Supervisory controller synthesis and implementation for safety PLCs

Ferdie F. H. Reijnen¹ · Toby R. Erens² · Joanna M. van de Mortel-Fronczak¹ · Jacobus E. Rooda¹

Received: 8 January 2021 / Accepted: 25 August 2021 / Published online: 6 November 2021
© The Author(s) 2021

Abstract

The development of supervisory controllers for cyber-physical systems is a laborious and error-prone process. Supervisor synthesis enables control designers to automatically synthesize a correct-by-construction supervisor from a model of the plant combined with a model of the control requirements. From the supervisor model, controller code can be generated which is suitable for the implementation on a programmable logic controller (PLC). Supervisors for industrial systems that operate in close proximity to humans have to adhere to strict safety standards. To achieve these standards, safety PLCs (SPLCs) are used. For SPLC implementation, the supervisor has to be split into a regular part and a safety part. In previous work, a method is proposed to automatically split a supervisor model for this purpose. The method assumes that the provided plant model is a collection of finite automata. In this paper, the extension to extended finite automata is described. Additionally, guidelines are provided for modeling the plant and the requirements to achieve a favorable splitting. A case study on a rotating bridge is elaborated which has been used to validate the method. The case study spans all development steps, including the implementation of the resulting supervisor to control the real bridge.

Keywords Supervisory control theory · Programmable logic controllers · Industrial automation

This article belongs to the Topical Collection: *Topical Collection on Control 2022*
Guest Editors: Joerg Raisch, Carla Seatzu and Shigemasa Takai

This work is supported by Rijkswaterstaat, part of the Ministry of Infrastructure and Water Management of the Government of the Netherlands.

✉ Ferdie F. H. Reijnen
ferdie.reijnen@gmail.com

¹ Department of Mechanical Engineering, Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

² Central Information Technology, Rijkswaterstaat, Delft, The Netherlands

1 Introduction

Due to the large number of sensors and actuators present in modern cyber-physical systems and the growing functionality that has to be provided by the control system, the development of the associated supervisory controllers is a laborious and error-prone process. Supervisor synthesis, introduced in Ramadge and Wonham (1987), enables control designers to automatically construct a supervisor from a discrete-event model of the uncontrolled system (the plant) combined with a model of the control requirements. Subsequently, from this supervisor a supervisory controller can be derived which can be used for controller code generation. The use of supervisor synthesis significantly reduces the development time and increases the quality of the supervisory controller, as — by construction — it adheres to the requirements and it is nonblocking. This way of developing a supervisor for a system is referred to as synthesis-based engineering in Baeten et al. (2016).

In industrial systems, supervisory controllers are often implemented on programmable logic controllers (PLCs). A PLC is connected to sensors and actuators in the system. Based on the sensor observations, the PLC program controls the actuators. Derivation of controller software from a synthesized supervisor has extensively been studied in the literature. In Fabian and Hellgren (1998), undesired situations were identified that can occur when such a supervisor is used. In Malik (2003), conditions are given to establish if a supervisor can be used for controller software generation. In Zaytoon and Riera (2017), a review on implementation of discrete-event supervisors on PLCs is provided. Examples of tools that, given a discrete-event model, can automatically generate PLC code are CIF (van Beek et al. 2014) and Supremica (Malik et al. 2017; Prenzel and Provost 2018).

Industrial systems that operate in close proximity to humans are required to adhere to safety standards. For example, control systems 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 62061 standard regarding machine safety (IEC 62061 2005). Safety functions of the control systems used to be hard-wired, but are now generally implemented in the controller software. PLC manufacturers support this by offering so-called safety PLCs (SPLCs), providing the hardware which adheres to the IEC standards.

SPLCs differ from regular PLCs (RPLCs — PLCs without safety functionality), as they have additional dedicated inputs and outputs for safety certified sensors and actuators, and they require separate safety controller code. In other words, the supervisory controller and the inputs and outputs are both split into a safety part and a regular part. The safety part contains the safety-critical control functions and the regular part all other functions. The developed safety controller code has to be certified to comply with the required safety standards.

The previously mentioned code generation tools do not support distributed implementation of a single supervisor on the regular part and the safety part of the SPLC. A solution is to separately develop a regular supervisor and a safety supervisor. This is difficult, as the behavior of the system depends on the joint control of both supervisors. As a result, reasoning about the behavior of each of these two supervisors separately is complicated. Another solution is to synthesize a single supervisor and then split this supervisor in the two parts. Following Reijnen et al. (2020a), the latter option is further investigated in this paper. It should be noted that beforehand, it is not known how the inputs and outputs are split. In the proposed method, this is automatically derived from the requirement model.

1.1 Related work

In the literature, SPLCs have been studied mostly in the context of formal verification. For example, in Soliman and Frey (2011) and Darvas et al. (2016), formal verification methods of safety-critical PLC programs are proposed. These methods are based on model checking and equivalence checking using formal specifications and apply to manually developed function blocks or supervisory controller code. A complementing aspect is elaborated in Ljungkrantz et al. (2012), namely a procedure for developing specifications for safety PLC program components. Resulting formal specifications dispose of ambiguity and enable the use of model checking for automatic verification. In Thönnessen et al. (2019) and Khan et al. (2019), automatic testing of SPLCs is investigated. There, the focus is on generating and executing test cases to determine whether an implementation conforms to a specification. To this end, the *QuickCheck* tool is used that generates randomized tests from a specification. Differently, in our approach, when supervisor synthesis is applied, the implementation conforms to the specification by construction.

Splitting a global supervisor in multiple local supervisors has also been studied before, for example, in the context of distributed control in Cai and Wonham (2010). There, the goal is to control multiple individual systems, such that they collectively realize the controlled behavior identical to that achieved by optimal and nonblocking monolithic control. For SPLCs, a single system has to be controlled with two supervisors. In the former study, the system decomposition is given, whereas for SPLCs, the decomposition is not known beforehand.

In other studies, such as de Queiroz and Cury (2000), division of the plant and the requirement models is proposed to reduce the computational effort of synthesizing the supervisor. State-space explosion can be avoided by computing many small supervisors instead of a single large supervisor. However, in our paper, the goal is to divide the supervisor to adhere to certain safety standards, not to reduce the computation effort. In fact, synthesis is done on the original, undivided model. Afterwards, the supervisor is divided.

In Reijnen et al. (2020a), a splitting method for implementing a supervisor on an SPLC has been presented. There, finite automata (FAs) and Boolean variables are used to model the plant and event-condition requirements are used to model the requirements. The requirements are manually partitioned into regular requirements and safety requirements. From that partitioning, the supervisor is split into a regular part and a safety part.

1.2 Contributions

The contribution of this paper is as follows. Firstly, the method of Reijnen et al. (2020a) is generalized by proposing a splitting method for extended finite-state automata (EFAs), which are FAs enhanced with discrete variables. EFAs offer advantages over FAs as they result in a more compact model representation and they are more suitable to model signal-based systems such as PLCs. We prove that splitting does not influence the behavior of the supervisory controller, by expanding the proof given in Reijnen et al. (2020a). Secondly, we provide guidelines for modeling the plant and for modeling the requirements that result in a favorable splitting. Lastly, a case study on a rotating bridge is described in detail. For the case study, all process steps, i.e., modeling, splitting, code generation, implementation, and testing, are performed.

Fig. 1 RPLC component architecture

1.3 Paper outline

This paper is structured as follows. In Section 2, the way of working of RPLCs and SPLCs is discussed. The preliminaries for discrete-event modeling, supervisor synthesis, and supervisor implementation are provided in Section 3. Modeling guidelines that result in a favorable splitting are provided in Section 4. The method to split a synthesized supervisor on an SPLC is provided in Section 5. In Section 6, the case study on a rotating bridge is described. Finally, Section 7 concludes this paper.

2 Programmable logic controllers

In this section, the way of working of a PLC is discussed. First, it is shown how a regular PLC operates, then the safety PLC operation is explained.

2.1 Regular PLCs

An RPLC can be represented by six components, of which a schematic overview is shown in Fig. 1. 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 control code (the supervisor), and M is the data memory of C. The arrows indicate the direction of data flows, e.g., C reads data from I, M, and Q and writes data to M and Q. The control code of C can be programmed using ladder diagrams, function block diagrams, structured text, instruction lists, or sequential function charts, as standardized in IEC 61131-3 (2013).

An RPLC operates in so-called scan cycles, as shown in Fig. 2. In an RPLC cycle, first, the ‘get’ action copies the values of the input signals to the input image. Then, the control code is executed. Finally, the ‘put’ 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. One full scan cycle takes a few milliseconds.

Fig. 2 Two RPLC scan cycles

2.2 Safety PLCs

In Fig. 3, the component architecture of an SPLC is shown. Compared to the architecture of an RPLC, an SPLC has an additional layer of safety components (depicted in yellow) 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 exists 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. Variables in D_R can be written by C_R and read by C_S , and vice versa for D_S . As an example, D_R can be used by C_R to request C_S to switch on an actuator, C_S then determines whether it is safe to do so.

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 . Secondly, C_S may not read from Q_S and Q_R , as determining safety should be derived from the sensors entirely (PLCopen 2018).

An SPLC scan cycle consists of an execution of the safety part, followed by an execution of the regular part. Each execution is preceded by its associated get action and succeeded by its associated put action. The get action copies the values of the input signals to the input image and the put action copies the values of the output image to the output signals. Here,

Fig. 3 SPLC component architecture

Fig. 4 An SPLC scan cycle

it is assumed that the safety part and the regular part are executed alternately, as shown in Fig. 4¹.

Within C_S , three parts can be distinguished: signal preprocessing, safety logic, and signal postprocessing. Preprocessing verifies the validity of the input signals, for example by one-out-of-two diagnostics. Postprocessing acts as a feedback loop that checks whether the output 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 encompasses the controller code. In general, the controller code of C_S can be programmed using either ladder diagrams or function block diagrams. In this paper, 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).

3 Preliminaries

In this section, modelling, synthesis, and implementation of supervisors is discussed. Firstly, it is shown how EFAs are used to model the components of the plant. Secondly, using event conditions to model requirements is discussed. Lastly, synthesis and implementation of supervisors is explained.

3.1 Modelling discrete-event systems with EFAs

In Cheng and Krishnakumar (1996), Chen and Lin (2000), and Sköldstam et al. (2007), extended finite-state automata are introduced for modeling systems. EFAs are finite-state automata parameterized by bounded discrete variables. Compared to regular finite-state automata, EFAs result in more compact and elegant models. For modeling large-scale systems, the use of EFAs is essential. In Reijnen et al. (2018), an example is shown where a finite-state automaton needs $2.8 \cdot 10^{11}$ locations to express a requirement that can be expressed in a simple guard, using an EFA. Moreover, EFAs are more suitable to model signal-based systems such as PLCs

A model consists of several interacting EFAs, called an EFA system, $\mathcal{E} = \{E_1, \dots, E_m\}$ together with a set of variables $X_{\mathcal{E}} = \{x_1, \dots, x_n\}$. 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, $\text{dom}(x)$. A valuation $v \in V$ is a function $v : X_{\mathcal{E}} \rightarrow \bigcup_{x \in X_{\mathcal{E}}} \text{dom}(x)$, for which $v(x) \in \text{dom}(x)$ for every $x \in X_{\mathcal{E}}$. The set of all valuations is V , the initial valuation is $v^0 \in V$.

Formally, an EFA is defined as $E = (L, X, \Sigma, \rightarrow, l^0, L^m)$, where L is the set of locations, $X \subseteq X_{\mathcal{E}}$ the set of local variables, Σ the set of events, $\rightarrow$ is the transition relation,

¹Alternatively, the safety part may interrupt the regular part. However, this may lead to synchronization errors.

$l^0 \in L$ the initial location, and $L^m \subseteq L$ the set of marked locations. The transition relation is defined as $\rightarrow \subseteq L \times G \times \Sigma \times U \times L$, 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. To this variable, values may only be assigned by that EFA. However, any other EFA may read its value.

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.

A guard is a function, $g : V \rightarrow \{\mathbf{T}, \mathbf{F}\}$, 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 (including the variables that are not local to that EFA). It is assumed that each EFA implicitly contains a variable that represents the current location, called the location pointer LP , with $\text{dom}(\text{LP}) = L$. As a result, locations of other EFAs can be used in the guards. An example of a guard is $x > 5 \wedge \text{E.On}$, 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 $v(\text{LP}_{\text{E}}) = \text{On}$.

An update is a function $u : V \rightarrow V$. 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 $x := \text{if}(x < 5) : x + 1 \text{ else } : x$, 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.

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

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

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

- 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.

An example of synchronizing EFAs is shown in Fig. 5. The EFA system is $\mathcal{E} = \{\text{Switch}, \text{Lamp}\}$, and $X_{\mathcal{E}} = \{Q\}$, where Q is a Boolean variable. In the graphical representation, edges labeled with controllable and uncontrollable events are depicted with solid arrows and dashed arrows, respectively. 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. Filled circles denote marked locations. To differentiate between location names and event names, the former are written with a capital initial letter. In this example, the events **on** and **off** are only enabled when EFA **Switch** is in location **Active** and in location **Inactive**, respectively. When events **on** or **off** occur, the value

Switch:

Lamp:

Fig. 5 Graphical representation of EFAs

of Q is negated. The synchronous product is shown in Fig. 6. Transitions with a guard that always evaluates to **F** are not displayed.

3.2 Modeling of requirements

For supervisor synthesis, the desired behavior of a system is given by a requirement model. A requirement 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 event conditions, introduced in Ma and Wonham (2006) and extended in Markovski et al. (2010).

Event conditions provide conditions for an event to be enabled. For event e and condition c , e **when** c defines that e may only occur whenever c evaluates to **T**. Conditions are defined similar to guards used in the extended transition relation. An event condition can also be represented as an EFA, such that the synchronous product with another EFA can be computed. The EFA representation of an event condition is shown in Fig. 7.

Based on Ouedraogo et al. (2011), the behavior of a collection $\mathcal{E}$ of EFAs with respect to an associated requirement set $\mathcal{R}$ is defined as the pair of languages, the closed behavior T and the marked behavior T_m , of the finite state automaton equivalent to the synchronous product of the $\mathcal{E}$ elements refined by strengthening the event guards using the associated conditions from $\mathcal{R}$. This refined synchronous product is denoted by $\mathcal{E}_{\parallel}$ and defined by $(L, X, \Sigma, \rightarrow, l^0, L^m)$.

Definition 1 Closed behavior $T(\mathcal{E}, \mathcal{R})$ is defined as the set of event sequences $t = \sigma_1 \dots \sigma_n$, $n \geq 0$, for which there is a run $(l_1, v_1) \xrightarrow{\sigma_1} (l_2, v_2) \dots \xrightarrow{\sigma_n} (l_{n+1}, v_{n+1})$ in $\mathcal{E}_{\parallel}$

Fig. 6 Synchronous product of EFAs Switch and Lamp

Fig. 7 EFA representation of an event condition e **when** c

such that $(l_1, v_1) = (l^0, v^0)$ and in every state (l_i, v_i) , $i \leq n$, for the associated event $(l_i, g_i, \sigma_i, u_i, l_{i+1})$ holds that g_i evaluates to **T**.

Definition 2 Marked behavior $T_m(\mathcal{E}, \mathcal{R})$ is defined as the set of event sequences $t = \sigma_1 \dots \sigma_n$, $n \geq 0$, for which there is a run $(l_1, v_1) \xrightarrow{\sigma_1} (l_2, v_2) \dots \xrightarrow{\sigma_n} (l_{n+1}, v_{n+1})$ in $\mathcal{E}_{\parallel}$ such that $(l_1, v_1) = (l^0, v^0)$, $l_{n+1} \in L^m$, and in every state (l_i, v_i) , $i \leq n$, for the associated event $(l_i, g_i, \sigma_i, u_i, l_{i+1})$ holds that g_i evaluates to **T**.

3.3 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 1) the requirements are always met, 2) it is controllable, 3) it is nonblocking, and 4) it is maximally permissive (see e.g., Cassandras and Lafontaine (2009) or Wonham and Cai (2019)). For synthesizing a supervisor for an EFA system, the algorithm proposed in Ouedraogo et al. (2011) is used.

Using the method of Miremadi et al. (2008), the synthesis result is represented as additional event conditions. The supervisor is the collection of the component models, the requirement model, and the additional event conditions. Without loss of generality, in the remainder of this paper we assume that the plant and the requirements together satisfy the properties listed above and synthesis does not generate additional event conditions. This is justified by the fact that whenever synthesis does generate additional event conditions, they can be included as requirements.

3.4 Supervisor implementation

From the model, controller code can be generated for a PLC. To prevent state-space explosion, the synchronous product of the models is not calculated. Instead, the linearization procedure given in Swartjes et al. (2014) is used.

For connecting the variables in the input image of the PLC to the supervisor, a mapping $\mathcal{I} : I \rightarrow X_{\mathcal{E}}$ is defined, where I is the set of input-image variables, and $X_{\mathcal{E}}$ the set of variables in the model. For example, consider EFA *Switch*, shown in Fig. 5 and Boolean input-image variable I_B . A mapping can be defined such that *Switch* is in location *Inactive* when $I_B = \mathbf{F}$ and in location *Active* when $I_B = \mathbf{T}$. This implies that when the value of I_B changes from **F** to **T** or from **T** to **F**, event *activate* or event *deactivate* occurs, respectively.

For connecting the variables in the output image of the PLC to the supervisor, a mapping $\mathcal{Q} : Q \rightarrow X_{\mathcal{E}}$ is defined, where Q is the set of output-image variables. This implies that for every $q \in Q$ there should be a corresponding variable in the model. For example, consider EFA *Lamp*, shown in Fig. 5, and Boolean output-image variable Q_L . A mapping can be

defined such that Q_L equals the value of Q . This implies that when event `on` or `off` occurs, the value of Q_L changes to **T** or **F**, respectively.

It should be noted that the supervisor is implemented as a supervisory controller. In Malik (2003) and Reijnen et al. (2019) solutions are proposed to handle subtle differences between a supervisory controller and a supervisor obtained via synthesis.

A supervisor can only forbid events, whereas a supervisory controller can force some events (typically, the controllable events). Methods exist to transform a supervisor into a supervisory controller (i.e., PLC code) automatically, see, e.g., Fabian and Hellgren (1998), Zaytoon and Riera (2017), Balemi et al. (1993), Malik (2003). In essence, what is done is to keep forcing controllable events among the enabled events of the supervisor, until no more controllable events are enabled. Under certain conditions (see below), the state that is eventually reached is always the same, independent of which enabled event is forced. When the values of the outputs are only based on the current state, the choice between enabled events can be made arbitrarily. In the new state, the supervisory controller waits for a new plant input (e.g., a sensor that switches on).

For this method, the supervisor needs to satisfy additional properties not guaranteed by synthesis. These properties are finite response, confluence, and nonblocking under control, as described in, e.g., Malik (2003), Reijnen et al. (2019). Finite response ensures that always eventually 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 adhere to these properties, the requirements should be altered.

In the literature (e.g., Fabian and Hellgren (1998), Zaytoon and Riera (2017)), differences are described between the execution of a supervisory-controller model and a PLC implementation. These differences originate from the real-time execution of the code. The PLC executes a repeating cycle, as explained in Section 2. 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 scan cycle), the above situations are never encountered, as demonstrated by Malik (2003) and Forschelen et al. (2012).

4 Modeling

This section introduces the component-based modeling method that is used for modeling. When component-based modeling is used, the plant is modeled as many small EFAs that do not share events (loosely coupled). The requirement model the desired interactions. The splitting method, shown in Section 5 works particularly well with such models.

Component-based modeling is a modeling paradigm that uses the fact that large systems can be obtained by assembling smaller components. Each component can be modeled separately. This way of modeling has proven to be useful for modeling the plant for supervisor synthesis, as shown in Kovács et al. (2012), Huang et al. (2015), and Reijnen et al. (2020b). In Goorden et al. (2019), modeling guidelines are proposed that help in obtaining the

necessary models. Two guidelines that help in achieving a favorable SPLC splitting are given below.

Guideline 1 Model independent plant components as asynchronous component models.

This entails that component models do not share events or refer to variables local to other EFAs. For infrastructural systems, it has been shown in Reijnen et al. (2020b) that a realistic plant model can be created by composing asynchronous component models consisting of 2 to 4 states each.

For splitting the supervisor, having many small component models is advantageous as for each model it can be determined whether it should be included in the safety part. By having smaller models, no excess model parts have to be included in the safety part.

Guideline 2 Use the abstraction level of the inputs and outputs of the control hardware for the plant model.

This entails that each sensor and actuator in the plant is modeled with a separate component EFA. With this perspective, events relate to the change of signal value sent to actuators or received from sensors. Furthermore, all events related to actuators are controllable and all events related to sensors are uncontrollable. The way of modeling is similar to the input-output perspective introduced in Balemi et al. (1993).

In the rest of this paper, we call an automaton a *sensor* EFA if its event set contains only uncontrollable events, i.e., $\Sigma = \Sigma_u$, and an *actuator* EFA if its event set contains only controllable events, i.e., $\Sigma = \Sigma_c$.

Using the input-output perspective offers two advantages. Firstly, textual control requirements formulated by engineers can be more easily translated into event-condition models, as the states of actuators and sensors are directly available in the plant model (as demonstrated in Reijnen et al. (2020b)). Secondly, for implementation the component models can easily be coupled to the available sensors and actuators as there exist dedicated component models for these.

For splitting the supervisor, a third advantage is observed. For SPLCs it is necessary to split the sensors and actuators into the safety or regular category. When these are available as component models, we can split these component models. From that split, we can derive which sensors and actuators should be put into the safety category.

5 Splitting

In this section, the method is described that can be used to split a supervisor for the implementation on an SPLC. The method is similar to the method described in Reijnen et al. (2020a). Equations have been adapted such that the method is applicable to EFA models.

The general idea is as follows. First, a plant model $\mathcal{P}$ consisting of automata representations of system components and a requirement set $\mathcal{R}$ are defined (and if needed extended with additional requirements following the synthesis step, as explained in Section 3.3) that together form the supervisor. Some of the variables and locations in the supervisor will be connected to the input image I of the PLC and some other to the output image Q of the PLC. These are defined via mappings $\mathcal{I}$ and $\mathcal{Q}$, respectively (see Section 3.4). The goal is to distribute the elements of these sets over the regular part and safety part. The distribution is derived from a (manual) partitioning of the requirements into disjoint sets of

Fig. 8 Desired result after splitting

safety requirements $\mathcal{R}'_S$ and regular requirements $\mathcal{R}'_R$. Based on this, the plant model, the requirement model, the input image, and the output image are split and the necessary data communication between the parts is derived, as shown in Fig. 8.

In the figure, $\mathcal{P}_R$ and $\mathcal{P}_S$ denote the set of plant components that belong in the regular part and in the safety part, respectively. Similarly, $\mathcal{R}_R$ and $\mathcal{R}_S$ denote the set of requirements that belong in the regular part and in the safety part, respectively. $\mathcal{P}_R$ and $\mathcal{R}_R$ form the *regular supervisor* and $\mathcal{P}_S$ and $\mathcal{R}_S$ form the *safety supervisor*. I_R , I_S , Q_R , Q_S are the regular and safety input and output image variables, respectively. Lastly, D_R and D_S are the sets of variables communicated via the data buffers.

Notice that $\mathcal{R}'_S$ and $\mathcal{R}'_R$ may differ from $\mathcal{R}_S$ and $\mathcal{R}_R$, where the latter denote the requirements in the safety part and the requirements in the regular part, respectively. By definition, $\mathcal{R}'_S$ should be included in $\mathcal{R}_S$. However, some requirements from $\mathcal{R}'_R$ may also be included in $\mathcal{R}_S$.

For describing the splitting procedure, some additional notations are introduced. For all conditions c , the functions $\text{comps}(c)$ and $\text{var}(c)$ are defined that return the set of EFAs that are referenced and the set of variables that are used. An EFA is referenced if either its location is used or a local variable is used in c . Given a set of EFAs $\mathcal{E}$, $\text{sens}(\mathcal{E})$ and $\text{act}(\mathcal{E})$ return the sets of sensors and actuators, respectively. Given an EFA E , $\text{evt}(E)$ and $\text{var}(E)$ return the event set of E and the set of local variables of E , respectively. Given a variable x , $\text{comp}(x)$ denotes the component where x is local to.

The rules that the splitting has to adhere to are given in Section 5.1. An illustrative example is provided in Section 5.2. The proposed splitting procedure is given in Section 5.3. In Section 5.4, it is proven that splitting does not influence the behavior of the supervisor.

5.1 Objectives of the splitting

For splitting, it is assumed that each condition for a requirement in $\mathcal{R}'_S$ is derived entirely from sensors. This is necessary, as in PLCopen (2018) it is defined that the actuators have to be monitored by sensors directly (instead of actuators (see also Fig. 3) or internal variables). This means that:

$$\forall(\sigma, c) \in \mathcal{R}'_S : \text{comps}(c) = \text{sens}(\text{comps}(c)) \quad (1)$$

In PLCopen (2018), rules are defined for the safety program. These rules lead to a clear separation between the safety functionalities and the non-safety functionalities. Moreover, by following these rules, the relevant aspects of the IEC 62061 standard are also met. These are translated in the following six rules used for splitting. First of all, by definition:

(r1) Each safety requirement has to be included in the safety part.

Secondly, sensors that are used in the condition part of safety requirements, have to be included in the safety part. To this end, the EFAs that model these sensors have to be included in the safety part.

(r2) Each sensor EFA that is used in a condition of a safety requirement has to be included in the safety part.

Thirdly, actuators that are subjected to safety requirements, have to be included in the safety part. To this end, the EFAs that model these actuators have to be included in the safety part.

- (r3) Each actuator EFA that has an event which is subjected to a safety requirement has to be included in the safety part.

Finally, because the safety program should be as small as possible (e.g., for fast execution and to obtain a clear separation between the safety part and the non-safety part), the following should hold:

- (r4) $\mathcal{P}_S$ and $\mathcal{R}_S$ should be as small as possible.

- (r5) D_S and D_R 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 through r6, a solution would seem to put everything in the regular part, but it is not due to r1 through r3. The other way around, when only looking at r1 through r3, a solution would seem to put everything in the safety part, but it is not due to r4 through r6.

5.2 An illustrative example

Consider two switches, Switch1 and Switch2, a lamp, and a motor. The plant component models are shown in Fig. 9.

Assume there are two regular requirements ($\mathcal{R}'_R$) and one safety requirement ($\mathcal{R}'_S$):

reg1:	Lamp.on when Switch1.Active
reg2:	Motor.on when Switch1.Active
safel:	Motor.on when Switch2.Active

These requirements state that if switch 1 is active, the lamp switches on. Whenever both switches are active the motor switches on. It is a safety requirement that switch 2 has to be active for the motor to switch on.

From this information, the following can be concluded regarding the partitioning of the plant components and the requirements. By definition, requirement safel must be part of the safety part. Secondly, since this requirement depends on information received from Switch2, Switch2 has to be a safety component. Thirdly, since this requirement restricts the behavior of Motor, Motor has to be a safety component.

Fig. 9 Plant component models for two switches, a lamp, and a motor

Since *Lamp* and *Switch1* are not related to any safety requirement, they are regular components. Moreover, *reg1* and *reg2* are regular requirements. Finally, the data buffer D_R must be used to communicate whether *reg2* is satisfied, as this is needed to determine whether *Lamp.on* may be enabled.

Subsequently, assume the following mappings $\mathcal{I} = \{I_1 : \text{Switch1.Active}, I_2 : \text{Switch2.Active}\}$ and $\mathcal{Q} = \{Q_1 : \text{Lamp.On}, Q_2 : \text{Motor.On}\}$. Since *Switch1* and *Lamp* are regular components, I_1 and Q_1 are regular inputs and outputs, respectively. Because *Switch2* and *Motor* are safety components, I_2 and Q_2 are safety inputs and outputs, respectively.

5.3 Splitting

In this subsection, a splitting procedure is proposed that satisfies the rules defined in the previous subsection.

It is assumed that the plant model is a product system, which means that the event sets of component models are pairwise disjoint and only local variables are used in guards. 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 or use a variable from another EFA, they are replaced by their synchronous product. Moreover, we assume that the input-image variables are coupled to sensor EFAs (and vice versa) and output-image variables are coupled to actuator EFAs (and vice versa). When component-based modeling is used, both assumptions should be satisfied.

From (r2) and (r6), definitions for the safety-sensor components, safety input-image variables and the regular input-images variables are derived:

$$\text{sens}(\mathcal{P}_S) = \{E \in \mathcal{P} \mid \exists(\sigma, c) \in \mathcal{R}'_S, E \in \text{sens}(\text{comps}(c))\} \quad (2)$$

$$I_S = \{i \in I \mid \text{comp}(\mathcal{I}(i)) \in \text{sens}(\mathcal{P}_S)\} \quad (3)$$

$$I_R = I \setminus I_S \quad (4)$$

From (r3) and (r6), definitions for the safety-actuator components, safety output-image variables and regular output-image variables are derived:

$$\text{act}(\mathcal{P}_S) = \{E \in \mathcal{P} \mid \exists(\sigma, c) \in \mathcal{R}'_S, \sigma \in \text{evt}(E)\} \quad (5)$$

$$Q_S = \{q \in Q \mid \text{comp}(\mathcal{Q}(q)) \in \text{act}(\mathcal{P}_S)\} \quad (6)$$

$$Q_R = Q \setminus Q_S \quad (7)$$

From (r2) and (r3), it follows that the safety-sensor components and safety-actuator components belong in the safety part. From (r4) it follows that the other components belong in the regular part.

$$\mathcal{P}_S = \text{sens}(\mathcal{P}_S) \cup \text{act}(\mathcal{P}_S) \quad (8)$$

$$\mathcal{P}_R = \mathcal{P} \setminus \mathcal{P}_S \quad (9)$$

From the previous two equations, the disjoint set of variables in the safety part and in the regular part can be derived.

$$X_S = \{x \mid \exists P \in \mathcal{P}_S, x \in \text{var}(P)\} \quad (10)$$

$$X_R = \{x \mid \exists P \in \mathcal{P}_R, x \in \text{var}(P)\} \quad (11)$$

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

$$\Sigma_S = \{\sigma \mid \exists P \in \mathcal{P}_S, \sigma \in \text{evt}(P)\} \quad (12)$$

$$\Sigma_R = \{\sigma \mid \exists P \in \mathcal{P}_R, \sigma \in \text{evt}(P)\} \quad (13)$$

From Eqs. 5, 8, and 12, 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, $\mathcal{R}'_R$ can be split into regular requirements for Σ_S and regular requirements for Σ_R .

$$\mathcal{R}_R^S = \{(\sigma, c) \in \mathcal{R}'_R \mid \sigma \in \Sigma_S\} \quad (14)$$

$$\mathcal{R}_R^R = \{(\sigma, c) \in \mathcal{R}'_R \mid \sigma \in \Sigma_R\} \quad (15)$$

Because of (r4), requirements in $\mathcal{R}_R^S$ 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 of a component in $\mathcal{P}_S$ is used to evaluate a condition and this value changes in the safety part. 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

$$\mathcal{R}_R^{SS} = \{(\sigma, c) \in \mathcal{R}_R^S \mid \text{comps}(c) \cap \mathcal{P}_S \neq \emptyset\} \quad (16)$$

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

$$\mathcal{R}_R^{SR} = \mathcal{R}_R^S \setminus \mathcal{R}_R^{SS} \quad (17)$$

The event conditions in $\mathcal{R}_R^{SR}$ 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 defined in Eq. 18. This ensures that each event has exactly one condition evaluation that has to be communicated (r4, r5).

$$\forall \sigma \in \Sigma_S, c_\sigma = \bigwedge_{(\sigma, c) \in \mathcal{R}_R^{SR}} c \quad (18)$$

Given a valuation v for the variables, we define $v_\sigma = v \models c_\sigma$. Practically, v_σ acts as a signal from the regular part to the safety part, indicating that all regular requirements are satisfied. As a result, event σ may be enabled whenever it is safe to do so. In the example in Section 5.2, v_σ is the result of the second requirement.

The following requirements are added to replace $\mathcal{R}_R^{SR}$. These requirements state that for event σ , all regular requirements must be satisfied (i.e., $v_\sigma = \mathbf{T}$).

$$\mathcal{R}_D = \{(\sigma, v_\sigma) \mid \sigma \in \Sigma_S\} \quad (19)$$

The requirements in the safety program and the regular program are

$$\mathcal{R}_S = \mathcal{R}'_S \cup \mathcal{R}_R^{SS} \cup \mathcal{R}_D \quad (20)$$

$$\mathcal{R}_R = \mathcal{R}_R^R \quad (21)$$

It should be noted that $\mathcal{R}_R^{SR}$ is effectively replaced by $\mathcal{R}_D$.

Finally, the programs cannot access variables from the other program part, as these are saved in M_R or M_S (see Fig. 3). Therefore, some variables have to be communicated via D_R and D_S . From Eqs. 1 and 2, it follows that variables used in conditions of $\mathcal{R}'_S$ are always contained in the safety part. Similarly, because of Eqs. 16, 17, and 18, it follows that v_σ can be calculated with the variables contained in the regular part.

From the regular part to the safety part, the v_σ variables have to be communicated. Secondly, the regular variables used in conditions of $\mathcal{R}_R^{SS}$ have to be communicated. From the safety part to the regular part, references to safety variables used in conditions of $\mathcal{R}_R^R$ have

to be communicated. To be precise, the variables contained within D_R and D_S are given in Eqs. 22 and 23.

$$D_R = \{v_\sigma \mid \exists(\sigma, v_\sigma) \in \mathcal{R}_D\} \cup \{x \mid \exists(\sigma, c) \in \mathcal{R}_R^{SS}, x \in \text{var}(c) \cap X_R\} \quad (22)$$

$$D_S = \{x \mid \exists(\sigma, c) \in \mathcal{R}_R^R, x \in \text{var}(c) \cap X_S\} \quad (23)$$

Above, all sets shown in Fig. 8 are defined.

5.4 Proof of equal behavior

The result is that the supervisors before and after the splitting have equal behavior. In Reijnen et al. (2020a), this proof has already been given for FAs. The proof is similar for EFAs, but has been included for completeness. For the proof, first notions of equal restrictiveness are defined.

Definition 3 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 logically equivalent.

Definition 4 Two sets of requirements $\mathcal{R}_1$ and $\mathcal{R}_2$ over the same event set are said to be equally restrictive iff for each event σ , $(\sigma, \bigwedge_{(\sigma, c) \in \mathcal{R}_1} c)$ is equally restrictive to $(\sigma, \bigwedge_{(\sigma, c) \in \mathcal{R}_2} c)$.

Definition 5 Let $\mathcal{P}$ be a plant and $\mathcal{R}_1$ and $\mathcal{R}_2$ be sets of requirements. The behavior of $\mathcal{P}$ together with $\mathcal{R}_1$ is equal to the behavior of $\mathcal{P}$ together with $\mathcal{R}_2$ if $T(\mathcal{P}, \mathcal{R}_1) = T(\mathcal{P}, \mathcal{R}_2)$ and $T_m(\mathcal{P}, \mathcal{R}_1) = T_m(\mathcal{P}, \mathcal{R}_2)$.

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 1 Let S_1 be $(\mathcal{P}, \mathcal{R})$ and S_2 be $(\mathcal{P}_R \cup \mathcal{P}_S, \mathcal{R}_R \cup \mathcal{R}_S)$, as defined in Section 5.3. Assuming that each c_σ is evaluated when its value is needed, such that v_σ in Eq. 19 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 Eqs. 8 and 9 that $\mathcal{P} = \mathcal{P}_R \cup \mathcal{P}_S$. Hence, the plant models are equal. As defined in Section 5.3, $\mathcal{R} = \mathcal{R}'_R \cup \mathcal{R}'_S$. From this and Eqs. 14, 15, 16, and 17, it follows that the requirements of S_1 are $\mathcal{R}_1 = \mathcal{R}^R_R \cup \mathcal{R}^{SS}_R \cup \mathcal{R}^{SR}_R \cup \mathcal{R}'_S$. From Eqs. 20 and 21, it follows that the requirements for S_2 are $\mathcal{R}_2 = \mathcal{R}^R_R \cup \mathcal{R}'_S \cup \mathcal{R}^{SS}_R \cup \mathcal{R}_D$. From Definition 4, it follows that $\mathcal{R}_1$ is equally restrictive to $\mathcal{R}_2$ if $\mathcal{R}_D$ is equally restrictive to $\mathcal{R}^{SR}_R$. So now it remains to be proven that $\mathcal{R}_D$ is equally restrictive to $\mathcal{R}^{SR}_R$. Equations 14 and 17 assure that all requirements in $\mathcal{R}^{SR}_R$ apply to events in Σ_S . In combination with Eqs. 18 and 19, Definition 4, and the assumption that v_σ can be substituted by c_σ , this shows that $\mathcal{R}_D$ is equally restrictive to $\mathcal{R}^{SR}_R$. Therefore, $\mathcal{R}_1$ is equally restrictive to $\mathcal{R}_2$.

Subsequently, it is proven that for all event sequences t :

$$t \in T(\mathcal{P}, \mathcal{R}_1) \Leftrightarrow t \in T(\mathcal{P}, \mathcal{R}_2)$$

and

$$t \in T_m(\mathcal{P}, \mathcal{R}_1) \Leftrightarrow t \in T_m(\mathcal{P}, \mathcal{R}_2).$$

Fig. 10 Model of the execution of an SPLC

This evidently holds for $t = \epsilon$. Assume $t = \sigma_1 \dots \sigma_n$, for some $n \geq 1$, belongs to $T(\mathcal{P}, \mathcal{R}_1)$. According to Definition 1, this means that there is a run $(l_1, v_1) \xrightarrow{\sigma_1} (l_2, v_2) \dots \xrightarrow{\sigma_n} (l_{n+1}, v_{n+1})$ in the synchronous product of EFAs in $\mathcal{P}$ refined w.r.t. R_1 for which $(l_1, v_1) = (l^0, v^0)$ and in every state (l_i, v_i) , for $i \leq n$, event $(l_i, g_i, \sigma_i, u_i, l_{i+1})$ is such that g_i evaluates to **T**. Since $\mathcal{R}_1$ is equally restrictive to $\mathcal{R}_2$, in the synchronous product of EFAs in $\mathcal{P}$ refined w.r.t. R_2 there is a corresponding run where in every state (l_i, v_i) , for $i \leq n$, event $(l_i, g'_i, \sigma_i, u_i, l_{i+1})$ is such that g'_i evaluates to **T**. Hence, $t \in T(\mathcal{P}, \mathcal{R}_2)$. Similar reasoning can be applied to prove the implication the other way around and, using Definition 2, for equality of $T_m(\mathcal{P}, \mathcal{R}_1)$ and $T_m(\mathcal{P}, \mathcal{R}_2)$. Hence, according to Definition 5, S_1 and S_2 have equal behavior. $\square$

To investigate how the SPLC scan cycle (shown in Fig. 4) 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 Fig. 10. 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 are updated. Similarly, the event *safe* denotes a switch to the safety part. When this event occurs, the values of the variables in D_R are updated.

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 Eq. 23 that D_S contains references to variables local to components in $\mathcal{P}_S$ only. Figure 10 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 Eq. 22 that D_R contains variables local to components in $\mathcal{P}_R$. Figure 10 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 Eqs. 16, 17, and 18 that their values are calculated from variables local to components in $\mathcal{P}_R$. 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 Proposition 1 and the above reasoning, it is concluded that the splitting and data communication do not influence the behavior of the supervisor.

6 Case study

To demonstrate the practical applicability of the proposed method, a case study on the Oissterwijksebaan bridge (OBB) has been performed. The OBB, shown in Fig. 11, is a rotating bridge that provides a way for land traffic to cross the canal when closed and a way for waterborne vessels to pass when open (rotated 90 degrees). The bridge deck consists of a

Fig. 11 The Oosterwijksebaan bridge

two-way vehicle lane and two pedestrian lanes. The bridge is operated and maintained by Rijkswaterstaat.

In Section 6.1, the way of working of the OBB is described. Modeling of the components is discussed in Section 6.2. How the safety requirements and regular requirements are formalized is shown in Subsections 6.3 and 6.4, respectively. The results of splitting the models are given in Section 6.5. How the results are validated is explained in Section 6.6.

6.1 Way of working

A schematic overview of the bridge is shown in Fig. 13. 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 provide a safe situation before the bridge is opened. The vessels on the Wilhelmina Canal are informed via red-green-red vessel traffic lights (VTL1 - VTL8). A VTL can show four different sign aspects, see Fig. 12. The double-red aspect indicates that the lock is out-of-service. The red and red-green aspects indicate that approaching the bridge not allowed and allowed, respectively. The green aspect indicates that a vessel may pass the bridge. 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 (open or close) and speed of this motor can be controlled.

Fig. 12 Sign aspects of a VTL: double red, red, red-green, and green

Fig. 13 A schematic overview of the Oisterwijksebaan bridge

An operator controls the bridge via a graphical user interface (GUI). The GUI has two purposes: providing information to the operator about the system's state, such as the displayed sign aspects, and acting as a control panel to provide a way for the operator to give commands. 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 51 sensors and 26 control commands.

The desired controlled behavior (Fig. 13) is as follows. When a vessel wants to pass the bridge, the operator initiates the open bridge procedure. Then, all stop signs switch on, and 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 may be shown. The close bridge procedure is similar to the procedure described above. When the bridge is not being operated, both red lamps of a VTL are enabled to indicate this.

6.2 Components

For the plant model, the component-based modeling method 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 Fig. 14. In this figure, the number of component EFAs is denoted. The model can be partitioned into two parts: the physical part representing all 'real' components and the GUI part representing the symbols available on a GUI to give commands.

For the Oisterwijksebaan bridge, 27 actuator component models, 48 sensor component models, and 7 command models are needed. One component model can represent multiple

Fig. 14 Plant model decomposition of the OBB

inputs or multiple outputs. For example, one VTL actuator EFA represents the outputs for the top two lamps. As an illustration, the model of that actuator is shown in Fig. 15. The model consists of three locations (circles), representing the legal aspects that can be shown.

6.3 Safety requirements

Due to the human-machine interaction, the bridge has to adhere to strict standards to guarantee human safety. Rijkswaterstaat uses a risk assessment procedure (Smith and Simpson 2020) that calculates the safety integrity level (SIL-level) for certain situations based on four criteria: severity, frequency, probability of incident, and probability of avoidance. From that, dangerous situations are identified (e.g., a person stands on the bridge when the bridge rotates), and requirements are defined to avoid these situations. For this case study, 22 textual requirements were already available, given below.

1. A stop sign will only turn off when the barriers are open.
2. A barrier will only close when the stop signs are on.
3. A barrier will only open when the bridge is closed.
4. A barrier will only open when the bridge is locked.
5. The bridge will only unlock when the barriers are closed.
6. The bridge will only lock when the VTLs show a red aspect.
7. The brake will only be released when the barriers are closed.
8. The brake will only be released when the VTLs show a red aspect.
9. The motor will only start on when the barriers are closed.
10. The motor will only start when the VTLs show a red aspect.
11. A VTL will only show a green aspect when the brake is applied.
12. A VTL will only show a green aspect when the bridge is open.
13. A VTL will only show a green aspect when the opposite VTLs show a red aspect.
14. A VTL will only show a red-green aspect when the opposite VTLs do not show a red-green aspect.

Fig. 15 Component model of the vessel traffic light actuator

15. A barrier will only close when the emergency stop is not pressed.
16. A barrier will only open when the emergency stop is not pressed.
17. The bridge will only unlock when the emergency stop is not pressed.
18. The bridge will only lock when the emergency stop is not pressed.
19. The brake will only be released when the emergency stop is not pressed.
20. The motor will only start on when the emergency stop is not pressed.
21. A VTL will only show a green aspect when the emergency stop is not pressed.
22. A VTL will only show a red-green aspect when the emergency stop is not pressed.

These requirements are formalized using event-condition models. The first part of the requirement is always an actuator action (i.e., a stop sign will only turn off when. . .) and the second part of the requirements is always an condition (i.e., . . . when the barriers are open). For example, Requirement 1 is modeled as:

SS.A.c_off when BB1.Open.On and BB2.Open.On,

In the expression, SS.A.c_off is the c_off event of the stop signs actuator component. BB1.Open.On and BB2.Open.On are the On locations of the components that represent the sensor that measures the open position of barrier 1 and 2, respectively.

In total, 48 event conditions are used to model the safety requirements. Notice that for some requirements, multiple event conditions are used. For example, requirements 22 refers to ‘A VTL’, whereas there are eight VTLs and thus eight event conditions.

6.4 Regular requirements

To achieve the desired functionality, additional (non-safety) requirements are necessary. For the bridge, the following regular requirements are encountered.

1. Actuators will only start when an operator gives the corresponding command.
2. VTL aspects will only switch when an operator gives the corresponding command.
3. Actuators will stop when they reach their end position.
4. Actuators will only start when they have not reached their end position.
5. When the locking procedure starts, the hydraulic valves will open.
6. The position of the hydraulic valves will only be set when the unlock-bridge procedure or lock-bridge procedure starts.
7. The rotation direction will only be set when the close-bridge procedure or open-bridge procedure starts.
8. The rotation speed will only be set when the close-bridge procedure or open-bridge procedure starts.

Similar to the safety requirements, all these requirements are modeled using event conditions. In total, 118 event conditions are used to model the regular requirements.

6.5 Splitting results

The results of splitting the model are shown in Table 1. To obtain these results, the splitting method has been implemented in CIF.² The results show that after splitting there are 60 requirements in the regular part and 99 requirements in the safety part. These numbers can

²CIF is the tool used for modeling, synthesis, splitting, and code generation, and is available at <https://projects.eclipse.org/projects/technology.escet>

Table 1 Results of partitioning the model

	Safety	Regular
Requirements	99	60
Data buffers	0	51
Actuators	24	15
Sensors	22	29

be explained as follows. Of the regular requirements, 58 contain safety events, of which 51 unique events, which explains the 51 data buffers. The resulting number of $\mathcal{R}_R^R$ requirements is then 60 ($118 - 58$). The number of safety requirements is 48 $\mathcal{R}_S'$ and 51 $\mathcal{R}_D$ requirements. Note that 58 $\mathcal{R}_R$ requirements are replaced by 51 $\mathcal{R}_D$ requirements, which explains the total number of requirements (166 before the split, and 159 after the split).

The 24 safety actuators are related to the stop signs, the barriers, the locking mechanism, the brake, the rotating mechanism, the top red VTL lamps, and the green VTL lamps. The 15 regular actuators are related to the barrier lights, the hydraulic valves, the rotation direction of the bridge, the rotation speed of the bridge, and the bottom red VTL lamps.

The 22 safety sensors are related to the state of the stop signs, the end position of the barriers, the end positions of the bridge, the state of the top red VTL lamps, and the state of the green VTL lamps. The 29 regular sensors are related to the non-end position of the barriers, the non-end positions of the bridge, and the bottom red VTL lamps. The EFAs modelling the GUI are not shown in the table. These are all included in the regular part.

6.6 Validation of the results

The results achieved by applying the proposed method of splitting the supervisor are validated in two ways.

First, the input image and output image partitioning is compared to the manually defined input image 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 2. 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 in the same manner as was chosen manually (the main-diagonal elements). For the outputs, the partitioning was identical. For the inputs, only four variables differ. Here, four of the safety inputs in the manual split were determined to be regular according to the presented method.

Table 2 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

Upon closer inspection of the controller code, none of these inputs were used in the safety part. Therefore, there was no reason to make them safety sensors. After discussing this with the engineers, it was concluded that they should have been implemented as regular inputs. In that case, exactly the same result is obtained. It is concluded that the method achieves the desired partitioning for the OBB.

Second, the split supervisor is used to generate controller code (for a Siemens SIMATIC S7-300 CPU315F-2 PN/DP SPLC) that has been implemented on the real bridge. Function block diagrams (FBDs) are used to implement the safety supervisor and structured text is used to implement the regular supervisor. For generating FBDs, the method as shown in Reijnen et al. (2020a) is used. The size of the generated code is around 100 KB, which is only a small part of the available memory on the PLC (2 MB).

To realize a system test, the OBB was closed for traffic for one night. During this period, the original controller code has been replaced by the generated code. During the system test, the bridge was operated by professional bridge operators. The operators did not experience differences between the generated controller and the original controller.

To validate the behavior, existing site acceptance tests (SATs) have been performed. A SAT protocol describes how the system should react under a specific circumstance. The performed tests are categorized into two groups: 1) does the supervisory controller restrict unsafe behavior (i.e., are all safety requirements satisfied) and 2) does the supervisory controller perform required behavior (i.e., the way of working). Of course, the supervisory control satisfied all safety requirements and did not reach blocking situations, as that is guaranteed by the synthesis algorithm. During the tests no anomalies were observed and all tests were successfully performed.

Problems related to the differences between a supervisor and a supervisory controller, as described in Subsection 3.4, were negligible in this case study, as the control hardware is much faster than the process to be controlled.

7 Concluding remarks

This paper generalizes the supervisor splitting method of Reijnen et al. (2020a) by extending it to extended finite-state automata and event-condition requirements. Two modelling guidelines are given that help in obtaining a model which, subsequently, results in a favorable splitting. The splitting method is based on the partitioning of the requirement model into safety requirements and regular requirements. The splitting as proposed allows for a supervisor to be implemented on an SPLC.

It is proven 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. The advantage of the proposed method is that the partitioning of the input and output images is done in a systematic manner which can be automated. Finally, implementing the supervisor on a real bridge shows a proof of concept for practical applicability of this way of working.

While the use of synthesis techniques has not yet been widespread in industry, this case study shows that they are mature enough to be applicable in practice and can be embedded in an engineering process delivering a ready to use implementation. In case safety standards have to be met, the presented method can be used to generate controller code for SPLCs from the synthesis result.

Acknowledgements The project described in this paper was supported by Rijkswaterstaat, part of the Ministry of Infrastructure and Water Management of the Government of the Netherlands. We thank Han Vogel, Maria Angenent, Stijn ten Pas, Anne van der Heiden for their involvement in this project, John van Dinther, Joep Schatorjé, Roel Driessens for their help on implementing the supervisor on the real bridge, and Pascal Etman for the encouraging coordination of project activities.

Declarations

Conflict of Interests The authors declare that they have no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- Baeten JCM, Van de Mortel-Fronczak JM, Rooda JE (2016) Integration of supervisory control synthesis in model-based systems engineering
- Balemi S, Hoffmann GJ, Gyugyi P, Wong-Toi H, Franklin GF (1993) Supervisory control of a rapid thermal multiprocessor. *IEEE Trans Autom Control* 38(7):1040–1059
- Cai K, Wonham WM (2010) Supervisor localization: a top-down approach to distributed control of discrete-event systems. *IEEE Trans Autom Control* 55(3):605–618
- Cassandras CG, Lafortune S (2009) *Introduction to Discrete Event Systems*. Springer Science + Business Media
- Chen YL, Lin F (2000) Modeling of discrete event systems using finite state machines with parameters. In: *Proceedings of the 2000 Conference on Control Applications*, IEEE, pp941–946
- Cheng KT, Krishnakumar AS (1996) Automatic generation of functional vectors using the extended finite state machine model. *ACM Trans Des Autom Electron Syst* 1(1):57–79
- Darvas D, Majzik I, Viñuela EB (2016) Formal verification of safety PLC based control software. In: *Proceedings of the 12th Conference on Integrated Formal Methods*, Springer, pp 508–522
- de Queiroz MH, Cury JER (2000) Modular control of composed systems. In: *Proceedings of the 2000 American Control Conference*, IEEE, 6, pp 4051–4055
- European Commission (2006) Machinery directive. *Official Journal of the European Union* L 157:24–86
- Fabian M, Hellgren A (1998) PLC-based implementation of supervisory control for discrete event systems. In: *Proceedings of the 37th Conference on Decision and Control*, IEEE, vol 3, pp 3305–3310
- Forschelen STJ, Van de Mortel-Fronczak JM, Su R, Rooda JE (2012) Application of supervisory control theory to theme park vehicles. *Discrete Event Dynamic Systems* 22(4):511–540
- Goorden MA, van de Mortel-Fronczak JM, Reniers MA, Fokkink WJ, Rooda JE (2019) Modeling guidelines for component-based supervisory control synthesis. In: *Proceedings of the Conference on Formal Aspects of Component Software*, Springer, pp 3–24
- Huang Y, Seck MD, Verbraeck A (2015) Component-based light-rail modeling in discrete event systems specification. *Simulation* 91(12):1027–1051
- IEC 61131-3 (2013) IEC 61131-3: Programmable Controllers – Part 3: Programming Languages (3rd). Standard, International Electrotechnical Commission
- IEC 62061 (2005) IEC 61508: Safety of machinery – Functional safety of safety-related electrical, electronic and programmable electronic control system. Standard, International Electrotechnical Commission
- Khan A, Thönnessen D, Fabian M (2019) On-the-fly conformance testing of safety PLC code using QuickCheck. In: *Proceedings of the 17th International Conference on Industrial Informatics*, IEEE, vol 1, pp 419–424
- Kovács G, Piétrac L, Bálint K (2012) A component-based approach for supervisory control. In: *Proceedings of 20th Mediterranean Conference on Control & Automation*, IEEE, pp 800–805

- Ljungkrantz O, Åkesson K, Yuan C, Fabian M (2012) Towards industrial formal specification of programmable safety systems. *IEEE Trans Control Syst Technol* 20(6):1567–1574
- Ma C, Wonham WM (2006) Nonblocking supervisory control of state tree structures. *IEEE Trans Autom Control* 51(5):782–793
- Malik P (2003) From supervisory control to nonblocking controllers for discrete event systems. PhD thesis, Universität Kaiserslautern
- Malik R, Åkesson K, Flordal H, Fabian M (2017) Supremica—an efficient tool for large-scale discrete event systems. *IFAC-PapersOnLine* 50(1):5794–5799
- Markovski J, van Beek DA, Theunissen RJM, Jacobs KGM, Rooda JE (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
- Miremadi S, Åkesson K, Lennartson B (2008) Extraction and representation of a supervisor using guards in extended finite automata. In: *Proceedings of Workshop on Discrete Event Systems, IEEE*, pp 193–199
- Ouedraogo L, Kumar R, Malik R, Åkesson K (2011) Nonblocking and safe control of discrete-event systems modeled as extended finite automata. *IEEE Transactions on Automation Science and Engineering* 8(3):560–569
- PLCopen (2018) Safety software technical specifications – Part 1: concepts and function blocks (2nd). Standard, PLCopen
- Prenzel L, Provost J (2018) PLC implementation of symbolic, modular supervisory controllers. *IFAC-PapersOnLine* 51(7):304–309
- Ramadge PJ, Wonham WM (1987) Supervisory control of a class of discrete event processes. *SIAM J Control Optim* 25(1):206–230
- Reijnen FFH, Goorden MA, van de Mortel-Fronczak JM, Reniers MA, Rooda JE (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 FFH, Hofkamp AT, Reniers MA, van de Mortel-Fronczak JM, Rooda JE (2019) 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 FFH, Erens TR, van de Mortel-Fronczak JM, Rooda JE (2020a) Supervisory control synthesis for safety PLCs. In: *Proceedings of the 15th Workshop on Discrete Event Systems, IFAC*, pp 151–158
- Reijnen FFH, Goorden MA, Van de Mortel-Fronczak JM, Rooda JE (2020b) Modeling for supervisor synthesis – a lock-bridge combination case study. *Discrete Event Dynamic Systems* 30(3):499–532
- Roussel JM, Giua A (2005) Designing dependable logic controllers using the supervisory control theory. *IFAC Proceedings* 38(1):56–61
- Sköldstam M, Åkesson K, 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
- Smith DJ, Simpson KG (2020) *The Safety Critical Systems Handbook*. Butterworth-Heinemann
- Soliman D, Frey G (2011) Verification and validation of safety applications based on plcopen safety function blocks. *Control Engineering Practice* 19:929–946
- Swartjes L, van Beek DA, Reniers MA (2014) Towards the removal of synchronous behavior of events in automata. *IFAC Proceedings* 47(2):188–194
- Thönnessen D, Smallbone N, Fabian M, Claessen K, Kowalewski S (2019) Testing safety PLCs using QuickCheck. In: *Proceedings of the 15th International Conference on Automation Science and Engineering, IEEE*, pp 1–6
- van Beek DA, Fokkink WJ, Hendriks D, Hofkamp AT, Markovski J, van de Mortel-Fronczak JM, Reniers MA (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
- Wonham WM, Cai K (2019) *Supervisory Control of Discrete-Event Systems*. Springer
- Zaytoon J, Riera B (2017) Synthesis and implementation of logic controllers—a review. *Annu Rev Control* 43:152–168

Ferdie F. H. Reijnen received the M.Sc. degree in mechanical engineering from Eindhoven University of Technology, Eindhoven, The Netherlands, in 2016, and the Ph.D. degree in mechanical engineering from Eindhoven University of Technology, Eindhoven, The Netherlands, in 2020. His research interests include discrete-event system modeling, supervisory control synthesis, and implementation of supervisory controllers.

Toby R. Erens received the M.Sc. degree in mechanical engineering from Eindhoven University of Technology, Eindhoven, The Netherlands, in 2019. Since 2019, he has been at the Central Information Technology, Rijkswaterstaat, Delft. His research interests include industrial automation.

Joanna M. van de Mortel-Fronczak received the M.Sc. degree in computer science from AGH University of Science and Technology, Cracow, Poland, in 1982 and the Ph.D. degree in computer science from Eindhoven University of Technology, Eindhoven, The Netherlands, in 1993. Since 1997, she has been with the Department of Mechanical Engineering, Eindhoven University of Technology. Her research interests include model-based engineering and synthesis of supervisory control systems.

Jacobus E. Rooda received the M.Sc. degree from Wageningen University of Agricultural Engineering, Wageningen, The Netherlands, and the Ph.D. degree from Twente University, Enschede, The Netherlands. Since 1985, he has been a Professor of (Manufacturing) Systems Engineering at the Department of Mechanical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands. Since 2010, he is a Professor Emeritus. He is still active in the research fields of engineering design for industrial systems, and of supervisory control thereof.