algorithm implemented in CIF is based on the algorithm proposed in Ouedraogo et al. (2011), see Section 2.5. The implementation of the synthesis algorithm in CIF uses binary decision diagrams to represent the models symbolically during synthesis. BDDs can be used to compactly and effectively represent a large state space (Vahidi et al., 2006). Even if the number of states is large, the number of nodes in its corresponding BDD can still be manageable. The results of the synthesis procedure for the lock, the bridge, and the lock-bridge combination are shown in Table 3.6. 'Plant state space' denotes the number of states in the synchronous product of all component models. 'Supervisor state space' denotes the number of states in the synthesized supervisor. These numbers are extracted from the size of the BDD. For this case study, the supervisor state space is smaller than the plant state space, as the supervisor restricts the plant from reaching undesired or unsafe states.

Table 3.6: State-space sizes, numbers of requirements, and computation times.

System	Plant state space	Number of requirements	Supervisor state space	Computation time [s]
Lock	1.2×10^{34}	306	1.1×10^{22}	2
Bridge	1.9×10^{23}	155	4.2×10^{12}	114
Lock-bridge	2.3×10^{57}	491	4.5×10^{34}	2,080

When analyzing the synthesized supervisor (available in the repository), i.e., the guards returned by the synthesis algorithm, it is observed that extra guards are imposed on opening the gates and closing the paddles. These guards are imposed to satisfy the state-invariant requirement (Requirement 6 of Section 3.3.3). There are no extra guards on the other events to satisfy nonblockingness or controllability. In this case, the computation time for the three supervisors is reasonable, considering the state-space sizes.

3.4 Simulation-based validation of the synthesized supervisor

Although the system is guaranteed to behave according to the requirements, the resulting controlled behavior might not be as expected. This can be caused by the fact that beforehand it is not known whether the textual requirements are complete and

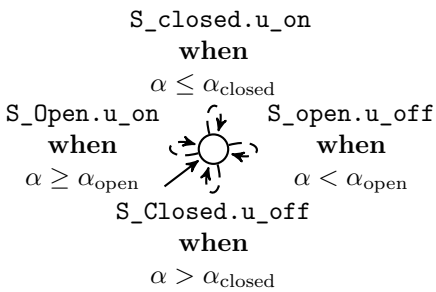
correct. For example, requirements could be too strict and as a result, the supervisor could prevent reaching parts of the desired behavior. Hence, the behavior of the controlled system has to be validated. Simulation with visualization is used to validate whether the behavior of the controlled system is consistent with the intended behavior. The separate modeling of the Algera lock and the Algera bridge is advantageous for validation. For this, both controlled systems have first been simulated independently, before being combined.

While it is possible to simulate the discrete-event model, it is more intuitive to validate the behavior using a more advanced simulation model in which hybrid behavior is included. For this, the discrete-event plant model is enriched with continuous behavior, using hybrid automata as defined in Henzinger (2000). The model of the hybrid plant is discussed in the next subsection and the validation in the subsection thereafter.

3.4.1 Hybrid plant model

A hybrid plant model is obtained by extending the discrete-event model used for synthesis with continuous behavior. Continuous behavior is modeled by introducing continuous variables that change their values due to the passing of time. How the value of a continuous variable evolves is defined by a differential equation that can depend, for example, on the current location of a component model (i.e., a location variable). Additionally, continuous variables may change their values during a state transition, which is modeled by an update.

For the components that move in two directions, such as boom barriers, which are represented by the double input - double output template of Section 3.3.1, the hybrid model is shown in Figure 3.17. Only the model of the physical relation between sensor and actuator is different from the discrete model. The left-hand side depicts this relation. The continuous variable α represents the angle of movement. Here, the sensor events occur depending on the value of α compared to a constant value, representing the fully closed or fully open movement angle, α_{closed} and α_{open}, respectively. The right-hand side lists the differential equations of continuous variable α. It states that α increases if the actuator is in the state `Opening` and the value of α is smaller than α_{open}. When the actuator is in the `Closing` location and larger than α_{closed}, the value of α decreases. In all other situations, the value of α remains constant.



$$\frac{d\alpha}{dt} = \begin{cases} +1 & \textbf{if } \texttt{A.Opening} \wedge \ \alpha < \alpha_{\text{open}} \\ -1 & \textbf{if } \texttt{A.Closing} \wedge \ \alpha > \alpha_{\text{closed}} \\ 0 & \textbf{otherwise} \end{cases}$$

Figure 3.17: Hybrid model of the two input - two output physical relation. `A.` denotes a reference to actuator `A`.

The discrete-event model of the timer, see Section 3.2.3, is extended with continuous behavior as well. The hybrid model is shown in Figure 3.18. On the left-hand side, the automaton model is shown. A continuous variable y is introduced to represent the remaining time of the timer. On the right-hand side, the differential equation of continuous variable y is shown. The value of y decreases when the timer is in the `Running` location. The event `c_start` updates the value of y to the desired timer value T. When $y \leq 0$, a transition to the `Finished` state occurs.

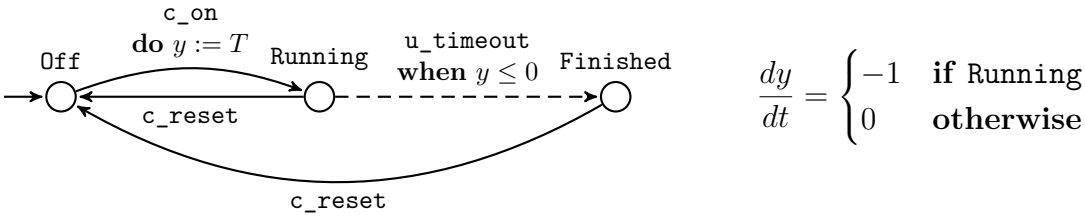


Figure 3.18: Hybrid model of a timer with duration T.

3.4.2 Visualization

The hybrid plant model is connected to a visualization of the system, shown in Figures. 3.2, 3.4, and 3.5. The properties of the objects in this image, e.g., color, visibility, rotation, and dimensions, are connected to the locations of automata and the values of continuous variables in the model. Here, we animate the behavior of the gates, the traffic lights, the boom barriers, and the bridge deck. The use of a simulation-based validation allows to visualize the behavior of the system, and in turn, makes validation more straightforward (Theunissen et al., 2013; Korssen et al., 2018).

3.4.3 Validation steps

The behavior of the (hybrid) plant model with respect to the real system has been validated as follows. Firstly, we derived all functionalities of the sensors and actuators from the design documentations. In these documents, the function of each actuator and sensor is described. Secondly, we consulted both the control engineers who maintain the current control system of the complex and the mechanical engineers that built the civil part.

The validation of the controlled system is accomplished by performing Factory Acceptance Tests (FAT) on the simulation model. The FAT protocols were obtained from Rijkswaterstaat. The protocols describe operator scenarios (e.g., which commands to give) and the required system's response. Typically, responses are starting a process when a command is given, or not executing a command when it is unsafe to do so. By subjecting the simulation to these tests, it can be checked whether the supervisor adheres to the requirements that Rijkswaterstaat specified for the control system. Furthermore, for Rijkswaterstaat, supervisor synthesis provides an analysis of the completeness of their requirements. In other words, it answers the question whether the set of specified requirements leads to desired controlled behavior described in the protocols. This can be checked as the supervisor is synthesized from these requirements.

The FAT focuses on three categories: 1) the behavior under normal conditions, 2) the behavior when the emergency stop is pushed, and 3) the behavior under component malfunctions. In this project, we focused on the first two categories. All tests showed the behavior as described in the FAT protocols, except one test. This was due to a missing requirement in the original specification. This requirement is related to the safe functioning of the gates in combination with the bridge traffic lights. We proposed a new requirement to obtain the correct behavior (Requirement 3b, of Section 3.3.3). In the meantime, Rijkswaterstaat has added this requirement to their set of safety requirements.

3.4.4 Discussion

While simulating the test scenarios increases the confidence in the correct behavior of the controlled system, it is not exhaustive, because only parts of the state space are explored. Although synthesis guarantees the absence of unsafe behavior in the other parts, it cannot guarantee the presence of desired behavior. For example, in general, it cannot be guaranteed that something should always happen. The validation could further be improved by verifying the desired behavior with properties from modal logic. For example, it would be useful to determine if the actuators always stop when the emergency button is pushed. Another approach would be to synthesize the supervisor such that it guarantees these properties by construction. For example, in Rawlings et al. (2014), synthesis is extended to work with CTL specifications, which makes specifying that something should happen possible.

Furthermore, it is known that the resulting controlled behavior is conform the requirements; yet, it is not always known if the requirements are correct and complete. For instance, in this case study we found a missing requirement and, therefore, the behavior of the controlled system was unsafe. In this case, we found this requirement because the test protocols described this behavior. However, that is not always the case. It would be beneficial to have a more systematic approach to validate the requirements beforehand.

3.5 Concluding remarks

The complexity and size of infrastructural systems in combination with the required functionality and demands on verified safety makes designing supervisors for these systems a challenging task. Supervisor synthesis is a useful method to obtain a supervisor that adheres to the specified requirements. However, control engineers lack acquaintance with modeling and specifying in the framework of automata. Besides this, in the related literature, no clear guidelines for obtaining the necessary models for synthesis are found.

In this chapter, guidelines for obtaining the plant and the requirements models are proposed. A case study on the Algera complex illustrates this way of modeling. The plant model has been obtained by representing all the sensors, actuators, commands, and physical relations as small component models. On the abstraction level of control inputs and outputs, many of these component models are similar. These similarities

allow for the use of templates, which greatly increases the quality of the models, while decreasing the modeling time. For this case study, 16 templates are used to model 239 components.

For the requirements model, the textual requirements are represented by event-condition models. This type of models allows for a straightforward translation of the textual requirements to logic-based expressions. Furthermore, the logic-based expressions relate closely to the way control engineers are acquainted with in practice. Aside from their similarity to the textual requirements, the size of the models is also considerably smaller than when FA models are used.

Simulation-based visualization is used to validate the resulting supervisors. Simulation allows to compare the behavior of the supervisor with the expected behavior that is, for example, described in FAT protocols. In this specific case study, we were able to identify a missing requirement by comparing the behavior of the controlled system with the expected behavior described in the FAT protocols. In the meantime, Rijkswaterstaat has added this requirement to the set of safety requirements.

The results described in the case study show that supervisor synthesis is applicable to large infrastructural system. Even though the plant model consists of 239 components, and is subjected to 491 requirements, a monolithic BDD-based synthesis procedure was able to derive the supervisor in about 35 minutes.

Chapter 4

Implementation of supervisory controllers for PLCs

In the design process of supervisory controllers (referred to as "controllers" in this chapter), implementation is the last step. For implementation, PLCs are commonly used in industry. When synthesis is used to obtain a supervisor, a controller and subsequently PLC code can be derived from that supervisor. In that way, the PLC code also satisfies the requirements by construction. However, using the synthesized supervisor for this purpose is no trivial task.

A supervisor as described in Ramadge and Wonham (1987) is meant to be implemented together with a separate controller. The supervisor monitors the behavior of the plant, and based on its observations, decides which actions to enable and which not. The task of the controller is to execute some of the enabled events, such as switching a motor on. The structure where the supervisor and the controller are two separate entities is referred to as supervised control (Charbonnier et al., 1995; Basile and Chiacchio, 2007; Pichard et al., 2018). This means that for supervised control, a separate controller has to be designed.

A different approach that is often taken (see Balemi et al. (1993), Reijnen et al. (2017), Vieira et al. (2017), and Prenzel and Provost (2018)), is to use supervisor synthesis to obtain a supervisor, and then 'interpret' this supervisor as a controller. In this way, the additional effort of designing a controller is avoided. This means that the controller monitors the behavior of the plant, enables some of the events, and then chooses which enabled event to execute. A downside of this approach is that even if the supervisor is nonblocking, this does not guarantee that the controller is also nonblocking, as shown in Dietrich et al. (2002). A blocking controller may have serious impact on the process it controls, such as it being impossible to achieve the desired behavior.

An additional aspect is the real-time implementation of the controller on a PLC (Fabian and Hellgren, 1998; Zaytoon and Riera, 2017). For supervisor synthesis, it is assumed that there is no communication delay between the plant and the

This chapter is based on: Reijnen, F.F.H., Hofkamp, A.T., van de Mortel-Fronczak, J.M., Reniers, M.A., and Rooda, J.E. (2020). "Implementation of state-based supervisory controllers", In preparation.

supervisor. In reality, there is a small delay. As a result, the guarantees provided by supervisor synthesis might not hold for the realization, see e.g., Balemi et al. (1993) and Rashidinejad et al. (2019).

In the literature, a few interesting case studies have been published that utilize supervisor synthesis and then interpret the supervisor as a controller, see e.g., Forschelen et al. (2012), Theunissen et al. (2013), Reijnen et al. (2017), Vieira et al. (2017), Korssen et al. (2018), and Prenzel and Provost (2018). All of these papers mention the aforementioned difficulties. Properties exist that can be used to verify that the controller is not affected by these difficulties (Fabian and Hellgren, 1998; Malik, 2003). For these cases, the properties were not verified. One of the reasons for this is that current synthesis and code generation tools cannot check them automatically, as mentioned in Vieira et al. (2017). Moreover, we believe that the solution to the delay problem as proposed in the literature (Fabian and Hellgren, 1998; Zaytoon and Riera, 2017) is often not applicable to real systems.

The main contributions of this chapter are as follows. First, an overview of the steps to go from a supervisor model to controller code is given, including potential problems that can be encountered during these steps. For each problem, it is shown how to verify that the controller is not affected by it. Second, it is shown how PLC code can be derived from the controller model. Finally, it is shown how optimization techniques can be applied to structure the generated code in such a way that the execution time and its variability can be reduced.

The chapter is structured as follows. The way of working of PLCs is explained in Section 4.1. The difference between a supervisor and a controller, and the steps to go from a supervisor model to controller code are provided in Section 4.2. Section 4.3 discusses the potential problems that may occur when a controller is derived from a supervisor and the solutions proposed in the literature. Moreover, it presents procedures to verify whether a given supervisor satisfies these properties. In Section 4.4, the problems related to the real-time implementation of the controller are shown. How to generate PLC code from the controller model is explained in Section 4.5. Section 4.6, proposes a way to minimizing the required computation time of the PLC. Finally, Section 4.7 concludes this chapter.

4.1 Programmable logic controllers

Programmable logic controllers are computers commonly used in industry to implement controllers. A PLC consists of a CPU that runs the controller code, input modules to which sensors are connected, and output modules to which actuators are connected. An input image and an output image are used to represent the sensor and actuator signals from the modules as variables in the PLC, such that the signals are usable in the program. A PLC operates in so-called scan cycles, as shown in Figure 4.1. In one PLC cycle, first, the get action copies the values of the input signals to the input image of the PLC program. Then, the controller code is executed. The controller reads from the input image and writes to the output image. Finally, the put action copies the values of the output image of the PLC program to the output signals. During the program execution, changes in the input signals are not registered. One full scan cycle

typically takes a few milliseconds. Fabian and Hellgren (1998), Hasdemir et al. (2008),

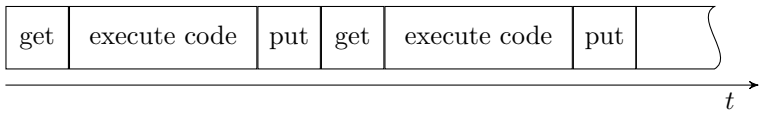


Figure 4.1: Two PLC scan cycles.

Leal et al. (2012), Vieira et al. (2017), and Prenzel and Provost (2018) demonstrate methods of executing an FA model on a PLC. In this chapter, a similar execution method is used. The code execution consists of the following steps.

1. Determine if the value of an input-image variable has changed and translate this change to an uncontrollable event.
2. Update the state of the controller by executing this event.
3. Determine if a controllable event is enabled in the new state, if so, execute this event, and update the state. Repeat this until no controllable event is enabled anymore.
4. Update the value of the output-image variables, depending on the state of the controller.

In step 3, a choice is made to execute all controllable events that are enabled. Compared to executing only one controllable event per cycle, the reaction time of the controller increases substantially when multiple events are possible (Prenzel and Provost, 2018).

In step 4, the output-image variables are updated according to the state of the controller, independent of which events have been executed to reach this state. When using the component-based modeling method described in Chapter 3, this means that the values of the output image variables are coupled to the locations of actuator components. For example, when a component model is in location `On`, the variable's value is true.

The IEC 61131-3 PLC standard (International Electrotechnical Commission, 2013) defines five standard programming languages for PLCs. From a discrete-event system model, code can be generated for most of these languages, as shown in the literature (Fabian and Hellgren, 1998; Hellgren et al., 2001; Hasdemir et al., 2008; Swartjes et al., 2017; Vieira et al., 2017; Prenzel and Provost, 2018; Reijnen et al., 2020d). In this chapter, structured text is used, being a textual language. Ladder diagrams, function block diagrams, and sequential function charts are graphical languages, and automatic code generation for these representations is more complicated.

The methods described in this chapter are also applicable to other real-time platforms. Matlab/Simulink implementations (Sharma and Reniers, 2016; Korssen et al., 2018) and micro-controller implementations (Torrico et al., 2016) work similar to the PLC implementations and are also prone to the issues described in the next section.

4.2 From supervisor model to controller code

A controller is a special kind of supervisor. The task of a controller is to execute events in the plant (events related to actuators) based on events received from the plant (events related to sensors). It is assumed that the controller executes controllable events as soon as they are enabled, to decrease its reaction time. A supervisor might allow multiple controllable events in a state. The task of the controller is then to choose one of these events.

Depending on how the controller chooses the controllable events, the derived controller may be blocking, even though the supervisor is not. For example, consider the supervisor shown on the left-hand side of Figure 4.2. In state `3` there is a choice between controllable events `b` and `c`. The controller that always chooses event `b` over event `c` is shown on the right-hand side of Figure 4.2. This controller is blocking as it is unable to return to the marked state `1`.

Figure 4.2: A supervisor that is nonblocking (left) and a derived controller that is blocking (right).

In general, a controller can be derived from a supervisor by choosing a controllable event each time more than one controllable event is enabled. To guarantee progress, a controller may not disable all of these controllable events. In Malik (2003), a derived controller is defined as follows.

Definition 4 (Derived controller)**.** *Let E_1 and E_2 be EFAs with state sets Q_1 and Q_2, marked state sets $Q_1^m \subseteq Q_1$ and $Q_2^m \subseteq Q_2$, initial states $q_1^0 \in Q_1$ and $q_2^0 \in Q_2$, and transition functions δ_1 and δ_2, respectively, both with the same event set Σ, that can be partitioned into controllable events Σ_c and uncontrollable events Σ_u. E_2 is said to be a controller derived from E_1 if $\forall(q_1, q_2) \in Q_1 \times Q_2$, such that $\exists s \in \Sigma^* : q_1 = \delta_1(q_1^0, s) \wedge q_2 = \delta_2(q_2^0, s)$, it holds that:*

- $Elig_1(q_1) \supseteq Elig_2(q_2)$
- $Elig_1(q_1) \cap \Sigma_u = Elig_2(q_2) \cap \Sigma_u$
- $|Elig_1(q_1) \cap \Sigma_c| \geq 1 \implies |Elig_2(q_2) \cap \Sigma_c| = 1$
- $q_1 \in Q_1^m \Leftrightarrow q_2 \in Q_2^m$

where $|\ \ |$ denotes the set size.

Note, that the choice between events does not necessarily have to be made based on the states of the supervisor. For example, for the supervisor shown in Figure 4.2, a different way to choose between events is to alternate between controllable events `b` and `c`. This results in the derived controller shown in Figure 4.3, which is nonblocking.

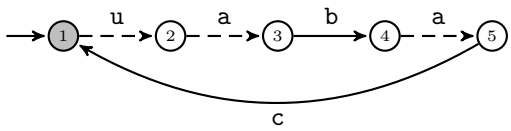


Figure 4.3: A derived controller that alternates between controllable events `b` and `c`.

In Malik (2003), three properties are provided for a supervisor, ensuring that any derived controller is nonblocking. These properties are: confluence, finite response, and nonblocking under control. In the next section, these properties are explained in detail. If a supervisor satisfies these properties, the choice between controllable events can be made arbitrarily, without having to worry about obtaining a blocking controller. Moreover, independently of the choices made, the same final state (the state where no controllable events are enabled) is reached. This ensures that all controllers that can possibly be derived exhibit the same behavior. Such a supervisor is called an implementable supervisor. The implementable supervisor can be used to derive a controller and, subsequently, the controller code.

Figure 4.4 shows the proposed method to obtain controller code from a supervisor model. First, it is verified whether supervisor S satisfies the three aforementioned properties. If it does, the supervisor is an implementable supervisor S'. If it does not satisfies all these properties, the supervisor has to be adapted (for example, by changing the requirements model and synthesizing a new supervisor, not depicted in the figure). If the supervisor is implementable, the supervisor can be used to derive a controller C. From this controller, controller code C' can be generated. In this controller code, the EFA in the controller model and the events in the EFAs are represented by blocks of code. To decrease the cycle time of the controller code on the PLC, the order of these EFAs and the events in the EFAs can be optimized. After optimizing, optimized controller code C'' is obtained. These steps are explained in the next sections.

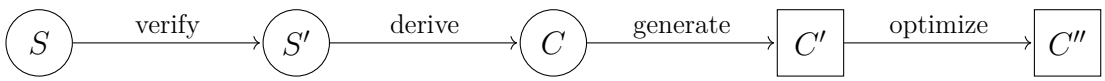


Figure 4.4: Method to obtain controller code from a supervisor model.

4.3 Implementable supervisor

When a supervisor is used to derive a controller, several issues have to be solved. In the following sections, these issues are addressed and solutions found in the literature are reported on. Most of these solutions require the supervisor to satisfy some property. However, the current synthesis tools (Feng and Wonham, 2006; Moor et al., 2008; van Beek et al., 2014; Malik et al., 2017) cannot verify these properties, as noted in Vieira et al. (2017). One of the reasons is that the properties are defined for a single automaton. Checking the properties for an EFA system requires computation of the synchronous product. For many realistic applications, computing the synchronous

product explicitly is infeasible. In this section, methods are given that can be used to determine if an EFA system satisfies the properties, without computing the synchronous product. This makes the verification feasible, even for large systems.

While in the literature the properties are defined on languages, here they are defined on EFAs. The reason is that for a PLC implementation, the actuators are controlled according to the state of an EFA. A different state can mean that a different actuator is enabled. The differences in the properties are mostly that when the language-based properties require the same continuations in the languages, the EFA-based properties require that the same state is reached.

4.3.1 Confluence

The supervisor might include states where multiple controllable events are allowed. When a controller is derived from a supervisor, one of these events has to be chosen, each time this state is reached. As shown in Figure 4.2, a 'wrong' choice can result in a blocking controller. This is referred to as the choice problem.

In Malik (2003), the confluence property is proposed. It states that whenever a choice between two controllable events exists, each event can be extended by a sequence of controllable events such that both paths end in the same state. This means that at the end of step 3 of the execution, described in Section 4.1, always the same state is reached. As the output signals depend on which state is reached, this means that the output is independent of the choices. Here, confluence is defined for EFAs.

Definition 5 (Confluence)**.** *Let E be an EFA with state set Q, transition function δ, and controllable events Σ_c. E is said to be confluent if for every reachable state $q \in Q$ and all $\sigma_1, \sigma_2 \in \Sigma_c$ such that $\delta(q, \sigma_1)!$ and $\delta(q, \sigma_2)!$ there exist $s_1, s_2 \in \Sigma_c^*$ such that $\delta(q, \sigma_1 s_1) = \delta(q, \sigma_2 s_2)$.*

As an example, consider the supervisor shown on the left-hand side of Figure 4.5. In state `3`, a choice can be made between controllable events `b` and `c`, resulting in different states from which a common state cannot be reached via controllable events. The supervisor shown on the right-hand side of Figure 4.5 is confluent. In state `3`, there is a choice between controllable events `b` and `c`. When `c` is chosen, state `1` is reached, and when `b` is chosen, state `1` is reachable via controllable event `d`.

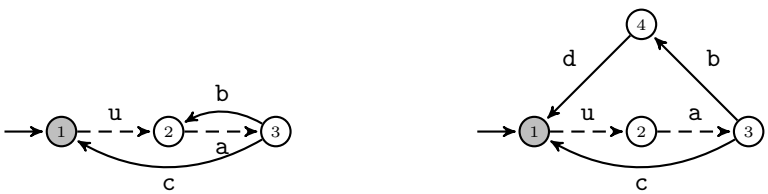


Figure 4.5: A supervisor that is not confluent (left) and a supervisor that is confluent (right).

In the literature, other methods exist to deal with the choice problem. In Leal et al. (2012), the choice problem has been solved by randomly selecting a controllable event each time there is a choice. In this way, all the behavior of the supervisor is kept in

the controller. A downside is that the reaction of the controller may be different every time. When the system is controlled by human operators this might be undesirable. In Morgenstern and Schneider (2007), the forcible nonblocking property is proposed. It requires that the supervisor always reaches a marked state, irrespective of the plant behavior. This also ensures that the derived controller will always reach a marked state, independent of the choices made. However, it might be difficult to enforce such a property. In many human-operated systems, the operator can postpone reaching a marked state indefinitely (e.g., by not giving the right command). Therefore, such a supervisor might not exist. In Huang and Kumar (2008), the notion of directed control is used: when presented with a choice, the event that brings the system closer (in terms of events) to a marked state is preferred. This property is very useful when the meaning of the marked state is that it is the 'end goal', and it is desired to stay in that state. However, for systems such as in Forschelen et al. (2012) and Reijnen et al. (2017), the interpretation of the marked state is that it is a safe resting state. There, it should also be possible to leave the marked state once it has been reached.

Verifying confluence

To check confluence for an EFA system, the global guard expression and the global update function derived from the NLEFA (see Section 2.4, at page 17) representation are used. The states where an event is enabled are derived from the global guard expressions. The global update functions denote how the state changes when an event occurs. The global guard expressions and the global update functions can be used to identify five cases where we can prove to have a confluent supervisor. These cases are shown below together with examples. In the examples, the locations are also variables, represented by the location pointer `LP`. For illustrative purposes, the locations are also explicitly visualized.

First of all, if two events are not enabled simultaneously, then there cannot be a choice, referred to as mutual exclusiveness.

Definition 6 (Mutual exclusiveness)**.** *Two different events $\sigma_1, \sigma_2 \in \Sigma$ with global guards g_1 and g_2, respectively, are said to be mutually exclusive with respect to a valuation v if $v \not\models g_1$ or $v \not\models g_2$.*

In Figure 4.6, σ_1 and σ_2 are mutually exclusive for all valuations.

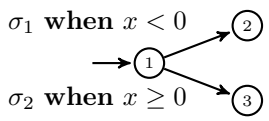


Figure 4.6: Example of mutually exclusive events.

Second, if two events are enabled simultaneously and the effect of the updates is the same, then it does not matter which event is chosen, as this implies that the same state is reached. This is referred to as update equivalence.

Definition 7 (Update equivalence)**.** *Two events $\sigma_1, \sigma_2 \in \Sigma$ with global updates u_1 and u_2, respectively, are said to be update equivalent with respect to a valuation v if $u_1(v) = u_2(v)$.*

In Figure 4.7, σ_1 and σ_2 are update equivalent for all valuations where $v(x) = 2$.

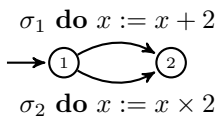


Figure 4.7: Example of update equivalent events.

Third, two events are called independent when they are enabled simultaneously and after the execution of either event, the other event is still enabled and both orders reach the same state.

Definition 8 (Independence)**.** *Two different events $\sigma_1, \sigma_2 \in \Sigma$, with global guards g_1 and g_2 and global updates u_1 and u_2, respectively, are said to be independent with respect to a valuation v if $u_1(v) \models g_2$, $u_2(v) \models g_1$, and $u_1(u_2(v)) = u_2(u_1(v))$.*

In Figure 4.8, σ_1 and σ_2 are independent events for all valuations.

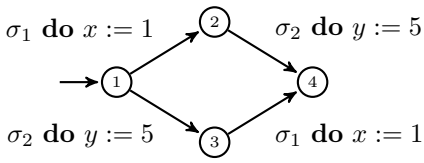


Figure 4.8: Example of independent events.

Fourth, if two events are enabled simultaneously and executing event 1 reaches the same state as executing event 2 followed by event 1, event 2 can be skipped.

Definition 9 (Skippable)**.** *For two different events $\sigma_1, \sigma_2 \in \Sigma$, with global guards g_1 and g_2 and global updates u_1 and u_2, respectively, σ_2 is said to be skippable by σ_1 with respect to a valuation v if $u_2(v) \models g_1$ and $u_1(u_2(v)) = u_1(v)$.*

In Figure 4.9, σ_2 is skippable by σ_1, for all valuations.

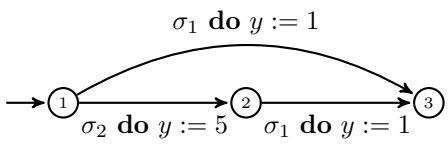


Figure 4.9: Example of skippable events.

Finally, if two events are enabled simultaneously, it can be the case that after the execution of both events, the update of the first event is reversed by a third event. In that case, it suffices to only execute the second event. The first event is reversible after the second event.

Definition 10 (Reversible)**.** *For two different events $\sigma_1, \sigma_2 \in \Sigma$, with global guards g_1 and g_2 and global updates u_1 and u_2, respectively, σ_2 is said to be reversible after σ_1 with respect to a valuation v if there exists a $\sigma_3 \in \Sigma$, with global guard g_3 and global update u_3, such that $u_2(v) \models g_1$, $u_1(u_2(v)) \models g_3$, and $u_3(u_1(u_2(v))) = u_1(v)$.*

In the EFA system of Figure 4.10, σ_2 is reversible after σ_1. Here, σ_3 is the reverse event of σ_2. In this example, the sequences $\sigma_2\sigma_1\sigma_3$ and σ_1 end in the same location $(\mathtt{1}, \mathtt{4})$, with the same value for x, for all valuations.

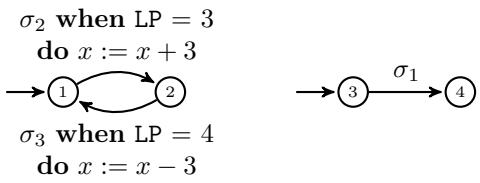


Figure 4.10: Example of reversible events.

Proposition 1. *Let $\mathcal{E}$ be a deterministic EFA system, with set $X_{\mathcal{E}}$ of variables that includes the location variables, and set V of all valuations, and let Σ_c be the set of controllable events. If each combination $(\sigma_1, \sigma_2, v) \in \Sigma_c \times \Sigma_c \times V$ satisfies one of the following conditions: mutually exclusive, update equivalent, independent, σ_2 is skippable by σ_1 or vice versa, or σ_2 is reversible after σ_1 with an event $\sigma_3 \in \Sigma_c$, or vice versa, then $\mathcal{E}_{\|}$ is confluent.*

Proof. Let Q be the state set of $\mathcal{E}_{\|}$. To prove that $\mathcal{E}_{\|}$ is confluent, we need to argue that for every reachable state $q \in Q$ whenever two controllable events $\sigma_1, \sigma_2 \in \Sigma_{\mathrm{c}}$ exist such that $\delta(q, \sigma_1)!$ and $\delta(q, \sigma_2)!$, then there exist sequences $s_1, s_2 \in \Sigma_{\mathrm{c}}^*$ such that $\delta(q, \sigma_1 s_1) = \delta(q, \sigma_2 s_2)$.

For (σ_1, σ_2, v) that is *mutually exclusive*, this is true, as either $\delta(q, \sigma_1)!$ or $\delta(q, \sigma_2)!$ is not true. For (σ_1, σ_2, v) that is *update equivalent* this is true for $s_1 = s_2 = \epsilon$. For (σ_1, σ_2, v) that is *independent* this is true for $s_1 = \sigma_2$ and $s_2 = \sigma_1$. For (σ_1, σ_2, v) that is *skippable* this is true for $s_1 = \epsilon$ and $s_2 = \sigma_1$. For (σ_1, σ_2, v) that is *reversible* this is true for $s_1 = \epsilon$ and $s_2 = \sigma_1\sigma_3$. Hence, if each combination $(\sigma_1, \sigma_2, v) \in \Sigma_{\mathrm{c}} \times \Sigma_{\mathrm{c}} \times V$ satisfies one of the following conditions: mutually exclusive, update equivalent, independent, σ_2 is skippable by σ_1 or vice versa, or σ_2 is reversible after σ_1 with an event $\sigma_3 \in \Sigma_{\mathrm{c}}$, or vice versa, then $\mathcal{E}_{\|}$ is confluent. □

To check if an EFA system is confluent, each pair of controllable events is checked for each valuation for the aforementioned properties, as defined in Algorithm 1. Of course, in the implementation, only those valuations are considered for which at least one variable that occurs in g_1, g_2, u_1, or u_2 has a different value (or g_3 or u_3 for the reversible property), as typically only a small subset of the variables is used. An event is also not checked with itself, because this combination is always update equivalent for a deterministic EFA. First, it is determined for which valuations two events are enabled simultaneously, i.e. for which they are not mutually exclusive (line 4). Second, if two events are enabled simultaneously, it is determined if they satisfy any of the aforementioned properties. When two events do not satisfy any of the properties,

the algorithm returns **F** (note that since these are sufficient conditions, this does not necessarily indicate that the EFA system is not confluent). Because skippable and reversible properties are not symmetrical, they are checked in both directions. The checks that are most often satisfied are checked first (based on experience from case studies).

Algorithm 1 Confluence check

Input: EFA system $\mathcal{E}$, variable set $X_{\mathcal{E}}$, valuation set V, and controllable event set Σ_{c}.
Output: True indicates that $\mathcal{E}_{\|}$ is confluent.

1: Transform $\mathcal{E}$ into its NLEFA representation, to obtain for each $\sigma \in \Sigma_{\mathrm{c}}$ its global guard g_σ and global update u_σ
2: **for all** $\sigma_1, \sigma_2 \in \Sigma_{\mathrm{c}} : \sigma_1 \neq \sigma_2$ **do**
3: **for all** $v \in V : v \models g_1$ and $v \models g_2$ **do**
4: **if** $\neg$ update equivalent(σ_1, σ_2, v) $\wedge$
5: $\neg$ independent(σ_1, σ_2, v) $\wedge$
6: $\neg$ (skippable$(\sigma_1, \sigma_2, v) \vee$ skippable$(\sigma_2, \sigma_1, v)) \wedge$
7: $\neg$ (reversible$(\sigma_1, \sigma_2, v) \vee$ reversible$(\sigma_2, \sigma_1, v))$
8: **then return F**
9: **end if**
10: **end for**
11: **end for**
12: **return T**

4.3.2 Finite response

As described in Section 4.1, a controller executes controllable events until no controllable event is enabled anymore (step 3). If an infinite sequence of controllable events exists in the controller, then it never stops executing events. In general, for a controller it is desired to generate a finite number of events and then reach a state where it waits for new inputs from the plant (e.g., an operator command, a sensor switch, or an elapsed timer). To accomplish this, the finite response property is proposed in Malik and Malik (2006). Here, finite response is defined for EFAs.

Definition 11 (Finite response)**.** *Let E be an EFA with state set Q, transition function δ, event set Σ, and controllable events $\Sigma_c \subseteq \Sigma$. E is said to have finite response if for every reachable state $q \in Q$ there exists an $n \in \mathbb{N}$ such that for every $s \in \Sigma^*$ and $\delta(q, s)!$ with $|s| > n$ it holds that $s \notin \Sigma_c^*$.*

In Malik and Malik (2006), it is shown that whenever no loops of controllable events exist in the supervisor, the controller always generates a finite response. A loop of controllable events is a sequence $s \in \Sigma_{\mathrm{c}}^+$ such that for a $q \in Q$, it holds that $\delta(q, s) = q$.

As an example, consider the supervisor shown on the left-hand side of Figure 4.11. This supervisor does not have finite response as there exists a loop of controllable events `ab` in state `2`. The supervisor shown on the right-hand side of Figure 4.11 does have finite response as there does not exist such a loop.

Figure 4.11: A supervisor that does not have finite response (left) and a supervisor that has finite response (right).

Verifying finite response

To check an EFA system for finite response, the following procedure is proposed. First, the possible controllable loops in the component models are overestimated by disregarding guards over variables that are not local. The overestimation of E is denoted by $\tilde{E}$. Then, these overestimated models are checked for controllable loops. The overestimation can be obtained by converting the guards on transitions to their disjunctive normal form and replacing the literals containing non-local variables with **T**. Second, if controllable loops are found, the global guard expressions of the events (derived from the NLEFA representation of the EFA system) are used to determine if these loops can exist in the EFA system. Third, it is checked if there is a controllable event σ in the event set of an EFA, but not in any of its controllable loops that can exist in the EFA system. If such an event is found, finite response is checked for $\Sigma_{\mathrm{c}} \setminus \{\sigma\}$, similar to the incremental method from Malik and Malik (2006).

In the first step, controllable loops are identified using Tarjan's algorithm (Tarjan, 1972), following the method from Malik and Malik (2006).

In the second step, a set of variables is defined that are not modified by controllable events in the EFA system. Typically, these are variables that belong to a sensor component. They are defined as follows.

Definition 12 (Σ_{c}-independent variables)**.** *Let $\mathcal{E}$ be an EFA system, with set $X_{\mathcal{E}}$ of variables, and set V of all valuations. Set $X_c \subseteq X_{\mathcal{E}}$ of variables that are never modified by controllable events, called Σ_c-independent variables, is defined as follows.*

$$X_c = \{x \in X_{\mathcal{E}} \mid \exists (L, X, \Sigma, \rightarrow, l^0, L^m) \in \mathcal{E} : x \in X \wedge \\ [\forall (l, g, \sigma, u, l') \in \rightarrow \wedge v \in V : \sigma \in \Sigma_c] \implies [v \models g \implies v(x) = u(v)(x)]\}$$

If for two events it can be proven that a variable in set X_{c} needs to have different values to satisfy their global guard expressions, then these events cannot be in the same controllable-event sequence. Such events are called Σ_{c}-unconnectable.

Definition 13 (Σ_{c}-unconnectable)**.** *Let $\mathcal{E}$ be an EFA system, let $\sigma_1, \sigma_2 \in \Sigma_c$ be controllable events and let X_c be the set of Σ_c-independent variables. Events σ_1 and σ_2 are said to be Σ_c-unconnectable if there do not exist $v_1, v_2 \in V$ such that $v_1 \models g_{\sigma_1}$, $v_2 \models g_{\sigma_2}$, and for every $x \in X_c$ it holds that $v_1(x) = v_2(x)$.*

If two controllable events are Σ_{c}-unconnectable then there does not exist a loop of controllable events that includes both events and is enabled in the EFA system. If a controllable loop consists of only events that are not Σ_{c}-unconnectable, then it can potentially be a loop in the EFA system.

Definition 14 (Potential controllable loop)**.** *Let $\mathcal{E}$ be an EFA system with controllable events Σ_c and with Σ_c-independent variables X_c. For an $E \in \mathcal{E}$ and an $s \in \Sigma_c^+$ such that s is a controllable loop in $\tilde{E}$, s is said to be a potential controllable loop in $\mathcal{E}_{\|}$ if there do not exist $\sigma_1, \sigma_2 \in s$ that are Σ_c-unconnectable.*

In this definition, $\sigma \in s$ denotes that event σ occurs at least once in sequence s.

In the third step, it is checked which events are in the event set of an EFA but not in any of its potential controllable loops. Then, finite response is checked for the remaining controllable events.

Proposition 2. *Let $\mathcal{E}$ be an EFA system with controllable events Σ_c and let X_c be the Σ_c-independent variables. If there exist an $E \in \mathcal{E}$ and a $\sigma \in \Sigma_c$ such that σ does not occur in any potential controllable loop of $\tilde{E}$, then $\mathcal{E}_{\|}$ has finite response for Σ_c if $\mathcal{E}_{\|}$ has finite response for $\Sigma_c \setminus \{\sigma\}$.*

Proof. Let Q be the state set of $\mathcal{E}_{\|}$. Assume that $\mathcal{E}_{\|}$ does not have finite response for Σ_c and that σ does not occur in any potential loop. Then there exist a $q \in Q$ and an $s \in \Sigma_\mathrm{c}^+$ such that $\delta(q, s) = q$, and consequently there cannot exist $\sigma_1, \sigma_2 \in s$ that are Σ_c-unconnectable in $\mathcal{E}_{\|}$. Moreover, there does also exist a q' in the state set of $\tilde{E}$ such that $\delta(q', s) = q'$. Hence, s must be a potential controllable loop of $\tilde{E}$, and thus $\sigma \notin s$. As a result, $\mathcal{E}_{\|}$ must also not have finite response for $\Sigma_\mathrm{c} \setminus \{\sigma\}$. □

To check if an EFA system has finite response, the results from Proposition 2 and the incremental approach from Malik and Malik (2006) are used in Algorithm 2. This algorithm consists of two parts:

1. [lines 5-9] For each EFA $\tilde{E}$, find all controllable loops. For each loop, determine if it is a potential loop, i.e., there are no Σ_c-unconnectable events in it. For each potential loop, add the events to the set Σ_loops that contains all events that occur in at least one potential loop.

2. [line 12] For each EFA, remove from Σ_c all controllable events in its alphabet that never occur in a potential loop.

In the algorithm, alph(..) returns the alphabet of an EFA or the events in a sequence, independent(Σ_c) returns set X_c as defined in Definition 12, cLoop($\tilde{E}$) returns the controllable loops in $\tilde{E}$, and unconnectable($\sigma_1, \sigma_2, X_\mathrm{c}$) refers to the property from Definition 13. Note that since these are sufficient conditions, the **F** result does not necessarily indicate that the EFA system does not have finite response. The **T** result indicates that the EFA system does have finite response.

4.3.3 Nonblocking under control

When a controller executes two controllable events in succession (in step 3 of the execution method discussed in Section 4.1), it is impossible that an uncontrollable event occurs in between. When reaching a marked state depends on the occurrence of this uncontrollable event, the marked state is unreachable in this execution method.

Algorithm 2 Finite-response check

Input: EFA system $\mathcal{E}$, variable set $X_{\mathcal{E}}$, and controllable event set Σ_{c}
Output: True indicates that $\mathcal{E}_{\|}$ has finite response

1: $\Sigma_c^{\mathrm{new}} := \Sigma_{\mathrm{c}}$
2: **repeat**
3: $\quad \Sigma_{\mathrm{c}}^{\mathrm{old}} := \Sigma_{\mathrm{c}}^{\mathrm{new}}$
4: $\quad X_c := \mathrm{independent}(\Sigma_{\mathrm{c}}^{\mathrm{new}})$
5: $\quad$ **for all** $E \in \mathcal{E}$ **do**
6: $\quad\quad \Sigma_{\mathrm{loops}} := \emptyset$
7: $\quad\quad$ **for all** $s \in \mathrm{cLoop}(\tilde{E})$ **do**
8: $\quad\quad\quad$ **if** $\forall \sigma_1, \sigma_2 \in \mathrm{alph}(s) : \neg\mathrm{unconnectable}(\sigma_1, \sigma_2, X_{\mathrm{c}})$ **then**
9: $\quad\quad\quad\quad \Sigma_{\mathrm{loops}} := \Sigma_{\mathrm{loops}} \cup \mathrm{alph}(s)$
10: $\quad\quad\quad$ **end if**
11: $\quad\quad$ **end for**
12: $\quad\quad \Sigma_{\mathrm{c}}^{\mathrm{new}} := \Sigma_{\mathrm{c}}^{\mathrm{new}} \setminus (\mathrm{alph}(E) \setminus \Sigma_{\mathrm{loops}})$
13: $\quad$ **end for**
14: **until** $\Sigma_{\mathrm{c}}^{\mathrm{new}} = \Sigma_{\mathrm{c}}^{\mathrm{old}}$
15: **return** $\Sigma_{\mathrm{c}}^{\mathrm{new}} = \emptyset$

As an example, consider the controller shown on the left-hand side of Figure 4.12. From state `1`, the controller executes controllable event `b`, and directly thereafter `c`. Hence, the behavior of the controller is actually as shown on the right-hand side of Figure 4.12. This controller is blocking, as it is unable to reach a marked state.

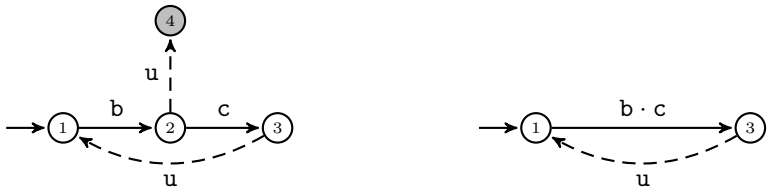


Figure 4.12: A nonblocking controller (left) and its blocking behavior when executed (right).

Another case is shown in Figure 4.13. There, the marked state is an in-between state that is left as soon as it is entered. Therefore, the system never actually is in the marked state.

Figure 4.13: A nonblocking controller (left) and its behavior for which the marked state is passed over.

In Malik (2003), the nonblocking under control property is proposed. It requires in every reachable state the existence of an event sequence to a marked state that prioritizes controllable events over uncontrollable events. Furthermore, in that marked state, no controllable event may be enabled. Such a sequence is called Σ_{c}-complete.

Definition 15 (Σ_c-complete)**.** *Let E be an EFA with state set Q, transition function δ, event set Σ, and controllable events $\Sigma_c \subseteq \Sigma$. A sequence $\sigma_1 \dots \sigma_n \in \Sigma^*$ from state q_0 to q_n:*

$$q_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} \dots \xrightarrow{\sigma_n} q_n$$

is said to be Σ_c-complete if it holds that

1. *for every $i = 1, \dots, n$ it holds that $\sigma_i \in \Sigma_c$ or $Elig(q_{i-1}) \cap \Sigma_c = \emptyset$.*
2. *$Elig(q_n) \cap \Sigma_c = \emptyset$.*

The nonblocking under control property is then defined as follows.

Definition 16 (Nonblocking under control)**.** *Let E be an EFA with state set Q, transition function δ, event set Σ, and controllable events $\Sigma_c \subseteq \Sigma$. E is said to be nonblocking under control if for every reachable state $q \in Q$ there exists a Σ_c-complete sequence $s \in \Sigma^*$ such that $\delta(q, s) \in Q^m$.*

Verifying nonblocking under control

To check if a system is nonblocking under control, the algorithm from Ouedraogo et al. (2011), see Section 2.5, can be used. For checking nonblocking under control, the algorithm is slightly changed.

The synthesis algorithm can be used for checking nonblockigness by changing step 2. Instead of considering only the transition labeled with uncontrollable events in step 2, all transitions are considered. Then, in step 3 the bad-state predicate of the initial location is evaluated for the initial valuation. If the bad-state predicate evaluates to **F**, the EFA is nonblocking, otherwise it is blocking.

Verifying nonblocking under control can be done with a similar procedure. Only the calculation of the good-state predicates is different, as for nonblocking under control a Σ_c-complete sequence to a marked state must exist instead of any sequence. The procedure to verify nonblocking under control is given below.

1. Initially, the good-state predicate for a location l consists of marked-location predicate $M_l = l \in L^{\mathrm{m}}$ that is **T** for marked locations and **F** otherwise, marked-valuation predicate $M_v = v \in V_{\mathrm{m}}$ that is **T** for marked valuations and **F** otherwise, and a conjunction over the negations of the guards for transitions (l, g, σ, u, l') labeled with controllable events. This last conjunction is added, as nonblocking under control considers only marked states in which no controllable events are enabled.
$$N_l := M_l \wedge M_v \wedge \bigwedge_{\substack{(l,g,\sigma,u,l') \\ \sigma \in \Sigma_{\mathrm{c}}}} (\neg g)$$
2. For each location, compute a new good-state predicate as follows. The new good-state predicate of a location is the disjunction of the old good-state predicate N_l, the good-state predicate for transitions labeled with controllable events $N_{l,\mathrm{c}}$, and the good-state predicate for transitions labeled with uncontrollable events $N_{l,\mathrm{u}}$.
$$N_l := N_l \vee N_{l,\mathrm{c}} \vee N_{l,\mathrm{u}}$$

The predicate for transitions (l, g, σ, u, l') labeled with controllable events is a disjunction of the form $g \wedge N_{l'}[u]$, where g is the guard of the transition and $N_{l'}[u]$ is the old good-state predicate of the target location l' where all occurrences of the variables are replaced by the expression they are assigned in the update u of the transition.

$$N_{l,\mathrm{c}} = \bigvee_{\substack{(l,g,\sigma,u,l') \\ \sigma \in \Sigma_\mathrm{c}}} (g \wedge N_{l'}[u])$$

The predicate for transitions (l, g, σ, u, l') labeled with uncontrollable events is similar to the predicate for transitions labeled with controllable events. It includes an additional conjunction of the form $\neg g_\mathrm{c}$, where g_c are the guards for transitions labeled with controllable events. In this way, transitions labeled with uncontrollable events are only taken into account when no transitions labeled with controllable events are enabled (i.e., it is Σ_c-complete).

$$N_{l,\mathrm{u}} = \bigwedge_{\substack{(l,g_c,\sigma,u,l') \\ \sigma \in \Sigma_\mathrm{c}}} (\neg g_c) \wedge \left(\bigvee_{\substack{(l,g,\sigma,u,l') \\ \sigma \in \Sigma_\mathrm{u}}} (g \wedge N_{l'}[u]) \right)$$

3. Repeat step 2 until the good-state predicates do not change anymore.

The differences between this procedure and the original procedure are: 1) in step 1, the negation of the guards for transitions labeled with controllable events is included and 2) in step 2, the predicates for transitions labeled with uncontrollable events are only take into account when no transitions labeled with controllable events are enabled. The good-state predicate indicates whether a Σ_c-complete sequence to a marked state exists.

Computing the bad-state predicate is the same as for the original verification algorithm, for completeness it is given below.

1. Define the initial bad-state predicate as the negation of the good-state predicate

$$B_l := \neg N_l$$

2. For each location of the EFA compute a new bad-state predicate as follows. The new bad-state predicate of a location l is the disjunction of the old bad-state predicate and, for each outgoing transition (l, g, σ, u, l') of the location, a disjunction of the form $g \wedge B_{l'}[u]$, where g is the guard of the transition and $B_{l'}[u]$ is the old bad-state predicate of location l' where all occurrences of the variables are replaced by the expression they are assigned in the update u of the transition.

$$B_l := B_l \vee \bigvee_{\substack{(l,g,\sigma,u,l') \\ \sigma \in \Sigma}} (g \wedge B_{l'}[u])$$

3. Repeat step 2 until the bad-state predicates do not change anymore.

The EFA is nonblocking under control if in the initial location and for the initial valuation, the bad-state predicate evaluates to **F**.

Since both the good-state and bad-state predicates are calculated iteratively, we proof that these calculations always terminate.

Proposition 3. *The provided procedure for verifying nonblocking under control terminates.*

Proof. To prove that the given procedure terminates we proof that calculation of the good-state predicates terminates and calculation of the bad-state predicates terminates. The good-state predicate is a mapping $N : V \to \{\mathbf{T}, \mathbf{F}\}$. Two predicates N_1 and N_2 are equal if $\forall v \in V : N_1(v) = N_2(v)$. Since the new good-state predicate N_2 is a disjunction of the old good-state predicate N_1, it holds that $N_1(v) \implies N_2(v)$. Hence, at the end of step 2 of the good-state predicate calculation: either $\forall v \in V : N_1(v) = N_2(v)$ (termination), or $\exists v' \in V : \neg N_1(v') \wedge N_2(v')$. Since V is finite, this can only happen a finite number of times before the first situation is reached (termination). The same proof can be used for the bad-state predicate calculation. Hence, the procedure terminates. □

4.4 Controller

The following sections describe three issues that can occur when any derived controller is implemented in a real-time platform, such as a PLC. It also mentions the solutions proposed in the literature.

4.4.1 Avalanche effect

An effect that is often regarded as problematic, is the avalanche effect, see, e.g., Fabian and Hellgren (1998) and Zaytoon and Riera (2017). It may occur if a specific event is used to trigger successive transitions labeled with the same uncontrollable event, thus producing an 'avalanche'. For example, as depicted in Figure 4.14, if uncontrollable event `a` is triggered in state `1`, the EFA should transit to state `2`. In an erroneous implementation, it might transit to state `3` directly thereafter, since event `a` is still enabled. However, it should only transit to state `3` when a second `a` event occurs.

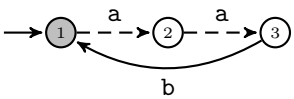


Figure 4.14: Avalanche effect illustration.

As already noticed in Leal et al. (2012) and Prenzel and Provost (2018), contrary to the other discussed problems, the avalanche effect is due to incorrect implementation of the controller: it is not a property of the controller itself. Each event should only trigger at most one transition per EFA. The controller code should make sure that this is handled correctly. In Section 4.5, a code generation method is shown that does not suffer from the avalanche effect. If this method is used, the EFA shown in Fig 4.14 can be used as a controller.

4.4.2 Inexact synchronization

During program execution, an uncontrollable event in the plant may occur. This event will only be recognized at the beginning of next scan cycle. Therefore, the communication between the PLC and the plant is subject to delays due to periodic reading of the inputs. Whenever such an event invalidates the choice made by the controller in the previous cycle, this may lead to problems.

In Balemi et al. (1993), the notion of delay insensitivity is defined. It requires that in every state where a controllable event is enabled, this event is also enabled in all states that can be reached by sequences consisting of uncontrollable events. Here, delay insensitivity is defined for EFAs.

Definition 17 (Delay insensitivity)**.** *Let E be an EFA with state set Q, transition function δ, controllable events Σ_c, and uncontrollable events Σ_u. E is said to be delay insensitive if for every reachable state $q \in Q$, $\sigma \in \Sigma_c$, and $s \in \Sigma_u^*$, such that $\delta(q, \sigma)!$ and $\delta(q, s)!$, holds $\delta(q, s\sigma) = \delta(q, \sigma s)$.*

As an example, consider the controller shown on the left-hand side of Figure 4.15. When uncontrollable event `a` occurs in state `1`, the controller responds by executing controllable event `c`. If during the execution of the controller code, uncontrollable event `b` occurs, the controller will observe it in the next cycle after event `c`, i.e., it will observe `acb`. In the plant, the order is `abc`, which is not allowed by the controller. The controller shown on the right-hand side of Figure 4.15 does not have this problem, as whenever a controllable event is allowed, it is also enabled in states that can be reached by sequences consisting of uncontrollable events.

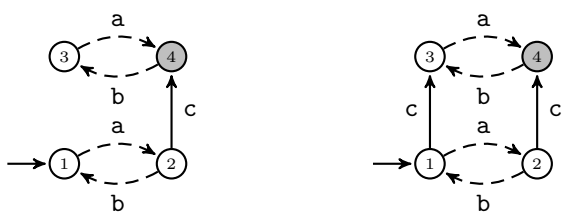


Figure 4.15: A controller that is delay sensitive (left) and a controller that is delay insensitive (right).

In the literature, other properties can be found, such as Σ_c-Σ_u-commuting in Malik (2002), delay nonconflictingness in Park and Cho (2006), and bounded-delay implementability in Xu and Kumar (2008). They are very similar to delay insensitivity, except that some define a delay bound (in terms of events) on the sequence of uncontrollable events. What they all have in common is that they require that the occurrence of uncontrollable events does not disable an enabled controllable event. As in general the enablement of controllable events depends on the state of sensors (and thus on uncontrollable events), it might be unrealistic to pursue a controller that satisfies such a property. In Leal et al. (2012), a different solution is suggested. There, the interrupt feature that some PLCs have, is utilized. In this way, every time an uncontrollable event occurs, the program is interrupted and the event is registered. However, they also note that many PLCs do not support this feature.

In Malik (2003), Forschelen et al. (2012), Reijnen et al. (2017), Prenzel and Provost (2018), and Swartjes (2018), it is argued that whenever the time scale of the system is much larger than that of the cycle time of the controller (seconds compared to milliseconds), inexact synchronization is not a problem. When the system operates faster than the controller, i.e., sensor changes happen faster than the cycle time of the controller, it is not suited for PLC implementation. In that case, either a faster processor is needed or the code must be optimized when the PLC program is too slow.

4.4.3 Simultaneity

Due to the periodic input reading, if between successive scan cycles two or more uncontrollable events occur, their order is lost. If the order of those events influences the allowed behavior, this can lead to problems.

In Fabian and Hellgren (1998), the interleave insensitivity property is proposed. Interleave insensitivity requires that after every interleaving of an uncontrollable event sequence, the same state has to be reached. Here, interleave insensitivity is defined for EFAs.

Definition 18 (Interleave insensitive)**.** *Let E be an EFA with state set Q, transition function δ, and uncontrollable events Σ_u. E is said to be interleave insensitive if for every reachable state $q \in Q$, $s_1, s_2 \in \Sigma_u^*$, such that $\delta(q, s_1 s_2)!$, there exists a $q' \in Q$, such that for every $s \in s_1 \mathbin{|||} s_2$ it holds that $\delta(q, s) = q'$, where $|||$ is the interleave operator.*

Interleave insensitivity also requires sequences that might not exist in the plant to be included in the controller, which might not be realistic. Therefore, we use a slightly different definition, called simultaneity insensitive.

Definition 19 (Simultaneity insensitive)**.** *Let E be an EFA with state set Q, transition function δ, and uncontrollable events Σ_u. E is said to be simultaneity insensitive if for every reachable state $q \in Q$ and sequences $s_1, s_2 \in \Sigma_u^*$, such that s_1 is a permutation of s_2, $\delta(q, s_1)!$, and $\delta(q, s_2)!$, holds $\delta(q, s_1) = \delta(q, s_2)$.*

Consider the controller shown on the left-hand side of Figure 4.16, if uncontrollable events `a` and `b` are observed simultaneously, it is unclear if the order was `ab` or `ba`. Hence, it is unclear whether the controller should transit to state `4` and execute controllable event `c` or transit to state `5` and do nothing. The controller shown on the right-hand side of Figure 4.16 does not have this problem, in both cases, controllable event `c` is executed and state `1` is reached.

In Leal et al. (2012), a solution similar to their solution for inexact synchronization is suggested. There, the interrupt feature that some PLCs have, is utilized. In this way, every time an uncontrollable event occurs, the program is interrupted and the event is registered, such that the order is not lost.

For simultaneity insensitivity, a similar remark can be made as for delay insensitivity. It might be unrealistic to pursue a controller that satisfies simultaneity insensitivity, as it requires that the order of uncontrollable events may not influence the behavior of the controller. As is the case for delay insensitivity, if the controller reacts sufficiently

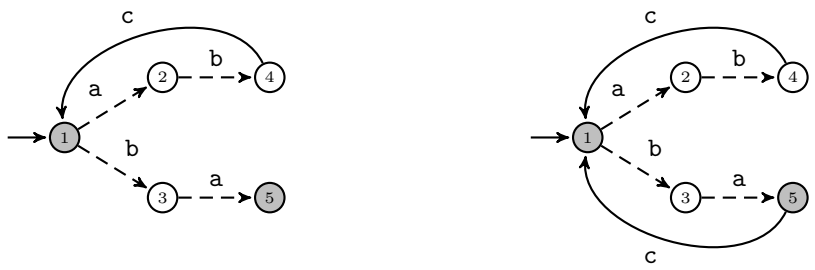


Figure 4.16: A controller that is simultaneity sensitive (left) and a controller that is simultaneity insensitive (right).

fast compared to the time scale of the system, simultaneity is not a problem. When the system operates faster than the controller, i.e., sensor changes happen faster than the cycle time of the controller, it is not suited for PLC implementation. In that case, either a faster processor is needed or the code must be optimized when the PLC program is too slow.

4.5 Controller code

For the implementation of a controller on a PLC, controller code has to be derived from the controller model. In Fabian and Hellgren (1998), controller code is generated by first computing the synchronous product of the models, such that one automaton is obtained that can be translated into code. For large models, this is infeasible as the state space is too large to compute. In Swartjes et al. (2014), a different method is proposed. There, the normalized representation is used, see Section 2.4. An advantage of normalization is that it does not require computing the whole state space. For each event in the system, a block of code is generated. A disadvantage is that the model structure is lost, and the code becomes difficult to interpret. Code readability is relevant for maintenance personnel in case the system fails because of, e.g., an unanticipated sensor failure.

To increase the readability of the code, the structure of the original model should be preserved. To this end, for each EFA a block of code and a set of variables is generated. Within the block of code, for each transition a block of code is generated. Subsequently, the event-condition requirements are placed in the block of code for the EFA that contains that event. In this way, each part of the code can be traced back to the original controller model.

The following sections describe how the concepts in the model are coupled to the hardware. Then it is shown how code for an EFA system can be generated.

4.5.1 Inputs and outputs

To couple the controller model to inputs and outputs of the PLC, the following procedure is used.

The inputs of the PLC are represented by set I of Boolean and integer variables. The uncontrollable events are coupled to these inputs by using the variables from I in the guards of these events. Typically, when a sensor model for a Boolean variable i contains events `u_on` and `u_off`, `u_on` indicates that the value of i changed from **F** to **T**, and vice verse for `u_off`. The guards '**when** i' and '**when** $\neg i$' are added to `u_on` and `u_off`, respectively, to make this coupling. Different event-variable combinations can exist (such as for integer variables, see Section 3.2.2), still such a mapping between events and variable values can be constructed.

The outputs of the PLC are represented by set Q of Boolean and integer variables. The actuator component models are coupled to these outputs by linking the variables from Q to the state of these models. Typically, when an actuator model for a Boolean variable q contains states `On` and `Off`. At the end of the program, a statement is added $q :=$ `LP` $=$ `On`. This means that the value of q is **T** whenever the model is in state `On` and **F** otherwise. Different state-variable combinations can exist (such as for integer variables), still such a mapping between states and variable values can be constructed.

4.5.2 Code generation

To generate code from a controller model consisting of multiple components and event-condition requirements, the following procedure is applied.

1. Transform the EFA system to an LEFA system (see Definition 1, on page 17).
2. If LEFAs share events, synchronize these LEFAs in one LEFA (see Definition 2, on page 17).
3. Create a variable `ready` to indicate whether the code is finished executing events.
4. Construct a function for each LEFA (see procedure below).
5. Construct a get input function that copies the value of the input-image variables to the controller variables.
6. Construct a put output function that copies the value of controller variables to the output-image variables.
7. Construct the initialization function.
8. Construct the program that consists of the get input function, a repeat statement that iterates over the LEFA functions, until `ready` equals **T**, and the put output function. The iteration loop stops when no more events are enabled that can be executed.

In step 2, different LEFAs are combined. Thanks to the modeling method presented in Chapter 3, the component models already have disjunctive event sets. In that case, step 2 only combines the LEFA with its requirements.

In step 8, the LEFAs are put in a repeat statement. A variable `ready` indicates whether the model finished executing events. At the start of the repeat statement, the value of `ready` is set to **T**. If an event is executed, this value is set to **F**. The code is repeated until `ready` is **T** at the end op the loop, This means that in that loop no

event was executed. The order of the LEFAs in the loop influences the number of repetitions. In Section 4.6.2, it is shown how this order can be optimized.

Step 4 is defined as follows for $(X, \Sigma, \rightarrow)$ an LEFA and v^0 the initial valuation.

4.1 For each $x \in X$ create a global variable, which is initialized as $x := v^0(x)$.

4.2 Create a variable `done` to indicate whether the LEFA is finished executing events.

4.3 For each transition $(g, \sigma, u) \in \rightarrow$, construct an if-then statement:
`if` g **`then`** u`; done := false;` **`end_if`**`;`.

4.4 Construct the LEFA function that consists of a repeat statement that iterates over the transition code blocks, until `done`.

4.5 Update the global `ready` variable, `ready := ready` **`and`** `done`.

In step 4.4, the transitions are put in a repeat statement. A variable `done` indicates whether the LEFA finished executing events. At the start of the repeat statement, the value of `done` is set to **T**. If an event is executed, this value is set to **F**. The code is repeated until `done` equals **T** at the end of the loop. The order of the transitions influences the number of repetitions. In Section 4.6.1, it is shown how this order can be optimized.

In the literature (Fabian and Hellgren, 1998; Vieira et al., 2017), the event on a transition is encoded as a Boolean variable to check if a transition is enabled. This can lead to the avalanche effect problem when one event can trigger successive transitions, see Section 4.4.1. In our method, we use the guards from the transitions. A transition is enabled only when its guard evaluates to **T**. Then, the avalanche effect does not exist anymore.

As an example for steps 4.1-4.5, consider EFA `E` shown on the left-hand side of Figure 4.17. Its LEFA representation, denoted by `L`, is shown on the right-hand side of this figure.

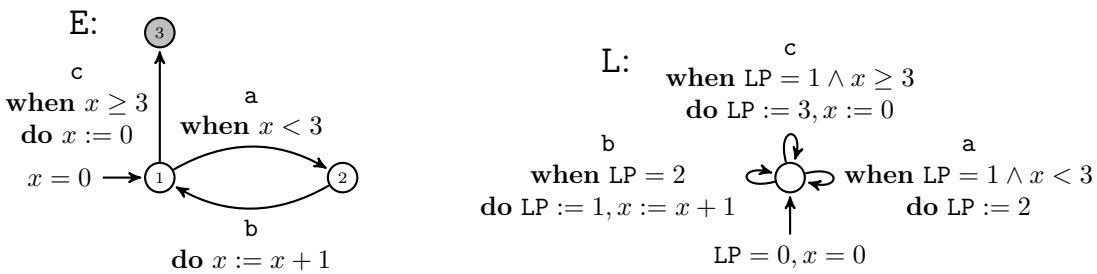


Figure 4.17: EFA `E` (left) and LEFA `L` (right).

The PLC code generated from `L` is shown in Listing 4.1. For each of the three transitions, a separate if-then statement is constructed. The variables with a dot (e.g., `L.LP`) are global variables, the other variables are local (e.g., `done`). The double slashes denote a comment.

Listing 4.1: Code for LEFA L.

```
repeat
  done := true;
  // Event a
  if L.LP = 1 and L.x < 3 then
    L.LP := 2;
    done := false;
  end_if;

  // Event b
  if L.LP = 2 then
    L.LP := 1;
    L.x := L.x + 1;
    done := false;
  end_if;

  // Event c
  if L.LP = 1 and L.x >= 3 then
    L.LP := 3;
    L.x := 0;
    done := false;
  end_if;

until done end_repeat;
```

As can be seen, the transformation from the model in Figure 4.17 to the code in Listing 4.1 is intuitive. In this example, the locations are numbered. In general, locations have names. In that case, named constants can be used to improve the readability.

As another example for code generation, consider the EFA system shown in Figure 4.18, consisting of two component models and two requirements. EFA S, shown on the left-hand side, represents a sensor and EFA A, shown on the right-hand side, represents an actuator. Variable i is a PLC input variable. Two requirements express that the actuator may only switch on when S is in location 2 and may only switch off when S is in location 1.

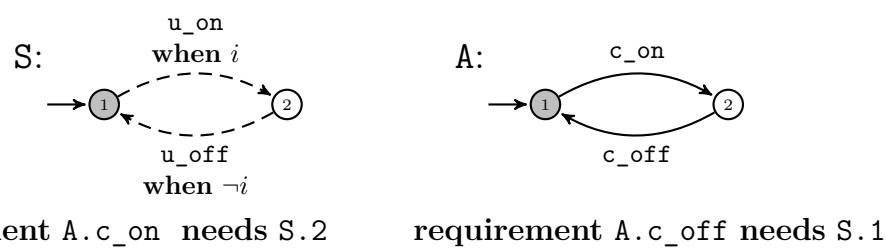


Figure 4.18: EFA system for which PLC code is generated.

The PLC code generated for this EFA system consists of four code parts, for EFA S (Listing 4.2), for EFA A (Listing 4.3), for the initialization (Listing 4.4), and for the program (Listing 4.5). Note that the requirements for c_on and c_off are moved to

the code for EFA `A` (step 2). In Listings 4.2 and 4.3, `ready` is the variable used in the repeat statement of Listing 4.5.

Listing 4.2: Code for EFA `S`.

```
repeat
  done := true;
  // Event u_on
  if S.LP = 1 and S.i then
    S.LP := 2;
    done := false;
  end_if;

  // Event u_off
  if S.LP = 2 and not S.i then
    S.LP := 1;
    done := false;
  end_if;

  ready := ready and done;
until done end_repeat;
```

Listing 4.3: Code for EFA `A`.

```
repeat
  done := true;
  // Event c_on
  if A.LP = 1 and S.LP = 2 then
    A.LP := 2;
    done := false;
  end_if;

  // Event c_off
  if A.LP = 2 and S.LP = 1 then
    A.LP := 1;
    done := false;
  end_if;

  ready := ready and done;
until done end_repeat;
```

The program consists of three parts. In the first part the PLC inputs are copied to the variables. In this case, the hardware address of i (in the program, denoted by `S.i`) is `%I0.0`. The second part is the iteration loop. Here, the code for `S` and the code for `A` is called (denoted by `S()` and `A()`) until `ready` equals **T**. The last part copies writes the outputs. In this case, the output at hardware address `%Q0.0` is switched on when EFA `A` is in location 2.

Listing 4.4: Code for the initialization.

```
S.LP := 1;
A.LP := 1;
```

Listing 4.5: Code for the program.

```
// Part 1, get inputs
S.i := %I0.0;

// Part 2, iteration loop
repeat
  ready := true;
  S();
  A();
until ready end_repeat;

// Part 3, put outputs
%Q0.0 := A.LP = 2;
```

4.6 Code optimization

In Section 4.4, it is shown that due to the real-time implementation, simultaneity and inexact synchronization might occur. To avoid these problems, the cycle time of the PLC and the variability of the cycle time of the PLC should be as short and as small as possible. In this section, we propose an optimization method to lower the cycle time.

The cycle time mostly depends on the time it takes to execute the generated code (see the previous section). The execution time of the code depends on the number of times the repeat statements in the LEFAs are executed and the number of times the repeat statement in the program is executed. Hence, to reduce the cycle time, the number of iterations should be reduced. The number of iterations depends on the order of the transitions in the LEFAs (the local orders) and the order of the LEFAs in the program (the global order), as shown in the following subsection.

If a supervisor satisfies the properties defined in Section 4.3, every derived controller is nonblocking and exhibits the same behavior. This allows to arbitrarily choose the local and global orders.

4.6.1 Local optimization

As an example of the ordering of events within an EFA, consider the controller model shown in Figure 4.19, and two event orderings $O_1 = [\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}]$ and $O_2 = [\mathtt{b}, \mathtt{d}, \mathtt{a}, \mathtt{c}]$. In case the controller is in state `1`, ordering O_1 updates the state of the supervisor to state `5` in two iterations. In the first iteration, controllable events `a`, `b`, `c`, and `d` are executed. In the second iteration, no event is executed. For ordering O_2, four iterations are required. In the first iteration, controllable event `a` is executed. In the second iteration, controllable events `b` and `c` are executed. In the third iteration, controllable event `d` is executed. In the fourth iteration, no event is executed. Since the number of iterations determines the execution time of the code, ordering O_1 is preferred.

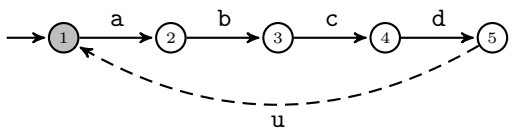


Figure 4.19: A controller model.

To analyze the event-execution order, so-called activity-based dependency structure matrices (DSMs) (Steward, 1981; Eppinger and Browning, 2012) are used. A DSM provides a concise representation of the relations between the events in the system in the form of a square matrix. The events are labeled along both axes of the matrix in the same order. Filled off-diagonal entries are used to indicate relationships between events. A filled element on row i, column j, indicates that event i depends on event j. Lower-diagonal elements ($i > j$) indicate feedforward information, whereas upper-diagonal elements ($i < j$) indicate feedback information. We define event dependency as follows.

Definition 20 (Event dependency)**.** *Event i depends on event j if there exist states $q, q' \in Q$ such that $j \in Elig(q)$, $i \notin Elig(q)$, $\delta(q, j) = q'$ and $i \in Elig(q')$. In other words, if in state q, event i is not enabled, but event j is, and after the execution of event j, event i becomes enabled, then event i depends on event j.*

The order can be optimized by applying sequencing techniques, such as described in Kehat and Shacham (1973) and Kusiak and Wang (1993). This is a form of DSM analysis that involves reordering the rows and columns of the DSM to minimize the number of feedback elements and to minimize the distance between the feedback elements and the main diagonal. Visually, this means that the elements are arranged such that as few filled entries as possible are above the main diagonal, and that those above the main diagonal are 'pushed' towards the main diagonal. Upper-diagonal marks are typically undesired as upon evaluating the enablement of event i, event i might not be enabled, whereas after executing event j it might be enabled, and the enablement of event i has to be re-evaluated.

As an example, consider the EFA shown in Figure 4.19. The DSM shown on the left-hand side of Figure 4.20, displays the event order O_2. Here, the filled square in row 1, column 3, indicates that event `b` depends on event `a`, i.e., after event `a`, event `b` may become enabled, as can be seen in the EFA. Uncontrollable events are not displayed, as these should always be checked first (Prenzel and Provost, 2018). The DSM shown on the right-hand side of Figure 4.19 is the result of sequencing the other DSM. The result is event ordering O_1. The DSM on the left-hand side has two feedback marks, whereas the DSM on the right-hand side has no feedback mark. For this example, the order is intuitive for the EFA. However, for more complex EFAs it is not always easy to derive the order by hand.

Figure 4.20: Unscheduled event-order DSM (left) and scheduled event-order DSM (right).

By sequencing the event-execution order, the number of iterations required to calculate the 'end' state of an LEFA is minimized.

4.6.2 Global optimization

Similar to the dependencies between events, dependencies between LEFAs in the code can be analyzed such that the LEFA-execution order can be minimized. We define an LEFA dependency as follows.

Definition 21 (LEFA dependency)**.** *LEFA L_1 depends on LEFA L_2 if there exists a transition in L_1 with a variable in a guard whose value is updated by an update in L_2.*

In Figure 4.21, the dependencies for the 37 LEFAs in the Oisterwijksebaan bridge model (Chapter 7) are shown, alphabetically. In total, there are 70 feedback marks.

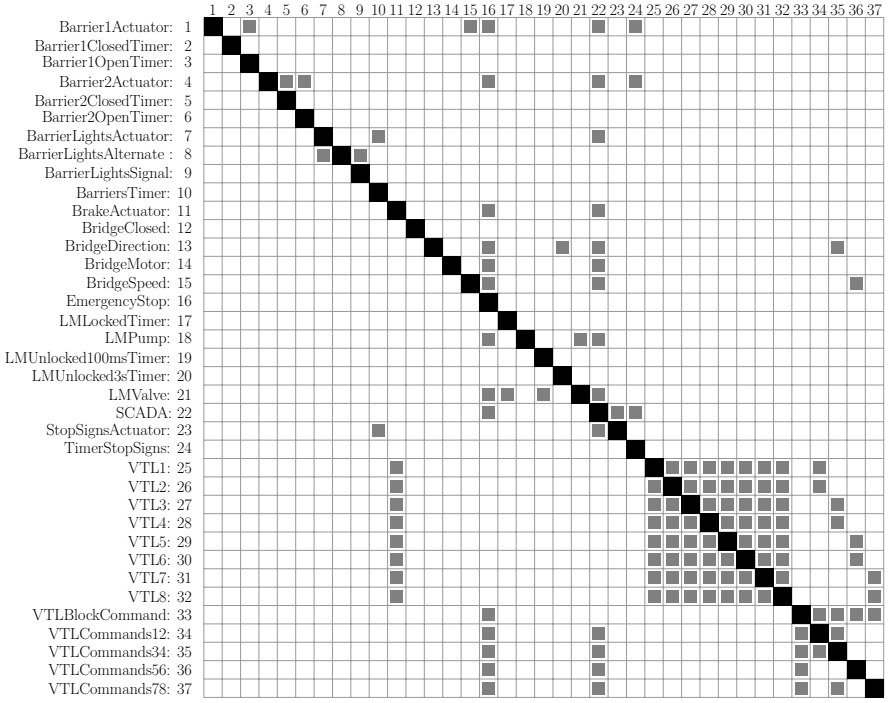


Figure 4.21: LEFA dependencies for the Oisterwijksebaan bridge, ordered alphabetically.

In Figure 4.22, the sequenced LEFA dependencies are shown. Here, there are only 34 feedback marks. Additionally, three feedback blocks are identified, depicted in red. When an event occurs in a feedback block, only the LEFAs in this block have to be re-evaluated. Hence, instead of having one global iteration loop, three smaller loops can be constructed. The LEFAs have to be iterated only for these smaller loops. The iteration loop for the LEFAs is shown in Listing 4.6. Here, `LEFA([1..3])` means that LEFA `1`, LEFA `2`, and LEFA `3` are called.

4.7 Concluding remarks

This chapter presents a method that allows to verify whether a supervisor, obtained via supervisor synthesis, is suitable to be used as a PLC controller. Compared to the supervised control setting (Charbonnier et al., 1995; Basile and Chiacchio, 2007), the additional step of designing a controller by hand is omitted. Instead, the controller is automatically derived from the supervisor. By using this method, the advantages of supervisor synthesis, such as correctness-by-construction and nonblockingness, automatically apply to the obtained controller. While other research also focuses on deriving controllers from supervisors, a problem that is often encountered there is that the derived controller can be blocking. Only after proving confluence, finite response, and nonblocking under control, it is guaranteed that any derived controller is nonblocking.

For the presented method, it has been shown which issues can be encountered. For these issues, it is shown how to verify that the obtained controller does not suffer from them. Furthermore, an event-execution ordering method and an LEFA-execution

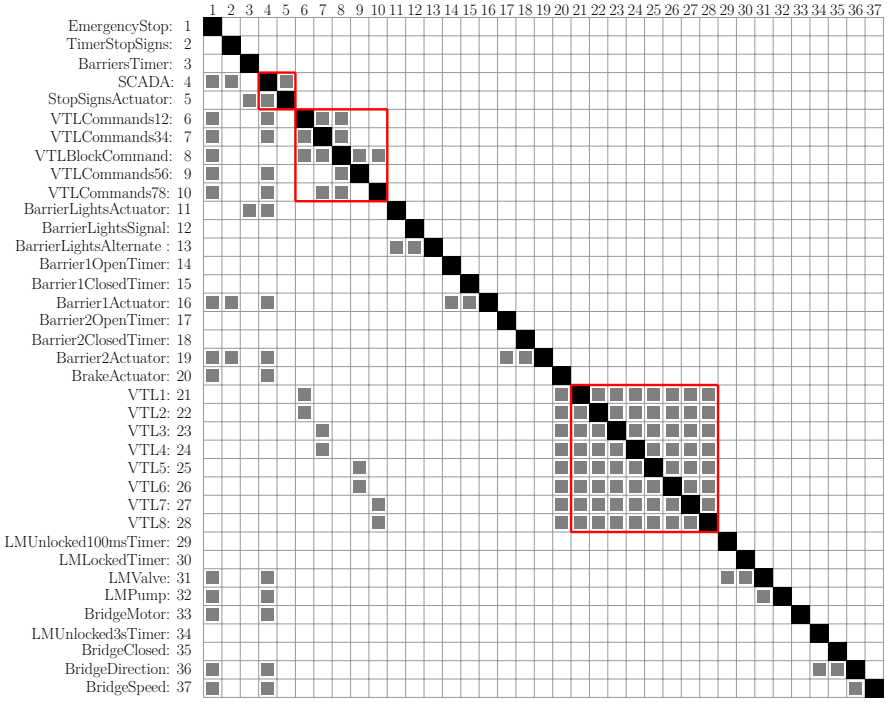


Figure 4.22: Sequenced LEFA dependencies for the Oisterwijksebaan bridge.

Listing 4.6: Pseudo code for the LEFA iteration.

```
LEFAs([1..3]);
repeat
  ready := true;
  LEFAs([4..5]);
until ready end_repeat;
repeat
  ready := true;
  LEFAs([6..10]);
until ready end_repeat;
LEFAs([11..20]);
repeat
  ready := true;
  LEFAs([21..28]);
until ready end_repeat;
LEFAs([29..37]);
```

ordering method are proposed that minimize the cycle time of the PLC and the variability in the cycle time. A low cycle time automatically reduces the possibility of inexact synchronization and simultaneity. Finally, a code generation method for PLC code is proposed that preserves the model structure as much as possible.

Chapter 5

Supervisor synthesis for safety PLCs

Industrial systems that operate in close proximity to humans are required to adhere to safety standards. For example, supervisory controllers for infrastructural systems such as waterway locks and movable bridges are required to adhere to the machinery directive (European Commission, 2006), by means of complying with the IEC 61508 standard regarding machine safety. Safety functions of the control systems used to be hard-wired, but are now generally implemented in (control) software. PLC manufacturers support this by offering so-called safety PLCs (SPLCs), for which the hardware adheres to the IEC standards.

SPLCs differ from regular PLCs (RPLCs), as they have additional dedicated inputs and outputs for safety certified sensors and actuators, and they have a separate safety program. In other words, the controller code and the inputs and outputs are split into a regular part and a safety part. The safety part contains the safety-critical control functions. The developed safety program has to be certified to comply with the required safety standards.

Synthesis tools such as CIF (van Beek et al., 2014) and Supremica (Malik et al., 2017) do not support distributed implementation of a single supervisory controller on the regular program and the safety program of the SPLC. A solution is to separately develop a regular supervisory controller and a safety supervisory controller. This can be difficult, as the behavior of the system depends on the joint control of both supervisory controllers. Due to this, it is difficult to analyze the behavior of the supervisory controllers separately. Another solution is to synthesize a single supervisor and then split it in the two parts, from where the supervisory controllers can be derived. The latter option is investigated in this chapter.

In the literature, SPLCs have been the focus of some studies, mostly in the context of formal verification. In Darvas et al. (2016), verification of safety functions has been studied. This method uses a requirements model to prove correctness of the controller code. In Ždánsky and Valigursky (2018), the diagnostic and performance functionality

This chapter is based on: Reijnen, F.F.H., Erens, T.R., van de Mortel-Fronczak, J.M., and Rooda, J.E. (2020). "Supervisory control synthesis for safety PLCs". In: *Proceedings of the 15th Workshop on Discrete Event Systems.* IFAC, In press.

of SPLCs has been studied. Splitting a supervisor in multiple supervisors has also been studied before, for example, in the context of distributed control in Cai and Wonham (2010). However, the goal there is to control multiple systems that have to work together, whereas for SPLCs, a single system is controlled with two supervisors. To the best of our knowledge, no work has previously been performed specifically on synthesizing, splitting, and implementing a supervisor on SPLCs.

The contribution of this chapter is a method to automatically split a supervisor into a supervisor for the regular part and a supervisor for the safety part, where both parts can be used to derive controller code for their respective programs on an SPLC. The applicability of this method is demonstrated by designing and implementing a supervisory controller for a real bridge.

This chapter is structured as follows. In Section 5.1, the way of working of an SPLCs is discussed. The modeling method used in this chapter is introduced in Section 5.2. The method to split and implement a synthesized supervisor on an SPLC is provided in Section 5.3. In Section 5.4, an illustrative example is provided. Validation of the method is discussed in Section 5.5. Finally, Section 5.6 concludes this chapter.

5.1 Safety PLCs

In this section, the way of working of an SPLC is discussed. In Figure 5.1, the component architecture of an SPLC is shown.

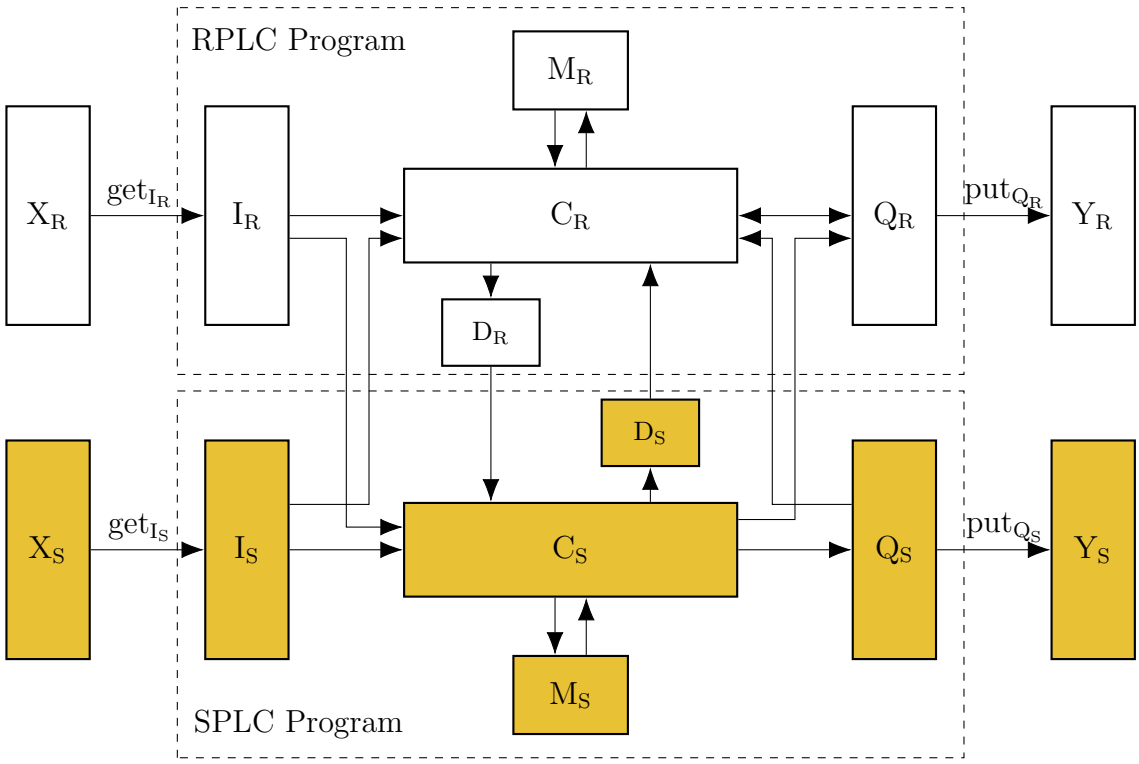


Figure 5.1: SPLC component architecture.

Compared to the architecture of an RPLC, an SPLC has an additional layer of safety components with interaction between the regular and safety component layers. Here, each component that is also present in an RPLC is subscripted with an R. In this figure, the components are as follows: X and Y are the input signals (sensors) and the output signals (actuators), respectively, I and Q are the images of the input and output, respectively, C contains the controller code, and M is the data memory of C. For each of these, there exist a regular variant and a safety variant. D_R and D_S are data buffers for the communication from the regular program to the safety program and vice versa, respectively. The arrows indicate the direction of data flows, e.g., C_S reads data from I_R, I_S, M_S, and D_R, and writes data to M_S, D_S, Q_R, and Q_S. It is important to note that C_R cannot write to Q_S, which ensures that all safety outputs are controlled by C_S.

An SPLC scan cycle consists of an execution of C_S followed by an execution of C_R. Each execution is preceded by its associated get_I and succeeded by its associated put_Q actions. The get_I action copies the values of the input signals to the input image and the put_Q action copies the values of the output image to the output signals. The images ensure that data available to the program is consistent during the program execution. Here, it is assumed that the safety part and the regular part are executed alternately, as shown in Figure 5.2. The assumption is based on a common mode of operation for SPLCs.

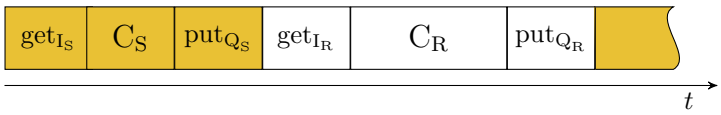


Figure 5.2: An SPLC scan cycle.

Within C_S, three parts can be distinguished: preprocessing, safety logic, and postprocessing. Preprocessing verifies the validity of the sensor signals, for example by one-out-of-two diagnostics. Postprocessing acts as a feedback loop that checks whether the actuator signals have been received by the actuator. When a discrepancy is detected in either the preprocessing or the postprocessing, the actuator signals are set to a predefined safe state. The safety logic is the controller code. In general, the controller code of C_S can be either a ladder diagram or a function block diagram. In this chapter, the focus is on the safety logic. The reason for this is that preprocessing and postprocessing are often embedded in SPLCs by the manufacturers, and otherwise standardized function blocks can be used, as defined in PLCopen (2018).

5.2 Modeling framework

This section provides the preliminaries on the modeling framework and implementation method used in this chapter. In this chapter, it is assumed that the plant is modeled using finite automata and Boolean input variables (BIVs), and that the requirements are modeled using event conditions.

A plant model consists of a set $\mathcal{P}$ of component models, $\mathcal{P} = \{P_1, \ldots, P_m\}$, and a set I of BIVs. A component is modeled as an FA. For all FAs P, the functions $loc(P)$

and $evt(P)$ are defined that return the set of locations and the set of events of P, respectively. The elements in I model the variables in the input image of the PLC.

For supervisor synthesis, BIVs are converted to their FA representation. The FA representation of a BIV is as follows: $P_{\text{BIV}} = (\{\mathbf{F}, \mathbf{T}\}, \{\texttt{on}, \texttt{off}\}, \{(\mathbf{F}, \texttt{u\_on}, \mathbf{T}), (\mathbf{T}, \texttt{u\_off}, \mathbf{F})\}, \mathbf{F}, \{\mathbf{F}, \mathbf{T}\})$. This FA is shown in Figure 5.3.

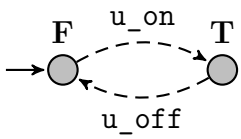


Figure 5.3: FA representation of P_{BIV}.

A requirements model consists of a set $\mathcal{R}$ of event conditions, $\mathcal{R} = \{R_1, \ldots, R_n\}$. An event condition is defined as a 2-tuple $R = (\sigma, c)$, where σ is an event defined in the plant model and c is a Boolean condition. Event σ is enabled only if c evaluates to $\mathbf{T}$. A condition is defined by the following grammar in Backus-Naur Form:

$$\langle c \rangle ::= \mathbf{F} \mid \mathbf{T} \mid v_l \mid i \mid \langle c \rangle \wedge \langle c \rangle | \langle c \rangle \vee \langle c \rangle | \neg \langle c \rangle \tag{5.1}$$

where v_l a is reference to a location in an FA which evaluates to $\mathbf{T}$ if and only if the current location of that FA is l, and $i \in I$ is a BIV. For all conditions c, the functions $loc(c)$ and $biv(c)$ are defined that return the set of all location references and the set of all BIVs in c, respectively.

Without loss of generality, in the remainder of this chapter it is assumed that $\mathcal{P}$, I and $\mathcal{R}$ together satisfy the safety, controllability, and nonblocking property. This can be justified, because whenever synthesis does generate additional guards, they can be included in $\mathcal{R}$. Furthermore, in Goorden and Fabian (2019), it has been shown that for realistic models this assumption often already holds, in other words, synthesis is not necessary.

5.2.1 Implementation

It should be noted that the supervisor is implemented as a controller. Subtle differences exist between a supervisor (that can only forbid events) and a controller (that can choose to execute some events), see, e.g., Fabian and Hellgren (1998), Zaytoon and Riera (2017), and Reijnen et al. (2019a). How to handle these differences is described in Chapter 4.

For connecting the variables in the input image of the PLC to the supervisor, the BIVs already defined in the plant model are used. The variables in the output image of the PLC are modeled by set Q of Boolean output variables. For connecting these variables to the supervisor, a hardware mapping is supplied. Here, a hardware mapping T_Q denotes the relation between an event and a value assignment to a variable in Q. Formally:

$$T_Q \subseteq \Sigma_{\text{c}} \times Q \times B \tag{5.2}$$

where B denotes the Boolean values, $B = \{\mathbf{T}, \mathbf{F}\}$. For example, $(\sigma, q, \mathbf{T}) \in T_Q$ defines that when σ occurs, the value $\mathbf{T}$ is assigned to q. Note that in the previous chapter,

the state is used to set the outputs. When using the modeling method proposed in Chapter 3, both can be used interchangeable.

5.3 Splitting method

In this section, the method is described that can be used to split a supervisor for the implementation on an SPLC. The general idea is as follows. First, a plant model and a requirements model are developed that represent a supervisor. Then, the requirements are (manually) partitioned into disjoint sets of regular requirements $\mathcal{R}'_{\mathrm{R}}$ and safety requirements $\mathcal{R}'_{\mathrm{S}}$. Based on this, the plant model, the requirements model, the input image, and the output image are split and the necessary data communication between the parts is derived, as shown in Figure 5.4. Here, D_{R} and D_{S} model the variables in $\mathrm{D_R}$ and $\mathrm{D_S}$, respectively.

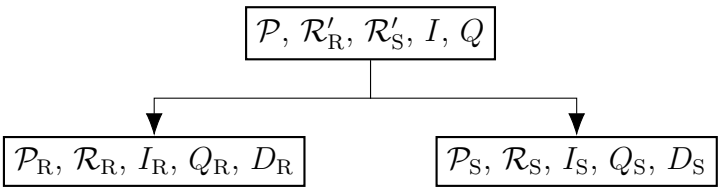


Figure 5.4: Desired result after splitting.

Notice that $\mathcal{R}'_{\mathrm{R}}$ and $\mathcal{R}'_{\mathrm{S}}$ may differ from $\mathcal{R}_{\mathrm{R}}$ and $\mathcal{R}_{\mathrm{S}}$, where the latter denote the requirements in the regular part and the requirements in the safety part, respectively

The objectives of the splitting are given in Sections 5.3.1. The proposed splitting procedure is given in Section 5.3.2. In Section 5.3.3, it is proven that splitting does not influence the behavior of the supervisor. Section 5.3.4 discusses code generation for the safety part.

5.3.1 Objectives of the splitting

For splitting, it is assumed that each condition for a requirement in $\mathcal{R}'_{\mathrm{S}}$ is derived entirely from (safety) BIVs. This is necessary, as in PLCopen (2018) it is defined that safety requirements have to be derived from sensors directly (instead of internal variables). Formally,

$$\forall(\sigma, c) \in \mathcal{R}'_{\mathrm{S}} : loc(c) = \emptyset \tag{5.3}$$

The result of the splitting has to adhere to the following six requirements. First of all:

(r1) Each safety requirement has to be included in the safety part, i.e., $\mathcal{R}'_{\mathrm{S}} \subseteq \mathcal{R}_{\mathrm{S}}$.

In PLCopen (2018), it is specified that if an output-image variable is associated with a safety requirement, that variable has to be included in the safety output image. This results in the following:

(r2) Each variable in the output image that is associated with an event subjected to a safety requirement has to be included in Q_{S}.

Due to the SPLC architecture, see Figure 5.1, events that result in the assignment of a value to a variable in the safety output image have to be included in the safety part. This gives rise to the third restriction.

(r3) Each event that results in that assignment of a value to a variable in Q_{S} has to be included in the safety part.

Finally, because the safety program should be as small as possible (e.g., for fast execution), the following should hold:

(r4) $\mathcal{P}_{\mathrm{S}}$ and $\mathcal{R}_{\mathrm{S}}$ should be as small as possible.
(r5) D_{R} and D_{S} should be as small as possible.
(r6) I_{S} and Q_{S} should be as small as possible.

These restrictions state that only the strictly necessary elements should be included in the safety part. When only looking at r4-r6, a solution with the empty set would seem an acceptable answer, but it is not due to r1-r3.

5.3.2 Splitting

It is assumed that the plant model is a product system, which means that the event sets of component models are pairwise disjoint. If the plant is not a product system, it can be obtained by performing the procedure described in de Queiroz and Cury (2000). In that procedure, if two component models share events, they are replaced by their synchronous product. When the modeling method described in Chapter 3 is used, only the components that share physical relations have to be combined.

From the assumption that requirements in $\mathcal{R}'_{\mathrm{S}}$ are derived from safety BIVs and (r6), definitions for the safety input image and the regular input images are derived:

$$I_{\mathrm{S}} = \{i \in I \mid \exists(\sigma, c) \in \mathcal{R}'_{\mathrm{S}},\ i \in \mathit{biv}(c)\} \tag{5.4}$$

$$I_{\mathrm{R}} = I \setminus I_{\mathrm{S}} \tag{5.5}$$

From (r2) and (r6), definitions for the safety output image and regular output image are derived:

$$Q_{\mathrm{S}} = \{q \in Q \mid \exists(\sigma, c) \in \mathcal{R}'_{\mathrm{S}},\ b \in B,\ (\sigma, q, b) \in T_Q\} \tag{5.6}$$

$$Q_{\mathrm{R}} = Q \setminus Q_{\mathrm{S}} \tag{5.7}$$

From (r2) and (r3), it follows that the component models that belong to $\mathcal{P}_{\mathrm{S}}$ are those that contain either an event subjected to a safety requirement or an event that results in the assignment of a value to a variable in the safety output image.

$$\begin{aligned}\mathcal{P}_{\mathrm{S}} = \{P \in \mathcal{P} \mid \exists(\sigma, c) \in \mathcal{R}'_{\mathrm{S}},\ \sigma \in \mathit{evt}(P)\} \cup \\ \{P \in \mathcal{P} \mid \exists(\sigma, q, b) \in T_Q,\ \sigma \in \mathit{evt}(P),\ q \in Q_{\mathrm{S}},\ b \in B\}\end{aligned} \tag{5.8}$$

$$\mathcal{P}_{\mathrm{R}} = \mathcal{P} \setminus \mathcal{P}_{\mathrm{S}} \tag{5.9}$$

Due to the product system assumption, two disjoint event sets for the safety and regular parts can be derived.

$$\Sigma_{\mathrm{S}} = \{\sigma \mid \exists P \in \mathcal{P}_{\mathrm{S}},\ \sigma \in evt(P)\} \tag{5.10}$$

$$\Sigma_{\mathrm{R}} = \{\sigma \mid \exists P \in \mathcal{P}_{\mathrm{R}},\ \sigma \in evt(P)\} \tag{5.11}$$

From (5.8) and (5.10), it follows that all events that are subjected to safety requirements belong to Σ_{S}. The regular requirements can be formulated for events in Σ_{S} and Σ_{R}. Hence, $\mathcal{R}'_{\mathrm{R}}$ can be split into regular requirements for Σ_{S} and regular requirements for Σ_{R}.

$$\mathcal{R}^{\mathrm{S}}_{\mathrm{R}} = \{(\sigma, c) \in \mathcal{R}'_{\mathrm{R}} \mid \sigma \in \Sigma_{\mathrm{S}}\} \tag{5.12}$$

$$\mathcal{R}^{\mathrm{R}}_{\mathrm{R}} = \{(\sigma, c) \in \mathcal{R}'_{\mathrm{R}} \mid \sigma \in \Sigma_{\mathrm{R}}\} \tag{5.13}$$

Because of (r4), requirements in $\mathcal{R}^{\mathrm{S}}_{\mathrm{R}}$ should be included in the regular program, whenever possible. This is only possible if the evaluation result of the requirement's condition does not change during the safety cycle (otherwise the splitting can influence the behavior of the supervisor). This can be violated when, e.g., the value of a variable in I_{S} is used to evaluate a condition and this value changes when the $\mathrm{get}_{I_{\mathrm{S}}}$ is performed. Therefore, the regular requirements which evaluations can change during the safety program cycle have to be included in the safety part. Formally, these requirements are

$$\begin{aligned}\mathcal{R}^{\mathrm{SS}}_{\mathrm{R}} = \{(\sigma, c) \in \mathcal{R}^{\mathrm{S}}_{\mathrm{R}} \mid & biv(c) \cap I_{\mathrm{S}} \neq \emptyset \vee \\ & loc(c) \cap \bigcup_{P \in \mathcal{P}_{\mathrm{S}}} loc(P) \neq \emptyset\}\end{aligned} \tag{5.14}$$

The other requirements in $\mathcal{R}^{\mathrm{S}}_{\mathrm{R}}$ can be included in the regular part.

$$\mathcal{R}^{\mathrm{SR}}_{\mathrm{R}} = \mathcal{R}^{\mathrm{S}}_{\mathrm{R}} \setminus \mathcal{R}^{\mathrm{SS}}_{\mathrm{R}} \tag{5.15}$$

The event conditions in $\mathcal{R}^{\mathrm{SR}}_{\mathrm{R}}$ can be evaluated in the regular part and then the evaluation result can be communicated to the safety part via the data buffer D_{R}. To this end, conditions for the same events are combined by putting them into conjunction, as in (5.16). This ensures that each event has exactly one condition evaluation that has to be communicated (r4, r5).

$$\forall \sigma \in \Sigma_{\mathrm{S}} : c_\sigma = \bigwedge_{(\sigma, c) \in \mathcal{R}^{\mathrm{SR}}_{\mathrm{R}}} c \tag{5.16}$$

Given a valuation v for the variables, we define $v_\sigma = v \models c_\sigma$. The reason to introduce v_σ is that sometimes c_σ has to be evaluated before it is used, e.g., it is evaluated in the regular part and used later in the safety part. The following requirements are added to replace the requirements in $\mathcal{R}^{\mathrm{SR}}_{\mathrm{R}}$. These requirements state that for event σ the evaluation result of c_σ has to be $\mathbf{T}$.

$$\mathcal{R}_D = \{(\sigma, v_\sigma) \mid \sigma \in \Sigma_{\mathrm{S}}\} \tag{5.17}$$

The requirements in the safety program and the regular program are

$$\mathcal{R}_{\mathrm{S}} = \mathcal{R}'_{\mathrm{S}} \cup \mathcal{R}^{\mathrm{SS}}_{\mathrm{R}} \cup \mathcal{R}_D \tag{5.18}$$

$$\mathcal{R}_{\mathrm{R}} = \mathcal{R}_{\mathrm{R}}^{\mathrm{R}} \tag{5.19}$$

It should be noted that $\mathcal{R}_{\mathrm{R}}^{\mathrm{SR}}$ is effectively replaced by c_σ, v_σ, and $\mathcal{R}_D$.

Finally, the programs cannot access variables from the other program part, as these are saved in $\mathrm{M_R}$ or $\mathrm{M_S}$ (see Figure 5.1). Therefore, some evaluation results and location references have to be communicated, which are given in (5.20) and (5.21) for D_{R} and D_{S}, respectively. From (5.3), it follows that conditions of requirements in $\mathcal{R}'_{\mathrm{S}}$ cannot contain location references. Similarly, because of (5.14), (5.15), and (5.16), conditions c_σ cannot contain location references to components in $\mathcal{P}_{\mathrm{S}}$.

$$D_{\mathrm{R}} = \{v_\sigma \mid \exists(\sigma, v_\sigma) \in \mathcal{R}_D\} \cup \{v_l \mid \exists(\sigma, c) \in \mathcal{R}_{\mathrm{R}}^{\mathrm{SS}}, l \in loc(c) \cap \bigcup_{P \in \mathcal{P}_{\mathrm{R}}} loc(P)\} \tag{5.20}$$

$$D_{\mathrm{S}} = \{v_l \mid \exists(\sigma, c) \in \mathcal{R}_{\mathrm{R}}^{\mathrm{R}}, l \in loc(c) \cap \bigcup_{P \in \mathcal{P}_{\mathrm{S}}} loc(P)\} \tag{5.21}$$

Now, all sets shown in Figure 5.4 are defined.

5.3.3 Proof of equal behavior

The result is that the supervisors before and after the splitting have equal behavior. To prove this, first notions of equal restrictiveness are defined.

Definition 22. *Two requirements $R_1 = (\sigma, c_1)$ and $R_2 = (\sigma, c_2)$ are said to be equally restrictive iff c_1 and c_2 are logical equivalent.*

Definition 23. *Two sets of requirements $\mathcal{R}_1$ and $\mathcal{R}_2$ are said to be equally restrictive iff for each event σ, $(\sigma, \bigwedge_{(\sigma,c)\in\mathcal{R}_1} c)$ is equally restrictive as $(\sigma, \bigwedge_{(\sigma,c)\in\mathcal{R}_2} c)$.*

Definition 24. *Let $\mathcal{P}$ and I model a plant and $\mathcal{R}_1$ and $\mathcal{R}_2$ be sets of requirements. The behavior of $(\mathcal{P}, I, \mathcal{R}_1)$ is equal to the behavior of $(\mathcal{P}, I, \mathcal{R}_2)$ if $\mathcal{R}_1$ and $\mathcal{R}_2$ are equally restrictive.*

Using these definitions, it can be proven that the behavior of the supervisor before the split is equal to the behavior of the supervisors after the split.

Proposition 4. *Let S_1 be $(\mathcal{P}, I, \mathcal{R})$ and S_2 be $(\mathcal{P}_R \cup \mathcal{P}_S, I_R \cup I_S, \mathcal{R}_R \cup \mathcal{R}_S)$, as defined in Section 5.3.2. Assuming that each c_σ is evaluated when its value is needed, such that v_σ in (5.17) can be substituted by c_σ, then, S_1 and S_2 have equal behavior.*

Proof. To prove this proposition, it is shown that the plant models of S_1 and S_2 are equal and that the requirements of S_1 and S_2 are equally restrictive. It follows from (5.8) and (5.9) that $\mathcal{P} = \mathcal{P}_{\mathrm{R}} \cup \mathcal{P}_{\mathrm{S}}$. It follows from (5.4) and (5.5) that $I = I_{\mathrm{R}} \cup I_{\mathrm{S}}$. Hence, the plant models are equal. As defined in Section 5.3, $\mathcal{R} = \mathcal{R}'_{\mathrm{R}} \cup \mathcal{R}'_{\mathrm{S}}$. From this and (5.12), (5.13), (5.14), and (5.15), it follows that the requirements of S_1 are $\mathcal{R}_1 = \mathcal{R}_{\mathrm{R}}^{\mathrm{R}} \cup \mathcal{R}_{\mathrm{R}}^{\mathrm{SS}} \cup \mathcal{R}_{\mathrm{R}}^{\mathrm{SR}} \cup \mathcal{R}'_{\mathrm{S}}$. From (5.18) and (5.19), it follows that the requirements for S_2 are $\mathcal{R}_2 = \mathcal{R}_{\mathrm{R}}^{\mathrm{R}} \cup \mathcal{R}'_{\mathrm{S}} \cup \mathcal{R}_{\mathrm{R}}^{\mathrm{SS}} \cup \mathcal{R}_{\mathrm{D}}$. From Definition 2, it follows that $\mathcal{R}_1$ is equally restrictive as $\mathcal{R}_2$ if $\mathcal{R}_D$ is equally restrictive as $\mathcal{R}_{\mathrm{R}}^{\mathrm{SR}}$. So now it remains to be proven that $\mathcal{R}_D$ is equally restrictive as $\mathcal{R}_{\mathrm{R}}^{\mathrm{SR}}$. (5.12) and (5.15) assure that all requirements in $\mathcal{R}_{\mathrm{R}}^{\mathrm{SR}}$ apply to events in Σ_{S}. In combination with (5.16) and (5.17), Definition 23,

and the assumption that v_σ can be substituted by c_σ, this shows that $\mathcal{R}_D$ is equally restrictive as $\mathcal{R}_{\mathrm{R}}^{\mathrm{SR}}$. Therefore, $\mathcal{R}_1$ is equally restrictive as $\mathcal{R}_2$. It is concluded that S_1 and S_2 have equal behavior. □

To investigate how the SPLC scan cycle (show in Figure 5.2) influences the behavior of the supervisor on the SPLC, the execution of the SPLC is modeled. This is similar to the method suggested in Roussel and Giua (2005), where the scan cycle of the RPLC has been modeled. The model of the SPLC is shown in Figure 5.5. The location `S` denotes the execution of the safety part, whereas `R` denotes the execution of the regular part. Events in Σ_{S} and Σ_{R} can only be executed in their respective part, as denoted by the self loops. The event `regular` denotes a switch to the regular part. When this event occurs, the values of the variables in D_{S} and I_{R} are updated (similar to the $\mathrm{get}_{I_{\mathrm{R}}}$ action in Figure 5.2). Similarly, the event `safe` denotes a switch to the safety part. When this event occurs, the values of the in D_{R} and I_{S} are updated.

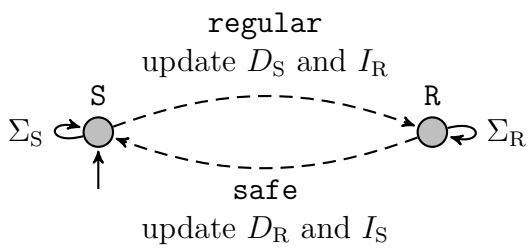


Figure 5.5: Model of the execution of an SPLC.

The variables in D_{S} and D_{R} are only updated when events `regular` and `safe` occur, respectively. Therefore, for variables in D_{S} and D_{R}, it has to be shown that their value cannot change in locations `R` and `S`, respectively.

It follows from (5.21) that D_{S} contains location references to components in $\mathcal{P}_{\mathrm{S}}$ only. Figure 5.5 shows that the values of these variables cannot change in location `R`, as Σ_{S} is not enabled here. Hence, the value of the variables in D_{S} cannot change in location `R`.

It follows from (5.20) that D_{R} contains location references and v_σ variables. D_{R} contains location references to components in $\mathcal{P}_{\mathrm{R}}$ only. Figure 5.5 shows that the values of these variables cannot change in location `S`, as Σ_{R} is not enabled here. For the v_σ variables in D_{R}, it follows from (5.14), (5.15), and (5.16) that their values are calculated from I_{R} and location references to components in $\mathcal{P}_{\mathrm{R}}$. These values remain unchanged in location `S`. As a result, the values of v_σ variables cannot change in location `S`. Hence, the value of the variables in D_{R} cannot change in location `S`.

Because of the results from Propositions 4 and the above reasoning, it is concluded that the splitting and data communication do not influence the behavior of the supervisor.

5.3.4 Safety implementation

Currently, tools such as CIF and Supremica are capable of generating code for a regular program, but not for a safety program. Below, a procedure to generate

function block diagrams (FBDs) is defined. This procedure is similar to the ladder diagram implementation described in Fabian and Hellgren (1998). Here, FBDs are generated as they are used more frequently nowadays. The following procedure is applied:

1. Introduce Boolean location variables for each location of each plant component, the initial value of the initial location variable is **T** and for all other locations **F**.

2. For each event, an FBD is created that encodes whether an event is enabled. An event is enabled when 1) the component model is in a location where the event is enabled, and 2) all event conditions for this event evaluate to **T**. Conditions as defined in (5.1) are translated into FBDs as follows: **F** and **T** are PLC literals, v_l is a location variable, i is a reference to a variable in the input image, $c \wedge c$ is an AND function block, $c \vee c$ is an OR function block, and $\neg c$ is an INVERSION function.

3. When an event is enabled, the transition and output update are encoded with SET and RESET function blocks. The origin location of the transition is reset (set to **F**) and the terminal location of the transition is set (set to **T**). Additionally, the variables in Q are reset and set according to T_Q.

5.4 Method illustration

To illustrate the method presented in the previous section, the splitting of a part of a supervisor for a bridge is shown[1]. Consider the following components: a boom barrier, a bridge deck, a vessel traffic light, an emergency stop, and three commands (open bridge, stop bridge, and close bridge). For controlling the position of the bridge deck, an actuator is present. The model of this actuator is shown in Figure 5.6.

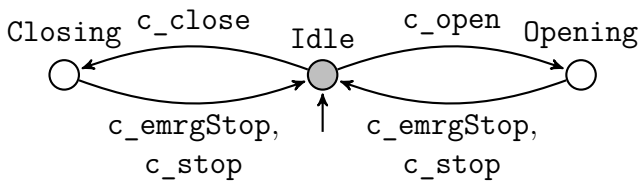


Figure 5.6: Component model of the bridge actuator P_{bridge}.

The following sensor signals are available: boom barrier is closed, vessel traffic light shows a red aspect, emergency stop, open command, close command, and stop command, denoted by the following BIVs $I = \{i_{\text{bbc}}, i_{\text{vtlr}}, i_{\text{es}}, i_{\text{oc}}, i_{\text{cc}}, i_{\text{sc}}\}$. Two actuator signals are controlled: opening the bridge and closing the bridge, so $Q = \{q_{\text{open}}, q_{\text{close}}\}$. The hardware mapping is as follows: $T_Q = \{($`c_open`, q_{open}, **T**$)$, $($`c_close`, q_{close}, **T**$)$, $($`c_emrgStop`, q_{open}, **F**$)$, $($`c_emrgStop`, q_{close}, **F**$)$, $($`c_stop`, q_{open}, **F**$)$, $($`c_stop`, q_{close}, **F**$)\}$.

[1]A more elaborate example is provided at www.github.com/ffhreijnen/WODES2020

The desired behavior is modeled using 12 requirements, provided below. Requirements in $\mathcal{R}'_{\mathrm{S}}$ are indicated by (S), other requirements belong to $\mathcal{R}'_{\mathrm{R}}$. It can be shown that the plant in combination with these requirements is safe, controllable, nonblocking, and maximally permissive.

r1: `c_open` **needs** i_{bbc} (S)
r2: `c_open` **needs** $\neg i_{\mathrm{es}}$ (S)
r3: `c_open` **needs** i_{oc}
r4: `c_open` **needs** $\neg i_{\mathrm{cc}}$
r5: `c_open` **needs** $\neg i_{\mathrm{sc}}$
r6: `c_stop` **needs** i_{sc}
r7: `c_close` **needs** i_{vtlr} (S)
r8: `c_close` **needs** $\neg i_{\mathrm{es}}$ (S)
r9: `c_close` **needs** i_{cc}
r10: `c_close` **needs** $\neg i_{\mathrm{oc}}$
r11: `c_close` **needs** $\neg i_{\mathrm{sc}}$
r12: `c_emrgStop` **needs** i_{es} (S)

To split the supervisor, the method described in the previous section is used. First, it is verified that (5.3) is satisfied. Then, the input image is partitioned as defined in (5.4) and (5.5): $I_{\mathrm{S}} = \{i_{\mathrm{bbc}}, i_{\mathrm{vtlr}}, i_{\mathrm{es}}\}$ and $I_{\mathrm{R}} = \{i_{\mathrm{oc}}, i_{\mathrm{cc}}, i_{\mathrm{sc}}\}$. Next, the output image is partitioned as defined in (5.6) and (5.7): $Q_{\mathrm{S}} = \{q_{\mathrm{open}}, q_{\mathrm{close}}\}$ and $Q_{\mathrm{R}} = \emptyset$. From (5.8) and (5.9), it can be derived that $\mathcal{P}_{\mathrm{S}} = \{P_{\mathrm{bridge}}\}$ and $\mathcal{P}_{\mathrm{R}} = \emptyset$. As a result, $\Sigma_{\mathrm{S}} = \{\texttt{c\_open}, \texttt{c\_close}, \texttt{c\_stop}, \texttt{c\_emrgStop}\}$ and $\Sigma_{\mathrm{R}} = \emptyset$.

The following requirement partitioning can be obtained. Since $\Sigma_{\mathrm{R}} = \emptyset$, it follows from (5.13) that no requirements are part of $\mathcal{R}^{\mathrm{R}}_{\mathrm{R}}$, and from (5.12) it follows that $\mathcal{R}^{\mathrm{S}}_{\mathrm{R}} = \{\mathrm{r3}, \mathrm{r4}, \mathrm{r5}, \mathrm{r6}, \mathrm{r9}, \mathrm{r10}, \mathrm{r11}\}$. From (5.14) and (5.15), it follows that $\mathcal{R}^{\mathrm{SR}}_{\mathrm{R}} = \mathcal{R}^{\mathrm{S}}_{\mathrm{R}}$ and $\mathcal{R}^{\mathrm{SS}}_{\mathrm{R}} = \emptyset$.

The requirements in $\mathcal{R}^{\mathrm{SR}}_{\mathrm{R}}$ are merged as in (5.16), such that the following conditions are obtained:

- $c_{\texttt{c\_open}} = i_{\mathrm{oc}} \wedge \neg i_{\mathrm{cc}} \wedge \neg i_{\mathrm{sc}}$
- $c_{\texttt{c\_close}} = i_{\mathrm{cc}} \wedge \neg i_{\mathrm{oc}} \wedge \neg i_{\mathrm{sc}}$
- $c_{\texttt{c\_stop}} = i_{\mathrm{sc}}$

The additional $\mathcal{R}_D$ requirements, as defined in (5.17) are

r13: `c_open` **needs** $v_{\texttt{c\_open}}$
r14: `c_close` **needs** $v_{\texttt{c\_close}}$
r15: `c_stop` **needs** $v_{\texttt{c\_stop}}$

As a result, the requirements in the safety part and in the regular part are: $\mathcal{R}_{\mathrm{S}} = \{\mathrm{r1}, \mathrm{r2}, \mathrm{r7}, \mathrm{r8}, \mathrm{r12}, \mathrm{r13}, \mathrm{r14}, \mathrm{r15}\}$ and $\mathcal{R}_{\mathrm{R}} = \emptyset$.

In this example, there is no location reference communication necessary, as defined in (5.20) and (5.21). The variables $v_{\texttt{c\_open}}$, $v_{\texttt{c\_close}}$, and $v_{\texttt{c\_stop}}$ are communicated via D_{R}, as defined in (5.20).

To illustrate the proposed safety implementation, the FBD for the `open` event is shown in Figure 5.7. Here, the outputs on the right-hand side are enabled whenever the inputs on the left-hand side are all true. The circle indicates the INVERSION function. $v_{\texttt{Idle}}$ and $v_{\texttt{Opening}}$ are location variables.

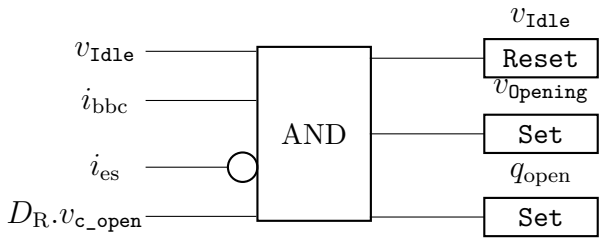


Figure 5.7: Function block diagram for the `open` event.

5.5 Validation of the method

The proposed method of splitting the supervisor is validated in three ways. To this end, the plant and requirements models for the Oisterwijksebaan bridge (shown in Figure 5.8), a swing bridge located in the Wilhelmina Canal in Tilburg, are used. The plant model of this bridge consists of 39 components and 51 BIVs. The requirements model consists of 118 regular requirements and 48 safety requirements. There are 39 outputs to be controlled. This supervisor has been split with the method proposed in Section 5.3. The currently implemented controller code of the bridge is available such that it can be compared to the controller code obtained from the model.

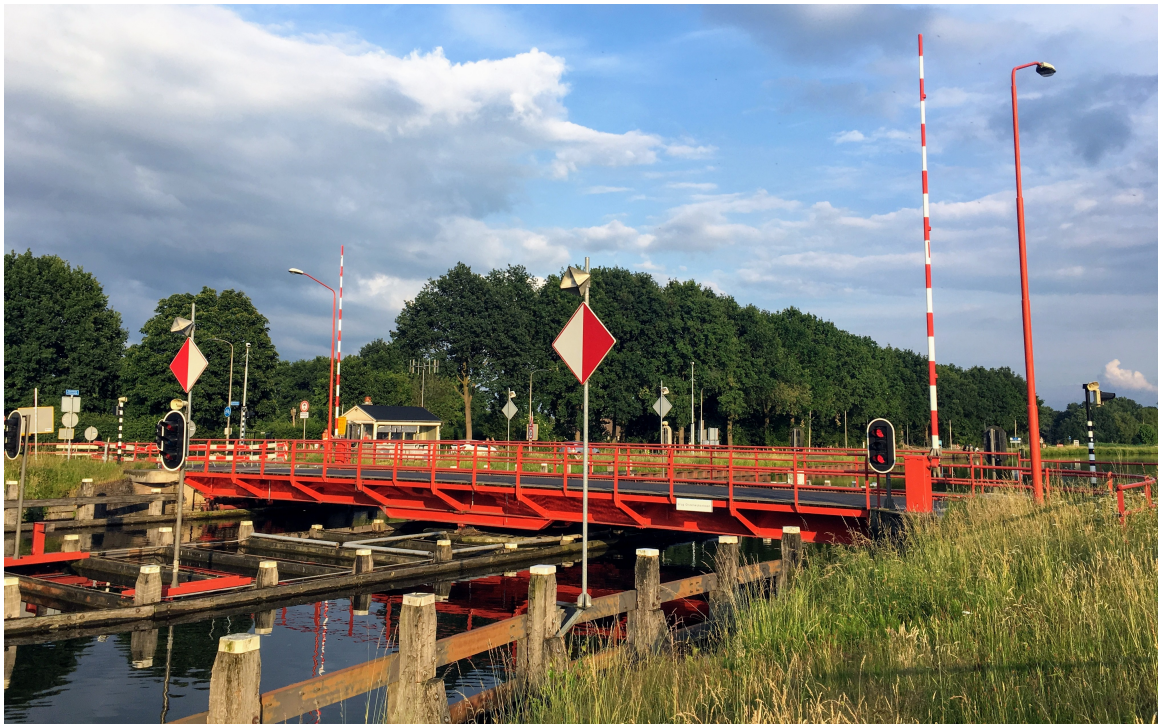


Figure 5.8: The Oisterwijksebaan bridge.

First, the split supervisor is used to generate controller code (for a SIMATIC S7-300 CPU315F-2 PN/DP SPLC from Siemens) that has been implemented on the real bridge. The size of the generated code is around 100 kB. The available size is 2 MB. To validate the behavior, existing site acceptance tests (SATs) have been

performed. A SAT protocol describes how the supervisory controller should react under a specific circumstance. The performed tests are categorized into two groups: 1) does the supervisory controller restrict unsafe behavior and 2) does the supervisory controller perform required behavior. During the tests no anomalies were observed, and it was concluded that the behavior of the supervisory controller was in accordance with the desired behavior.

Second, the input and output image partitioning is compared to the manually defined input and output image partitioning. The manually defined partitioning is the one currently used to control the bridge. The results for the splitting of I and Q are as shown in Table 5.1. The off-diagonal elements in this table denote differences between the automatic and manual splitting. As can be seen in the table, almost all variables are partitioned the same as was chosen manually (the main-diagonal elements). For the outputs, the partitioning was identical. For the inputs, only four variables differ. Here, four safety inputs in the manual split were chosen to be regular according to the presented method. Upon closer inspection of the code, none of these inputs were used in the safety part. Therefore, they could have been implemented as regular inputs.

Table 5.1: I and Q partitioning comparison.

		Automatic			
		I_S	I_R	Q_S	Q_R
Manual	I_S	22	4		
	I_R	0	25		
	Q_S			24	0
	Q_R			0	15

Third, the generated FBDs are compared with the manually programmed FBDs. Both FBDs adhere to a similar structure and implement the same safety constraints. A difference noticeable in the generated FBDs compared to the manually programmed FDBs are the extra variables required for keeping track of the automata locations, e.g., the variables $v_{\texttt{Idle}}$ and $v_{\texttt{Opening}}$ in Figure 5.7. It should be investigated whether the output-image variables can be used to keep track of the current location. For example, for the supervisory controller in the previous section, the value of $v_{\texttt{Opening}}$ is always equal to the value of q_{open}.

5.6 Concluding remarks

This chapter presents a method that can be used to split a given supervisor into a supervisor for the regular part and a supervisor for the safety part. This splitting method is based on the partitioning of the requirements models into regular requirements and safety requirements. The splitting as proposed allows for a supervisor to be implemented on an SPLC. Furthermore, it is shown that the behavior of the supervisor before the split is equal to the behavior of the supervisors after the split. This ensures that the properties guaranteed by supervisor synthesis still hold. Comparing a supervisor obtained via this method to a manually designed supervisor shows that a similar

partitioning can be obtained. Finally, implementing the supervisor (as a supervisory controller) on a real bridge shows a proof of concept for practical applicability of this method.

This method assumes that the regular program and safety program are alternately executed. In some cases, the safety program interrupts the regular program at certain time intervals. More research is required to determine whether this method is still applicable in this case.

Chapter 6

Hardware-in-the-loop set-up for supervisory controllers

Although synthesis guarantees that the supervisory controller adheres to the requirements, it is not always clear if the requirements are complete and correct. Therefore, the behavior of the supervisory controller has to be validated. Model simulation is often used for this purpose. It is a quick and intuitive method for validation of the supervisory controller's behavior. For simulation, the model of the supervisory controller is combined with a model of the plant. Model simulation allows an engineer to analyze the controlled behavior of a system under different scenarios. When undesired behavior is observed, the plant or the control requirements can be adapted, a new supervisor can be synthesized, and a new supervisory controller can be derived. While simulation is a valuable tool for early validation, it provides only a partial analysis. One of the main shortcomings is that a model of the supervisory controller is simulated, and not the actual implementation code. Another shortcoming is that the simulation platform and the implementation platform, often a programmable logic controller, have different operating semantics. Also, performance metrics such as cycle times cannot be derived from the simulation. Finally, communication to other subsystems such as an interface for operators cannot be tested. To overcome all these shortcomings, hardware-in-the-loop (HIL) simulation can be performed after model simulation and before implementation in the real system.

HIL simulation is a method in which the supervisory controller is implemented on the hardware, but instead of the hardware being connected to the real plant, it is connected to a model of the plant. HIL simulation offers the same advantages as model simulation, such as early validation and the possibility to simulate scenarios that would be unsafe to test on the real system. In addition, as the implementation code is run on the hardware, the performance metrics can be measured and the interaction with other subsystems can be evaluated. The increase in test coverage contributes towards improving the supervisory controller in the early design phase. Because of this, HIL simulation has been proven to be useful for a range of applications, such as

Reijnen, F.F.H., Verbakel, J.J., van de Mortel-Fronczak, J.M., and Rooda, J.E. (2019). "Hardware-in-the-loop set-up for supervisory controllers with an application: The Prinses Marijke complex". In: *Proceedings of the 3rd Conference on Control Technology and Applications*. IEEE, pp. 843-850.

wind turbines (Li et al., 2010), mineral grinding (Dai et al., 2016), and traffic control (Bullock et al., 2004).

HIL simulation in combination with synthesis has been explored before in Theunissen et al. (2009) and Diogo et al. (2012). In Theunissen et al. (2009), a supervisory controller for a patient support system for an MRI has been developed. The supervisory controller was first validated using model simulation. Subsequently, the model of the supervisory controller was connected to the real hardware. Different from the HIL simulation described above, a model of the supervisory controller was connected to the plant realization, instead of connecting the realization of the supervisory controller to a model of the plant. This is a different type of HIL simulation. While this method can be beneficial for rapid prototyping, it requires the plant to be available. In Diogo et al. (2012), an implementation environment for automated manufacturing systems, which uses HIL concepts, is presented. Here, parts of the plant are simulated, while the supervisory controller is implemented on the real PLC. In that study, a separate simulation environment has been coded in a SCADA system to represent all the components. The simulated components can gradually be replaced by the real components. In Diogo et al. (2012), manual coding of both the supervisory controller and the simulation environment has been applied, which is time consuming and error prone.

Supervisor synthesis, model simulation, and HIL simulation all require a model of the plant. Specifically for synthesis, a discrete-event model without time is used. For model simulation and HIL simulation the model is enriched with continuous-time behavior. Remodeling the plant for each step is very time consuming and error prone. Remodeling also is unnecessary as these models are very similar. Moreover, when the design of the plant changes, the changes have to be implemented in all models separately. Hence, to streamline the development process, reuse of the already available models is desired.

The main contribution of this chapter is twofold. Firstly, it presents a method that combines synthesis-based engineering and HIL simulation. For this method, the models already available for supervisor synthesis are refined and re-used to automatically generate the plant model for HIL simulation. Compared to Diogo et al. (2012), this approach decreases the development time and reduces coding errors. The second contribution of this chapter is the description of an application of the proposed method to an industrial case, the Prinses Marijke complex. For this case, the necessary models have been developed and implemented in an experimental HIL set-up. The models and additional information are available in the repository[1].

This chapter is structured as follows. The Prinses Marijke complex used as the case study is introduced in Section 6.1. The set-up used for HIL simulation is presented in Section 6.2. In Section 6.3, the proposed method is used to develop a supervisory controller and a HIL simulation for the Prinses Marijke complex. Finally, Section 6.4 concludes this chapter.

[1] www.github.com/ffhreijnen/PrinsesMarijkeComplex

6.1 The Prinses Marijke complex

As a case study, the Prinses Marijke complex is considered. The complex, depicted in Figure 6.1, consists of two waterway locks (left-hand side) and a floodgate (right-hand side). Under normal conditions, the floodgate is open and vessels can pass under it. In case of high water levels, the floodgate is closed to regulate the water flow in the Amsterdam-Rhine Canal. In this case, vessels have to travel via the locks.

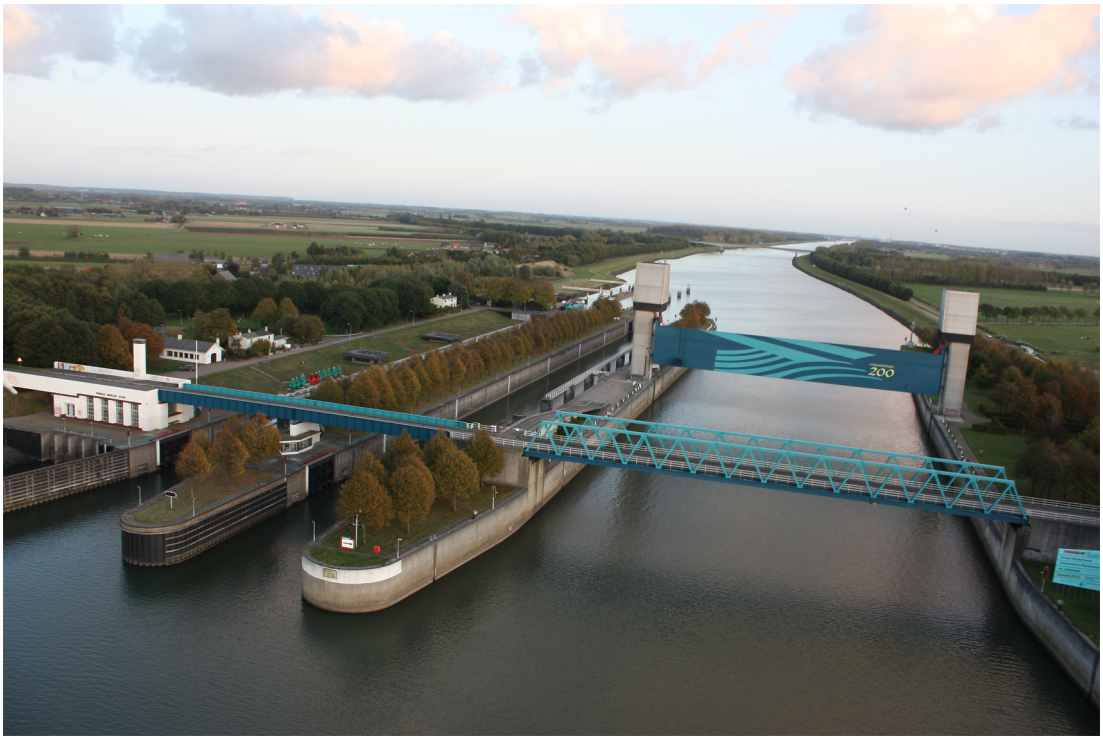


Figure 6.1: Overview of the Prinses Marijke complex [https://hofstraheersche.nl].

A waterway lock is used in canals and rivers to raise or lower vessels between different water heights. Moreover, similar to the floodgate, it is used to block and regulate the water flow. To do this, a lock consists of a large chamber (here: 300 m × 18 m), that is separated from the rest of the canal by watertight gates at both sides. The water level inside the chamber can be varied by opening or closing paddles in the gates. Vessels entering and leaving the lock are signaled by lock traffic lights. An operator is responsible for controlling the lock, i.e., opening and closing the gates and paddles, and switching the traffic light aspects.

Building of the Prinses Marijke locks was completed in 1939, while the floodgate was completed in 1981. Many of the technical installations are approaching the end of their life, and a complete renovation of the complex is planned. Part of this renovation includes the supervisory control system. Currently, the supervisory controller is implemented using a relay control box. Operators interact with the supervisory controller via a control panel consisting of push buttons and feedback lamps. To increase safety and functionality, the relay circuits will be replaced by a PLC and the control panel by a graphical user interface.

6.2 Hardware-in-the-loop simulation

As discussed before, model simulation is a valuable tool for early validation, yet it is only a partial analysis. HIL simulation offers a method to validate the behavior of the supervisory controller implemented on a PLC (in contrast to a model of the supervisory controller) with its subsystems, e.g., a GUI or a logging system, together with a model of the plant. To implement this method, a (Java) program, that is used as the plant model in HIL simulation, is automatically generated from the simulation model, such that no additional modeling or coding effort is required. For generating this program, the method from Swartjes et al. (2014) is used. The PLC can be connected to the program, to perform HIL simulation. Additional, external subsystems, such as a GUI can be connected to the PLC, to validate its functionality. An operator can now control the HIL simulation via the GUI, as if he were operating the real system.

6.2.1 Hardware-in-the-loop set-up

Here, the set-up that has been developed for HIL simulation of supervisory controllers is described. A schematic representation of the set-up is depicted in Figure 6.2. The set-up consists of three parts: the GUI for the operator implemented on a PC, the supervisory controller implemented on a PLC, and a model of the system implemented on another PC. The GUI communicates with the PLC by sending commands given by the operator. The PLC communicates status information to the GUI to inform the operator about the system's state. The PLC can influence the behavior of the plant by switching actuators on or off. The plant reacts by switching sensors on or off. Physically, the communication between parts is realized using the OPC UA protocol over Ethernet.

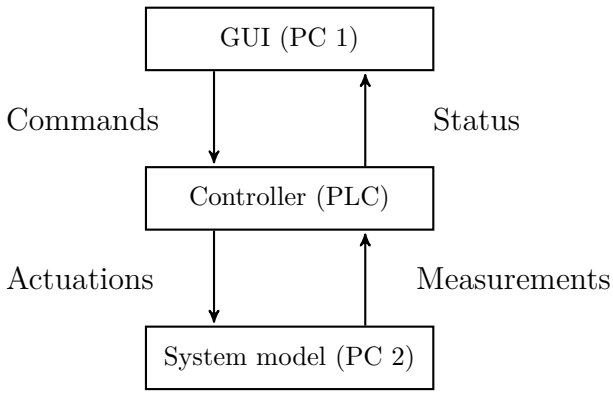


Figure 6.2: Schematic representation of the HIL set-up.

The GUI is implemented on a supervisory control and data acquisition (SCADA) system, which is the industrial standard for GUIs. Be aware, the term is misleading, as the supervisory controller is not implemented here. Designing the GUI and connecting the GUI to the supervisory controller is still a manual task. All the connections to the PLC should already be included in the discrete-event plant model for synthesis,

such that all required information is available and the supervisory controller knows which commands are possible.

The supervisory controller is implemented on a Siemens PLC, and its code in the structured control language (SCL), is automatically generated from the model of the supervisory controller, as described in Chapter 4. For the PLC, it does not matter whether it is communicating with the system model or with the real system.

Communicating with the real system is realized via physical inputs X and physical outputs Y of the PLC that are connected via wires to, respectively, sensors and actuators in the plant P. Additionally, a PLC has a set of internal variables M, that is not connected to the physical inputs or outputs. These variables are primarily used as memory, but also for communication to and from the GUI. The PLC cycle works as follows. At the start, the get action copies the values of the input signals X to the input image I. The PLC program is run, which can update the values of M and output image Q. At the end of the cycle, the put action copies the values of the output image Q to the output signals Y and the cycle restarts. This is schematically depicted in Figure 6.3. If the inputs and output hardware is not physically connected, the images I and Q can still be accessed. However, the get and put operations from X and to Y, respectively, are not performed.

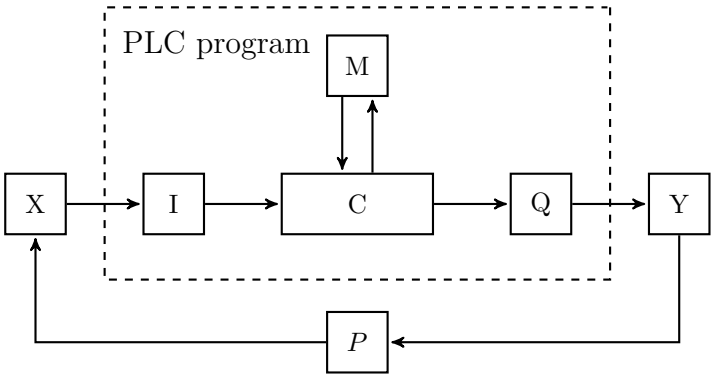


Figure 6.3: A schematic overview of the PLC in combination with the real plant.

Communicating with the HIL simulation is realized via virtual inputs and outputs. The system model P_{HIL} is also implemented on a SCADA system. Here, we used the Ignition SCADA software to execute code. The SCADA application is able to access and manipulate values of variables in I and Q. The code operates similarly to the PLC code. It cyclically reads all actuator signals from Q, calculates all state transitions, and then writes all the sensor signals to I, as schematically depicted in Figure 6.4. Hence, instead of the inputs being provided by X and outputs being supplied to Y, they are provided by and supplied to the HIL model.

A hardware mapping is needed for generating code of the system model. As the system model is a refinement of the plant model used for the synthesis, it uses the same events as the supervisory controller. Hence, the hardware mapping is exactly inverse (denoted by H_{map}^{-1}), and can therefore be automatically generated from the hardware mapping of the supervisory controller. To do so, the guards for sensors are substituted by assignments, and the assignments for actuators are substituted by guards. For example, when in the hardware mapping of the supervisory controller is

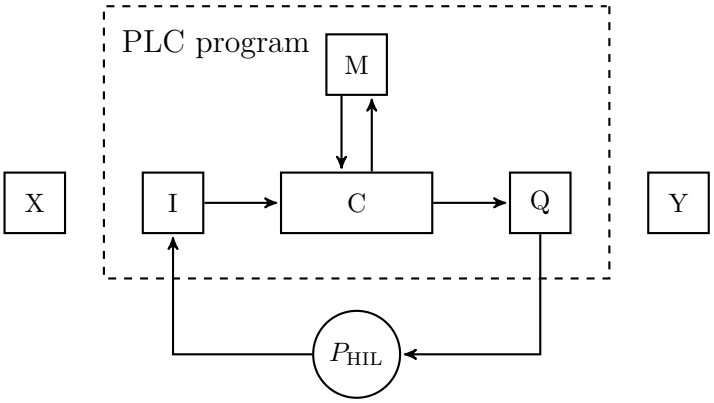


Figure 6.4: A schematic overview of the PLC in combination with the HIL set-up.

defined that a signal switches on when a actuator component is in the location `On`, in the inverse hardware mapping it is defined that when the signal is switched on, the controllable event that transits to location `On` has occurred. A similar inversion exists for uncontrollable events.

The process of generating code for the HIL set-up is depicted in Figure 6.5. As can be seen, during the SCL code generation, addresses of the variables in I, Q, and M are created (from H_{map}) that are used by P_{HIL} and the GUI. Because of this, there is no manual coding required to implement P_{HIL}. A step-by-step guide describing how to set-up the simulation is available in the repository.

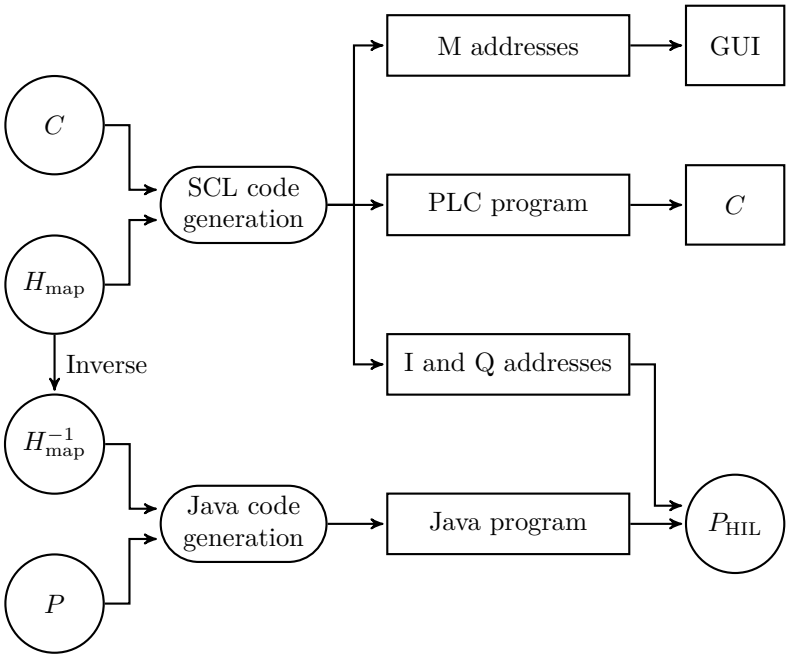


Figure 6.5: A schematic overview of code generation for the HIL set-up.

6.3 Case study: the Prinses Marijke complex

This section describes how a supervisor for the Prinses Marijke complex is synthesized and how a derived supervisory controller is implemented in the HIL set-up.

6.3.1 Modeling the plant and the requirements

To synthesize a supervisor, first the plant and the requirements are modeled. For the plant, many components can be represented by automata of the same type, i.e., automata that have the same location and transition structure. This shows the usefulness of using templates, i.e., standard models that can be instantiated per component, as first introduced in Grigorov et al. (2011). The use of templates is beneficial as it accelerates the modeling process, and also reduces the risk of modeling errors. In addition, many locks are similar, and consist of the same components, such that standard templates can be created for all locks. For this project, the templates that were developed for a different lock in Tilburg (Reijnen et al., 2017), have been re-used. Similarly, for the floodgate, the components have been modeled by the templates already defined for locks. The modeled components are: gates, paddles, entering lock traffic lights, leaving lock traffic lights, water-height sensors, and the GUI. These components are further decomposed into standardized sensor and actuator models. All templates and component models are implemented in CIF (van Beek et al., 2014) and are available in the repository.

To illustrate the process of developing a requirements model, consider a lock traffic light. A traffic light is positioned before a set of lock gates (at each side of the lock there are two mitre gates, also visualized in Figure 6.6). Because of safety, it should only be possible to switch to a green sign aspect (modeled as event `TL.c_g`), when a gate is fully open (modeled as location `Open`, in the component model of the gate sensor `Gate.S`). This requirement is modeled as the top requirement, given below. Furthermore, the traffic light should not spontaneously switch its aspect, but only when the respective command is activated from the GUI (modeled as location `GreenAspect`, in the component model of the GUI `Command`). This requirement is modeled as the bottom requirement, given below. All requirement models are implemented in CIF and are available in the repository.

$$\texttt{TL.c\_g}\ \textbf{needs}\ \texttt{EGate.S.Open} \wedge \texttt{WGate.S.Open}$$
$$\texttt{TL.c\_g}\ \textbf{needs}\ \texttt{Command.GreenAspect}$$

6.3.2 Supervisor synthesis

To synthesize a supervisor from the plant model and the requirements model, CIF is used. The synthesis algorithm in CIF uses a BDD implementation (Miremadi et al., 2012) of the algorithm given in Ouedraogo et al. (2011). BDDs allow to efficiently represent and perform calculations on automaton models with large state spaces.

In Table 6.1, the number of component models in each subsystem are given, as well as the number of requirements. It should be noted that the models for the north lock and the south lock are identical. The state-space sizes of the uncontrolled plant,

as well as of the plant under control of the supervisor are given. Depending on the number of PLCs that will be used, either three supervisors, or one supervisor can be synthesized, both results are provided in the table. For the one supervisor, the synthesis step took around 13 seconds on an i7, 2.60GHz, 8GB laptop.

Table 6.1: Model sizes for the Prinses Marijke complex.

System	Number of components	Plant state space	Number of requirements	Supervisor state space
Lock north	76	3.7×10^{19}	144	1.9×10^{11}
Lock south	76	3.7×10^{19}	144	1.9×10^{11}
Floodgate	24	5.9×10^{5}	24	6.6×10^{4}
Complex	176	8.1×10^{44}	312	2.4×10^{27}

6.3.3 Model simulation

For model simulation, the discrete-event plant model is refined by adding continuous-time behavior. In this case, the position and velocity of the gates, of the leveling blades, and of the floodgate are modeled. Additionally, a model of the water level inside the lock is included, that increases or decreases depending on the position of the leveling blades.

From the supervisor for the complex, a supervisory controller has been derived. This supervisory controller is simulated together with the refined plant model. The states of the automata during simulation are visualized via the schematic image given in Figure 6.6. Elements, such as color and position, change depending on the current location of each automaton and the values of variables. Commands given via the GUI can be simulated by clicking on interactive buttons in the image. Some control windows are hidden and can be opened by clicking on a component, e.g., clicking on a traffic light opens the traffic light control window.

To validate the behavior of the supervisory controller, several use cases have been analyzed. Whenever undesired behavior was found, the plant model or the requirements model was updated and a new supervisor was synthesized. Because the models developed for another lock were re-used, no missing or contradicting requirements were identified.

6.3.4 Controller code generation

To generate controlled code, a hardware mapping has been defined. In total, 82 inputs, 87 outputs, and 55 GUI commands are described in the hardware mapping. From the models, a PLC program has been generated that consists of 5500 lines of SCL (available in the repository). For the generation, we used CIF. The controller code has been implemented on an industrial Simatic S7-315F-2 PN/DP fail-safe PLC of Siemens.

It is known that several difficulties exist when generating controller code from a supervisor model and implementing it on a PLC, as discussed in Chapter 4. Some problems can arise due to the difference between a supervisor and a supervisory

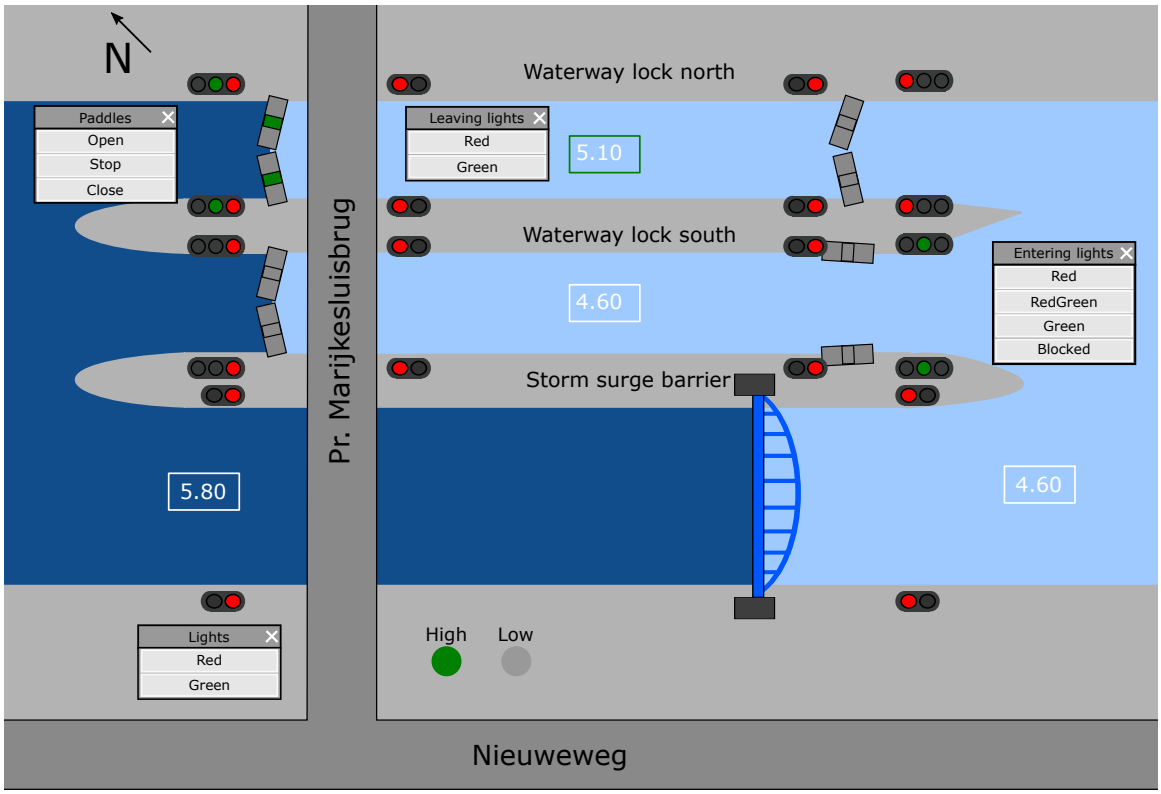


Figure 6.6: Model simulation of the Prinses Marijke complex.

controller, these are choice, infinite response, and causality. Some other problems can arise due to the behavior of a PLC, these are simultaneity and inexact synchronization. It has been verified that the supervisor is confluent, has finite response, and is nonblocking under control. During model simulation and HIL simulation no anomalies were observed due to delay.

6.3.5 Hardware-in-the-loop simulation

For HIL simulation, the set-up described in Section 6.2 is used. The generated controller code is implemented in the PLC. The GUI, shown in Figure 6.7, is designed according to the standardized format described by RWS, and connected to the PLC. The states of the traffic lights, gates, water level and the floodgate are obtained from the PLC memory M.

The Java program that was generated from the simulation model contains 10,300 lines of code (available in the repository). For the generation, we used CIF. To run the Java program and to connect the program to the PLC, we used the Ignition SCADA software. Connecting the program to the PLC interfaces was straightforward as the PLC input and output addresses were also generated. No additional coding was required to implement the program.

To validate the behavior of the supervisory controller, several use cases were tested. Some minor errors were found that relate to the communication between the GUI and the supervisory controller. These errors had to do with commands that could be given

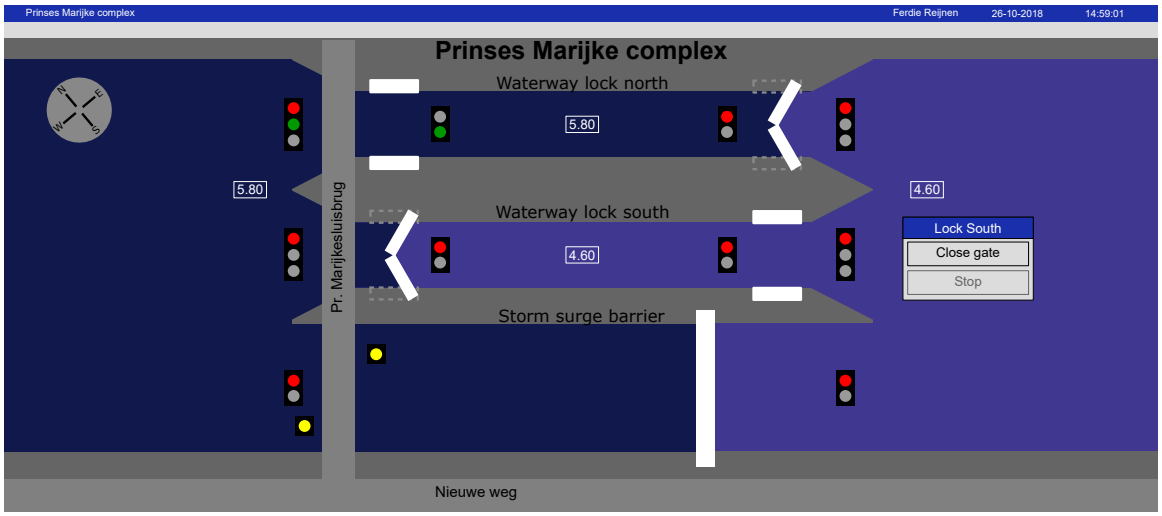


Figure 6.7: GUI for the Prinses Marijke complex.

to the supervisory controller when they should have been forbidden by the SCADA application. No errors were found that originated from the supervisory controller, as the supervisory controller was already extensively validated with model simulation before.

The HIL set-up is an effective tool for validation of the PLC controller code. Moreover, the HIL set-up can also function as a tool to demonstrate the whole development process of modeling, synthesis, and implementation. Especially the possibility to connect the supervisory controller to the GUI designed by RWS was valuable. In this way, the supervisory controller could be validated by RWS in the same manner (e.g., with factory acceptance tests) as they would validate supervisory controllers that were coded by hand. Also here, it was concluded by RWS that the behavior of the supervisory controller is as intended.

6.4 Concluding remarks

In this chapter, a method that integrates HIL simulation into the synthesis-based engineering framework is presented. HIL simulation offers the possibility to validate the behavior of the supervisory controller implemented on the hardware, e.g., a PLC, before the system is built. Compared to model simulation, it executes the controller code, instead of a model of the supervisory controller. Moreover, it is also possible to connect the supervisory controller to external subsystems such as a GUI or a logging system. For supervisor synthesis, a plant model is already available. Hence, the additional modeling effort required to obtain a simulation model and perform model simulation is small. A model suitable for HIL simulation can automatically be generated from this simulation model.

A case study on the Prinses Marijke complex was described that successfully uses this method to obtain the necessary models for supervisor synthesis, controller code generation, and HIL simulation. For this project, the HIL set-up was also successfully used as a demonstration set-up.

Chapter 7

A synthesized fault-tolerant supervisory controller for a swing bridge

When designing supervisory controllers, a relevant aspect is the robustness against faults (e.g., a broken actuator) in the plant. It is important that the supervisory controller is able to compensate faults to some degree, to maintain (degraded) functionality while still guaranteeing safety properties. Such a supervisory controller is called a fault-tolerant supervisory controller.

In Baeten et al. (2016), it has been shown how supervisor synthesis techniques and tools can be integrated into the engineering process for supervisory controllers. There, the focus is mostly on the specification and the design of the supervisory controller. A difficulty that is not addressed in Baeten et al. (2016) is how to obtain a fault-tolerant supervisory controller. Moreover, the implementation step, which has proven to be complicated (see, e.g., Fabian and Hellgren (1998), Zaytoon and Riera (2017), and Chapter 4) is also not addressed there. Even though supervisor synthesis techniques have been under development for over 30 years, the use in industry has not been widespread (Wonham et al., 2018; Laing et al., 2020). One of the main reasons for this is that until recently synthesis techniques were unable to scale to industrial-size problems. However, research such as Vahidi et al. (2006), Miremadi et al. (2011), Ouedraogo et al. (2011), and Goorden (2019) and advances in computation power and memory availability facilitated the use of supervisor synthesis for industrial-size problems, as demonstrated in, e.g., Moormann et al. (2020a) and Reijnen et al. (2020a).

The contribution of this chapter is as follows. It demonstrates the use of supervisor synthesis for an industrial case study, namely a swing bridge, for which a fault-tolerant supervisory controller has been synthesized, validated, implemented, and tested. With this case study, it is shown that synthesis techniques in combination with fault-tolerant control (FTC) have matured to a point where they are powerful enough to be applied for industrial-size problems.

This chapter is based on: Reijnen, F.F.H., Leliveld, E.-B.M.L., van de Mortel-Fronczak, J.M., van Dinther, M.J.T., Rooda, J.E., and Fokkink W.J. (2020). "A synthesized fault-tolerant supervisory controller for a swing bridge". Submitted.

The chapter is structured as follows. FTC is discussed in Section 7.1. The swing bridge, for which a fault-tolerant supervisory controller has been synthesized and implemented is introduced in Section 7.2. In Section 7.3, the modeling and synthesis of the supervisory controller is described. Implementation and validation of the supervisory controller are described in Sections 7.4 and 7.5, respectively. The results are evaluated in Section 7.6.

7.1 Fault-tolerant control

In systems, it is possible that components fail. For safety and availability of the system, it is important to take failures into account when developing a supervisory controller. Examples of faults are a broken wire, a blocking actuator, or a vessel hitting a bridge. These faults belong to the of class of permanent faults (Blanke et al., 2016). For a permanent fault, there is a specific point in time from whereon it is present in the system. Drift-like faults such as wear are not considered. A reason for this is that such faults are generally resolved by resource controllers, and not by the supervisory controller.

A fault-tolerant supervisory controller is able to satisfy the requirements both in nominal operation and after the occurrence of a fault. In Moor (2016), an overview on FTC in combination with supervisor synthesis is provided. Two types of FTC can be distinguished, active and passive. In active FTC, the identification of the fault is used in the control logic. Whereas in passive FTC, the control logic does not change after identification. In this chapter, active FTC is used, as in general this leads to the most permissive supervisory controller (Jiang and Yu, 2012).

Active FTC consists of two steps, detecting a fault and reacting upon the detection. For the first step, inputs and outputs of the system can be monitored by diagnosers, as demonstrated in Sampath et al. (1996). Depending on the diagnoser observations, faults can be identified and reported to the supervisory controller. Different methods and techniques to automatically derive diagnosers are discussed in Zaytoon and Lafortune (2013). Another way of identifying faults is by sensors embedded in the control unit, for example, PLCs can detect a broken wire, or sensors in the plant, such as an overheat sensor.

While fault diagnosis is concerned with how to identify faults, FTC is concerned with what the supervisory controller should do after such an identification. A schematic representation of the diagnosers in combination with the system and the supervisory controller is given in Figure 7.1.

In this set-up, diagnosers are modeled as observers that provide additional inputs, i.e., fault identifications, to the supervisory controller. Since identification events cannot be effected by the supervisory controller, they are regarded as uncontrollable. The identification event is triggered by a diagnoser or by embedded sensors in the control unit.

In the requirements model, the identification of a fault can be used in the guard expression. In this way, the behavior of the supervisory controller changes when a fault has been identified. Here, the modeling method of Reijnen et al. (2018b) is used.

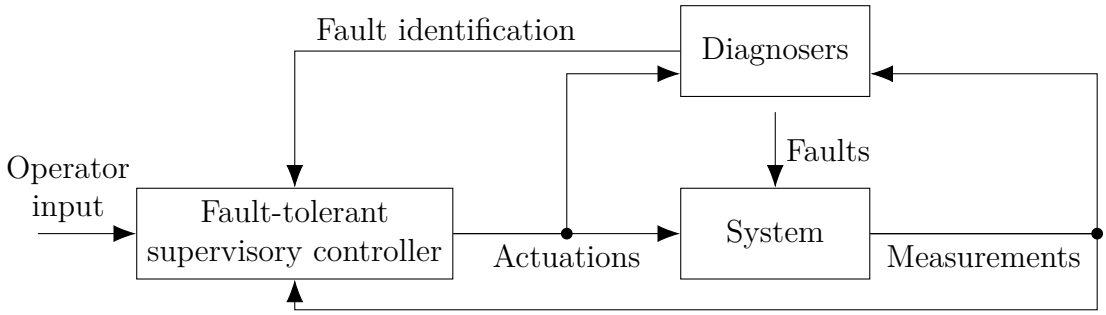


Figure 7.1: Fault-tolerant control set-up.

7.2 The Oisterwijksebaan bridge

As a case study, a fault-tolerant supervisory controller for the Oisterwijksebaan bridge (OBB), located in Tilburg, the Netherlands, has been modeled, synthesized, validated, and implemented. The OBB, shown in Figure 7.2, is a swing bridge that provides a way for vehicles to cross the Wilhelmina Canal. It consists of a two-way vehicle lane and two pedestrian lanes. The bridge is operated and maintained by RWS.

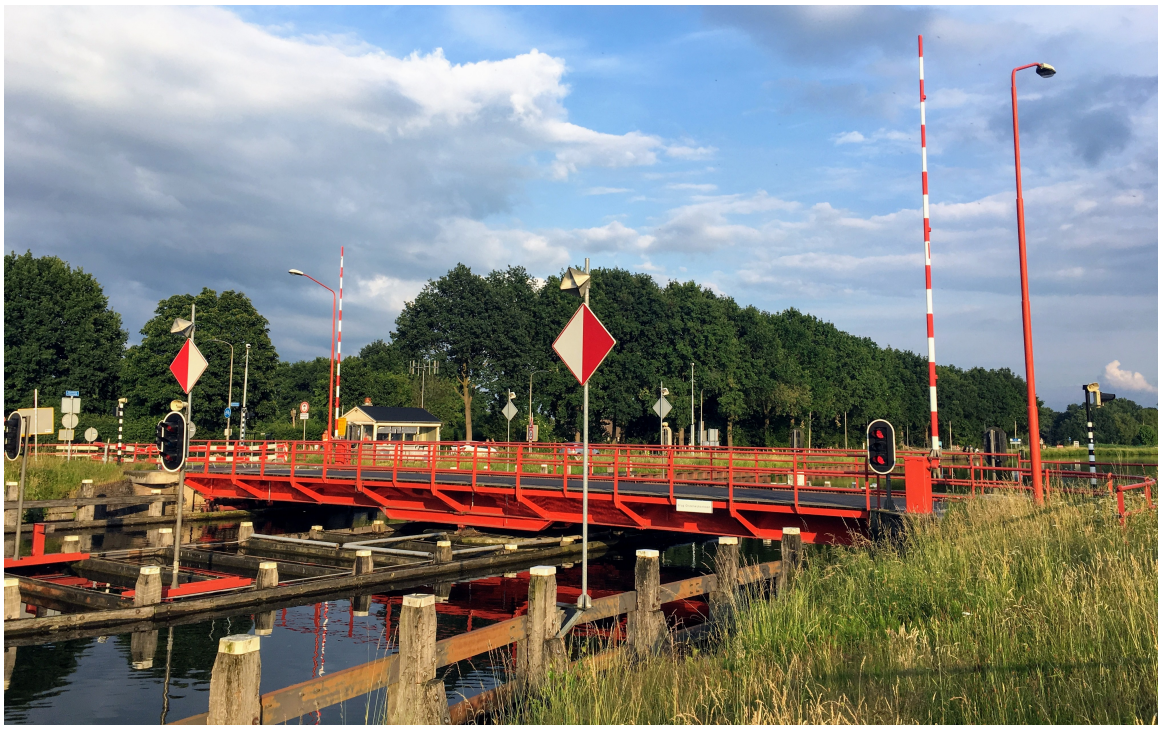


Figure 7.2: The Oisterwijksebaan bridge.

A schematic overview of the bridge is shown in Figure 7.3. On both sides of the bridge, stop signs are located (SS1 - SS5), which warn the land traffic to stop before the boom barriers are lowered. These boom barriers (BB1 - BB2) are placed on both sides of the bridge, to guarantee a safe situation before the bridge is opened. The vessels on the Wilhelmina Canal are informed by red-green-red vessel traffic lights

(VTL1 - VTL8). The bridge deck (BD) consists of a locking mechanism, a brake, and a rotating mechanism. To unlock the bridge, a hydraulic pump is used to lower the bridge into its bearings. The bridge is raised out of its bearings to lock it. Whenever the brake is applied, rotation is not possible. For rotation, an electric motor is used. The direction and speed of this motor can be controlled.

An operator controls the bridge via a GUI. The GUI has two purposes: providing information to the operator about the system's state, such as the displayed sign aspects and identified faults, and acting as a control panel. The commands that can be given are, e.g., stopping land traffic, rotating the bridge, and switching sign aspects. In total, the bridge is controlled by 39 actuators, based on the inputs of 53 sensors and 26 control commands.

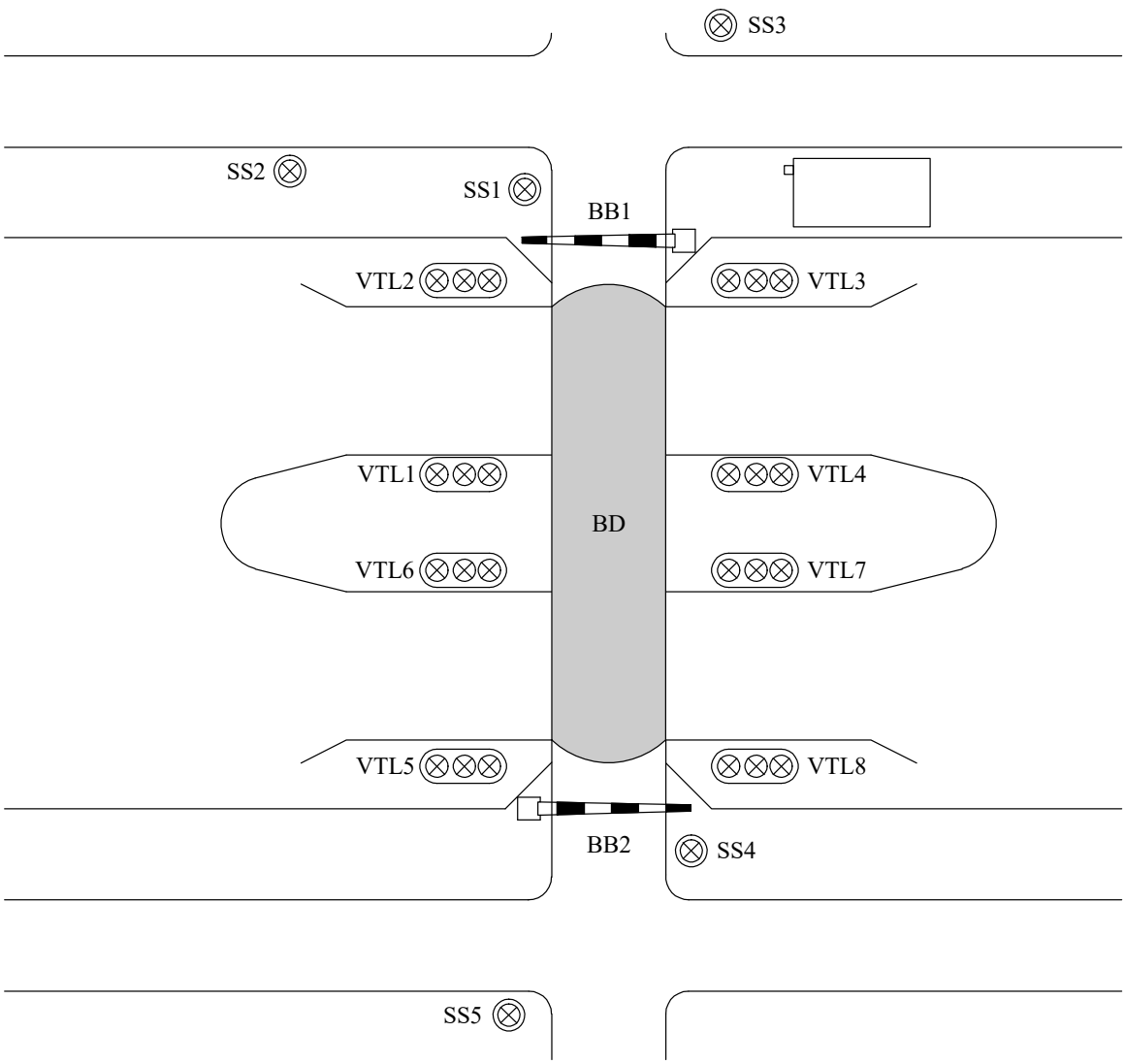


Figure 7.3: A schematic overview of the Oisterwijksebaan bridge.

7.2.1 Desired controlled behavior

The desired controlled behavior when no faults are present is as follows. When a vessel wants to pass the bridge, the operator initiates the *open bridge* procedure. Then, all five stop signs switch on. After 14 seconds, the two boom barriers close. Then, the bridge is lowered into its bearings, the brake is released, and the electric motor is started. The bridge rotates at a constant speed until it is almost open, then the bridge decelerates. When the bridge is fully open, the brake is applied. During the bridge rotation, the operator switches the VTLs to a red-green aspect, indicating a vessel can approach the bridge. Once the bridge is fully open, a green aspect is shown. The *close bridge* procedure is similar to the procedure described above.

Due to the human-machine interaction, the bridge has to adhere to strict standards to guarantee human safety. Some of these are: 'A boom barrier may not be raised when the bridge is not fully closed' and 'The bridge may only rotate when the VTLs display a red aspect'.

For safety and availability, the bridge should be robust against faults. A component failure may not lead to dangerous situations. Moreover, when a fault occurs, the bridge should be able to continue its operation, with possibly degraded functionality. Two examples of such situations are: 'When a red lamp of a VTL fails, this should be detected and signaled to the operator, and the VTL should be deactivated' and 'When one out of two VTLs in a passage fails, the bridge may still be closed'.

7.3 Supervisor synthesis for the OBB

In this section, modeling of the components, diagnosers, and requirements is discussed. Subsequently, the results of the supervisor synthesis are presented. A description, visualization, and the CIF code of all models are available in the repository[1].

7.3.1 Component models

For the plant model, the component-based modeling method as proposed in Chapter 3 is used. In this method, every component (actuator, sensor, or operator command) is modeled as a separate EFA. These automata typically consist of a small number (2-5) of locations and do not share events with other EFAs. An overview of the component models is shown in Figure 7.4. In this figure, the number of component EFAs is denoted. The model can be partitioned into two parts: the physical part and the GUI part.

For the Oisterwijksebaan bridge, 27 actuator component models, 48 sensor component models, and 7 command models are needed. A model can represent multiple inputs, outputs, or control commands. For example, one VTL actuator EFA represents the outputs for all three lamps. As an illustration, the model of the VTL actuator is shown in Figure 7.5. The model consists of four locations, representing the four legal aspects that can be shown.

[1] www.github.com/ffhreijnen/OBB

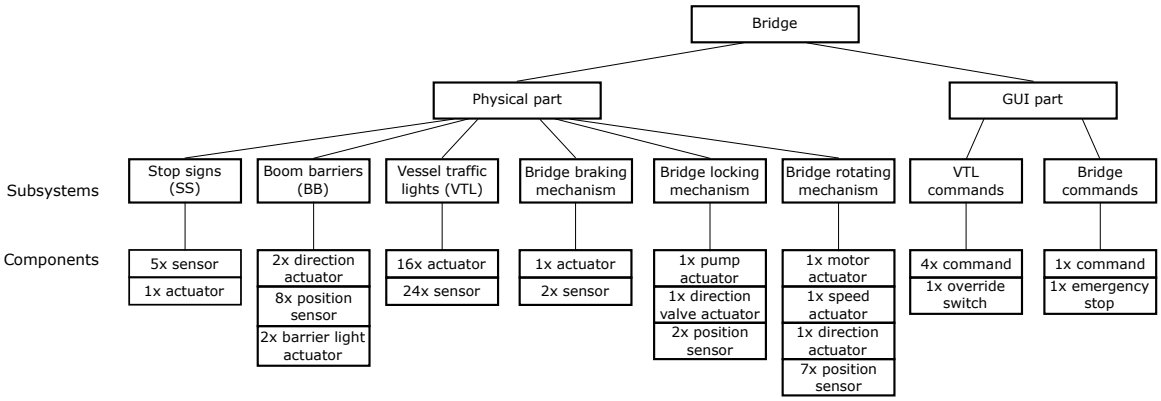


Figure 7.4: Plant model decomposition of the OBB.

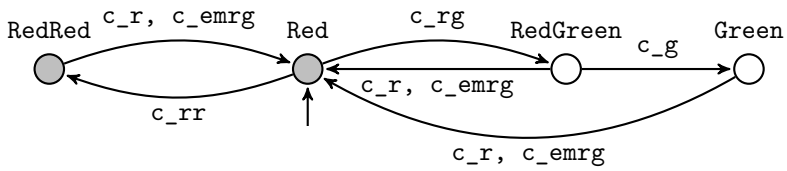


Figure 7.5: Component model of the vessel traffic light actuator.

7.3.2 Diagnoser models

For safety and availability, it is necessary to compensate for faults in the system. To detect faults, diagnosers are used. Here, the diagnosers are used to detect three types of faults:

- Discrepancy faults - the sensor measurement of an action performed by an actuator is different from what is expected (e.g., a lamp is broken).
- Duration-monitoring faults - performing an action takes longer (shorter) than expected (e.g., the bridge is not accelerating (decelerating)).
- Unexpected-movements faults - a component starts moving without being actuated (e.g., a car hits a boom barrier).

When a fault is diagnosed, this is signaled to the supervisory controller and thereafter to the operator. Based on the formalized requirements, the supervisory controller can influence the behavior of the system after the fault identification.

As an example for a diagnoser model for a discrepancy fault, consider a land-traffic sign that is actuated via Boolean signal Q. The status of the lamp is measured via Boolean signal I. It is expected that within t_f seconds after actuation the sensor should switch on, otherwise a fault has occurred. The EFA in Figure 7.6 models the behavior of this diagnoser. When I and Q have different values, the EFA will transition to location `Diagnosing` and a timer value t will be set to t_f. In this location, the timer value decreases (denoted by $t' = -1$). If $t \leq 0$ and I and Q have different values, the EFA transitions to location `Broken`. The diagnoser can be reset via the `reset` event.

As an example for a diagnoser model for a duration-monitoring fault, consider a boom barrier that is actuated via Boolean signals Q_o and Q_c for opening and closing,

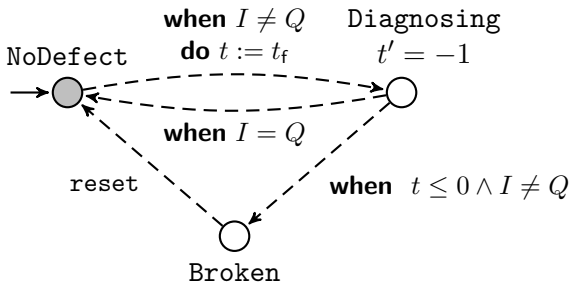


Figure 7.6: Model of the diagnoser for a broken traffic sign.

respectively. The position of the barrier is measured via Boolean signals I_o and I_c for the fully open position and the fully closed position, respectively. It is expected that the barrier opens in at most t_o seconds and closes in at most t_c seconds, otherwise a fault the barrier is stuck. The EFA in Figure 7.7 models the behavior of this diagnoser.

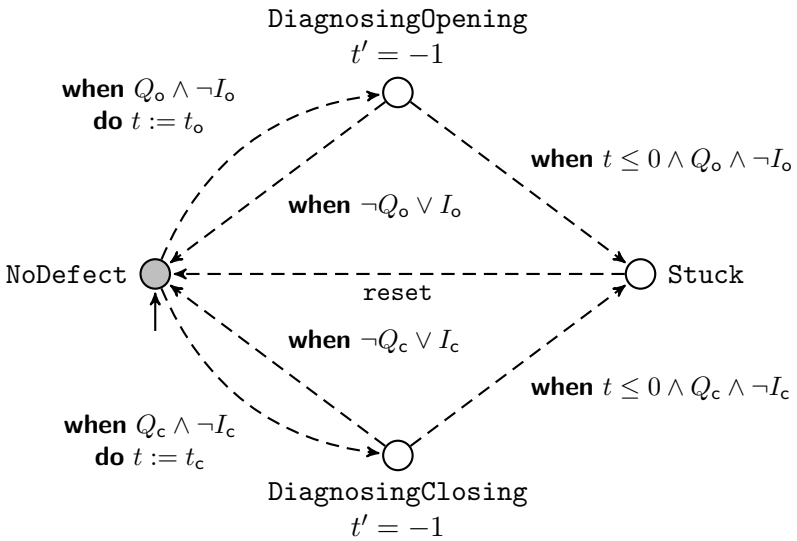


Figure 7.7: Model of the diagnoser for a stuck barrier fault.

7.3.3 Requirement models

The desired behavior of the system is formalized using event-condition requirements. Such a requirement enforces that a certain event is only allowed to occur when the condition is satisfied. For example, consider VTL1, modeled as the EFA in Figure 7.5. The event `green` is only allowed to occur when the bridge is fully open, represented by location `On` in EFA `Bridge.Open` (this EFA is not shown). This is modeled by the following event condition.

`VTL1.c_g` **needs** `Bridge.Open.On`

To achieve fault tolerance, it is specified what has to happen when faults have been diagnosed. For example, a VTL may only show the red-green aspect when its

red lamp has not failed. This is modeled by the following event condition.

`VTL1.c_rg` **needs** ¬`VTL1.RedLampDignoser.Broken`

The complete desired behavior of the OBB can be formalized by defining many small event conditions. In total, the OBB model consists of 167 event-condition requirements, provided in the repository.

7.3.4 Supervisor synthesis

From the described models, a supervisor has been synthesized using CIF. For this case study, the plant model consists of 82 components and 40 diagnosers. The requirements model consists of 167 event conditions. A supervisor has been synthesized in a few minutes (on a standard laptop). The supervisor represents a state space of $2.1 \cdot 10^{46}$ states. Even though the number of component, requirement, and diagnoser models is large, the synthesis procedure is fast enough.

7.4 From supervisor to supervisory controller

There exist subtle differences between a supervisory controller and a supervisor as obtained via synthesis, also discussed in Chapter 4. A supervisor can only forbid events, whereas a supervisory controller can can choose when to execute the controllable events. Methods exist to derive a supervisory controller from a supervisor automatically, see, e.g., Balemi et al. (1993), Fabian and Hellgren (1998), Malik (2003), and Zaytoon and Riera (2017), and Chapter 4. In essence, what is done is to keep executing controllable events among the enabled events of the supervisor, until no more controllable events are enabled. Then, the supervisory controller waits for a new plant input (e.g., a sensor that switches on).

For this method to be applicable, the supervisor needs to satisfy additional properties not guaranteed by synthesis. These properties are finite response, confluence, and nonblocking under control, as described in Section 4.3. Finite response ensures that always a state is reached where no controllable events are enabled. Confluence ensures that each time the supervisory controller can choose between multiple controllable events, the choice can be made arbitrarily. Nonblocking under control requires that a marked state can be reached without being dependent on events that happen only in rare situations. When the supervisor does not satisfy these properties, the requirements should be altered. In Reijnen et al. (2019a), guidelines are provided for this. To verify that the synthesized supervisor for the OBB adheres to these properties, CIF has been used.

7.4.1 Supervisory controller code generation

The supervisory-controller model has been used to generate controller code. For this, the procedure in Section 4.5 is used. A major advantage of this algorithm compared to others, such as Fabian and Hellgren (1998), is that it does not require explicit coding

of each state in the supervisory controller, which would be infeasible for $2.1 \cdot 10^{46}$ states. Instead, for every event in the model, an if-then statement is generated. The size of all these statements on the PLC is about 200 kB.

CIF has been used to generate the implementation code. The generated code can be automatically imported by the PLC vendor software, in this case, TIA portal from Siemens.

7.4.2 Supervisory controller code implementation

The generated code has been implemented on a Siemens S7-300 PLC. Because for each component a model exists (shown in Figure 7.4), the generated code can easily be connected to the input and output signals of the PLC. As, each state in the model corresponds to specific values of the PLC signals. For example, in state `RedRed` in Figure 7.5, the signals belonging to the activation of the top red lamp, middle green lamp, and bottom red lamp of the VTL, are on, off, and on, respectively.

In the literature (e.g., Fabian and Hellgren (1998) and Zaytoon and Riera (2017)), differences are described between the execution of a supervisory-controller model and a PLC implementation (see also Section 4.4). These differences originate from the real-time execution of the code. The PLC executes a repeating cycle, consisting out of three parts: reading the inputs, executing the code, and setting the outputs. As a result, when a sensor signal changes, this is only noticed during the next input reading.

This delay can lead to a situation where an undetected change in a sensor invalidates the choice of the action made by the supervisory controller, referred to as inexact synchronization. A delay can also lead to a situation where between successive PLC scan cycles two or more events occur. If this happens, the supervisory controller is unable to recognize the exact order of the events, referred to as simultaneity.

In general, when the process time of the system is much higher than the cycle time (the time it takes the PLC to finish the three phases), the above situations are never encountered, as demonstrated by Malik (2003), Forschelen et al. (2012), and Reijnen et al. (2017). In case of the OBB, the process time is in the order of seconds, whereas the cycle time of the PLC is in the order of milliseconds.

7.5 Validation of the supervisory controller

Although the supervisory controller is guaranteed to behave according to the requirements, the resulting controlled behavior might not be as expected, due to incomplete or incorrect requirements. For example, requirements could be too strict, resulting in the supervisory controller not exhibiting the desired behavior. Also the model of the plant can differ from the real system. Hence, the behavior of the supervisory controller and the plant have to be validated.

There are three stages in the validation process: model simulation, HIL simulation, and system testing.

7.5.1 Model simulation

For model simulation, the supervisory-controller model, the plant model (containing the physical part and the GUI part), and a visualization are used. The visualization is connected to the component EFAs and represents their states. Using a visualization is more intuitive for validation than only showing the state of each EFA. Figure 7.8 shows the visualization of the plant. Figure 7.9 shows the visualization of the GUI. Two visualizations have been used to differentiate between what happens with the real system and what an operator sees on the GUI. Simulation has been performed using CIF.

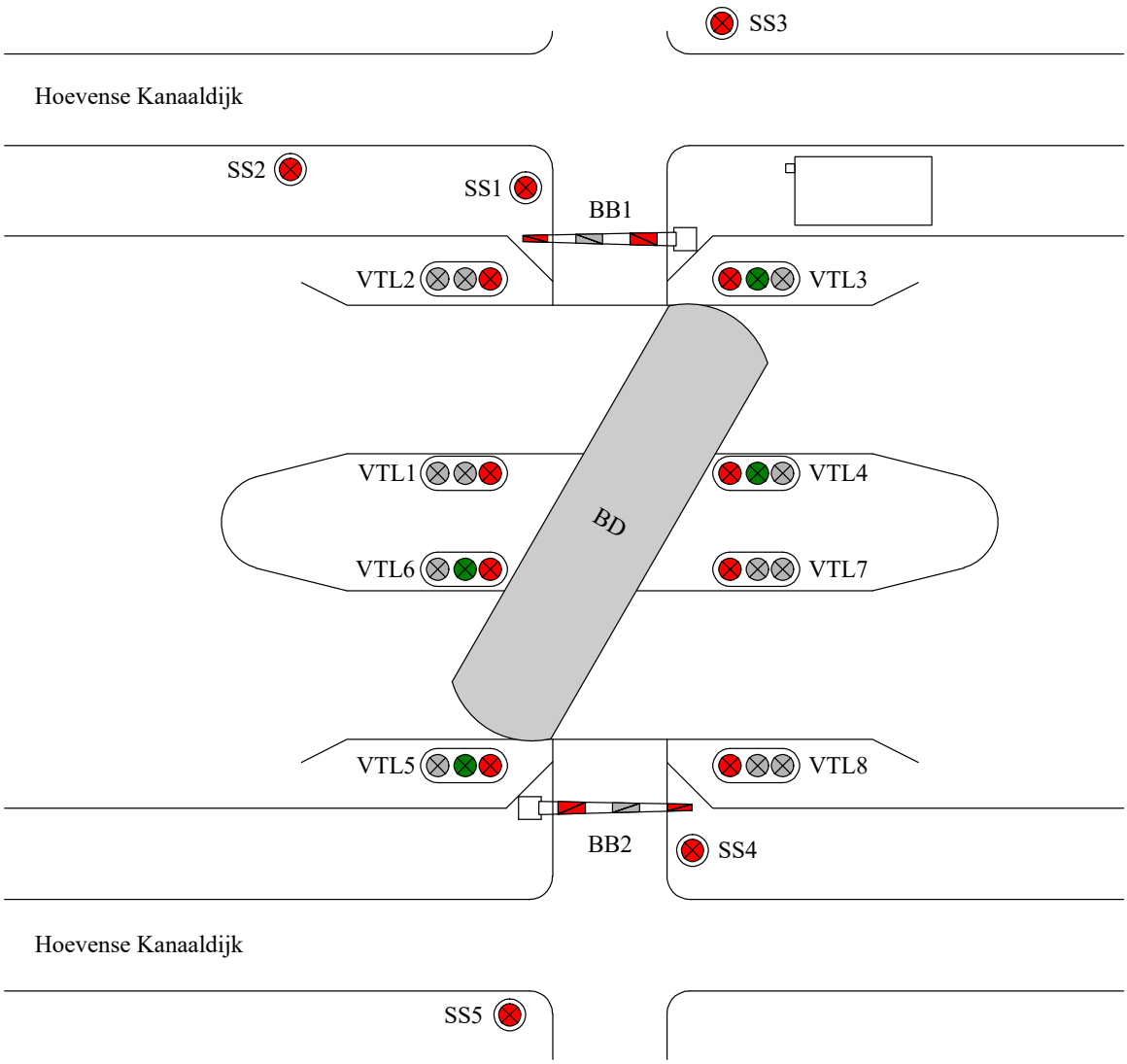


Figure 7.8: Visualization of the plant used for model simulation.

Simulation has been used extensively to validate the behavior of the controlled system. For this, test cases, also called factory acceptance test (FAT) protocols, with predefined sequences have been used. The protocols consist of actions for the operator to perform and expected reactions from the supervisory controller. These

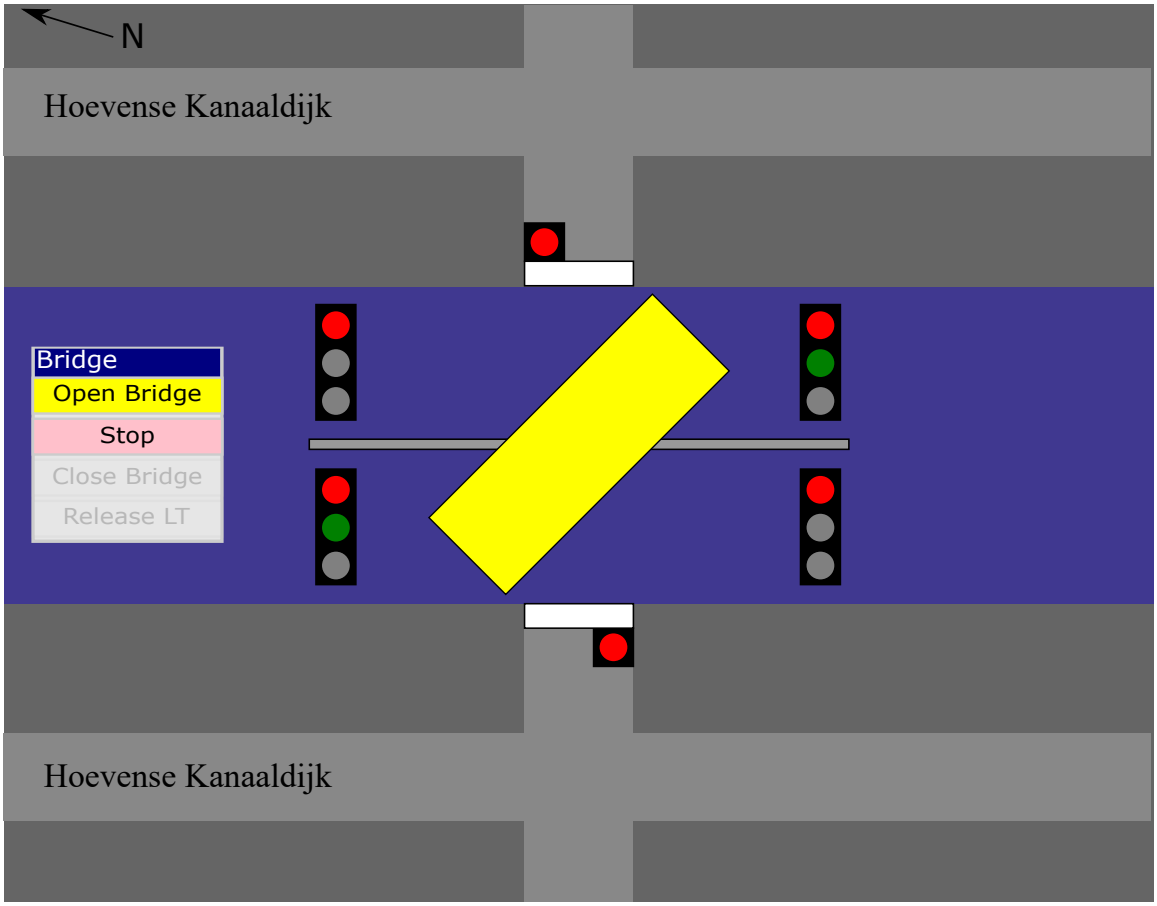


Figure 7.9: Visualization of the GUI used for model simulation.

test cases have been formulated by RWS. Whenever the behavior was not as desired, the requirements model was altered. This led to an iterative process of 1) adapting the requirements, 2) synthesizing a supervisor, and 3) simulating the result.

During the simulation less than five small errors where observed. Two of these are: 'when the operator clicks on the bottom red lamp of any VTL in the GUI, all VTLs should display a double red aspect' and 'when a VTL displays a red-green aspect when the bridge deck reaches its open position, the VTL should automatically show the green aspect'. Additionally requirements have been added to the requirements model to obtain this behavior.

7.5.2 Hardware-in-the-loop simulation

For HIL simulation, the supervisory controller is implemented in the real PLC. The PLC is connected to two computers. On one computer, the physical part of the bridge is simulated. On the other computer, the GUI is implemented, manually. The model used for HIL simulation is implemented in Java code, and is automatically generated from the physical part of the plant model. This set-up allows the supervisory controller and the GUI to be validated as if they were implemented in the real system. A detailed description of the process is given in Chapter 6 and Reijnen et al. (2019b).

To validate the behavior, the same FAT protocols were used. As the behavior had already been validated with model simulation, no anomalies were observed. These tests show the supervisory controller has been implemented correctly on the PLC and that the interaction between the GUI and the supervisory controller is as expected. It has also been verified that the cycle time is sufficiently short. Additionally, bridge operators provided feedback on the design of the GUI. Based on their feedback, a feature was added that gives priorities to different kinds of faults, such that faults with a high priority are displayed in a different color than faults with a low priority.

7.5.3 System testing

For system testing, the supervisory controller is implemented on the PLC connected to the real bridge. To realize this test, the OBB was closed for traffic for one night. During this period, the original supervisory controller has been replaced by the generated code. Since the controller code had already been validated in the HIL set-up, the installation required little effort. The GUI that had been developed for the HIL set-up was used here as well. During the system test, the bridge was operated by professional RWS operators. The operators did not experience differences between the synthesized supervisory controller and the original supervisory controller.

To validate the behavior of the real bridge, controlled by the supervisory controller, site acceptance test protocols have been used. The protocols are similar to the FAT protocols, except that for generating faults, the physical system has been modified. For example, wires are disconnected to simulate failing sensors. The only error that has been observed during the test was a value of the program not being converted correctly to an analog signal for the speed actuator. This was fixed in the output module settings. The error was not observed before because in the HIL set-up the value of the program is used, not the analog signal (i.e., the put action from Q to Y is not performed, see Figure 6.4, on page 96).

This test illustrates that the supervisory controller is able to control the real bridge. The difference in behavior between the simulation model and the real bridge was minimal, e.g., hysteresis in position sensors of the bridge deck was observed. This means that when the bridge reaches an angle where a position sensor should switch on or off, it would switch on and off rapidly, for a few PLC cycles. However, this did not lead to any problems.

7.6 Evaluation of the engineering method

Model-driven software engineering combined with formal methods provides a powerful way of working for the design of supervisory controllers. It provides a structured and systematic technique for the design and specification of the system's behavior. Moreover, it provides more consistency and allows for less ambiguity than documents written in a natural language. The use of formal models in an early stage of the product development process, forces the engineers to clarify all aspects of the system. Clarity contributes to a good design of the control software.

The use of synthesis guarantees that the requirements are always satisfied by construction. As a result, the supervisory controller does not have to be verified against these requirements. Instead, the focus lies on assuring that the requirements are complete and correct. For this, simulation-based validation using predefined test protocols is a powerful and intuitive tool. In this case study, all errors related to the behavior of the supervisory controller were found during model simulation. During HIL simulation and system testing, only errors related to the GUI and the hardware were found. These errors could be repaired without altering the supervisory controller.

In the literature, there are concerns about the technical implementation of the supervisory controller obtained via synthesis, e.g., Fabian and Hellgren (1998) and Zaytoon and Riera (2017). In this project, these topics have been addressed systematically.

- In Zaytoon and Riera (2017), it was argued that finding the right abstraction level for the plant is difficult. They note that a higher-level description leads to a synthesis problem that is easy to solve but the result may be difficult to implement on a PLC. On the other hand, a more detailed model may lead to unrealistic-size controllers, due to a possible explosion of the state space. For our case, the component-based modeling method from Chapter 3 has been used. This is an intuitive abstraction level for developing models and makes implementation on the PLC straightforward. Moreover, by generating code for each EFA model separately, as explained in Section 4.5, the resulting code size is only 200 kB.

- Robustness against faults has been achieved by including fault diagnosis and fault-tolerant control in the models, making it possible to guarantee that the behavior of the supervisory controller is safe, even when faults occur.

- Problems related to the differences between a supervisor and a supervisory controller have been solved by using the way of working described in Chapter 4.

- The delay between the supervisory controller and the plant is negligible in this case study, where the control hardware is much faster than the process to be controlled.

While the use of synthesis techniques has not yet been widespread in industry, this case study shows that these techniques have matured to a point where they are applicable in practice.

Chapter 8

Design of a supervisor platform for movable bridges

In this chapter, the design of supervisors for a product platform is considered. Following the definition of Meyer and Lehnerd (1997), a product platform is a set of common components, modules, or parts from which a stream of derivative products (i.e., a family) can be efficiently created. In this way, several similar systems can be composed out of standardized components.

Inspired by the work of Ekberg and Krogh (2006), Grigorov and Rudie (2010), Grigorov et al. (2011), and Malik et al. (2011), the use of (domain-specific) modules for the design of the necessary models for synthesis and simulation is investigated. Modules can be compared to classes in object-oriented programming languages such as C++ and Java. A module acts as a blueprint that can be instantiated to create a part of the model. A library of commonly used modules can be established, such that a model can be composed from these modules. In Ekberg and Krogh (2006), templates are introduced as a means to instantiate copies of similar FAs. Templates have also been used in Chapter 3 for this purpose. This work has been extended in Grigorov et al. (2011), where a template-based design method has been proposed. In that method, templates for components and requirements are used to design the plant model and the requirements model. These models are in the form of FAs. Interaction between templates is modeled using event-name maps that link events in the component models to events in the requirement models. In Grigorov et al. (2011), it was concluded that template-based design significantly increases the accessibility of supervisor synthesis, and is useful for less error-prone and rapid model development. In Grigorov and Rudie (2010), the work of Grigorov et al. (2011) has been extended by including templates with parameters. Then, a template is instantiated for some value of the parameters. In Malik et al. (2011), modules with defined interfaces were introduced. Modules group several automata and requirements. Within these modules some events are local, whereas other events are global, which provides the infrastructure needed for modular modeling. Modules provide a more general way of instantiating than templates.

This chapter is based on: Reijnen, F.F.H., van de Mortel-Fronczak, J.M., Reniers, M.A., and Rooda, J.E. (2020). "Design of a supervisor platform for movable bridges". In: *Proceedings of the 16th Conference on Automation Science and Engineering.* IEEE, pp. 1298-1304.

This chapter describes a method to develop supervisors for a product platform. In this method, a graphical model of the plant is built from a predefined set of modules. A tool has been developed that implements this method for the design of supervisors for movable bridges. The tool has been used for the design of supervisors for seventeen movable bridges. Different from Grigorov et al. (2011), the requirements are modeled as state-based (logic) expressions. From these graphical model and the expression, the necessary models for synthesis and simulation are generated. Compared to the method of Grigorov et al. (2011), EFAs are used, such that also variables can be used. In ter Beek et al. (2016), a method is presented that is able to synthesize a supervisor that is able to control all products of a family. In our method, a supervisor is synthesized for each individual product, as this leads to smaller supervisors.

Compared to manually designing the models, the method as described in this chapter increases the following aspects.

- **Model quality:** In Frakes and Kang (2005), Selby (2005), and Åkesson et al. (2020), it is argued that the reuse of components increases the quality because of more careful design and testing and more extensive usage than that of single-use equivalents. While this effect was not observed in the study performed by Grigorov et al. (2011), they argue that this may be due to the investigated problem being too easy.
- **Development productivity:** Reuse of models increases productivity by avoiding redevelopment, as stated in Selby (2005) and Åkesson et al. (2020). For supervisor synthesis, it was experimentally shown in Grigorov et al. (2011) that the modeling time is significantly reduced when modules are reused.
- **Accessibility:** By having a module library, the user does not have to be an expert in automata-based modeling, as modules can be instantiated from the library. In Grigorov et al. (2011), it was reported that participants found it much easier to use this way of modeling.

The use of domain-specific modules requires an initial investment in terms of effort, as discussed in Mellegård et al. (2015), Liebel et al. (2018), and Åkesson et al. (2020). However, if sufficient instances are created, the saved effort outweighs the initial effort. If supervisors for a large number of similar systems have to be created, it makes the proposed method worthwhile.

This chapter is structured as follows. A description of the modeling method is provided in Section 8.1. A tool that implements this method for the design of models for a family of bridges is described in Section 8.2. The results from the case study are presented in Section 8.3. Finally, concluding remarks are provided in Section 8.4.

8.1 Modeling

In case studies where supervisor synthesis has been applied, it can be observed that models often consist of similar modules. For example, the waterway lock of Reijnen et al. (2017) consists of six modules that are reused multiple times. Similarly, the lock-bridge system described in Chapter 3 uses only a few different EFAs to model 239 components. Because of this, the use of standardized modules is investigated.

8.1.1 Method

The proposed modeling method is as follows. When a supervisor for a system has to be synthesized, firstly, the subsystems in the plant are modeled via instantiating modules from a library. The modules can be customized via parameters. For these modules, the plant model, the hybrid plant model (using hybrid automata (HA)), and the visualization of the plant are all standardized. Secondly, the requirements model is partitioned into *internal requirements*, i.e., requirements inside a module, and *external requirements*, i.e., requirements between modules. The internal requirements are also standardized. The external requirements are formalized manually. For each module, a set of external events and external states is defined that can be used in the model of the external requirements. A supervisor for the system can now be synthesized from the plant (composed out of the modules), the internal requirements, and the external requirements. The supervisor can be used to derive a supervisory controller, which can be validated via simulation-based validation, for which the hybrid plant model and visualization are used. The objects of the method are visualized in Figure 8.1. An arrow indicates that an element is contained in another element and [0..*] means zero or more times.

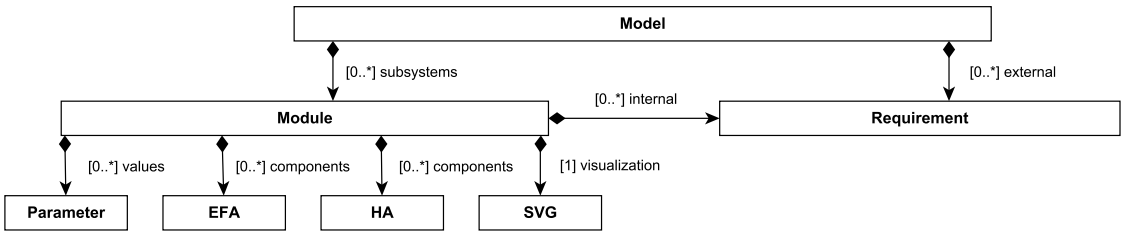


Figure 8.1: Overview of the object of the modeling method.

8.1.2 Modules

In the proposed method, a library of commonly encountered modules is created. For each module, the plant model, the internal requirements, the hybrid plant model, and the visualization are standardized. Within a module, EFAs synchronize using shared events. Modules do not share events with other modules. Similarly, variables are not shared between modules. A module consists of a set of parameters used to customize the module.

As an example for a parameter, consider a boom barrier for a movable bridge. Its customization parameter is a checkbox indicating whether barrier lights are available. In case barrier lights are present, an additional EFA is created that represents the barrier-lights actuator. A model can be created by instantiating the module for some values. The other parts of the boom-barrier module are as follows.

The plant model consists of the EFAs depicted in Figure 8.2. There is a component model for the closed sensor (EFA `S_Closed`), the open sensor (EFA `S_Open`), the actuator (EFA `A`), two models for the physical relations between `S_Closed`, `S_Open`,

and A (EFAs P_1 and P_2), and, optionally as indicated above, a model for the barrier lights (EFA L).

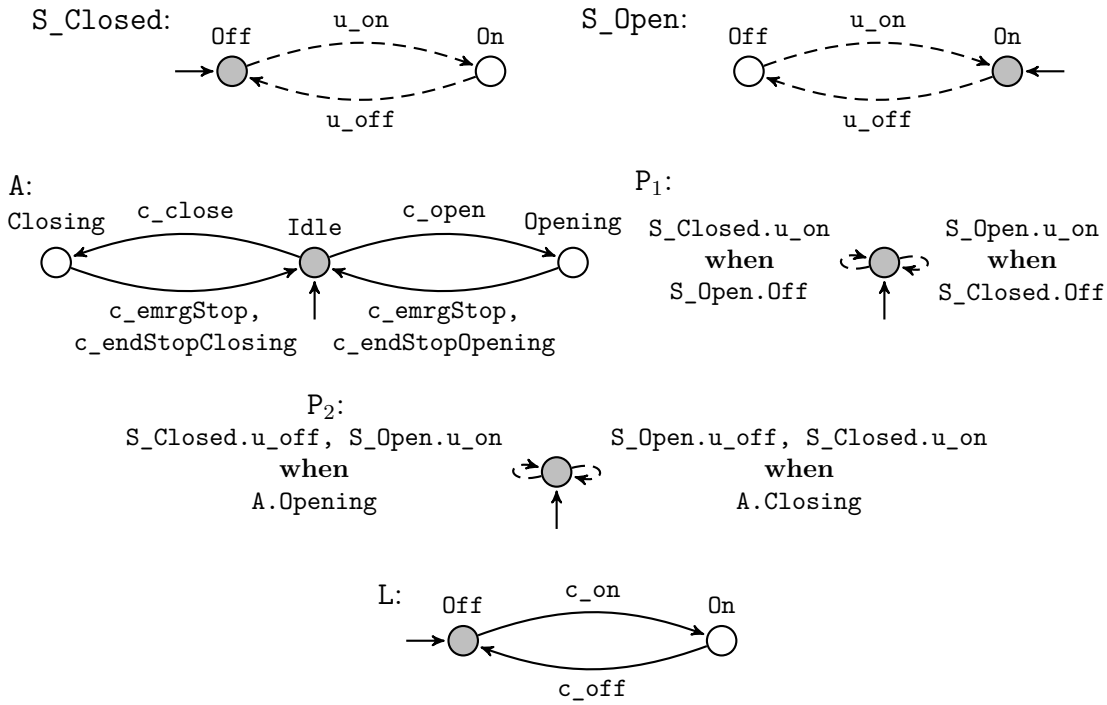


Figure 8.2: Plant model for a boom barrier.

For the correct functioning of a boom barrier, two internal requirements are defined. These requirements model that the actuator can switch off when the barrier has reached its end position, as show below.

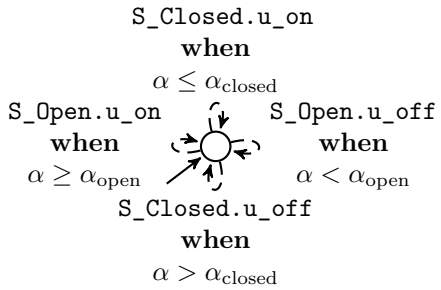
A.c_endStopClosing **needs** S_Closed.On

A.c_endStopOpening **needs** S_Open.On

For simulation, a hybrid plant model is necessary. For the boom-barrier models S_Open, S_Closed, A, and L, the HA models and the EFA models are identical. Continuous-time behavior is used to model the rotation of the boom barrier. For this, P_1 and P_2 are replaced by model P_{hyb}, as shown in Figure 8.3. Here, α represents the angle and α_{closed} and α_{open} represent the fully closed and fully open angle, respectively. A differential equation describes the angular velocity (positive if the actuator is opening and the barrier is not fully open, negative if the actuator is closing and the barrier is not fully closed, and zero otherwise).

The visualization of the boom barrier is shown in Figure 8.4. For simulation, the variables in the hybrid plant model are connected to properties in the visualization. For the boom barrier, parts of the boom barrier get hidden whenever α increases, simulating the barrier rotating upwards. The colors of the lights, depicted by the red boxes, depend on the location of EFA L. In the model, this connection is denoted as below. It means that the fill attribute of the SVG-element 'Lamp' is red when EFA L is in location on and white otherwise.

'Lamp' **attribute** fill: **if** L.On: red **else** white;



$$\frac{d\alpha}{dt} = \begin{cases} +1 & \textbf{if } \texttt{A.Opening} \wedge \ \alpha < \alpha_{\text{open}} \\ -1 & \textbf{if } \texttt{A.Closing} \wedge \ \alpha > \alpha_{\text{closed}} \\ 0 & \textbf{otherwise} \end{cases}$$

Figure 8.3: Hybrid plant model P_{hyb} for a boom barrier.

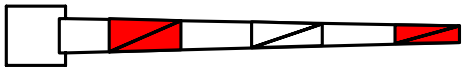


Figure 8.4: Visualization of a boom barrier.

8.1.3 External requirements

External requirements are used to model the interaction between modules. For each module, the event set is partitioned into internal and external events. The internal events are only available inside the module, whereas the external events can also be used in the event part of an external requirement. Similarly, variables are partitioned into internal and external variables. Internal variables are only available inside the module, whereas external variables can also be used in the condition part of an external requirement. In this way, modules are 'connected' via external requirements that use external events and external variables.

Consider the boom-barrier module from the previous subsection. The internal events are `A.c_endStopOpening` and `A.c_endStopClosing`, and all the sensor events. The external events are `A.c_open`, `A.c_close`, `A.c_emrgStop`, `L.c_on`, and `L.c_off`. For the requirements model, it is necessary to know whether a boom barrier is fully open, fully closed, or moving, which are represented by external variables $x_{\texttt{fo}}$ ($=$ `S_Open.On` $\wedge$ `A.Idle`), $x_{\texttt{fc}}$ ($=$ `S_Closed.On` $\wedge$ `A.Idle`), and $x_{\texttt{m}}$ ($= \neg$`A.Idle`), respectively.

8.2 Tool

A tool has been developed that is used as a front-end for the CIF toolset (van Beek et al., 2014). CIF can be used for supervisor synthesis and simulation. With the tool, CIF model code and SVG images can be generated automatically.

The tool consists of a graphical interface where a user can instantiate modules on a canvas, using the mouse cursor, and an area where the external requirements can be modeled. A screenshot of the interface, displaying the Oisterwijksebaan bridge, is shown in Figure 8.5. In the figure, label A is the plant canvas, label B is the module library, label C is where the external requirements are formalized, and label D shows the parameter for the selected component.

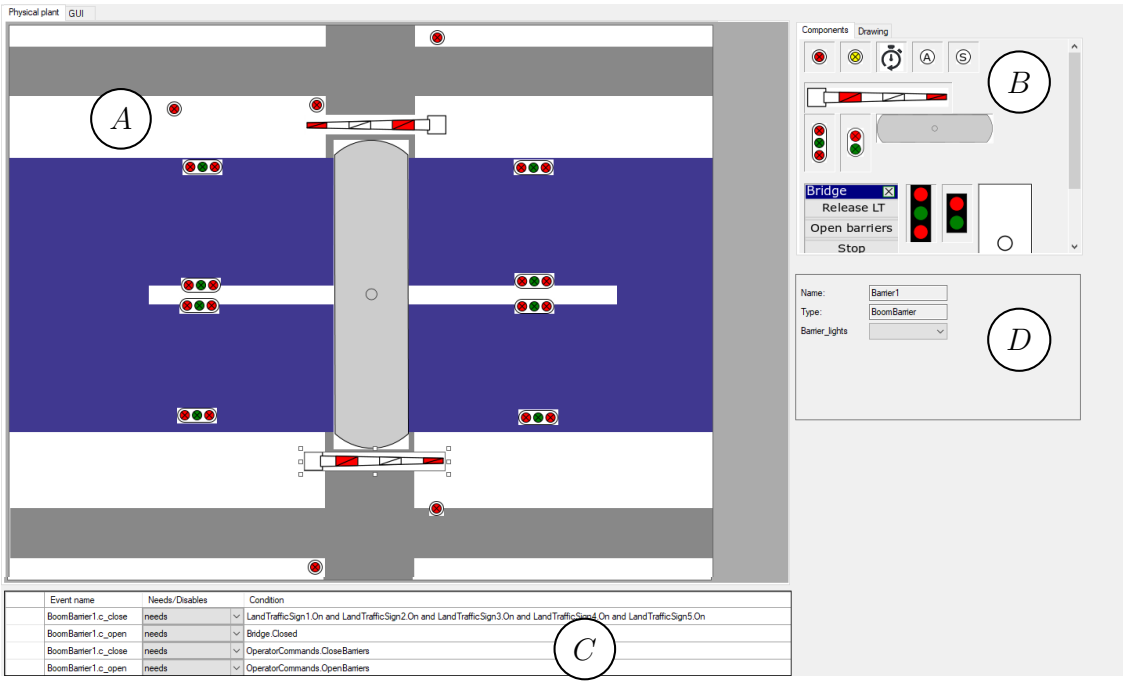


Figure 8.5: The interface of the tool, displaying a model of the Oisterwijksebaan bridge.

8.2.1 Plant canvas

On the canvas, the modules are instantiated. There are two canvases: the canvas for the modules of the GUI and the canvas for the modules of the physical part of the system. The model of the GUI and the model of the physical part are presented graphically. The modules of these two canvases constitute the plant model. A user can change between the two canvases via the top-left tabs, see Figure 8.5. Instantiated modules on the canvas can be moved, rotated, and scaled.

When an instantiated module is selected on a canvas, a menu displaying the parameters of the module is opened, denoted by label *D* in Figure 8.5. In this case, the bottom barrier is selected, this can be seen by the eight white squares that are displayed around this component. These squares can be used to scale the component. For the boom barrier component, the parameters are its name, its type, and whether barrier lights are present.

8.2.2 Module library

The module library contains the available modules. Each module has a unique icon to represent it. By hovering over a module, a short description appears. A module can be dragged from the library to the canvas to create a model from it. Then, a pop-up appears where its name can be defined. To enhance the visualization, a library of additional tools, such as background colors and labels, is available. The user can change between these libraries via tabs. For example, in the figure, these tools are used to draw the water and the roads.

8.2.3 External requirements

A table is used to display the external requirements. Each row represents a requirement. A row consists of cells for the event name and the condition. Right-clicking an instantiated module opens a window from where a new requirement for an external event can be added. By right-clicking an instantiated module when a requirement is selected, external states can be selected to use in the condition part. For example, the right-click window for a boom-barrier component is shown in Figure 8.6. It is automatically checked if the created condition is a valid Boolean expression, otherwise a warning is displayed.

See al requirements
Add requirement for c_open
Add requirement for c_close
Add requirement for c_emrgStop
Use state: Open
Use state: Closed
Use state: Moving

Figure 8.6: Right-click window of an instantiated boom barrier module.

8.3 Case study on a family of bridges

To evaluate the applicability of the method, the tool has been used to model a family of seventeen bridges located in the Wilhelmina Canal in North Brabant, the Netherlands. Three types of bridges are encountered, single leaf bascule bridges, vertical lifting bridges, and swing bridges. All of these bridges have a width between 25 m and 42 m. For defining the module library and the external requirements, the lists of inputs and outputs of the control unit, the electric drawings, and the maintenance manuals have been used. The following subsections describe the module library for these bridges and the case study.

8.3.1 Library for movable bridges

A library for commonly encountered modules in a movable bridge has been developed, with the main modules shown in Figure 8.7. The library consists of modules for physical components such as traffic lights, boom barriers, and bridge decks, and modules for GUI elements such as an operator window and a vessel traffic light. Some modules are not shown here, such as the timer, and the generic on/off actuator and on/off sensor, that can be used to represent components for which no dedicated module exists.

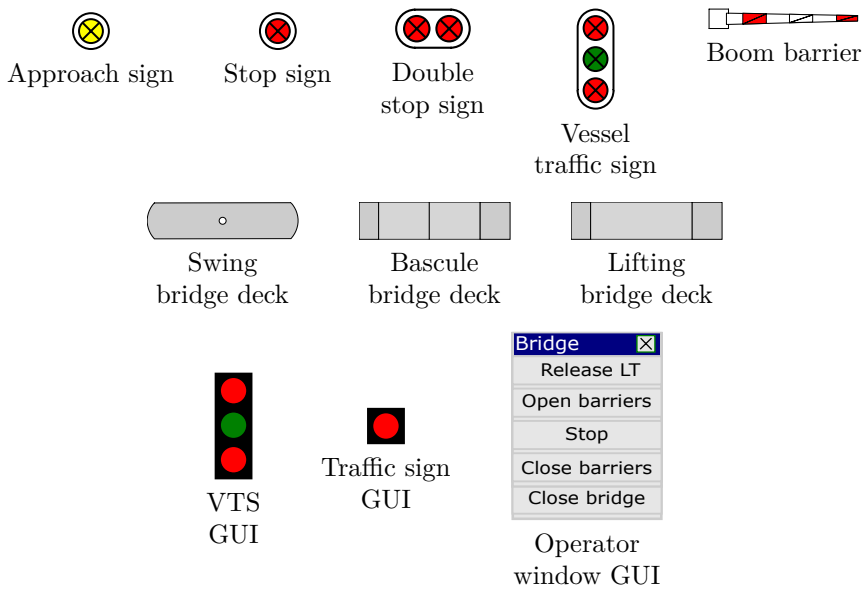


Figure 8.7: Part of the module library for movable bridges.

8.3.2 Case study

In Table 8.1, the bridge name, the bridge type, and the number of module instances are denoted. As can be seen, the majority of the modules are used in every bridge model. The difference is in the number of times each module is instantiated. As only one swing bridge has been included in the study, the swing bridge modules are used only once. Note that the modules used for bridges 4-7, 9, 15, and 16 are equal as they are very similar bridges.

For all bridges, the majority of the components can be represented by the modules in the library. However, in some cases there are components for which no module exists. In these cases, the on/off sensor and the on/off actuator are sufficient to model the additional components. Examples of such components are alarm sensors (e.g., motor overheating sensor) and auxiliary components (e.g., outdoor lighting).

The pneumatic and electric control of the bridge decks are unique for each system. For example, most of the bascule bridges use pneumatic actuators with multiple pumps, valves, and pressure controllers. Which actuator can be controlled based on which sensor is different each time. In the model, the detailed working principle is abstracted from. Instead, the events used for all bridge deck modules are: `c_open`, `c_close`, `c_stop`, `c_lock`, and `c_unlock`. These events are translated by the resource controller to the specific actuation of pumps and valves.

In all cases, it was possible to model the external requirements using the external events and variables. Typically, for each bridge around 100 external requirements are modeled. The CIF toolset was able to synthesize a supervisor for all bridges. Monolithic synthesis has been used, which takes a few seconds.

Table 8.1: Overview of the modules used for the seventeen bridges.

Bridge			Physical bridge modules								GUI bridge modules						
#	Name	Type	Approach sign	Stop sign	Double stop sign	Boom barrier	Vessel traffic sign	Bascule bridge	Lift bridge	Swing bridge	Traffic sign	Boom barrier	Vessel traffic sign	Bascule bridge	Lift bridge	Swing bridge	Operator window
1	Biest-Houtakker	Bascule		4		2	4	1			2	2	2	1			1
2	Bosscheweg	Lift		4	4	4	4		1		2	4	2		1		1
3	Enschotsestraat	Bascule		6		2	4	1			2	2	2	1			1
4	Groenewoud	Bascule		5		2	4	1			2	2	2	1			1
5	Heikantsebaan	Bascule		5		2	4	1			2	2	2	1			1
6	Heuvel	Bascule		5		2	4	1			2	2	2	1			1
7	Holenakker	Bascule		5		2	4	1			2	2	2	1			1
8	Hooijdonk	Bascule	2	5	4	2	4	1			2	2	2	1			1
9	Houtens	Bascule		5		2	4	1			2	2	2	1			1
10	Lijnsheike	Bascule		5		4	4	1			2	4	2	1			1
11	Oisterwijksebaan	Swing		5		2	8			1	2	2	4			1	1
12	Oranjelaan	Bascule	2	1	7	4	4	1			2	4	2	1			1
13	Sluis V	Bascule		3		2	6	1			2	2	3	1			1
14	Son	Lift		8	4	8	4		1		2	8	2		1		1
15	Stad van Gerwen	Bascule		4		2	4	1			2	2	2	1			1
16	Waalstraat	Bascule		4		2	4	1			2	2	2	1			1
17	Weert	Lift			4	8	4		1		4	8	2		1		1

8.4 Concluding remarks

This chapter describes a method for supervisor design for a product platform. A tool has been developed for the design of supervisors for a family of movable bridges. In a case study with seventeen bridges, it has been shown that with only a small set of modules, the necessary models for synthesis and simulation can be generated automatically. After defining requirements between the instantiated modules, a supervisor has been synthesized. The proof-of-concept delivered for a family of movable bridges shows that this method is suitable for designing supervisors for a family of similar systems.

In further research, it should be investigated how external requirement models can be reused. In this case study, the external requirements have been modeled separately each time. It is expected that also a library for external requirements can be composed, as for the case study many requirements for the bridges are similar. Furthermore, we are interested in using the tool to develop supervisors for other infrastructural systems, such as waterway locks.

Chapter 9

Concluding remarks

This thesis aimed to investigate the applicability of supervisor synthesis to the design of supervisory controllers for infrastructural systems, to increase quality and evolvability. Based on the results presented in this thesis, it can be concluded that the synthesis-based engineering method is particularly suitable for the design, validation, and implementation of supervisory controllers for infrastructural systems. This was especially demonstrated by Chapter 7, where the method is illustrated starting from textual specifications all the way to the implementation on a real system. Specifically, an improvement in the following three aspects has been observed:

1. The quality of the requirements.
2. The quality of the supervisory controller.
3. The reusability and evolvability of the models.

First, the synthesis-based engineering method provides a structured and systematic way for the specification and design of the system's behavior. The required models provide more consistency and allow for less ambiguity than documents written in natural language, which increases their quality. Moreover, supervisor synthesis helps in identifying missing and incorrect requirements, as shown in Chapter 3 where a missing requirement for a waterway lock was found. Even in case synthesis is not used, the models still contribute to a more consistent design, as demonstrated in Goorden (2020) and Moormann (2020), where component and requirement models are used as a design specification for waterway locks and road tunnels, respectively.

Second, the use of synthesis guarantees that the requirements are always met by the supervisor and the supervisory controller derived from that. As a result, the supervisory controller does not have to be verified against these requirements. Instead, the focus lies on assuring that the formalized requirements are complete and correct. Simulation-based validation and HIL simulation are powerful tools for this purpose. The use of synthesis and simulation increases the quality and the confidence in the correctness of the supervisory controller before implementation in the real system. For the supervisory controller discussed in Chapter 7, all errors related to its behavior were found during simulation and no errors were found during testing on the real system.

Third, the component-based modeling method in combination with standardized modules allows for easy reuse and evolvability of models. The development time of the supervisory controllers decreased significantly during the project, from a few months for the waterway lock described in Reijnen et al. (2017), to a few hours for the bridges described in Reijnen et al. (2020b). Furthermore, the developed modules have also been reused to model other systems, like road tunnels, roadside systems, and manufacturing lines.

The research questions defined in Section 1.5 are answered below.

Research question 1

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

Chapter 3 presents a component-based modeling method for supervisor synthesis. Here, all actuators, sensors, and commands are modeled as small, loosely-coupled component models. The interactions between the components are formalized as physical relation models. Various case studies show that this method is very suitable for modeling waterway locks and movable bridges. Additionally, it is shown how the control requirements for locks and bridges can be formalized with event conditions. For this, we identified textual requirements that can straightforwardly be transformed into a model. When component-based modeling is used, the formalized requirements relate closely to the specifications control engineers are acquainted with in practice, which increases the accessibility of the method. Chapter 7 addresses the modeling and design of fault-tolerant supervisory controllers and fault diagnosers. Fault diagnosers are used as additional inputs to the supervisory controller, and their identifications are used in the requirements model.

Various case studies illustrate this modeling method. In Chapter 3, a lock-bridge combination model is shown, consisting of 80 actuators and 96 sensors, and that has to respond to 63 operator commands. The waterway lock and the movable bridge described in Chapters 6 and 7, respectively, are also modeled using this method. The supervisory controller for the bridge was also implemented on the hardware. The method is also suitable for modeling other infrastructural systems, such as road tunnels (Moormann et al., 2020a) and roadside systems (de Vos, 2018). In Reijnen et al. (2018a), it is described how a supervisory controller for a manufacturing line is modeled, synthesized and implemented on hardware, using the same method.

Research question 2

> *Which steps are necessary to derive a supervisory controller, and subsequently, implementation code, from a synthesized supervisor model?*

In Chapter 4, an overview of the steps to go from a (synthesized) supervisor to a controller implementation is given. It is based on the method from Malik (2003) to

verify that any supervisory controller derived from the supervisor is nonblocking. For this, a supervisor needs to have finite response, be confluent, and be nonblocking under control. This thesis proposes sufficient conditions that can be used to check these properties for systems modeled as EFAs.

From a supervisory-controller model, PLC code can be generated, as discussed in Chapter 4. The required cycle time of the PLC can be minimized by optimizing the event order and the EFA order. In that way, situations related to inexact synchronization and delay insensitivity are mitigated. Furthermore, the original model structure is preserved.

In Chapter 7, it is described how a supervisory controller for a swing bridge has been derived from a synthesized supervisor, based on the method described in Chapter 4. Subsequently, the derived supervisory controller has been used to generate implementation code for a PLC. The supervisory controller has successfully been used to control the real bridge.

Research question 3

> *Which additional steps have to be performed to use a synthesized supervisor as a safety PLC controller?*

For safety PLCs, two supervisors are necessary, one for the regular part and one for the safety part. In Chapter 5, a method is described that allows a supervisor to be split for this purpose. This splitting method is based on the partitioning of the requirements. From this partitioning, it is derived which sensors and actuators belong to the safety part and how the supervisor has to be split. It is shown that the behavior of the supervisor before the split is equal to the behavior of the supervisors after the split. This assures that the properties guaranteed by synthesis still hold afterwards.

The Oisterwijksebaan bridge, described in Chapter 7, has been used to demonstrate the applicability of this method. First, it is shown that the partitioning of the sensors and actuators is similar to the partitioning made by experts. Second, it is shown that for both parts PLC code can be generated and that the generated code can be used to control the real bridge.

Research question 4

> *How can a supervisory controller for a real infrastructural system be synthesized, implemented, and tested?*

In Chapter 7, it is described how a fault-tolerant supervisory controller for a real swing bridge has been synthesized, implemented, and tested. For this, the modeling method of Chapter 3 and the supervisory-controller derivation method of Chapter 4 are used. Simulation-based validation with factory acceptance tests, defined by RWS, show that the behavior is as intended. To bridge the gap between simulation and implementation, a hardware-in-the-loop set-up has been developed. With that set-up,

the behavior of the supervisory controller implemented on the PLC was validated before it was connected to the real system. This set-up and how it has been integrated in the engineering method are described in Chapter 6. After HIL simulation, the supervisory controller has been implemented on the real system. To validate the behavior of the supervisory controller on the real system, site acceptance tests with predefined scenarios have been performed by RWS bridge operators. These tests show that the supervisory controller behaves as specified in the scenarios, both in nominal behavior and after the occurrence of faults.

Research question 5

> *In what way can the similarities between similar systems be exploited to efficiently develop the plant and the requirements models?*

Chapter 8 describes a graphical method for the development of supervisory controllers for a product platform. A product platform is a set of common components, modules, or parts from which a stream of derivative products can be efficiently created. In this method, a library of standardized modules for commonly encountered components is created (according to the modeling method described in Chapter 3). Modules in this library are used to quickly compose a plant model. For the requirements model, an interface consisting of events and states is defined for each module. The desired interaction between modules is defined using these events and states.

As a proof of concept, a prototype tool has been developed that implements this method for a family of movable bridges. Using this tool, the plant and the requirements models can be developed graphically. A case study with a family of seventeen bridges shows that with a small set of modules, the necessary models for synthesis and simulation can be generated automatically.

9.1 Future work

This thesis shows how modeling, synthesis, validation, implementation, and testing can be combined into a design process of supervisory controllers for infrastructural systems. There are various ways to further build on the results of this research project. Below, two possible extensions are described.

Improving synthesis performance

For the case studies described in this thesis, it was always possible to synthesize a supervisor. The time the synthesis procedure takes depends on the application. For example, it takes around 3 minutes to synthesize the supervisor for the Oisterwijksebaan bridge described in Chapter 7 and around 35 minutes to synthesize the supervisor for the Algera complex described in Chapter 3. Similarly, the computer memory required for the synthesis procedure differs between applications. It is expected that for larger systems, monolithic synthesis becomes infeasible due to the computation time being too long and the required memory being too high. In Goorden et al. (2020b), it

has been shown that in some cases only a part of the plant model and a part of the requirements model are needed for synthesis. In addition, in Moormann et al. (2020b), it has been shown that if parts of the system are symmetrical (such as the upstream side and downstream side of a waterway lock), only one part is needed for synthesis. It should be investigated whether these techniques can be used to reduce the computation time and the required memory for waterway locks and movable bridges. Another factor that influences the computation time and the required memory is the ordering of variables in the binary decision diagram. Thuijsman et al. (2019) shows that for different orderings for the BDD-variables of the waterway lock described in Reijnen et al. (2017), the synthesis time ranges from a few seconds to a few hours. Research is needed to consistently choose a favorable ordering.

Data logging and reconstruction of behavior

In case of malfunctions or accidents related to infrastructural systems, it is useful to reconstruct and analyze the behavior that led to such an undesired situation. Current practice consists of monitoring and storing the actuator signals, the sensor signals, and the GUI signals, and plotting them in a time series chart. Reconstruction and analysis using a time series chart is laborious and difficult to perform for engineers not familiar with the system. When a model of the plant is available, it can be used for several purposes. Firstly, it can be used to visualize the logged data via simulation, making the reconstruction and analysis more intuitive and accessible. Secondly, as shown in Roth et al. (2011), the logged behavior can be compared with the expected nominal behavior defined in the plant model. In that way, faulty behavior can be identified automatically. Thirdly, if a model of the supervisory controller is also available, it can be used to determine whether the implemented supervisory controller behaves as intended. Preliminary results for the applicability of these techniques to infrastructural systems are presented in Reijnen et al. (2020c). More research is needed to integrate these techniques into the incident analysis process.

Bibliography

Åkesson, B., Hooman, J., Sleuters, J., and Yankov, A. (2020). "Reducing design time and promoting evolvability using Domain-Specific Languages in an industrial context". In: *Model Management and Analytics for Large Scale Systems*. Elsevier, pp. 245–272.

Alenljung, T., Sköldstam, M., Lennartson, B., and Åkesson, K. (2007). "PLC-based implementation of process observation and fault detection for discrete event systems". In: *Proceedings of the 3rd Conference on Automation Science and Engineering*. IEEE, pp. 207–212.

Atampore, F., Dingel, J., and Rudie, K. (2019). "A controller synthesis framework for automated service composition". In: *Discrete Event Dynamic Systems* vol. 29, no. 3, pp. 297–365.

Auer, A., Dingel, J., and Rudie, K. (2014). "Concurrency control generation for dynamic threads using discrete-event systems". In: *Science of Computer Programming* vol. 82, pp. 22–43.

Baeten, J. C. M., van de Mortel-Fronczak, J. M., and Rooda, J. E. (2016). "Integration of supervisory control synthesis in model-based systems engineering". In: *Complex Systems*. Studies in Systems, Decision and Control. Springer, pp. 39–58.

Balemi, S., Hoffmann, G. J., Gyugyi, P., Wong-Toi, H., and Franklin, G. F. (1993). "Supervisory control of a rapid thermal multiprocessor". In: *IEEE Transactions on Automatic Control* vol. 38, no. 7, pp. 1040–1059.

Basile, F. and Chiacchio, P. (2007). "On the implementation of supervised control of discrete event systems". In: *IEEE Transactions on Control Systems Technology* vol. 15, no. 4, pp. 725–739.

van Beek, D. A., Fokkink, W. J., Hendriks, D., Hofkamp, A. T., Markovski, J., van de Mortel-Fronczak, J. M., and Reniers, M. A. (2014). "CIF 3: Model-based engineering of supervisory controllers". In: *Proceedings of the 20th Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, pp. 575–580.

ter Beek, M. H., Reniers, M. A., and de Vink, E. P. (2016). "Supervisory controller synthesis for product lines using CIF 3". In: *Proceedings of the 7th Symposium on Leveraging Applications of Formal Methods*. Springer, pp. 856–873.

Blanke, M., Kinnaert, M., Lunze, J., Staroswiecki, M., and Schröder, J. (2016). *Diagnosis and Fault-Tolerant Control*. 3rd. Springer-Verlag, Berlin Heidelberg.

von Bochmann, G., Hilscher, M., Linker, S., and Olderog, E.-R. (2017). "Synthesizing and verifying controllers for multi-lane traffic maneuvers". In: *Formal Aspects of Computing* vol. 29, no. 4, pp. 583–600.

Brandin, B. A. (1996). "The real-time supervisory control of an experimental manufacturing cell". In: *IEEE Transactions on Robotics and Automation* vol. 12, no. 1, pp. 1–14.

Bullock, D., Johnson, B., Wells, R. B., Kyte, M., and Li, Z. (2004). "Hardware-in-the-loop simulation". In: *Transportation Research Part C: Emerging Technologies* vol. 12, no. 1, pp. 73–89.

Cai, K. and Wonham, W. M. (2010). "Supervisor localization: A top-down approach to distributed control of discrete-event systems". In: *IEEE Transactions on Automatic Control* vol. 55, no. 3, pp. 605–618.

Cassandras, C. G. and Lafortune, S. (2009). *Introduction to Discrete Event Systems*. Springer Science + Business Media.

Chandra, V. and Kumar, R. (2002). "A event occurrence rules based compact modeling formalism for a class of discrete event systems". In: *Mathematical and Computer Modelling of Dynamical Systems* vol. 8, no. 1, pp. 49–73.

Chandra, V., Huang, Z., and Kumar, R. (2003). "Automated control synthesis for an assembly line using discrete event system control theory". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* vol. 33, no. 2, pp. 284–289.

Charbonnier, F., Alla, H., and David, R. (1995). "The supervised control of discrete event dynamic systems: A new approach". In: *Procedings of the 34th Conference on Decision and Control*. IEEE, pp. 913–920.

Chen, Y.-L. and Lin, F. (2000). "Modeling of discrete event systems using finite state machines with parameters". In: *Proceedings of the 2000 Conference on Control Applications*. IEEE, pp. 941–946.

Cheng, K.-T. and Krishnakumar, A. S. (1996). "Automatic generation of functional vectors using the extended finite state machine model". In: *ACM Transactions on Design Automation of Electronic Systems* vol. 1, no. 1, pp. 57–79.

Crnkovic, I. (2001). "Component-based software engineering – New challenges in software development". In: *Software Focus* vol. 2, no. 4, pp. 127–133.

Dai, W., Zhou, P., Zhao, D., Lu, S., and Chai, T. (2016). "Hardware-in-the-loop simulation platform for supervisory control of mineral grinding process". In: *Powder Technology* vol. 288, pp. 422–434.

Darvas, D., Majzik, I., and Viñuela, E. B. (2016). "Formal verification of safety PLC based control software". In: *Proceedings of the 12th Conference on Integrated Formal Methods*. Springer, pp. 508–522.

Dietrich, P., Malik, R., Wonham, W. M., and Brandin, B. A. (2002). "Implementation considerations in supervisory control". In: *Synthesis and Control of Discrete Event Systems*. Springer, pp. 185–201.

Dijsselbloem, J. R. V. A., van Asselt, M. B. A., Zouridis, S., and Verheij, C. A. J. F. (2019). *Veiligheid van op afstand bediende bruggen*. Tech. rep. De Onderzoeksraad voor Veiligheid.

Diogo, R. A., Santos, E. A. P., Vieira, A. D., Loures, E. d. F. R., and Busetti, M. A. (2012). "A computational control implementation environment for automated manufacturing systems". In: *International Journal of Production Research* vol. 50, no. 22, pp. 6272–6287.

Ekberg, G. and Krogh, B. H. (2006). "Programming discrete control systems using state machine templates". In: *Proceedings of the 8th Workshop on Discrete Event Systems*. IEEE, pp. 194–200.

Eppinger, S. D. and Browning, T. R. (2012). *Design Structure Matrix Methods and Applications*. MIT press.

European Commission (2006). "Machinery Directive". In: *Official Journal of the European Union* vol. L 157, pp. 24–86.

Fabian, M., Fei, Z., Miremadi, S., Lennartson, B., and Åkesson, K. (2014). "Supervisory control of manufacturing systems using extended finite automata". In: *Formal Methods in Manufacturing*, pp. 295–314.

Fabian, M. and Hellgren, A. (1998). "PLC-based implementation of supervisory control for discrete event systems". In: *Proceedings of the 37th Conference on Decision and Control*. Vol. 3. IEEE, pp. 3305–3310.

Feng, L. and Wonham, W. M. (2006). "TCT: A computation tool for supervisory control synthesis". In: *Proceedings of the 8th Workshop on Discrete Event Systems*. IEEE, pp. 388–389.

Forschelen, S. T. J., van de Mortel-Fronczak, J. M., Su, R., and Rooda, J. E. (2012). "Application of supervisory control theory to theme park vehicles". In: *Discrete Event Dynamic Systems* vol. 22, no. 4, pp. 511–540.

Frakes, W. B. and Kang, K. (2005). "Software reuse research: Status and future". In: *IEEE Transactions on Software Engineering* vol. 31, no. 7, pp. 529–536.

Frey, G. and Litz, L. (2000). "Formal methods in PLC programming". In: *Proceedings of the 2000 Conference on Systems, Man and Cybernetics*. IEEE, pp. 2431–2436.

Göbe, F., Ney, O., and Kowalewski, S. (2016). "Reusability and modularity of safety specifications for supervisory control". In: *Proceedings of the 21st Conference on Emerging Technologies and Factory Automation*. IEEE, pp. 1–8.

Goorden, M. A. (2019). "Supervisory control synthesis for large-scale infrastructural systems". PhD thesis. Eindhoven University of Technology.

Goorden, M. A. (2020). *Generieke beschrijving van de logische besturing van een sluis*. Tech. rep. Rijkswaterstaat.

Goorden, M. A. and Fabian, M. (2019). "No synthesis needed, we are alright already". In: *Proceedings of the 15th Conference on Automation Science and Engineering*. IEEE, pp. 195–202.

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. (2020a). "The road ahead for supervisor synthesis". In: *Proceedings of the Symposium on Dependable Software Engineering*, In press.

Goorden, M. A., van de Mortel-Fronczak, J. M., Reniers, M. A., Fabian, M., Fokkink, W. J., and Rooda, J. E. (2020b). "Model properties for efficient synthesis of nonblocking modular supervisors". In: *arXiv preprint arXiv:2007.05795*.

Gössler, G. and Sifakis, J. (2005). "Composition for component-based modeling". In: *Science of Computer Programming* vol. 55, no. 1-3, pp. 161–183.

Grigorov, L., Butler, B. E., Cury, J. E., and Rudie, K. (2011). "Conceptual design of discrete-event systems using templates". In: *Discrete Event Dynamic Systems* vol. 21, no. 2, pp. 257–303.

Grigorov, L. and Rudie, K. (2010). "Techniques for the parametrization of discrete-event system templates". In: *IFAC Proceedings Volumes* vol. 43, no. 12, pp. 370–375.

Guillet, S., Bouchard, B., and Bouzouane, A. (2014). "Designing smart homes dedicated to disabled people using modular discrete controller synthesis". In: *IFAC Proceedings Volumes* vol. 47, no. 2, pp. 54–59.

Hasdemir, İ. T., Kurtulan, S., and Gören, L. (2008). "An implementation methodology for supervisory control theory". In: *The International Journal of Advanced Manufacturing Technology* vol. 36, no. 3-4, pp. 373–385.

van der Heide, J. (2019). *Basisspecificatie beweegbare brug, versie 4.5.0*. Tech. rep. Rijkswaterstaat.

Hellgren, A., Fabian, M., and Lennartson, B. (2001). "Modular implementation of discrete event systems as sequential function charts applied to an assembly cell". In: *Proceedings of the 2001 Conference on Control Applications*. IEEE, pp. 453–458.

Henzinger, T. A. (2000). "The theory of hybrid automata". In: *Verification of Digital and Hybrid Systems*. Springer, pp. 265–292.

Huang, J. and Kumar, R. (2008). "Directed control of discrete event systems for safety and nonblocking". In: *IEEE Transactions on Automation Science and Engineering* vol. 5, no. 4, pp. 620–629.

Huang, Y., Seck, M. D., and Verbraeck, A. (2015). "Component-based light-rail modeling in discrete event systems specification". In: *Simulation* vol. 91, no. 12, pp. 1027–1051.

International Electrotechnical Commission (2013). *IEC 61131-3: Programmable Controllers – Part 3: Programming Languages (3rd)*. Standard.

Jiang, J. and Yu, X. (2012). "Fault-tolerant control systems: A comparative study between active and passive approaches". In: *Annual Reviews in Control* vol. 36, no. 1, pp. 60–72.

Kehat, E. and Shacham, M. (1973). "Chemical process simulation programs-2". In: *Process Technology International* vol. 18, no. 3, pp. 115–118.

Kim, S., Park, J., and Leachman, R. C. (2001). "A supervisory control approach for execution control of an FMC". In: *International Journal of Flexible Manufacturing Systems* vol. 13, no. 1, pp. 5–31.

Korssen, T., Dolk, V. S., van de Mortel-Fronczak, J. M., Reniers, M. A., and Heemels, W. P. M. H. (2018). "Systematic model-based design and implementation of supervisors for advanced driver assistance systems". In: *IEEE Transactions on Intelligent Transportation Systems* vol. 19, no. 2, pp. 533–544.

Kovács, G. and Piétrac, L. (2009). "Multi-face modeling for rapid prototyping of discrete event control systems". In: *Proceedings of the 10th European Control Conference*. IEEE, pp. 1463–1468.

Kovács, G., Piétrac, L., and Bálint, K. (2012). "A component-based approach for supervisory control". In: *Proceedings of the 20th Mediterranean Conference on Control and Automation*. IEEE, pp. 800–805.

Kusiak, A. and Wang, J. (1993). "Efficient organizing of design activities". In: *International Journal of Production Research* vol. 31, no. 4, pp. 753–769.

Laing, C., David, P., Blanco, E., and Dorel, X. (2020). "Questioning integration of verification in model-based systems engineering: An industrial perspective". In: *Computers in Industry* vol. 114.

Lauzon, S. C., Ma, A. K. L., Mills, J. K., and Benhabib, B. (1996). "Application of discrete-event-system theory to flexible manufacturing". In: *IEEE Control Systems* vol. 16, no. 1, pp. 41–48.

Leal, A. B., da Cruz, D. L. L., and da Silva Hounsell, M. (2012). "PLC-based implementation of local modular supervisory control for manufacturing systems". In: *Manufacturing System*. InTech, pp. 159–182.

Leduc, R. J. and Wonham, W. M. (1995). "Discrete event systems modeling and control of a manufacturing testbed". In: *Proceedings of the 1995 Canadian Conference on Electrical and Computer Engineering*. Vol. 2. IEEE, pp. 793–796.

Li, W., Joós, G., and Bélanger, J. (2010). "Real-time simulation of a wind turbine generator coupled with a battery supercapacitor energy storage system." In: *IEEE Transaction on Industrial Electronics* vol. 57, no. 4, pp. 1137–1145.

Liao, H., Wang, Y., Stanley, J., Lafortune, S., Reveliotis, S., Kelly, T., and Mahlke, S. (2013). "Eliminating concurrency bugs in multithreaded software: A new approach based on discrete-event control". In: *IEEE Transactions on Control Systems Technology* vol. 21, no. 6, pp. 2067–2082.

Liebel, G., Marko, N., Tichy, M., Leitner, A., and Hansson, J. (2018). "Model-based engineering in the embedded systems domain: An industrial survey on the state-of-practice". In: *Software and Systems Modeling* vol. 17, no. 1, pp. 91–113.

Ljungkrantz, O., Åkesson, K., Richardsson, J., and Andersson, K. (2007). "Implementing a control system framework for automatic generation of manufacturing cell controllers". In: *Proceedings of the 24th Conference on Robotics and Automation*. IEEE, pp. 674–679.

Lopes, Y. K., Trenkwalder, S. M., Leal, A. B., Dodd, T. J., and Groß, R. (2016). "Supervisory control theory applied to swarm robotics". In: *Swarm Intelligence* vol. 10, no. 1, pp. 65–97.

Ma, C. and Wonham, W. M. (2006). "Nonblocking supervisory control of state tree structures". In: *IEEE Transactions on Automatic Control* vol. 51, no. 5, pp. 782–793.

Malik, P. (2002). "Generating controllers from discrete-event models". In: *Proceedings of the Summer School on Modelling and Verification of Parallel Processes*, pp. 337–342.

Malik, P. (2003). "From supervisory control to nonblocking controllers for discrete event systems". PhD thesis. Universität Kaiserslautern.

Malik, P. and Malik, R. (2006). "Modular control-loop detection". In: *Proceedings of the 8th Workshop on Discrete Event Systems*. IEEE, pp. 119–124.

Malik, R., Åkesson, K., Flordal, H., and Fabian, M. (2017). "Supremica–An efficient tool for large-scale discrete event systems". In: *IFAC-PapersOnLine* vol. 50, no. 1, pp. 5794–5799.

Malik, R., Fabian, M., and Åkesson, K. (2011). "Modelling large-scale discrete-event systems using modules, aliases, and extended finite-state automata". In: *IFAC Proceedings Volumes* vol. 44, no. 1, pp. 7000–7005.

Markovski, J., van Beek, D. A., Theunissen, R. J. M., Jacobs, K. G. M., and Rooda, J. E. (2010). "A state-based framework for supervisory control synthesis and verification". In: *Proceedings of the 49th Conference on Decision and Control*. IEEE, pp. 3481–3486.

Mellegård, N., Ferwerda, A., Lind, K., Heldal, R., and Chaudron, M. R. (2015). "Impact of introducing domain-specific modelling in software maintenance: An industrial case study". In: *IEEE Transactions on Software Engineering* vol. 42, no. 3, pp. 245–260.

Meyer, M. H. and Lehnerd, A. P. (1997). *The Power of Product Platforms: Building Value and Cost Leadership*. Free Press.

Miremadi, S., Åkesson, K., and Lennartson, B. (2011). "Symbolic computation of reduced guards in supervisory control". In: *IEEE Transactions on Automation Science and Engineering* vol. 8, no. 4, pp. 754–765.

Miremadi, S., Åkesson, K., Lennartson, B., and Fabian, M. (2010). "Supervisor computation and representation: A case study". In: *IFAC Proceedings Volumes* vol. 43, no. 12, pp. 275–280.

Miremadi, S., Lennartson, B., and Åkesson, K. (2012). "A BDD-based approach for modeling plant and supervisor by extended finite automata". In: *IEEE Transactions on Control Systems Technology* vol. 20, no. 6, pp. 1421–1435.

Moor, T. (2016). "A discussion of fault-tolerant supervisory control in terms of formal languages". In: *Annual Reviews in Control* vol. 41, pp. 159–169.

Moor, T., Schmidt, K., and Perk, S. (2008). "libFaudes–An open source C++ library for discrete event systems". In: *Proceedings of the 9th Workshop on Discrete Event Systems*. IEEE, pp. 125–130.

Moor, T., Schmidt, K., and Perk, S. (2010). "Applied supervisory control for a flexible manufacturing system". In: *IFAC Proceedings Volumes* vol. 43, no. 12, pp. 253–258.

Moormann, L. (2020). *Generieke beschrijving van de logische besturing van een tweebuis tunnel met midden-tunnel-kanaal*. Tech. rep. Eindhoven University of Technology.

Moormann, L., Maessen, P., Goorden, M. A., van de Mortel-Fronczak, J. M., and Rooda, J. E. (2020a). "Design of a tunnel supervisory controller using synthesis-based engineering". In: *Proceedings of World Tunnel Congress 2020*. ITA-AITES, pp. 573–578.

Moormann, L., van de Mortel-Fronczak, J. M., Fokkink, W. J., and Rooda, J. E. (2020b). "Exploiting symmetry in dependency graphs for model reduction in supervisor synthesis". In: *Proceedings of the 16th Conference on Automation Science and Engineering*. IEEE, pp. 660–667.

Morgenstern, A. and Schneider, K. (2007). "Synthesizing deterministic controllers in supervisory control". In: *Informatics in Control, Automation and Robotics II*. Springer, pp. 95–102.

Nieman, W. (2016). *Basisspecificatie schutsluis, versie 4.1.1*. Tech. rep. Rijkswaterstaat.

van Nieuwenhuizen, C. (2019). *Bediening sluizen en bruggen Afsluitdijk*. Tweede Kamer der Staten-Generaal.

Nourelfath, M. and Niel, E. (2004). "Modular supervisory control of an experimental automated manufacturing system". In: *Control Engineering Practice* vol. 12, no. 2, pp. 205–216.

Ouedraogo, L., Kumar, R., Malik, R., and Åkesson, K. (2011). "Nonblocking and safe control of discrete-event systems modeled as extended finite automata". In: *IEEE Transactions on Automation Science and Engineering* vol. 8, no. 3, pp. 560–569.

Park, S.-J. and Cho, K.-H. (2006). "Delay-robust supervisory control of discrete-event systems with bounded communication delays". In: *IEEE Transactions on Automatic Control* vol. 51, no. 5, pp. 911–915.

Pena, P. N., Costa, T. A., Silva, R. S., and Takahashi, R. H. (2016). "Control of flexible manufacturing systems under model uncertainty using supervisory control theory and evolutionary computation schedule synthesis". In: *Information Sciences* vol. 329, pp. 491–502.

Pétin, J.-F., Gouyon, D., and Morel, G. (2007). "Supervisory synthesis for product-driven automation and its application to a flexible assembly cell". In: *Control Engineering Practice* vol. 15, no. 5, pp. 595–614.

Pichard, R., Philippot, A., Saddem, R., and Riera, B. (2018). "Safety of manufacturing systems controllers by logical constraints with safety filter". In: *IEEE Transactions on Control Systems Technology*, pp. 1659–1667.

PLCopen (2018). *Safety software technical specifications – Part 1: concepts and function blocks (2nd)*. Standard.

Prenzel, L. and Provost, J. (2018). "PLC implementation of symbolic, modular supervisory controllers". In: *IFAC-PapersOnLine* vol. 51, no. 7, pp. 304–309.

de Queiroz, M. H. and Cury, J. E. R. (2000). "Modular control of composed systems". In: *Proceedings of the 2000 American Control Conference*. Vol. 6. IEEE, pp. 4051–4055.

de Queiroz, M. H. and Cury, J. E. R. (2002). "Synthesis and implementation of local modular supervisory control for a manufacturing cell". In: *Proceedings of the 6th Workshop on Discrete Event Systems*. IEEE, pp. 377–382.

Ramadge, P. J. and Wonham, W. M. (1987). "Supervisory control of a class of discrete event processes". In: *SIAM Journal on Control and Optimization* vol. 25, no. 1, pp. 206–230.

Ramos, A. L., Ferreira, J. V., and Barceló, J. (2011). "Model-based systems engineering: An emerging approach for modern systems". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* vol. 42, no. 1, pp. 101–111.

Rashidinejad, A., Reniers, M. A., and Fabian, M. (2019). "Supervisory control of discrete-event systems in an asynchronous setting". In: *Proceedings of the 15th Conference on Automation Science and Engineering*. IEEE, pp. 494–501.

Rawlings, B. C., Christenson, B., Wassick, J. M., and Ydstie, B. E. (2014). "Supervisor synthesis to satisfy safety and reachability requirements in chemical process control". In: *IFAC Proceedings Volumes* vol. 47, no. 2, pp. 195–200.

Reijnen, F. F. H., Goorden, M. A., van de Mortel-Fronczak, J. M., Reniers, M. A., and Rooda, J. E. (2018a). "Application of dependency structure matrices and multilevel synthesis to a production line". In: *Proceedings of the 2nd Conference on Control Technology and Applications*. IEEE, pp. 458–464.

Reijnen, F. F. H., Goorden, M. A., van de Mortel-Fronczak, J. M., and Rooda, J. E. (2017). "Supervisory control synthesis for a waterway lock". In: *Proceedings of the 1st Conference on Control Technology and Applications*. IEEE, pp. 1562–1568.

Reijnen, F. F. H., Goorden, M. A., van de Mortel-Fronczak, J. M., and Rooda, J. E. (2020a). "Modeling for supervisor synthesis – A lock-bridge combination case study". In: *Discrete Event Dynamic Systems* vol. 30, no. 3, pp. 499–532.

Reijnen, F. F. H., Hofkamp, A. T., Reniers, M. A., van de Mortel-Fronczak, J. M., and Rooda, J. E. (2019a). "Finite response and confluence of state-based supervisory controllers". In: *Proceedings of the 15th Conference on Automation Science and Engineering*. IEEE, pp. 509–516.

Reijnen, F. F. H., van de Mortel-Fronczak, J. M., Reniers, M. A., and Rooda, J. E. (2020b). "Design of a supervisor platform for movable bridges". In: *Proceedings of the 16th Conference on Automation Science and Engineering*. IEEE, pp. 1298–1304.

Reijnen, F. F. H., van de Mortel-Fronczak, J. M., and Rooda, J. E. (2020c). "Data logging and reconstruction of discrete-event system behavior". In: *Proceedings of the 16th Conference on Control, Automation, Robotics and Vision*. IEEE, In press.

Reijnen, F. F. H., Reniers, M. A., van de Mortel-Fronczak, J. M., and Rooda, J. E. (2018b). "Structured synthesis of fault-tolerant supervisory controllers". In: *IFAC-PapersOnLine* vol. 51, no. 24, pp. 894–901.

Reijnen, F. F. H., Erens, T. R., van de Mortel-Fronczak, J. M., and Rooda, J. E. (2020d). "Supervisory control synthesis for safety PLCs". In: *Proceedings of the 15th Workshop on Discrete Event Systems*. IFAC, In press.

Reijnen, F. F. H., Leliveld, E.-B. M. L., van de Mortel-Fronczak, J. M., van Dinther, M. J. T., Rooda, J. E., and Fokkink, W. J. (2020e). "A synthesized fault-tolerant supervisory controller for a swing bridge". In: Submitted.

Reijnen, F. F. H., Verbakel, J. J., van de Mortel-Fronczak, J. M., and Rooda, J. E. (2019b). "Hardware-in-the-loop set-up for supervisory controllers with an application: the Prinses Marijke complex". In: *Proceedings of the 3rd Conference on Control Technology and Applications*. IEEE, pp. 843–850.

Rijkswaterstaat (2019). *Landelijke brug- en sluisstandaard, vraagspecificatie eisen, versie 3.2*. Tech. rep. Rijkswaterstaat.

Rijkswaterstaat (2020). *Jaarbericht Rijkswaterstaat 2019*. Tech. rep. Rijkswaterstaat.

Roth, M., Lesage, J.-J., and Litz, L. (2011). "The concept of residuals for fault localization in discrete event systems". In: *Control Engineering Practice* vol. 19, no. 9, pp. 978–988.

Roussel, J.-M. and Giua, A. (2005). "Designing dependable logic controllers using the supervisory control theory". In: *IFAC Proceedings Volumes* vol. 38, no. 1, pp. 56–61.

Sampath, M., Sengupta, R., Lafortune, S., Sinnamohideen, K., and Teneketzis, D. C. (1996). "Failure diagnosis using discrete-event models". In: *IEEE Transactions on Control Systems Technology* vol. 4, no. 2, pp. 105–124.

Sampath, M., Sengupta, R., Lafortune, S., Sinnamohideen, K., and Teneketzis, D. (1995). "Diagnosability of discrete-event systems". In: *IEEE Transactions on Automatic Control* vol. 40, no. 9, pp. 1555–1575.

van der Sanden, L. J., Reniers, M. A., Geilen, M. C. W., Basten, A. A., Jacobs, J., Voeten, J. P. M., and Schiffelers, R. R. H. (2015). "Modular model-based supervisory controller design for wafer logistics in lithography machines". In: *Proceedings of the 18th Conference on Model Driven Engineering Languages and Systems*. IEEE, pp. 416–425.

Selby, R. W. (2005). "Enabling reuse-based software development of large-scale systems". In: *IEEE Transactions on Software Engineering* vol. 31, no. 6, pp. 495–510.

Sharma, A. and Reniers, M. A. (2016). "Integrated simulation of CIF3 and Simulink models". In: *Proceedings of the 1st Industry Track on Software Language Engineering*, pp. 33–37.

Silva, D. B., Vieira, A. D., Loures, E. F. R., Busetti, M. A., and Santos, E. A. P. (2011). "Dealing with routing in an automated manufacturing cell: A supervisory control theory application". In: *International Journal of Production Research* vol. 49, no. 16, pp. 4979–4998.

Sköldstam, M., Åkesson, K., and Fabian, M. (2007). "Modeling of discrete event systems using finite automata with variables". In: *Proceedings of the 46th Conference on Decision and Control*. IEEE, pp. 3387–3392.

Steward, D. V. (1981). *Systems Analysis and Management: Structure, Strategy and Design*. Petrocelli books.

Swartjes, L. (2018). "Model-based design of baggage handling systems". PhD thesis. Eindhoven University of Technology.

Swartjes, L., van Beek, D., Fokkink, W. J., and van Eekelen, J. (2017). "Model-based design of supervisory controllers for baggage handling systems". In: *Simulation Modelling Practice and Theory* vol. 78, pp. 28–50.

Swartjes, L., van Beek, D. A., and Reniers, M. A. (2014). "Towards the removal of synchronous behavior of events in automata". In: *IFAC Proceedings Volumes* vol. 47, no. 2, pp. 188–194.

Tarjan, R. (1972). "Depth-first search and linear graph algorithms". In: *SIAM Journal on Computing* vol. 1, no. 2, pp. 146–160.

Theunissen, R. J. M., Schiffelers, R. R. H., van Beek, D. A., and Rooda, J. E. (2009). "Supervisory control synthesis for a patient support system". In: *Proceedings of the 10th European Control Conference*. IEEE, pp. 4647–4652.

Theunissen, R. J. M. (2015). "Supervisory control in health care systems". PhD thesis. Eindhoven University of Technology.

Theunissen, R. J. M., Petreczky, M., Schiffelers, R. R. H., van Beek, D. A., and Rooda, J. E. (2013). "Application of supervisory control synthesis to a patient support table of a magnetic resonance imaging scanner". In: *IEEE Transactions on Automation Science and Engineering* vol. 11, no. 1, pp. 20–32.

Thramboulidis, K. (2015). "A cyber-physical system-based approach for industrial automation systems". In: *Computers in Industry* vol. 72, pp. 92–102.

Thuijsman, S., Hendriks, D., Theunissen, R. J. M., Reniers, M. A., and Schiffelers, R. R. H. (2019). "Computational effort of BDD-based supervisor synthesis of extended finite automata". In: *In proceedings of the 15th Conference on Automation Science and Engineering*. IEEE, pp. 486–493.

Torrico, C. R., Leal, A. B., and Watanabe, A. T. (2016). "Modeling and supervisory control of mobile robots: A case of a sumo robot". In: *IFAC-PapersOnLine* vol. 49, no. 32, pp. 240–245.

Vahidi, A., Fabian, M., and Lennartson, B. (2006). "Efficient supervisory synthesis of large systems". In: *Control Engineering Practice* vol. 14, no. 10, pp. 1157–1167.

Verbakel, J. J. (2017). *Behavioral model of two locks*. Tech. rep. Eindhoven University of Technology.

Vieira, A. D., Santos, E. A. P., de Queiroz, M. H., Leal, A. B., de Paula Neto, A. D., and Cury, J. E. R. (2017). "A method for PLC implementation of supervisory control of discrete event systems". In: *IEEE Transactions on Control Systems Technology* vol. 25, no. 1, pp. 175–191.

Vogel-Heuser, B., Diedrich, C., Fay, A., Jeschke, S., Kowalewski, S., Wollschlaeger, M., and Göhner, P. (2014). "Challenges for software engineering in automation". In: *Journal of Software Engineering and Applications* vol. 7, no. 5, pp. 441–451.

de Vos, K. (2018). *Supervisory control synthesis for roadside systems*. Tech. rep. Eindhoven University of Technology.

Vyatkin, V. (2013). "Software engineering in industrial automation: State-of-the-art review". In: *IEEE Transactions on Industrial Informatics* vol. 9, no. 3, pp. 1234–1249.

Wilschut, T. (2018). "System specification and design structuring methods for a lock product platform." PhD thesis. Eindhoven University of Technology.

Wonham, W. M., Cai, K., and Rudie, K. (2018). "Supervisory control of discrete-event systems: A brief history". In: *Annual Reviews in Control* vol. 45, pp. 250–256.

Wonham, W. M. and Cai, K. (2019). *Supervisory Control of Discrete-Event Systems*. Springer.

Xu, S. and Kumar, R. (2008). "Asynchronous implementation of synchronous discrete event control". In: *Proceedings of the 9th Workshop on Discrete Event Systems*. IEEE, pp. 181–186.

Zaytoon, J. and Carré-Ménéatrier, V. (2001). "Synthesis of control implementation for discrete manufacturing systems". In: *International Journal of Production Research* vol. 39, no. 2, pp. 329–345.

Zaytoon, J. and Lafortune, S. (2013). "Overview of fault diagnosis methods for discrete event systems". In: *Annual Reviews in Control* vol. 37, no. 2, pp. 308–320.

Zaytoon, J. and Riera, B. (2017). "Synthesis and implementation of logic controllers–A review". In: *Annual Reviews in Control* vol. 43, pp. 152–168.

Ždánsky, J. and Valigursky, J. (2018). "Application diagnostic of distributed control system with safety PLC". In: *Proceedings of the 12th ELEKTRO*. IEEE.