Discrete Event Dynamic Systems manuscript No.
(will be inserted by the editor)

Modeling for Supervisor Synthesis – A Lock-bridge Combination Case Study

Ferdie F.H. Reijnen · Martijn A. Goorden · Joanna M. van de Mortel-Fronczak · Jacobus E. Rooda

Received: date / Accepted: date

Abstract Designing supervisory controllers for high-tech systems is becoming increasingly complex due to demands for verified safety, higher quality and availability, and extending functionality. Supervisor synthesis is a method to automatically derive a supervisor from a model of the plant and a model of the control requirements. While supervisor synthesis is an active research topic, only a few reports exist on industrial applications. One of the reasons for this is the lack of acquaintance of control engineers with modeling and specifying in the framework of automata. In addition to this, there are no clear guidelines for obtaining the necessary models for synthesis. In this paper, we describe a general way of modeling for the plant and the requirements in order to contribute towards the acceptance of supervisor synthesis in industry. This way of modeling is illustrated with an industrial case study in which a supervisory controller is synthesized for the Algera complex. The Algera complex consists of a waterway lock and a movable bascule bridge. The supervisor has to control 80 actuators based on the observations from 96 discrete sensors, and subjected to 63 control commands available from the operator. We show how to model the plant as a collection of extended finite-state automata, how to model the requirement as a collection of event conditions, how to synthesize the monolithic supervisor, and how to validate the resulting supervisor using continuous-time simulation.

Keywords Supervisor synthesis · Extended finite-state automata · Industrial application · Infrastructural systems

1 Introduction

High-tech systems have become increasingly complex due to the high demands from the market in terms of functionality, quality, and safety. As a result, supervisory controllers (or supervisors) for these systems are getting more complex as well. At the same time, for

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

F.F.H. Reijnen · M.A. Goorden · J.M. van de Mortel-Fronczak · J.E. Rooda
Department of Mechanical Engineering, Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
E-mail: F.F.H.Reijnen@tue.nl

the development process, it is desired to decrease time-to-market and costs. Model-based development methods can help in overcoming these difficulties. The use of formal models has advantages over the traditional engineering process. Models can help in verifying and validating the supervisor design early in the process, resulting in a reduction of design errors found during the testing and integration phase, where error repair is more time-consuming and costly.

Supervisor synthesis of Ramadge and Wonham (1987) is a method to automatically derive a minimally restrictive supervisor from a discrete-event model of the plant (what can the system do) and a model of the control requirements (what may the system do). The synthesized supervisor controls the system, by disabling controllable events, such that the controlled system adheres to the requirements by construction, and is guaranteed to be nonblocking.

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 Zaytoon and Riera (2017) and Grigorov et al. (2011) 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); Pena et al. (2016). Other application domains 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. 2014), 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. 2016), and driver assistance systems (von Bochmann et al. 2015; 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. 2014). For this application, 3 actuators are controlled based on observations of 8 sensors, resulting in a plant state space of 5×10^4 states. 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 8 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, 6 actuators are controlled by a supervisor based on the observations of 11 sensors. The plant state space of the theme park vehicle is 1.7×10^{10} states. The requirement models are 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), Göbe et al. (2016), and Theunissen (2015). 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 efficient. 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 an event-condition models of the requirements. When comparing the two 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 requirement model can be derived.

The contribution of this paper is twofold. Firstly, it proposes guidelines to obtain the plant model and the requirement 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 large industrial system.

For modeling the plant, we choose the abstraction level of inputs and outputs of the control unit. First of all, because the models can be used for the generation of implementation code. Secondly, because 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 requirement models that relate 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 actuator based on the observations from 96 sensors, and is subjected 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, see Reijnen et al. (2019).

This paper is structured as follows. Section 2 describes the Algera complex. In Section 3, the preliminaries of discrete-event system modeling, requirement modeling, and supervisor synthesis are provided. In Section 4, 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 5. Section 6 discusses how the synthesized supervisor is validated. Finally, Section 7 concludes this paper.

2 Case study: the Algera complex

The Algera complex, shown in Fig. 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.

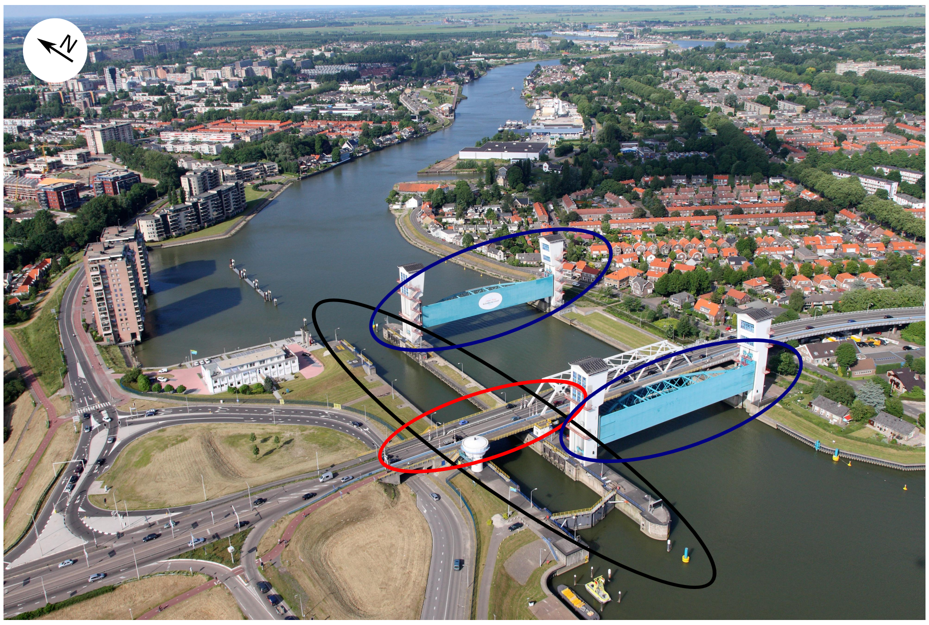


Fig. 1 The Algera complex consisting of a lock (encircled in black), a bascule bridge (encircled in red), and two storm surge barriers (encircled in blue) [https://beeldbank.rws.nl, Rijkswaterstaat / Joop van Houdt]

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 inlands. 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 for vessels that are too high to pass under the storm surge barriers. By opening the bridge, 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 in this paper.

2.1 Description and functionality of the Algera lock

The Algera lock, schematically depicted in Fig. 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.

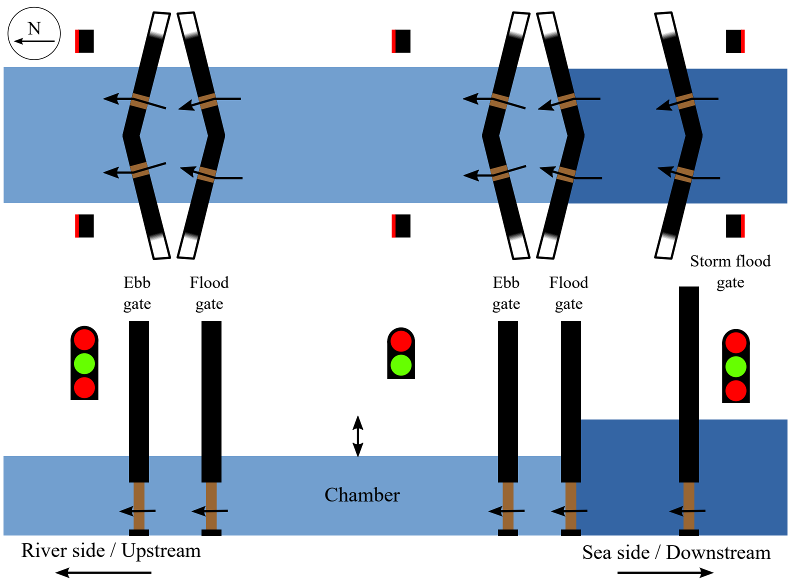


Fig. 2 A schematic representation of the Algera lock when all gates are closed, all paddles are open, and the sea side is the high-water side. View from above (top) and view from the side (bottom)

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 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 Fig. 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.

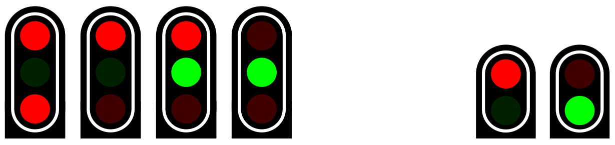


Fig. 3 The aspects of the lock traffic light: double-red, red, red-green, and green (left), and the aspects of the bridge traffic light: red and green (right)

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. This traffic light can display a red or green aspect, shown on the right-hand side of Fig. 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.

2.1.1 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 Fig. 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.

2.2 Description and functionality of the Algera bridge

The Algera bridge, schematically depicted in Fig. 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.

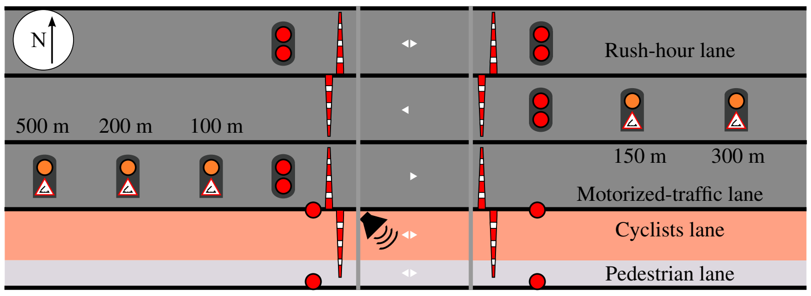


Fig. 4 A schematic representation of the Algera bridge. Orange lights represent approach signs and red lights represent stop signs. White triangles display the traffic direction

To warn motorized traffic in advance, approach signs are positioned before the bridge, visualized by orange circles in Fig. 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 Fig. 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.

2.2.1 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.

2.3 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 Fig. 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 Fig. 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.

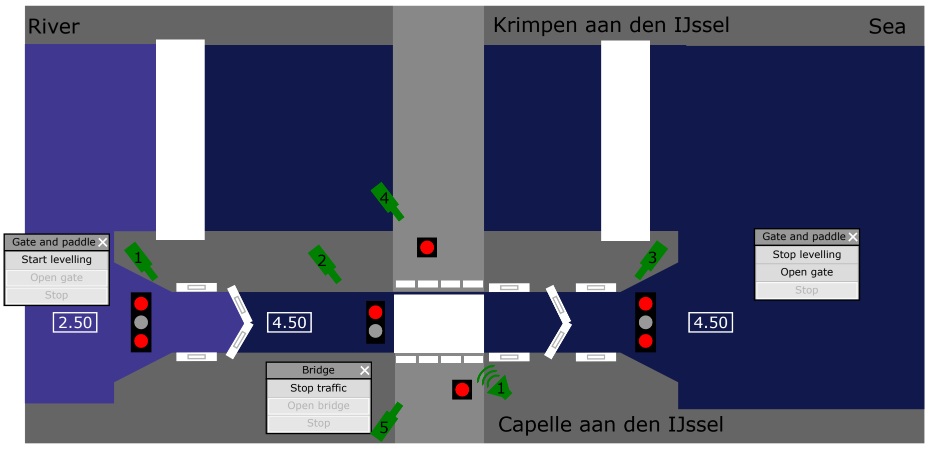


Fig. 5 Part of the graphical user interface of the Algera complex

3 Preliminaries

In this section, the concepts and notations of supervisory control theory used in this paper are summarized. First, modeling of discrete-event systems is discussed. Second, model of requirements is discussed. Finally, supervisor synthesis and implementation of supervisors is discussed.

3.1 Modeling of discrete-event systems

In the context of supervisor synthesis, systems are usually modeled by (extended) finite-state automata (FAs) or Petri nets. Both formalisms are used to represent event-driven behavior. In this paper, extended finite-state automata (EFAs) (Sköldstam et al. 2007) are used as the modeling formalism. First FAs are introduced, followed by EFAs.

3.1.1 Finite-state automata

An FA is formally defined as a 5-tuple:

$$A = (L, \Sigma, \delta, l_0, L_\mathrm{m}) \tag{1}$$

where L is a finite set of locations, Σ a finite set of events, $\delta \subseteq L \times \Sigma \times L$ the transition relation, $l_0 \in L$ the initial location, and $L_\mathrm{m} \subseteq L$ the set of marked locations. 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. The number of states in an FA is equal to the number of locations that can be reached from the initial location, usually equal to $|L|$.

For large systems, it is not feasible to model their behavior by a single FA, as the state space is often too large. Instead, a system can be modeled as a set of several interacting automata A_i (referred to as component models). The behavior of the combined set of automata is given by the synchronous product $A = A_1 \parallel \ldots \parallel A_n$ (Cassandras and Lafortune 2008), which requires simultaneous execution of transitions labeled by the same event.

Automata can be displayed graphically as well. Here, a (labeled) circle denotes a locations, an unconnected incoming arrow indicates the initial location, and a filled circle indicates a marked locations. Controllable and uncontrollable events are visualized by a (labeled) solid and dashed arrows, respectively. An example is shown in Fig. 6. The right-hand side automaton is the synchronous product of the other two automata. The automata synchronize over the `1to2` event.

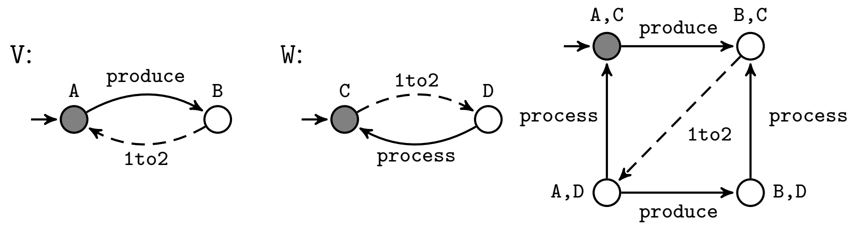


Fig. 6 Graphical representation of FAs (left and middle) and their synchronous product (right)

3.1.2 Extended finite-state automata

In Chen and Lin (2000) and Sköldstam et al. (2007) extended-finite state automata are used for modeling systems. EFSs are FAs parameterized by bounded discrete variables. Formally, an EFA is defined as $E = (L, V, \Sigma, \rightarrow, l_0, v_0, L_\mathrm{m})$, where V is the set of variables with initial valuation v_0, $\rightarrow$ the extended transition function, and the other elements equal to those for FAs. For an EFA, a state is a combination of a location and a valuation of the variables. The number of states in an EFA is equal to the number of combinations of locations and variable valuations that can be reached from the initial location. In this paper, we only consider Boolean variables, but in general bounded integers can also be used. Furthermore, EFAs extend the transition relation with guard expressions (conditions) and variable assignments (updates). Formally, the transition relation is defined as $\rightarrow \subseteq L \times C \times \Sigma \times U \times L$, where C is the set of all conditions and U is the set of all updates. A transition is only enabled when the associated condition evaluates to true. Whenever a transition is taken, some of the variables may be updated. A condition is defined by the following grammar in Backus-Naur Form:

$$\langle\text{cond}\rangle ::= \mathbf{T} \mid \mathbf{F} \mid v_b \mid v_l \mid \langle\text{cond}\rangle \wedge \langle\text{cond}\rangle \mid \langle\text{cond}\rangle \vee \langle\text{cond}\rangle \mid \neg\langle\text{cond}\rangle \tag{2}$$

where **T** and **F** are true and false, respectively, v_b is a Boolean variable, v_l is a location variable, $\wedge$ is the AND operator, $\vee$ is the OR operator, and $\neg$ is the NOT operator. A location variable is a reference to a location, denoted by <automaton>.<location>. It evaluates to **T** if and only if the automaton is in that location.

An update consists of zero or more variable assignments of the form $v_b := c$, where ':=' denotes an assignment of the value of c to variable v_b. It is not allowed for an update to have multiple assignments for the same variable.

Two EFAs can be combined by computing the synchronous product. Let $E^k = (L^k, V^k, \Sigma^k, \rightarrow^k, l_0^k, v_0^k, L_{\mathrm{m}}^k)$, $k = 1,2$ be EFAs. The synchronous product of E^1 and E^2 is

$$E^1 \parallel E^2 = (L^1 \times L^2, V^1 \times V^2, \Sigma^1 \cup \Sigma^2, \rightarrow, (l_0^1, l_0^2), (v_0^1, v_0^2), L_{\mathrm{m}}^1 \times L_{\mathrm{m}}^2) \tag{3}$$

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

- $((l_1^1, l_1^2), c^1 \wedge c^2, \sigma, (u^1, u^2), (l_2^1, l_2^2)) \in\rightarrow$ if $\sigma \in \Sigma^1 \cap \Sigma^2$, there exist $(l_1^1, c^1, \sigma, u^1, l_2^1) \in\rightarrow^1$ and $(l_1^2, c^2, \sigma, u^2, l_2^2) \in\rightarrow^2$, and u^1 and u^2 do not have multiple assignments for the same variable;
- $((l_1^1, l_1^2), c^1, \sigma, u^1, (l_2^1, l_1^2)) \in\rightarrow$ if $\sigma \in \Sigma^1 \setminus \Sigma^2$ and $(l_1^1, c^1, \sigma, u^1, l_2^1) \in\rightarrow^1$;
- $((l_1^1, l_1^2), c^2, \sigma, u^2, (l_1^1, l_2^2)) \in\rightarrow$ if $\sigma \in \Sigma^2 \setminus \Sigma^1$ and $(l_1^2, c^2, \sigma, u^2, l_2^2) \in\rightarrow^2$.

In case u^1 and u^2 have multiple assignments for the same variable, the synchronous product is undefined. A location variable for a location $l' \in L^1 \cup L^2$ in a transition condition is substituted by **T** if for that transition $l' = l_1^1$ or $l' = l_1^2$, and to **F** otherwise.

An example of synchronization is shown in Fig. 7. The keywords **when** and **do** are used to denote conditions and updates, respectively. In this example, the event `lamp_on` is only enabled when automaton `X` is in location `Pushed`. `Q` is a Boolean variable. The synchronous product is shown on the right-hand side. Transitions with condition **F** are not displayed.

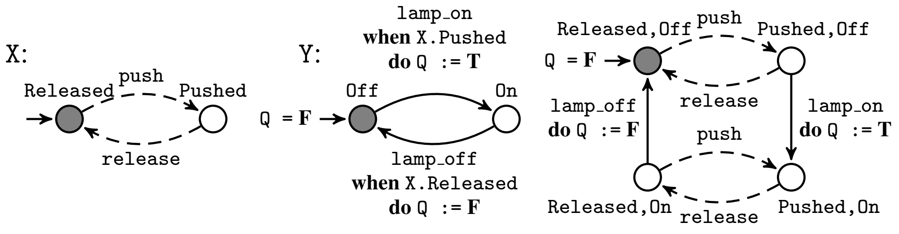


Fig. 7 Graphical representation of EFAs (left and middle) and their synchronous product (right)

3.2 Modeling of requirements

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

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 $\mathbf{T}$. A condition is defined as in (2). Event-condition requirements 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 requirement is as shown in Fig. 8.

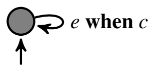


Fig. 8 EFA representation of an event-condition requirements e needs c

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 states where Y evaluates to $\mathbf{F}$ are prohibited. A condition is defined as in Equation 2. The synchronous product of an EFA and a state-invariant requirement can be computed by removing in the EFA all transitions to states where Y evaluates to $\mathbf{F}$.

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 control requirements, a supervisor can be synthesized automatically. The supervisor restricts the behavior of the system such that the following properties are always satisfied:

Safety The system cannot reach states or enable events that are forbidden by the requirements.
Controllability Only controllable events are restricted by the supervisor.
Nonblockingness The system is always able to reach a marked state.
Maximal permissiveness The supervisor imposes the minimal restriction on the system to satisfy safety, controllability, and nonblockingness.

By applying monolithic supervisor synthesis (see, e.g., Ouedraogo et al. (2011)), a single supervisor is synthesized to control the plant. Traditionally, a single (E)FA is returned that represents this supervisor.

As an example, consider automata `V` and `W` from Fig. 6 and requirement `R`: '`process` needs `A`'. The synchronous product of `V`, `W`, and `R` is shown on the left-hand side of Fig. 9. As can be seen, location (`B,D`) is a blocking state. Applying monolithic supervisor synthesis on `V` $\|$ `W` and `R` results in the right-hand side automaton in Fig. 9. As can be seen, event `produce` is disabled in state (`A,D`), to resolve the blocking issue.

For large state spaces, returning a supervisor represented by a single automaton 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 then an EFA with a single location and for each controllable event in the plant a selfloop with the derived guard. The derived guards can

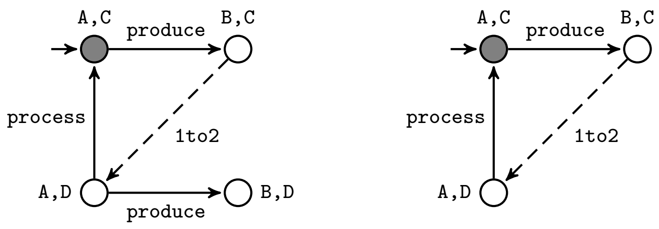


Fig. 9 Synchronous product of V, W, and R (left) and a supervisor synthesized for V || W and R (right)

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 the right-hand side supervisor from Fig. 9. Instead of returning the EFA in Fig. 9, the method of Miremadi et al. (2011) returns the EFA as shown in Fig. 10. The supervisor is now represented by V || W || R || G.

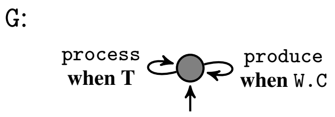


Fig. 10 The synthesis result for V || W and R, using the the method of Miremadi et al. (2011)

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 result in no extra guards. 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.

3.4 Implementation of supervisors

One of the main motivations of using supervisor synthesis is the possibility to generate implementation code from the synthesized supervisor. This allows the supervisor to be implemented on, e.g., a PLC controller or a microcontroller. Tools exist that can automatically generate controller code given a discrete-event model of the supervisor. Examples of such tools are CompileDES for libFAUDES (Moor et al. 2008), Supremica (Malik et al. 2017; Prenzel and Provost 2018), and CIF (van Beek et al. 2014). A detailed description of code generation from discrete-event models is provided in Zaytoon and Riera (2017).

4 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. Secondly, 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 vehicles (Forschelen et al. 2012), a patient support table for an MRI scanner (Theunissen et al. 2014), a FESTO production line (Reijnen et al. 2018), 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. Subsequently, a method for obtaining a plant model is given. Finally, different types of textual formats are discussed that can straightforwardly be modeled.

4.1 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 has been discussed. Here, 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 paper, we refer to physical relation models instead of interaction models.

In supervisory control papers, e.g., Balemi et al. (1993) and Roussel and Alessandro (2005), it has been 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 has been 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.

4.2 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).

4.2.1 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 Fig. 11, examples of models for Boolean signals are shown.

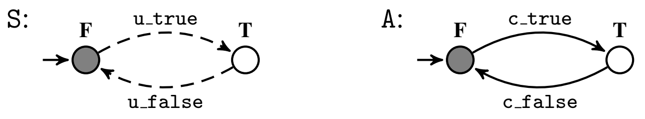


Fig. 11 Component models for a Boolean input (left) and a Boolean output (right)

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 5.1.3.

4.2.2 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 requirements 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 Fig. 12. The component model is as shown on the right-hand side of Fig. 12.

Value	State	Abbreviation
0 - 19	Low	L
20 - 80	Nominal	N
81 - 100	High	H

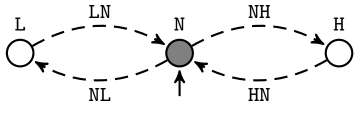


Fig. 12 Mapping between signal value and states (left) and component model for the sensor (right)

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 Fig. 13. The component model is as shown on the right-hand side of Fig. 13.

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

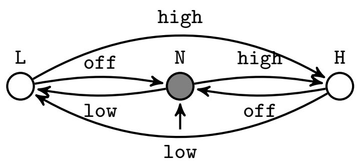


Fig. 13 Mapping between states and signal value (left) and component model for the actuator (right)

4.2.3 Physical relation models

Aside from modeling the behavior of the individual components, the physical relations between the different components need to be modeled as well. These relations are referred to as interaction models in Gössler and Sifakis (2005). 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.

An an example, consider the filling pump and the water-height sensor from Section 4.2.2. 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 Fig. 14, this physical relation model is shown. Here, S and A refer to the previous sensor and actuator model, respectively.

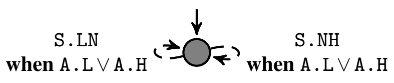


Fig. 14 The physical relation model

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 6.

4.3 The requirement 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 requirement 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. (2018), has shown that in general 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.

4.3.1 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 3.2. A *may only* requirement is modeled as in (4), whereas a *may not* requirement is modeled as in (5).

$$\texttt{component.event} \ \textbf{needs} \ \texttt{condition} \tag{4}$$

$$\texttt{component.event} \ \textbf{needs} \ \neg\texttt{condition} \tag{5}$$

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.

4.3.2 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 Fig. 15.

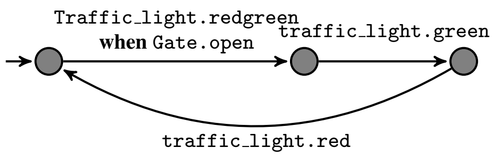


Fig. 15 Example of an event-order requirement

4.3.3 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:

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

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

To model a timer-based requirement, a timer 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 Fig. 16. The requirement is modeled as shown in (6). Notice that in this model the time is not explicitly modeled. This is included later, see Section 6.1.

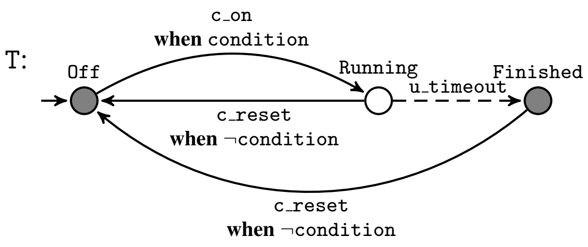


Fig. 16 Model of a timer for condition `condition`

$$\texttt{component.event}\ \textbf{needs}\ \texttt{condition}\ \wedge \texttt{T.Finished} \tag{6}$$

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.

4.3.4 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 notions from a plant model as: 'not (`gate1.sensor.open` and `gate2.sensor.open`)'.

5 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 4. In Subsection 5.1, the templates for the plant models are introduced. The plant and requirement models for the Algera lock and the Algera bridge are described in Subsections 5.2 through 5.6. In Subsection 5.7, the synthesis result is discussed.

5.1 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, see Reijnen et al. (2019). 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 a total of 16 templates are used to represent the behavior of 176 actuators and sensors and 63 commands. The templates are provided in the subsequent subsections.

5.1.1 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 Fig. 17, 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 modeled as the bottom EFA in Fig. 17.

5.1.2 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`, `Rest`, and `Opening` behavior is included in the actuator template, shown in the upper left of Fig. 18. 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

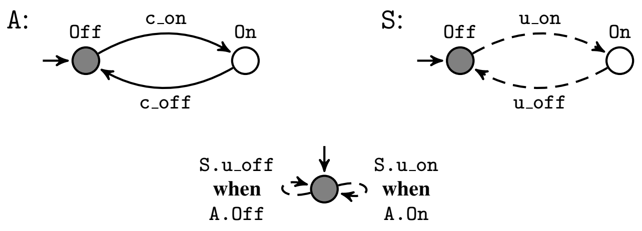


Fig. 17 Template of single output actuator A (left), single input sensor S (right), and the actuator-sensor physical relations (bottom)

modeled as the sensor from Fig. 17. There is a physical restriction that both sensors cannot be on simultaneously; this is modeled as the upper-right automaton in Fig. 18. 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 Fig. 18.

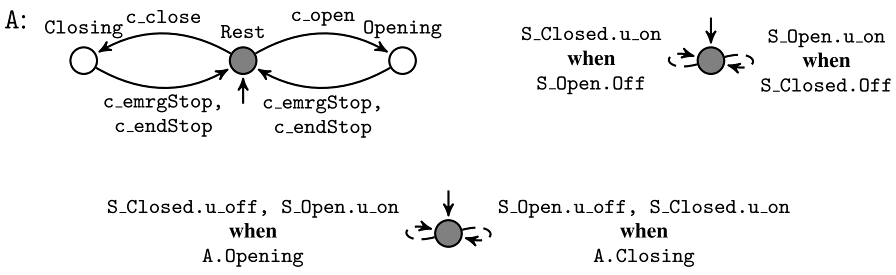


Fig. 18 Template of double output actuator A (left), the sensor-sensor physical relations (right), and the actuator-sensor physical relations (bottom)

5.1.3 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 Fig. 19 and Fig. 20, respectively. The template for the sensors is similar to the sensor template in Fig. 17 (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 Fig. 19 and Fig. 20, respectively.

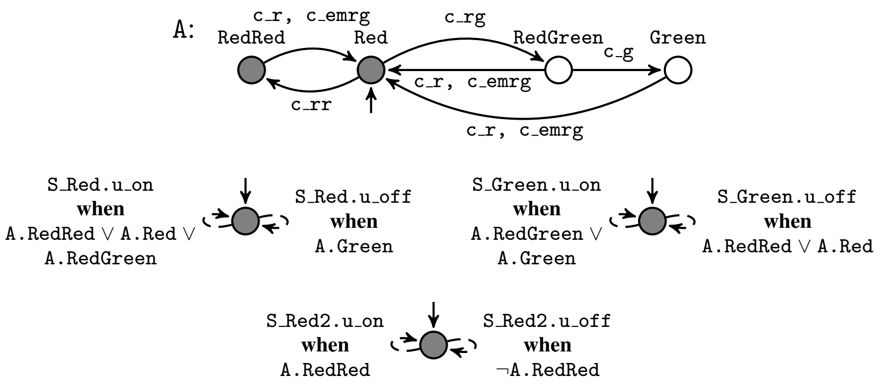

A:
RedRed
Red
RedGreen
Green
c_r, c_emrg
c_rg
c_g
c_r, c_emrg
c_rr
c_r, c_emrg
S_Red.u_on when A.RedRed ∨ A.Red ∨ A.RedGreen
S_Red.u_off when A.Green
S_Green.u_on when A.RedGreen ∨ A.Green
S_Green.u_off when A.RedRed ∨ A.Red
S_Red2.u_on when A.RedRed
S_Red2.u_off when ¬A.RedRed

Fig. 19 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)

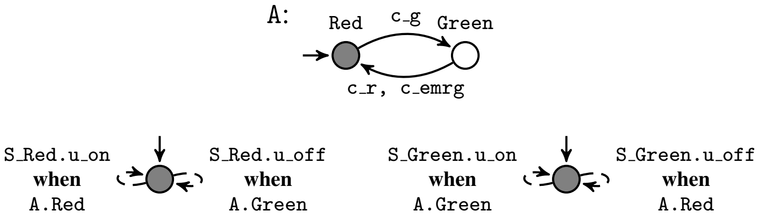


Fig. 20 Template of the bridge traffic light actuator `A` (top), the red lamp sensor-actuator physical relations (bottom left), and the green lamp sensor-actuator physical relations (bottom right)

5.1.4 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 Fig. 21. 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 Fig. 21. The commands for the lock and bridge traffic light are modeled on the bottom left-hand side and bottom right-hand side of Fig. 21, respectively.

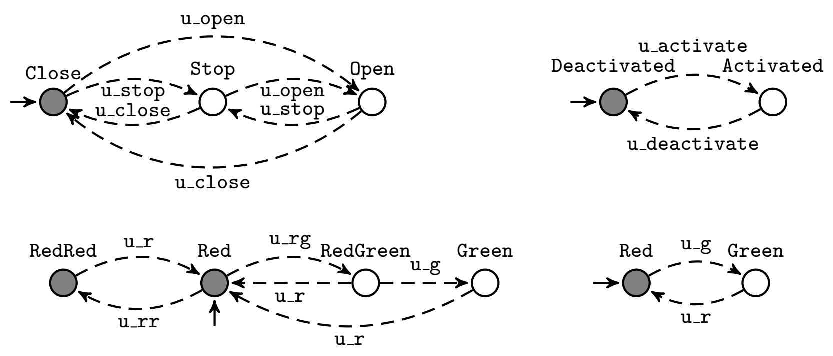


Fig. 21 Template for the movable components commands (top left), for the emergency stop (top right), for the lock traffic light commands (bottom left), and for the bridge traffic light commands (bottom right)

5.2 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 1, for each component, the model template, the number of instantiations, and the number of states are given. The number of states in the synchronous product of all the component models equals 1.2×10^{34}.

5.3 Requirement 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, see Reijnen et al. (2019)), 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 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 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, 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 4.3. 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.

There are two advantages for 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

Table 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

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.

5.4 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. Table 3 lists, for each component, the model template, the number of instantiations, and the number of states. The number of states in the synchronous product of all the component models equals 1.9×10^{23}.

Table 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

5.5 Requirement 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, see Reijnen et al. (2019)), are listed below. Furthermore, Requirements 8-10 from Section 5.3 are also imposed on the bridge, but are not repeated in this subsection, 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 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 4.3.

Table 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 Rest

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

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

In Table 5, these requirements are defined formally.

Table 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`

5.7 Supervisor synthesis

A supervisor has been synthesized from the plant and requirement models. For synthesis, the CIF 3 toolset (van Beek et al. 2014) has been used. The synthesis algorithm implemented in CIF 3 is based on the algorithm proposed in Ouedraogo et al. (2011). The implementation of the synthesis algorithm in CIF 3 uses the (BDD-based) method of Miremadi and Lennartson (2016) 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 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. 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 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, see Reijnen et al. (2019)), 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 5.3). 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.

6 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.

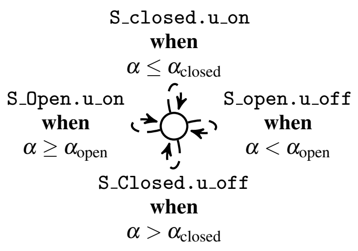
While it is possible to simulate the discrete-event model, it is easier to validate the behavior using a more advanced simulation model in which hybrid behavior is included. For this, the discrete-event plant model is enriched with continuous behavior, using hybrid automata as defined in Henzinger (2000).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

6.1 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 5.1.2, the hybrid model is shown in Fig. 22. 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.

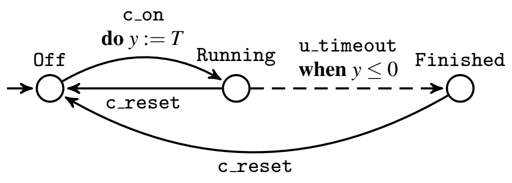
The discrete-event model of the timer, see Section 4.3.3, is extended with continuous behavior as well. The hybrid model is shown in Fig. 23. On the left-hand side, the automaton model is shown. A continuous variable y is introduced to represent the remaining time of the timer. On the right-hand side, 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`


S_closed.u_on
when
$\alpha \leq \alpha_{\text{closed}}$
S_Open.u_on
when
$\alpha \geq \alpha_{\text{open}}$
S_open.u_off
when
$\alpha < \alpha_{\text{open}}$
S_Closed.u_off
when
$\alpha > \alpha_{\text{closed}}$

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

Fig. 22 Hybrid model of the two input - two output physical relation. `A.` denotes a reference to actuator `A`

updates (denoted by the keyword **do**) the value of y to the desired timer value T. When $y \leq 0$, a transition to the `Finished` state occurs.



$$\frac{dy}{dt} = \begin{cases} -1 & \textbf{if } \texttt{Running} \\ 0 & \textbf{otherwise} \end{cases}$$

Fig. 23 Hybrid model of a timer with duration T

6.2 Visualization

The hybrid plant model is connected to a visualization of the system, shown in Figs. 2, 4, and 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.

6.3 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 component.

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 scenario's (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 standards that Rijkswaterstaat requires for their controllers. Furthermore, for Rijkswaterstaat, it provides an analysis of the completeness of their requirements. In other words, does the set of specified requirements lead 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 5.3). In the meantime, Rijkswaterstaat has added this requirement to their set of safety requirements.

6.4 Discussion

While simulating the test scenario's 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 interesting to determine if the actuators are always allowed to 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.

7 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 paper, guidelines for obtaining the plant and the requirement 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 input 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 requirement 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 automata models are used.

Simulation-based visualization is used to validate the resulting supervisors. Simulation allows to compare the behavior of the supervisor versus 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 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 systems of an industrial scale. 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.

Acknowledgements We thank the following Rijkswaterstaat employees: Maria Angenent, Gerrit Bruggink, Leendert-Jan Deurloo, John van Dinther, Robert de Roos, Bert van der Vegt, and Han Vogel for the support they provided for this project. We thank Bart van Willigen for assisting with the case study.

References

Atampore F, Dingel J, Rudie K (2016) Automated service composition via supervisory control theory. In: Proceedings of Workshop on Discrete Event Systems, IEEE, pp 28–35

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

Balemi S, Hoffmann GJ, Gyugyi P, Wong-Toi H, Franklin GF (1993) Supervisory control of a rapid thermal multiprocessor. IEEE Transactions on Automatic Control 38(7):1040–1059

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 Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, pp 575–580

von Bochmann G, Hilscher M, Linker S, Olderog ER (2015) Synthesizing controllers for multi-lane traffic maneuvers. In: Proceedings of International Symposium on Dependable Software Engineering: Theories, Tools, and Applications, Springer, pp 71–86

Brandin BA (1996) The real-time supervisory control of an experimental manufacturing cell. IEEE Transactions on Robotics and Automation 12(1):1–14

Cassandras CG, Lafortune S (2008) Introduction to Discrete Event Systems, 2nd edn. Springer US

Chandra V, Kumar R (2002) A event occurrence rules based compact modeling formalism for a class of discrete event systems. Mathematical and computer modelling of dynamical Systems 8(1):49–73

Chandra V, Huang Z, Kumar R (2003) Automated control synthesis for an assembly line using discrete event system control theory. IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) 33(2):284–289

Chen YL, Lin F (2000) Modeling of discrete event systems using finite state machines with parameters. In: Proceedings of Conference on Decision and Control, IEEE, pp 941–946

Crnkovic I (2001) Component-based software engineering - new challenges in software development. Software Focus 2(4):127–133

Fabian M, Fei Z, Miremadi S, Lennartson B, Åkesson K (2014) Supervisory control of manufacturing systems using extended finite automata. Formal Methods in Manufacturing pp 295–314

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

Göbe F, Ney O, Kowalewski S (2016) Reusability and modularity of safety specifications for supervisory control. In: IEEE 21st International Conference on Emerging Technologies and Factory Automation, IEEE, pp 1–8

Gössler G, Sifakis J (2005) Composition for component-based modeling. Science of Computer Programming 55(1-3):161–183

Grigorov L, Butler BE, Cury JE, Rudie K (2011) Conceptual design of discrete-event systems using templates. Discrete Event Dynamic Systems 21(2):257–303

Guillet S, Bouchard B, Bouzouane A (2014) Designing smart homes dedicated to disabled people using modular discrete controller synthesis. In: Proceedings of Workshop on Discrete Event Systems, IFAC, pp 54–59

Hasdemir IT, Kurtulan S, Gören L (2008) An implementation methodology for supervisory control theory. The International Journal of Advanced Manufacturing Technology 36(3):373–385

Henzinger TA (2000) The theory of hybrid automata. In: Verification of Digital and Hybrid Systems, Springer, pp 265–292

Huang Y, Seck MD, Verbraeck A (2015) Component-based light-rail modeling in discrete event systems specification. Simulation 91(12):1027–1051

Kim S, Park J, Leachman RC (2001) A supervisory control approach for execution control of an FMC. International Journal of Flexible Manufacturing Systems 13(1):5–31

Korssen T, Dolk VS, van de Mortel-Fronczak JM, Reniers MA, Heemels WPMH (2018) Systematic model-based design and implementation of supervisors for advanced driver assistance systems. IEEE Transactions on Intelligent Transportation Systems 19(2):533–544

Kovács G, Piétrac L (2009) Multi-face modeling for rapid prototyping of discrete event control systems. In: Proceedings of European Control Conference, IEEE, pp 1463–1468

Kovács G, Piétrac L, Bálint K (2012) A component-based approach for supervisory control. In: Proceedings of Mediterranean Conference on Control & Automation, IEEE, pp 800–805

Lauzon SC, Ma AKL, Mills JK, Benhabib B (1996) Application of discrete-event-system theory to flexible manufacturing. IEEE Control Systems 16(1):41–48

Leduc RJ, Wonham WM (1995) Discrete event systems modeling and control of a manufacturing testbed. In: Proceedings of Canadian Conference on Electrical and Computer Engineering, IEEE, vol 2, pp 793–796

Liao H, Wang Y, Stanley J, Lafortune S, Reveliotis S, Kelly T, Mahlke S (2013) Eliminating concurrency bugs in multithreaded software: A new approach based on discrete-event control. IEEE Transactions on Control Systems Technology 21(6):2067–2082

Ljungkrantz O, Åkesson K, Richardsson J, Andersson K (2007) Implementing a control system framework for automatic generation of manufacturing cell controllers. In: Proceedings of Conference on Robotics and Automation, IEEE, pp 674–679

Lopes YK, Trenkwalder SM, Leal AB, Dodd TJ, Groß R (2016) Supervisory control theory applied to swarm robotics. Swarm Intelligence 10(1):65–97

Ma C, Wonham WM (2006) Nonblocking supervisory control of state tree structures. IEEE Transactions on Automatic Control 51(5):782–793

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 Conference on Decision and Control, IEEE, pp 3481–3486

Miremadi S, Lennartson B (2016) Symbolic on-the-fly synthesis in supervisory control theory. IEEE Transactions on Control Systems Technology 24(5):1705–1716

Miremadi S, Åkesson K, Lennartson B (2011) Symbolic computation of reduced guards in supervisory control. IEEE Transactions on Automation Science and Engineering 8(4):754–765

Moor T, Schmidt K, Perk S (2008) libFaudes–An open source C++ library for discrete event systems. In: Proceedings of Workshop on Discrete Event Systems, IEEE, pp 125–130

Moor T, Schmidt K, Perk S (2010) Applied supervisory control for a flexible manufacturing system. In: Proceedings of Workshop on Discrete Event Systems, IFAC, pp 253–258

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

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

Pena PN, Costa TA, Silva RS, Takahashi RH (2016) Control of flexible manufacturing systems under model uncertainty using supervisory control theory and evolutionary computation schedule synthesis. Information Sciences 329:491–502

Pétin JF, Gouyon D, Morel G (2007) Supervisory synthesis for product-driven automation and its application to a flexible assembly cell. Control Engineering Practice 15(5):595–614

Prenzel L, Provost J (2018) PLC implementation of symbolic, modular supervisory controllers. IFAC-PapersOnLine 51(7):304–309

de Queiroz MH, Cury JER (2002) Synthesis and implementation of local modular supervisory control for a manufacturing cell. In: Proceedings of Workshop on Discrete Event Systems, IEEE, pp 377–382

Ramadge PJ, Wonham WM (1987) Supervisory control of a class of discrete event processes. SIAM journal on control and optimization 25(1):206–230

Rawlings BC, Christenson B, Wassick JM, Ydstie BE (2014) Supervisor synthesis to satisfy safety and reachability requirements in chemical process control. In: Proceedings of Workshop on Discrete Event Systems, IFAC, pp 195–200

Reijnen FFH, Goorden MA, van de Mortel-Fronczak JM, Rooda JE (2017) Supervisory control synthesis for a waterway lock. In: Proceedings of Conference on Control Technology and Applications, IEEE, pp 1562–1568

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 Conference on Control Technology and Applications, IEEE, pp 458–464

Reijnen FFH, Goorden MA, van de Mortel-Fronczak JM, Rooda JE (2019) Models and documentation for the Algera complex, www.github.com/ffhreijnen/AlgeraComplex

Roussel JM, Alessandro G (2005) Designing dependable logic controllers using the supervisory control theory. IFAC Proceedings Volumes 38(1):56–61

Sampath M, Sengupta R, Lafortune S, Sinnamohideen K, Teneketzis D (1995) Diagnosability of discrete-event systems. IEEE Transactions on automatic control 40(9):1555–1575

van der Sanden LJ, Reniers MA, Geilen MCW, Basten AA, Jacobs J, Voeten JPM, Schiffelers RRH (2015) Modular model-based supervisory controller design for wafer logistics in lithography machines. In: Proceedings of Conference on Model Driven Engineering Languages and Systems, IEEE, pp 416–425

Silva DB, Vieira AD, Loures EFR, Busetti MA, Santos EAP (2011) Dealing with routing in an automated manufacturing cell: A supervisory control theory application. International Journal of Production Research 49(16):4979–4998

Sköldstam M, Åkesson K, Fabian M (2007) Modeling of discrete event systems using finite automata with variables. In: Proceedings of Conference on Decision and Control, IEEE, pp 3387–3392

Theunissen RJM (2015) Supervisory control in health care systems. PhD thesis, Eindhoven University of Technology

Theunissen RJM, Petreczky M, Schiffelers RRH, van Beek DA, Rooda JE (2014) Application of supervisory control synthesis to a patient support table of a magnetic resonance imaging scanner. IEEE Transactions on Automation Science and Engineering 11(1):20–32

Vahidi A, Fabian M, Lennartson B (2006) Efficient supervisory synthesis of large systems. Control Engineering Practice 14(10):1157–1167

Wonham WM, Cai K, Rudie K (2018) Supervisory control of discrete-event systems: A brief history. Annual Reviews in Control 45:250–256

Zaytoon J, Carré-Ménéatrier V (2001) Synthesis of control implementation for discrete manufacturing systems. International Journal of Production Research 39(2):329–345

Zaytoon J, Riera B (2017) Synthesis and implementation of logic controllers–A review. Annual reviews in control 43:152–168