Application of supervisory control theory to theme park vehicles

Systems Engineering Group
Department of Mechanical Engineering
Eindhoven University of Technology
PO Box 513
5600 MB Eindhoven
The Netherlands
http://se.wtb.tue.nl/

SE Report: Nr. 2011-05

Application of Supervisory Control Theory to Theme Park Vehicles

S.T.J. Forschelen, J.M. van de Mortel-Fronczak,
R. Su and J.E. Rooda

ISSN: 1872-1567

SE Report: Nr. 2011-05
Eindhoven, June 2011
SE Reports are available via http://se.wtb.tue.nl/sereports

Abstract

Due to increasing system complexity, time-to-market and development costs reduction, new engineering processes are required. Model-based engineering processes are suitable candidates because they support system development by enabling the use of various model-based analysis techniques and tools. As a result, they are able to cope with complexity and have the potential to reduce time-to-market and development costs. Moreover, supervisory control synthesis can be integrated in this setting, which can further contribute to the development of control systems. To evaluate the applicability of recently developed supervisor synthesis techniques and to show how they can be integrated in an engineering process, a theme park vehicle is chosen as a case study. The supervisor synthesized for the theme park vehicle has successfully been implemented and integrated in the existing resource-control platform.[1]

[1]The work described has been performed in cooperation with NBG Industrial Automation B.V., in the framework of Twins 05004 project under the European Community's EUREKA cluster program ITEA 2.

1 Introduction

High-tech companies are often challenged to increase the functionality and quality of a product, while at the same time time-to-market and product costs should be reduced. Current practice shows that this is not straightforward. As a result, there is a need for new engineering processes. The purpose of this paper is to show how the supervisory control theory of [18] can contribute to the development of control systems and how it can be integrated in an engineering process.

Figure 1 shows a schematic overview of a high-tech system with the focus on control. At the bottom, the main structure is depicted that usually contains mechanical parts. Sensors and actuators are mounted on these mechanical parts to monitor their position or state and to actuate them. The sensor signals have to be processed and the actuators have to be controlled with feedback control to assure that they reach the desired position in a desired way. This happens at the resource control level. Above the resource control level, supervisory control is depicted. It coordinates the individual components and gives the desired functionality to the whole system.

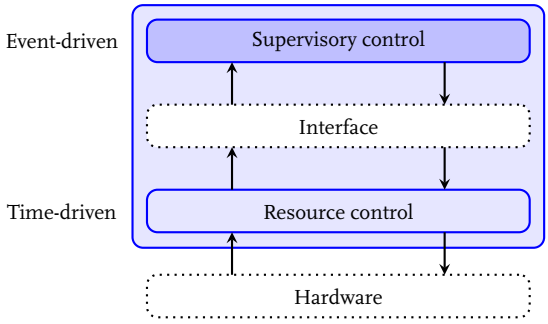


Figure 1: Positioning supervisory control

The model-based engineering process, as proposed in [2], enables the use of various model-based analysis techniques and tools to support system development. Moreover, supervisory control synthesis can be integrated in this setting. To this end, the system has to be decomposed into a plant P and a supervisor S. Note that this clear separation between plant and supervisor, is mostly not evident in traditional engineering. Although supervisory requirements are present, they are mostly intermixed with regulative control requirements. Figure 2 shows a graphical representation of this framework introduced in [20].

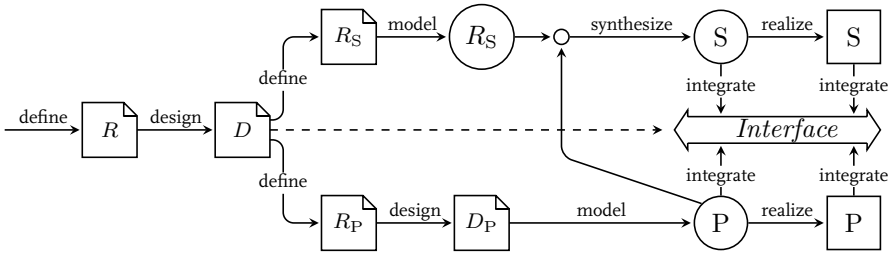


Figure 2: The engineering process with supervisory control synthesis

Initially, the requirements R of the system under supervision are defined. Based on these requirements, a design D of the system and a decomposition into the uncontrolled plant and

the supervisor are defined. After decomposition, the requirements for the supervisor R_S and for the uncontrolled plant R_P are specified. The requirements for the supervisor are formally modelled. From the plant requirements, a design D_P and one or more models of plant P can be defined. A discrete-event model of the plant together with the model of R_S can be used to synthesize a supervisor in the framework of supervisory control theory. Plant models can also be used to simulate the behaviour of the uncontrolled plant under supervision of the supervisor. If models of all system components are derived, several analysis techniques of the model-based engineering paradigm can be used to test the system in an early stage of the system development process.

In synthesis-based engineering, properties which are checked afterwards in traditional and model-based engineering, are used as input for generation of a design of a component that is correct by construction. As a consequence, the design and implementation do not need to be tested against the requirements, i.e., the verification can be eliminated. This changes the development process from implementing and debugging the design and the implementation, to designing and debugging the requirements.

To investigate the applicability of the supervisory control theory, in this paper we choose a theme park vehicle as a case study because it contains all basic features of a high-tech system. We make the following contributions from the implementation point of view. First, we investigate several implementation issues of the Ramadge-Wonham paradigm, e.g., satisfaction of basic assumptions such as asynchronous and instantaneous event firings, the potential negative effect of not achieving eventuality with nonblockingness, and asynchronous communication, etc. Second, we apply recently developed supervisor synthesis techniques to handle the potentially high complexity frequently encountered in high-tech systems, and illustrate their effectiveness. Third, we provide concrete evidence to show that the Ramadge-Wonham paradigm can speed up the controller design and significantly improve the quality of the relevant control software. We have been investigating the effectiveness of using distributed supervisor synthesis to improve the reconfigurability of a high-tech software control system.

The paper is structured as follows. Section 2 shortly summarizes the supervisory control concepts and synthesis techniques applied in the case study. In Section 3, the case study is introduced. The relevant models are defined in Section 4. In Section 5, supervisor synthesis, validation and implementation are presented. Finally, Section 6 concludes the paper by showing the impact supervisory control synthesis can have on a product development process. This paper is an extended version of [6].

2 Supervisory control theory

In the Ramadge-Wonham supervisory control paradigm, an open-loop system is modeled as one or several finite-state automata, and requirements are used to specify safety or liveness properties that the corresponding closed-loop system should possess. There are two basic assumptions about the system, namely event firings should be *asynchronous* and *instantaneous*. To capture the concepts of control and observation, events are distinguished as either *controllable* or *uncontrollable*, and *observable* or *unobservable*. The core of the control theory is to synthesize a supervisor, which disables only controllable events, updates control commands only after new observations are obtained, and always guarantees that the closed-loop system can reach a marker state regardless of its current state. These three features are captured by the main concepts of *controllability*, *observability* and *nonblockingness*, respectively.

In general, there are two basic control strategies: *state-based feedback control* and *event-based*

feedback control. In the former strategy, the supervisor observes only state information, and in the latter one only sequences of observable events are available to the supervisor. Based on observations the supervisor issues appropriate control commands accordingly, which determine the set of events that are allowed to be fired before new observations are obtained. Which strategy should be used completely depends on what can be observed in the system, i.e., states or events. In our case study, we apply both strategies.

When the system is not complex, a centralized approach can be used to synthesize a centralized supervisor, see e.g. [18]. Unfortunately, centralized approaches cannot overcome the state-space explosion phenomenon. To deal with this computational complexity issue, more advanced synthesis techniques have been introduced recently, e.g., the symbolic computation plus state-based control synthesis of [12], interface-based hierarchical synthesis of [10], compositional synthesis of [5] and [14], or distributed synthesis approaches of [23] and [22]. Among them modular (or distributed) techniques of [21], [4], [8], [23] and [22] are of particular interest for several reasons: (1) most (if not all) complex systems are naturally constructed in a modular way; (2) by deliberately masking out certain internal behaviors the plant model G can be significantly simplified for synthesis; (3) synthesized local controllers may be reused when G undergoes a structural change, which only affects part of it. The corresponding supervisory control architecture is depicted in Figure 3.

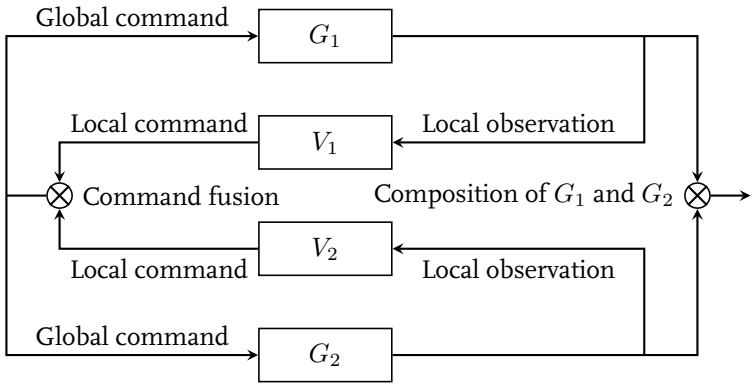


Figure 3: A distributed control architecture

The supervisor consists of a collection of local supervisors, each responsible for enforcing some local requirements in a few local components. The projections are used to model local observation channels for the corresponding local supervisors. The local control commands issued by V_1 and V_2 will be put together according to a certain fusion rule, e.g., the conjunctive rule which defines the global control command as the intersection of all local commands ([19]), or the disjunctive rule which defines the global control command as the union of local commands ([28]), or the combination of both conjunctive and disjunctive rules ([28]).

In our case study, we apply three synthesis techniques in order to determine the most suitable one for the case that can effectively handle complexity and reconfigurability of the system: the centralized state-based control synthesis and two automaton-based distributed synthesis techniques. In state-based synthesis, we follow the Ramadge-Wonham state-based supervisory control framework of [17] and [27] with a focus on STS symbolic technique introduced in [12]. This technique arranges component models in a state-tree structure and utilizes binary decision diagrams to efficiently manipulate states when performing reachability and coreachability search. In addition, we use a new logic system described in [15] that can precisely specify both state-based and event-based requirements. In event-based synthesis, we apply

the aggregative distributed synthesis technique from [23] and the coordinated distributed synthesis techniques from [22]. Technical details can be found in the Appendix.

Fig. 4 illustrates the aggregative distributed synthesis. To explain this approach, suppose we have three components and the ordering is G_1, G_2, G_3. Suppose there are three specifications H_1, H_2, H_3. Specification H_1 'touches' only G_1 in the sense that its alphabet Δ_1 is a subset of Σ_1. Specification H_2 touches only G_1 and G_2 but not G_3, namely its alphabet Δ_2 is a subset of $\Sigma_1 \cup \Sigma_2$ and $\Delta_2 \cap \Sigma_3 = \varnothing$. Additionally, specification H_3 touches not only G_1 and G_2 but also G_3, namely its alphabet Δ_3 is a subset of $\Sigma_1 \cup \Sigma_2 \cup \Sigma_3$ and $\Delta_3 \cap \Sigma_3 \neq \varnothing$. First, we compute the supremal nonblocking state-normal supervisor S_1 of G_1 under the specification H_1. (When there is no H_1 satisfying the conditions described, H_1 becomes the canonical recognizer of $(\Sigma_1^r)^*$. In this case only nonblockingness of the closed-loop behavior is the synthesis goal.) To achieve a nonblocking supervisor S_2, an abstraction W_1 of $G_1 \times S_1$ is created. The alphabet Σ' is chosen by whatever convenient reasons, as long as the condition $\Sigma_1 \cap (\Sigma_2 \cup \Sigma_3 \cup \Delta_2 \cup \Delta_3) \subseteq \Sigma' \subseteq \Sigma_1$ holds. The reason of imposing this condition is that in the subsequent computation, we can always use W_1 to replace $G_1 \times S_1$. If we do not want W_1 to lose too much information about controllability during abstraction, we can choose $\Sigma_{1,c} \subseteq \Sigma'$. Of course, too many events remaining in Σ' may result in an abstraction with only few states being removed from G_1. So there is a tradeoff issue that we need to deal with when we choose Σ'. Such a tradeoff is, in our opinion, case-dependent. We now have a plant $G_2 \times W_1$ and specification H_2. In a similar way, we can compute the supremal nonblocking state-normal supervisor S_2 of $G_2 \times W_1$ under H_2. Suppose S_2 exists, then we can create an abstraction W_2 of $G_2 \times W_1 \times S_2$ and a new plant $W_3 = G_3 \times W_2$. We then synthesize the supremal nonblocking state-normal supervisor S_3 of W_3 under H_3. If all supervisors are non-empty, the result of this procedure is a nonblocking distributed supervisor.

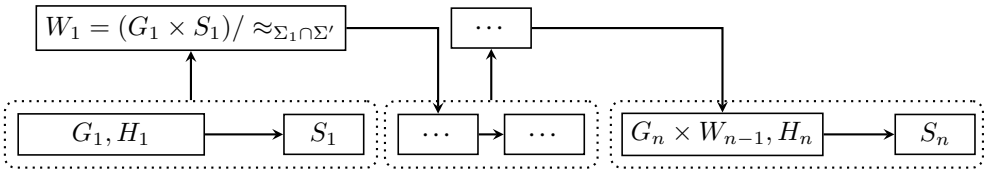


Figure 4: Synthesis of aggregative distributed supervisor

Fig. 5 illustrates the coordinated distributed synthesis. In this approach, we first synthesize a local supervisor S_i for each component G_i so that the local specification H_i can be enforced. Then we compute an abstraction so that we can synthesize a local supervisor to take care of H. We call each S_i a *local supervisor* and S a *coordinator*, which is mainly used to coordinate local supervisors $\{S_i | i \in I\}$ to avoid conflict. The existence of S gives rise to the term *coordinated distributed supervisor*. Of course, S itself is a supervisor, which enforces the specification H. Theorem 6.13 included in the Appendix allows us to synthesize a multiple-level multiple-coordinator distributed supervisor. For example, the system in Theorem 6.13 may be only a single module of a large system. Thus, after obtaining $\{S_i | i \in I\} \cup \{S\}$, we can compute an appropriate abstraction of $\times_{i \in I}(G_i \times S_i) \times S$ so that high-level local supervisors and/or coordinators can be synthesized.

3 Case study: a theme park vehicle

To show the effectiveness of the STS symbolic and the distributed synthesis techniques, we applied them to a real system: the multimover. The multimover, as shown in Figure 6a,

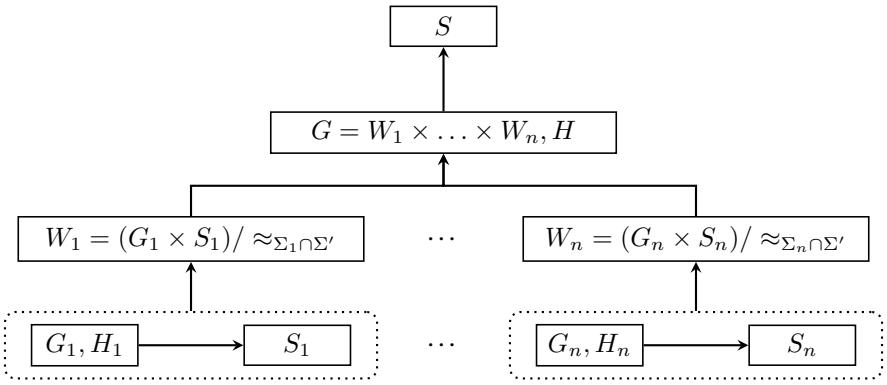
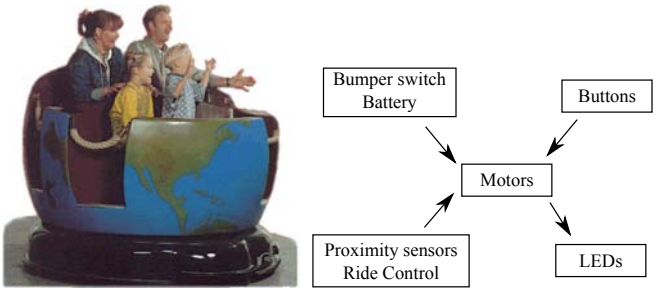


Figure 5: Synthesis of coordinated distributed supervisor

is an Automated Guided Vehicle that follows an electrical wire integrated in the floor. This track wire produces a magnetic field that can be measured by track sensors. Next to the track wire, floor codes are positioned, that can be read by means of a metal detector. These floor codes give additional information about the track, e.g., the start of a certain scene program, a switch, junction or a dead-end. The scene program, which is read by the scene program handler, defines when the vehicle should ride at what speed, when it should stop, rotate, play music and in which direction the vehicle should move (e.g., at a junction).



(a) The Multimover (b) Interaction between components

Figure 6: Theme park vehicle

An operator is responsible for powering up the vehicle and deploying it into the ride manually. The operator also controls the dispatching of the vehicles in the passenger boarding and outboarding area. The vehicle can receive messages from Ride Control. Ride Control coordinates all vehicles and sends start/stop commands to these vehicles. These messages are sent with wireless signals or by means of the track wire. Multimovers are not able to communicate with other vehicles. Safety is an important aspect of this vehicle. Therefore, several sensors are integrated in this vehicle to avoid collisions. First, proximity sensors are integrated in the vehicle to avoid physical contact with other objects. We can distinguish two types of proximity sensors. A long-range proximity sensor that senses obstacles in the vicinity of six meters and a short-range proximity sensor that senses obstacles in the vicinity of one meter. Second, a bumper switch is mounted on the vehicle that can detect physical contact with other objects. The interactions between vehicle components are shown schematically in Figure 6b.

The main requirement for supervisory control synthesis is safety. Three safety-related aspects can be distinguished:

- **Proximity handling** The supervisor has to assure that the multimover does not collide with other vehicles or obstacles. To this end, proximity sensors are integrated at the front and back which can detect an obstacle in the vicinity of the multimover. To avoid collisions, the multimover should drive at a safe speed and stop if the obstacle is too close to it.
- **Emergency handling** The system should stop immediately and should be powered off when a collision occurs. To detect collisions, a bumper switch is mounted on the multimover. The same applies when the battery level is too low. The LED interface should give a signal when an emergency stop has been performed. The multimover should be deployed back into the ride by an operator manually.
- **Error handling** When a system failure occurs (e.g., a malfunction of a motor), the system should stop immediately and should be powered off to prevent any further wrong behaviour. The LED interface should give a signal that an emergency stop has been performed. The multimover should be deployed back into the ride by an operator manually.

A divide-and-conquer strategy is often applied to get a good overview of control problems. This means that a large control problem can be divided into smaller control subproblems which can be solved more easily. We can divide the control problem of the multimover into five subproblems:

- **LED actuation** An operator must be able to check in which state the multimover is by looking at the Interface LEDs. This means that the states of the LEDs represent the current state of the multimover. It is a task of the supervisor to actuate the LEDs according to the state of the multimover.
- **Motor actuation** The drive motor, steer motor and scene program handler have to be switched on and off according to the state of the multimover. If the multimover is in the state **Active**, all motors can be switched on. If the multimover is in the state **Reset** or **Emergency**, all motors have to be switched off.
- **Button handling** The user interface of the multimover contains three buttons. The reset button is used to reset the vehicle if the multimover is active and deployed into the ride or it is in the state **Emergency**. The forward and the backward buttons are used to deploy the vehicle into the corresponding direction. The supervisor has to assure that the corresponding state is reached after a button is pushed.
- **Proximity and Ride Control handling** On each side of the multimover, two proximity sensors are mounted: one long-range and one short-range. If a long-range proximity sensor detects an object in the traveling direction, the multimover should react by slowing down to a safe driving speed. If an obstacle is detected by a short-range proximity sensor, the multimover should stop in order to prevent a collision. When the short-range proximity sensor no longer detects an object, the vehicle should start riding automatically. If the multimover receives a stop command from Ride Control, it should stop as in the case of short-range proximity handling. If Ride Control sends a start command, the multimover should automatically start riding with the speed depending on the state of the proximity sensors related to the current driving direction.
- **Emergency and error handling** In order to guarantee safety of passengers, the multimover should be deactivated immediately when an emergency situation occurs. It

should not be possible to reset the multimover if the bumper switch is still activated or battery power is still too low. A control task of the supervisor is to enter the **Emergency** state of the multimover when an emergency situation occurs.

Based on system functionality and the amount of interaction between components, components that have a lot of interaction with each other and are strongly coupled in the control problem belong to the same control subproblem. Fig. 7 shows a graphical representation of the partitioning of the multimover control problem. This forms the basis for distributed synthesis.

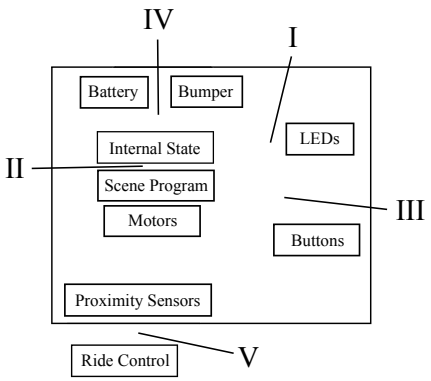


Figure 7: Partitioning of the control problem

4 Plant and requirement models

In our case study, the plant model represents an event-based abstraction of the actual behaviour of the physical components and their resource control, which is schematically shown in Figure 8. The arrows represent the information flow between the components, the resource controllers and the supervisor.

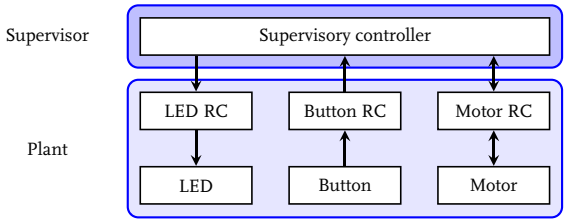


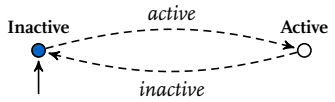
Figure 8: The control architecture

As usual in the context of supervisory control, plant models are defined by automata. Each component together with its resource controller is modelled by one automaton. Automata consist of states and transitions labeled by (controllable and uncontrollable) events. States of the plant model represent all relevant states of each resource (e.g. on, off, empty, active). Controllable events represent relevant discrete commands (function calls) to the resource

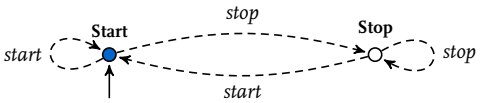
control (e.g. enable, disable). These actions can be enabled or disabled by the supervisor. Uncontrollable events represent messages that are sent from the resource control to the supervisor (e.g. a failure notification, a sensor event). These events cannot be disabled by the supervisor.

The plant model is made with the assumption that the resource control of the multimover is working correctly. This assumption is reasonable because the resource controllers are embedded in the existing implementation and have thoroughly been tested. This means that if a command is given, it is carried out correctly. Furthermore, the communication between the plant and the supervisor is sufficiently fast. If an event occurs at the plant (e.g., a button is pressed), the supervisor is synchronized immediately. In Section 5, we argue that this is also the case in our prototype implementation because the response time of the multimover controller is short enough to properly react to the changes in its environment.

In this paper, we use italic event labels (e.g. *active*) and bold state labels (e.g. **Start**). In the graphical automaton representation, initial states are denoted by an unconnected incoming arrow and marker states are denoted by filled vertices. Controllable and uncontrollable events are denoted by solid and dashed edges, respectively.



(a) Automaton of a sensor



(b) Automaton of Ride Control

Figure 9: Automata of input components

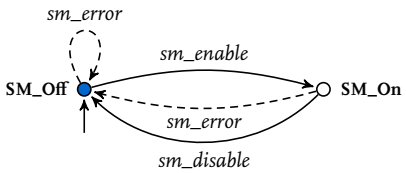
For brevity, we only show a few component and requirement models that are representative for our case study. The complete definition of the supervisory control problem under consideration can be found in [7]. All buttons and sensors are modelled by automata having the same structure, as depicted in Figure 9a for a sensor. The sensor can generate two events: *active* and *inactive*. Each event labels the transition from one state to another, e.g. if a sensor becomes active, the event *active* is generated.

Ride Control can send a general start/stop command to start or stop all the multimovers in an attraction. Ride Control sends these commands constantly with a certain interval. Therefore, it is possible that the same command is sent over and over again. This behaviour is captured by the automaton depicted in Figure 9b. Note that the events mentioned above are uncontrollable, since the supervisor cannot disable them.

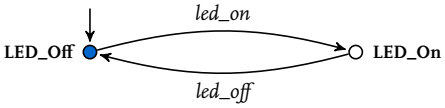
In Figure 10a, the automaton of the steer motor is given. The relevant states of the steer motor are **SM_On** and **SM_Off**. The actuation signals that are important for the supervisory controller are switching on the steer motor (*sm_enable*) and switching off (*sm_disable*). This motor contains a hardware safety in case the motor is short-circuited or has a hardware failure. If this hardware safety is activated (*sm_error*), the motor is automatically switched off. Since the hardware safety can also be activated when the motor is switched off and still slowing down, the event error is looped at state **SM_Off**.

All LEDs of the multimover are modelled by automata having the same structure, as depicted in Figure 10b. A LED of the multimover can be in two states: **LED_On** and **LED_Off**. The events *led_on* and *led_off* represent the function calls switching the LED on and off, respectively.

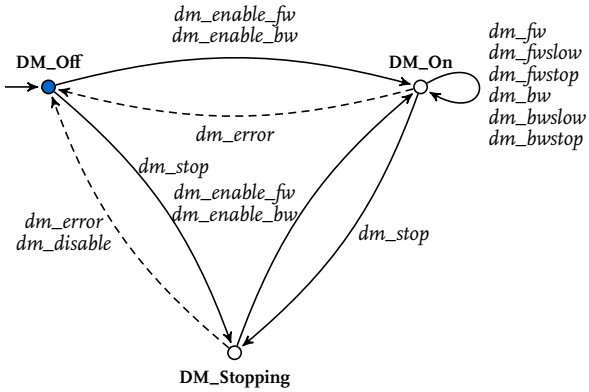
In Figure 10c, the automaton of the drive motor is given. It is basically the same as the automaton of the steer motor. However, it contains an extra state **DM_Stopping**, since for safety reasons the drive motor may not be switched off if the multimover is still moving (e.g., stopping). Therefore, an extra event *dm_stop* is introduced that stops the drive motor. If the drive motor has stopped, the uncontrollable event *dm_disable* occurs and the drive motor is switched off. Because we want to be able to set the maximum speed of the drive motor, the events *dm_fw*, *dm_fwslow*, *dm_fwstop*, *dm_bw*, *dm_bwslow* and *dm_bwstop* are introduced. The drive motor also contains a hardware safety in case the motor is short-circuited or has a hardware failure. When such a situation occurs, the motor is automatically switched off, which is modelled by the event *dm_error*.



(a) Automaton of the steer motor



(b) Automaton of an Interface LED



(c) Automaton of the drive motor

Figure 10: Automata of actuators

The multimover itself can also be in three states, namely **MM_Emergency**, **MM_Reset** and **MM_Active**, see Figure 11. **MM_Emergency** denotes the state of the multimover in which all components are switched off and the multimover has to be reset manually by pushing the reset button. If the reset button is pushed, the multimover should enter the state **MM_Reset**. From this state, the multimover can be deployed into the ride (**MM_Active**) or can switch back to **MM_Emergency** (if an emergency event occurs). Since a lot of control requirements

are based on the state of the multimover, this automaton is introduced for modelling convenience.

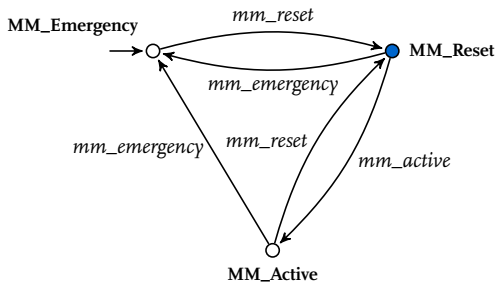


Figure 11: Plant model of the multimover

Marker states are used to describe completed tasks. They represent a set of states which we always want to be reachable by any behaviour [13]. Since in the model, the **MM_Reset** state is a marker state, the synthesized supervisor always assures that the multimover can be reset.

Behaviour of plant components can be modelled in different ways. For instance, component models can represent an already restricted behaviour (partially controlled), or all the physically possible behaviour (uncontrolled), with no restrictions. We have chosen for component models representing the uncontrolled behaviour. In this way, plant models are obtained that match exactly the behaviour of the interface of the components. Decomposing the system in an uncontrolled plant and a supervisor gives a clear view of the system functionality. Supervisor synthesis is slightly more difficult with these unrestricted plant automata, since more behaviour has to be restricted by means of requirement models.

The automata depicted in Figure 12 specify the control requirements of the emergency and error handling control module. Figure 12a and 12b specify that the events *mm_active* and *mm_reset* are only allowed to take place if the bumper switch is not activated and the power level of the battery is sufficient. This requirement can be defined by taking the plant automata of the bumper switch and the battery and adding loops with events *mm_active* and *mm_reset* at the states that represent the bumper switch not being activated and the power level of the battery being sufficient.

The last requirement, which is depicted in Figure 12c, specifies when the event *mm_emergency* is allowed to occur. It is only allowed to occur after activation of the bumper switch (*bs_press*), the power level of the battery becoming too low (*ba_empty*), a parse error of the scene program (*sh_error*), a failure of the drive motor (*dm_error*) or a failure of the steering motor (*sm_error*). If one (or a sequence) of these emergency events takes place, the requirement allows the occurrence of the event *mm_emergency*, all other events are allowed to take place in any order without restrictions.

The original event-based framework uses automata to describe plant models and requirement models. Sometimes it is easier or more intuitive to express requirements by state-based expressions instead of automata. Automata can only enumerate all possible behaviours, while state-based expressions can describe them more concisely. Moreover, system requirements are often expressed in terms of conditions over states. This possibility is provided in the state-based framework, where both state-based expressions and automata are available to specify the desired behaviour. However, as indicated in [15], deriving the suitable state-based expressions can be an error-prone and tedious task. To avoid this inconvenience, logical

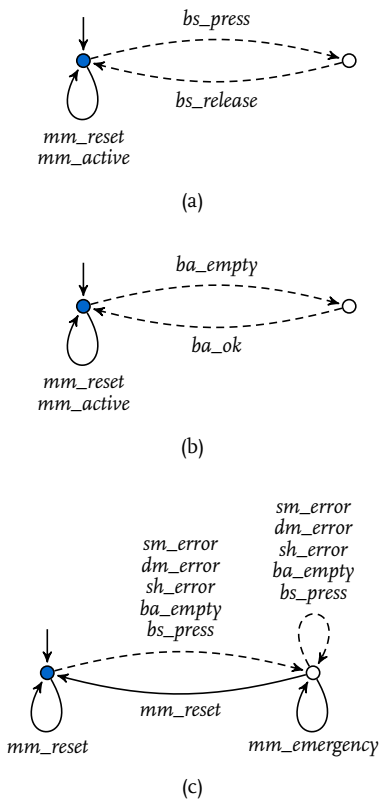


Figure 12: Requirement models of the emergency module

specifications are proposed for automatic generation of these state-based expressions. Design engineers can express requirements by logical specifications that naturally follow from informal, intuitive requirements.

The requirements that are depicted in Figure 12a and 12b can also be specified by a logical expression stating that if the events *mm_reset* and *mm_active* are enabled by the supervisor then the bumper switch is released (**BS_Released**) and the power level of the battery is sufficient (**BA_OK**). Using the syntax introduced in [15], this logical expression reads: $\rightarrow$ { *mm_reset, mm_active* } $\Rightarrow$**BS_Released** $\downarrow \wedge$**BA_OK** $\downarrow$.

5 Supervisor synthesis, validation and implementation

To indicate the complexity of the multimover supervisory control problem, Table 1 shows the numbers and sizes of the plant components and the requirements. In the event-based framework, it is not possible to derive a centralized supervisor for a problem of this size.

For the supervisory control problem of the multimover, a centralized state-based supervisor and two distributed event-based supervisors are synthesized. The centralized supervisor has been synthesized using the state-based approach of [12] based on state tree structures. The

Table 1: Complexity of the supervisory control problem

No. of components	17
No. of states per component	2-4
No. of control requirements	30
No. of states per requirement	2-7

state-based synthesis produces binary decision diagrams (BDD) for each controllable event. The maximum BDD size is 15 and the minimum BDD size is 1. The BDD size of the optimal controlled behaviour is 100. Furthermore, two distributed supervisors have been synthesized using the event-based coordinated approach of [22] and the event-based aggregative approach of [23]. The results of the event-based synthesis are shown in Tables 2 and 3 that illustrate the effectiveness of the distributed synthesis techniques in application to supervisory control problems of this size. Since only automata can be used for specifying the requirement models of a distributed supervisor, an automatic conversion of logical expressions to automata is used. This conversion allows design engineers to specify the formal requirements with automata and logical expressions and still synthesize a distributed supervisor. In [7], a detailed description of the results is provided.

Table 2: Distributed coordinated supervisors

Module	# states	# transitions
LED actuation	25	77
Motor actuation	41	222
Button handling	193	1541
Emergency handling	181	2149
Proximity handling	481	4513

Table 3: Distributed aggregative supervisors depending on synthesis order

Module	Order	# states	# trans.	Order	# states	# trans.
LED actuation	1	25	77	5	41	125
Motor actuation	2	41	222	2	257	1428
Button handling	3	465	3477	4	177	765
Emergency handling	4	89	626	3	118	609
Proximity handling	5	225	1953	1	481	4513

Synthesized supervisors have been evaluated to check whether the models of the controlled system are consistent with the intended behaviour. For this purpose, discrete-event simulation is used persistently. In this setting, the state-space stepper is used to explore the state space of the closed-loop system behaviour. The state-space stepper allows to check whether the supervisor disables right transitions in right states when evaluating a trace. Several scenarios relevant to the multimover have been tested to confirm the validity of the models used. These scenarios involve generation of control actions in reaction to generated events, such as a button is pressed or a sensor is activated. The CIF-toolset described in [26] is used for discrete-event simulation.

In the original supervisory control framework, a supervisor acts as a passive device that tracks events produced by the plant and restricts the behaviour of the plant by disabling the controllable events, see e.g., [1]. However, it is often the case that the plant does not generate all controllable events on its own without being initiated. Usually, machines do not start their work unless a start command is given. In this case, it is desirable to have a controller

which not only disables controllable events but also initiates the occurrence of particular controllable events, as indicated by [3]. Furthermore, supervisory control theory is based on the assumption that the supervisor is always synchronized with the state of the plant, i.e. there is no communication delay. However, in contrast to the synchronous communication used in models, real systems often use asynchronous communication. Hence, a supervisor can be seen as a dictionary of allowed events at each state of the plant, from which an associated controller can choose an appropriate control action. In this section, the implementation is explained of such a controller, which is referred to as a supervisory controller. The idea is similar to the notion of directed controller of [9].

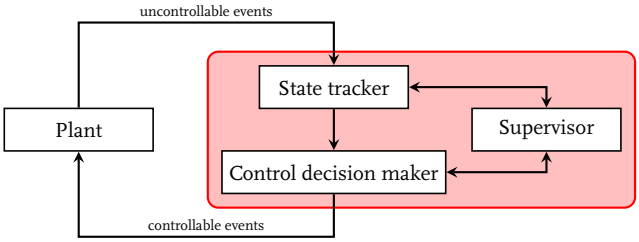


Figure 13: A supervisory controller

The functionality of a supervisory controller can be roughly divided in two tasks. The supervisory controller needs to track the state of the plant in order to give appropriate feedback to the plant. We call this part of the controller the state tracker. Next, the controller is responsible for sending appropriate control actions back to the plant based on the state of the plant. We refer to this part of the supervisory controller as the control decision maker. In Figure 13, a schematic overview of a supervisory controller is given. In this figure, we can distinguish the plant which represents the components and the low-level resource control, and a supervisory controller in the filled frame. This supervisory controller contains a state tracker which tracks the state, a control decision maker which sends appropriate actions back to the plant and a supervisor which contains all allowed behaviour.

At some point, the plant generates an event (e.g., a button is pressed or a sensor is activated). A notification must be sent to the state tracker, which updates the current state of the supervisor. This is done by looking in the supervisor what the new current state is. Only uncontrollable events are tracked by the state tracker, since the supervisory controller initiates the controllable events. If the state tracker is ready with updating the current state of the supervisor, the control decision maker has to search for an appropriate control action that can be sent back to the plant (e.g., turn the LED on or turn off the motor). If an appropriate control action is found, this action is carried out and the current state of the supervisor is updated again.

The communication problem is related to building controllers from supervisory control models, as discussed in [13]. This problem occurs when the controller sends a control action to the plant, but in the meantime, the state of the plant is changed. Hence, a control action is chosen based on an old state of the plant. Such a situation can occur because communication between the plant and the controller in the real system is not synchronous.

In order to prove the concept of synthesis-based engineering, a prototype of a supervisory controller with the synthesized supervisors is implemented in the existing control software of the multimover. This implementation is set up in such a way that it works with all supervisors synthesized. A schematic overview of the control architecture of the multimover with the supervisory controller is given in Figure 14. The implemented supervisory controller replaces the existing control software that has the same functionality.

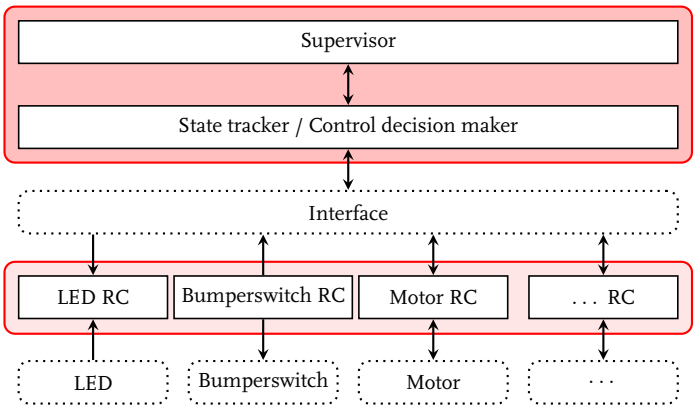


Figure 14: Control architecture of the implementation

The lowest layer includes all components and their resource controllers. Above the resource control, an interface is defined that is responsible for sending the correct events from the resource control to the supervisory controller and sending the correct events from the supervisory controller to the resource controllers. This interface makes use of a listener and notifier structure, which is a simple communication paradigm. The resource control of each component can publish messages of a certain topic and can subscribe to a certain topic, which means that they will receive all published messages of that topic. The interface is subscribed to all relevant events and will receive them. This interface has to be coded manually, since it is different for every system.

The next layer in Figure 14 is an implementation of a supervisory controller containing a state tracker and a control decision maker. This layer is set up in such a way that it is independent of the supervisor model, supervisory control framework used and the system itself. A generic implementation of a supervisory controller is shown as pseudo-code of Algorithm 1. As already mentioned, the functionality of the supervisory controller can be divided into two (parallel) tasks, namely tracking the state of the system by the state tracker (lines 3 through 7) and making appropriate control decisions by the control decision maker (lines 9 through 19).

All (uncontrollable) events that are generated by the plant (e.g. button and sensor signals) are inserted by the interface in a buffer (called "list"), which is a different process that also has access to this buffer. This buffer is emptied by the state tracker by taking and removing the first element of the buffer ($E \leftarrow$ pop(list)) and subsequently, the current state of the supervisor is updated (line 5, UpdateSupervisor(S)). If the list is empty, the state tracker knows the current state of the system. Based on this current state of the supervisor, a control decision can be calculated.

If the current state of the supervisor has changed ($S \leftarrow 1$), the control decision maker has to check if a control action is possible. Setting variable S to 1 (line 6) activates the control decision maker. First, a control decision is computed. If an appropriate control action is found (line 11, $E \neq 0$), the event list has to be checked (line 12), to ensure that the supervisory controller has made an appropriate control action based on the most actual state of the plant. If this is the case, the state of the supervisor is updated and the appropriate control action is executed. Note that this implementation does not prevent the execution of a control action based on an old state of the supervisor. The communication problem can still occur.

If no control action is possible (e.g. all controllable events are disabled by the supervisor), there is no need to search for a control action over and over again. So, the boolean variable

Algorithm 1 Supervisory controller implementation

```
    loop
      // State Tracker
      while len(list) > 0 do
        E ← pop(list);
 5:     UpdateSupervisor(E);
        S ← 1
      end while
      // Control Decision Maker
      if S = 1 then
10:     E ← ComputeControlAction;
        if E ≠ 0 then
          if len(list) = 0 then
            UpdateSupervisor(E);
            ExecuteEvent(E)
15:       end if
        else
          S ← 0
        end if
      end if
20: end loop
```

S is set to 0 (line 17), which means the control decision maker is not executed anymore. If the state of the system changes again due to the occurrence of an uncontrollable event, the control decision maker is activated again.

The next layer in Figure 14 is the supervisor itself, which contains the information about the allowed behaviour, according to the requirements. This information can be generated from the model of the supervisor. This is done by a script in Python, that reads the information from a CIF model and stores this information in a lookup table. A lookup table is used for this information, since a lookup table can be used with an efficient indexing operation, which can reduce processing time. The details of the implementation can be found in [7].

The prototype implementation described above is suitable for supervisors of both frameworks, either event-based or state-based. However, there are some differences with respect to how the state is tracked and the control decisions are made for both frameworks. A supervisor that is synthesized with the event-based framework contains the complete allowed language of the closed-loop system, stored in one or more automata. The state is tracked by updating the current states of the automata if an event occurred. An automaton is only updated if the event that has occurred is also in the language of this automaton. If an event occurs that is not allowed by automata, then the model is inadequate, since the state tracker cannot track the state of the system. If this happens, the supervisory controller and all components are switched off. Control decisions are calculated by searching for controllable events that are allowed by all automata. The first controllable event that is found and allowed by all automata is chosen as the control action. A supervisor that is synthesized with the state-based framework uses automata and BDDs to store the state feedback control (SFBC) map in. The automata are used to store the information when each controllable event is allowed by the plant models and the BDDs are used to store the information when each controllable event is allowed by the state-based expressions. A state-based implementation uses the plant models and event-based requirements to track the state of the system. All automata of the plant models and event-based requirements are updated if an uncontrollable event occurs. If an uncontrollable event occurs that is not allowed by an automaton, the state of the system cannot be tracked and the model of the supervisor is inadequate. If this happens, the

supervisory controller and all components are switched off. Control decisions are calculated by searching for a controllable event that is allowed by all automata and its BDD. The first controllable event that is allowed by all automata and its BDD is used as a control action.

Our implementation is first tested by means of simulation, then on a test rack and, finally, on a real vehicle. For testing, the same scenarios were used that comprised of pushing buttons and activating and deactivating sensors. All relevant situations were tested exhaustively, in order to validate the error handling, proximity handling and emergency handling. During the tests, the communication problem did not occur. This can be explained by the fact that the supervisor responds sufficiently fast to the state changes of the plant. In the prototype implementation, the choices in the supervisor do not depend on the order in which uncontrollable events happen. Additionally, as the state changes are evaluated within 80 ms (for the state-based supervisor; for the event-based supervisor within 20 ms), which is fast enough for the multimower, the assumptions about asynchronous and instantaneous event firings mentioned in Sections 2 and 4 are satisfied by our prototype implementation.

6 Conclusions

Formal models are a key element in the synthesis-based engineering process. They provide a structured and systematic approach to the component and system behaviour specification. Moreover, they allow more consistency and less ambiguity than documents, because formal semantics precisely defines what models mean. The use of formal models in an early stage of the product development process, forces the engineers to clarify all aspects of the system. Clarity contributes to a good design and correct control software. Furthermore, modelling systems and requirements by finite state machines or logical expressions is intuitive. However, time is needed to develop appropriate modelling skills.

The automatic synthesis of a supervisor changes the software development process from designing and debugging controller code into designing and debugging requirements, assuming correct plant models. Since these requirements are modelled formally, we do not need to test the model of the supervisor against the requirements, since it is correct by construction. Thus, the engineers can focus on validating the system, not on verifying the software design. Subsequently, the requirements of a system can change over time due to customer demands. As a consequence, in traditional engineering, all changes have to be made in the software design informally, and this is difficult to do without introducing errors or inconsistencies. Using synthesis-based approach, only plant models or requirement models have to be adapted and a new supervisor can be synthesized. This means that the system is evolvable, i.e. able to withstand changes.

In addition, the synthesized supervisors can be simulated immediately, which results in a short feedback loop in the development process. Furthermore, the usage of models allows the application of model-based techniques, such as simulation and formal verification, which can detect errors in an early stage of the system development process. As a result, the costs to develop expensive prototypes can possibly be reduced. Furthermore, since the desired behaviour is specified in models instead of in the software code, engineers can have a better understanding of the control software, which can lead to an easier validation with respect to the original informal specifications.

The results of the case study prove the effectiveness of the synthesis techniques used especially when requirements or plant components change, or if the system needs to be reconfigured. The evidence can be provided by the following observation. The engineering process used presently requires approximately two days for making changes to the control system

if the number of proximity sensors is extended. The synthesis-based engineering process described in this paper requires approximately four hours to cope with the same change.

Appendix

In this section we first provide an overview of Ramadge-Wonham supervisory control paradigm introduced in [18]. Then we provide technical details on the event-based aggregative and coordinated distributed synthesis approached introduced in [23] and [22].

Let Σ be a finite alphabet, and Σ^* the Kleene closure of Σ, i.e. the collection of all finite sequences of events taken from Σ. Given two strings $s, t \in \Sigma^*$, s is called a *prefix substring* of t, written as $s \leq t$, if there exists $s' \in \Sigma^*$ such that $ss' = t$, where ss' denotes the concatenation of s and s'. We use ϵ to denote the empty string of Σ^* such that for any string $s \in \Sigma^*$, $\epsilon s = s\epsilon = s$. A subset $L \subseteq \Sigma^*$ is called a *language*. $\overline{L} = \{s \in \Sigma^* | (\exists t \in L)\, s \leq t\} \subseteq \Sigma^*$ is called the *prefix closure* of L. Given two languages $L, L' \subseteq \Sigma^*$, the concatenation of L and L' is $LL' = \{ss' \in \Sigma^* | s \in L \wedge s' \in L'\}$.

Let $\Sigma' \subseteq \Sigma$. A map $P : \Sigma^* \to \Sigma'^*$ is called the *natural projection* with respect to (Σ, Σ'), if

1. $P(\epsilon) = \epsilon$
2. $(\forall \sigma \in \Sigma)\, P(\sigma) = \begin{cases} \sigma & \text{if } \sigma \in \Sigma' \\ \epsilon & \text{otherwise} \end{cases}$
3. $(\forall s\sigma \in \Sigma^*)\, P(s\sigma) = P(s)P(\sigma)$

Given a language $L \subseteq \Sigma^*$, $P(L) = \{P(s) \subseteq \Sigma'^* | s \in L\}$. The inverse image map of P is

$$P^{-1} : 2^{\Sigma'^*} \to 2^{\Sigma^*} : L \mapsto P^{-1}(L) = \{s \in \Sigma^* | P(s) \in L\}$$

Given $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$, the *synchronous product* of L_1 and L_2 is defined as:

$$L_1 || L_2 = P_1^{-1}(L_1) \cap P_2^{-1}(L_2)$$

where $P_1 : (\Sigma_1 \cup \Sigma_2)^* \to \Sigma_1^*$ and $P_2 : (\Sigma_1 \cup \Sigma_2)^* \to \Sigma_2^*$ are natural projections. It is clear that $||$ is commutative and associative.

A *finite-state automaton* is a 5-tuple $G = (X, \Sigma, \xi, x_0, X_m)$, where X stands for the state set, Σ for the alphabet, $\xi : X \times \Sigma \to X$ for the (partial) transition map, x_0 for the initial state and X_m for the marker state set. In the Ramadge-Wonham supervisory control paradigm, an open-loop system is modeled as one or several finite-state automata, and requirements are used to specify safety or liveness properties that the corresponding closed-loop system should possess. There are two basic assumptions about the system, namely event firings should be *asynchronous* and *instantaneous*. To capture the concepts of control and observation, each event $\sigma \in \Sigma$ can be either *controllable* or *uncontrollable* in the sense of whether or not its firing can be disabled, and σ may also be either *observable* or *unobservable* in the sense of whether or not its firing can be detected. Let $\Sigma = \Sigma_c \cup \Sigma_{uc} = \Sigma_o \cup \Sigma_{uo}$, where the disjoint sets Σ_c and Σ_{uc} denote respectively the collection of controllable and uncontrollable events, and the disjoint sets Σ_o and Σ_{uo} denote respectively the collection of observable and unobservable events. Let $\phi(\Sigma)$ be the collection of all finite-state automata whose alphabets are Σ.

The concept of supervisory control is captured by a supervisory control map $V : \Theta \to \Gamma$, where $\Gamma = \{\gamma \subseteq \Sigma | \Sigma_{uc} \subseteq \gamma\}$ is the collection of all *control patterns*, each of which contains

all uncontrollable events denoting that no uncontrollable events can be disabled, and Θ represents what can be observed in the system G based on which an appropriate control pattern can be chosen. When $\Theta = X$ or $L(G)$, we call the corresponding V a *state-based control map* or an *event-based control map* respectively. In the sequel, we focus on event-based control. The *closed behavior* of the supervised system V/G is defined as follows:

1. $\epsilon \in L(V/G)$
2. $(\forall s \in L(V/G))(\forall \sigma \in V(s))\, s\sigma \in L(G) \Rightarrow s\sigma \in L(V/G)$
3. all strings of $L(V/G)$ are obtained in Steps (1) and (2).

The *marked behavior* of V/G is defined as $L(V/G) \cap L_m(G)$.

The standard supervisory control problem in the Ramadge-Wonham paradigm is formulated as follows: given a plant $G \in \phi(\Sigma)$ and a requirement $H \in \phi(\Sigma)$, find a supervisory control map V such that (1) $L_m(V/G) \subseteq L_m(H)$ (i.e., "only legal behaviors can happen"), and (2) $\overline{L_m(V/G)} = L(V/G)$ (i.e., "each ongoing task can be finished"). It turns out that the existence of V depends on whether there exists a sublanguage $K \subseteq L_m(G) \cap L_m(H)$ which is *controllable* and *observable*. If we find such a K then we can construct the corresponding V. For this reason the control problem of finding a control map V is turned into the synthesis problem of computing a controllable and observable sublanguage K.

Definition 6.1. [18] $K \subseteq L(G)$ is *controllable* with respect to G if $\overline{K}\Sigma_{uc} \cap L(G) \subseteq \overline{K}$. □

Intuitively speaking, if K is controllable then no string can skid out of K via an uncontrollable extension. In supervisor synthesis K is usually interpreted as the collection of all "good" behaviors. Thus, we can use event disabling to prevent any "bad" behavior from happening if we start with a good one inside K (i.e., in $\overline{K}$). It can be shown that controllability is closed under set union.

Definition 6.2. [11] Let $K \subseteq L(G)$ and $P_o : \Sigma^* \to \Sigma_o^*$ be the natural projection. K is *observable* with respect to G and P_o if for all $s, s' \in \overline{K}$ and $\sigma \in \Sigma$,

$$s\sigma \in \overline{K} \wedge s'\sigma \in L(G) \wedge P_o(s) = P_o(s') \Rightarrow s'\sigma \in \overline{K}$$

We say K is *normal* with respect to G and P_o if $P_o^{-1}(P_o(K)) \cap L(G) = K$. □

Intuitively speaking, if K is observable then for any two strings $s, s' \in \overline{K}$ having the same observation and both extendable by σ, $s\sigma \in \overline{K}$ if and only if $s'\sigma \in \overline{K}$. This property allows the existence of a supervisor, which acts only upon observable events. We can check that normality is stronger than observability. But K is observable does not necessarily imply K is normal. This is true if $\{\sigma \in \Sigma | (\exists s \in \overline{K})\, s\sigma \in L(G) - \overline{K}\} \subseteq \Sigma_o$, which means a string can skid out of K only through observable extensions. Although normality looks quite strong, in practical applications it is as general as observability when we take controllability into consideration and set all controllable events observable (which is quite common in real systems). Furthermore, normality has a very good property — it is closed under set union, which allows us to compute the largest normal sublanguage. As a contrast, observability is neither closed under set union nor closed under set intersection. We have the following result, which solves the Ramadge-Wonham supervisory control problem.

Theorem 6.3. [18] Given a language $K \subseteq L_m(G)$, there exists an event-based supervisory control map $V : L(G) \to \Gamma$ such that $L_m(V/G) = K$ and $\overline{L_m(V/G)} = L(V/G)$ if and only

if (1) K is controllable with respect to G; (2) K is observable with respect to G and P_o; (3) K is $L_m(G)$-closed, i.e., $L_m(G) \cap \overline{K} = K$. □

In many practical applications K is not given explicitly. Instead, another language $H \subseteq \Delta^*$ with $\Delta \subseteq \Sigma$, which is called a *requirement*, is given. Let

$$\mathcal{C}(G,H) = \{K \subseteq L_m(G) || H \Big| K \text{ is controllable w.r.t. } G \text{ and normal w.r.t. } G \text{ and } P_o\}$$

We can show that there exists a unique largest element $K_* \in \mathcal{C}(G,H)$ such that for all $K \in \mathcal{C}(G,H)$, we have $K \subseteq K_*$. We call K_* the *supremal controllable and normal sublanguage of G under H*. How to compute K_* is the Ramadge-Wonham supervisor synthesis problem, whose solvability imposes a major challenge for practical applications.

Distributed synthesis

We first describe an automaton-based abstraction technique, then formulate a distributed synthesis problem, and finally provide two distributed synthesis techniques.

Automaton abstraction

In distributed synthesis we frequently encounter the situation of synthesizing a local supervisor for a large number of local components. The union of their alphabets is usually much larger than the alphabet of the supervisor - in other words, the supervisor can only observe and control a small number of events. To take the entire composition of those local components into consideration during synthesis is certainly not only unnecessary but also computationally intensive. For this reason it is quite natural to ask the following question:

Does there exist an abstraction map such that a nonblocking supervisor of an abstracted model of a plant is guaranteed to be a nonblocking supervisor of the plant?

The answer to this question is affirmative. One such a map is defined over *nondeterministic finite-state automata*. A *nondeterministic finite-state automaton* is an automaton $A = (Y, \Sigma, \eta, y_0, Y_m)$, where the transition map $\eta : Y \times \Sigma \to 2^Y$ is nondeterministic. A is called *deterministic* if for all $y \in Y$ and $\sigma \in \Sigma$, $\eta(y,\sigma)$ contains no more than one element.

Definition 6.4. [23] We say $G = (X, \Sigma, \xi, x_0, X_m)$ is *standardized* if Σ contains a special event τ, which is called the *initial event* such that the following hold

1. $x_0 \notin X_m \wedge (\forall x \in X)\,[\xi(x,\tau) \neq \varnothing \iff x = x_0] \wedge (\forall \sigma \in \Sigma - \{\tau\})\,\xi(x_0,\sigma) = \varnothing$
2. $(\forall x \in X)(\forall \sigma \in \Sigma)\, x_0 \notin \xi(x,\sigma)$ □

A standardized automaton is an automaton, in which x_0 is not marked (by condition 1), τ is only used at x_0, which has only outgoing transitions labeled by τ (by condition 1) and no incoming transitions (by condition 2). From now on let $\phi(\Sigma)$ be the set of all standardized finite-state automata, whose alphabets are Σ. We want to point out that any non-standardized automaton can be easily converted into a standardized one by adding a new initial state and connects this new initial state with the original one by a new initial event.

Definition 6.5. [25] Given $G = (X, \Sigma, \xi, x_0, X_m) \in \phi(\Sigma)$, let $\Sigma' \subseteq \Sigma$ and $P : \Sigma^* \to \Sigma'^*$ be the natural projection. A *weak observation equivalence* relation on X with respect to Σ' is an equivalence relation $R \subseteq \{(x,x') \in X \times X | x \in X_m \iff x' \in X_m\}$ such that for all $(x,x') \in R$ and all $s \in \Sigma^*$, if $P(s) \neq \epsilon$ then for all $y \in \xi(x,s)$,

$$(\exists s' \in \Sigma^*)\, P(s) = P(s') \wedge (\exists y' \in \xi(x',s'))\,(y,y') \in R$$

The largest weak observation equivalence relation on X with respect to Σ' is denoted as $\approx_{\Sigma',G}$. □

A weak observation equivalence relation is close to a weak bisimulation relation defined in, e.g. [21], which is extended from [16]. It says for all $(x, x') \in R \subseteq X \times X$, $s \in \Sigma^*$ and $y \in \xi(x, s)$, there is $s' \in \Sigma^*$ such that $P(s) = P(s')$ and

$$(\exists y' \in \xi(x', s'))\,(y, y') \in R \wedge [y \in X_m \iff y' \in X_m]$$

But Def. 6.5 has two different features: (1) it is possible that (x, x') can have inequivalent "unobservable" extensions (i.e. x can reach a state y via s with $P(s) = \epsilon$, but x' cannot reach any state y' equivalent to y via some s' with $P(s) = P(s')$), which is not allowed in the definition of [21], and (2) two equivalent states must have the same marking status, which does not need to be true in the definition of [21]. From now on, when G is clear from the context, we simply use $\approx_{\Sigma'}$ to denote $\approx_{\Sigma',G}$.

Definition 6.6. [25] Given $G = (X, \Sigma, \xi, x_0, X_m)$, let $\Sigma' \subseteq \Sigma$ with $\tau \in \Sigma'$. The *automaton abstraction* of G with respect to $\approx_{\Sigma'}$ is an automaton $G/\approx_{\Sigma'} = (Z, \Sigma', \lambda, z_0, Z_m)$ where

1. $Z := X/\approx_{\Sigma'} := \{< x >_{\Sigma'} := \{x' \in X | (x, x') \in \approx_{\Sigma'}\} | x \in X\}$
2. $z_0 :=< x_0 >_{\Sigma'} \in Z$
3. $Z_m := \{< x >_{\Sigma'} \in Z | < x >_{\Sigma'} \cap X_m \neq \varnothing\}$
4. $\lambda : Z \times \Sigma' \to 2^Z$, where for any $(z, \sigma) \in Z \times \Sigma'$,

$$\lambda(z, \sigma) = \{z' \in Z | (\exists x \in z)(\exists u, u' \in (\Sigma - \Sigma')^*)\, \xi(x, u\sigma u') \cap z' \neq \varnothing\}$$ □

In practical applications G may consists of a large number of automata, namely $G = G_1 \times \cdots \times G_n$ for a very large number $n \in \mathbb{N}$, where $G_i \in \phi(\Sigma_i)$ for $i = 1, 2, \cdots, n$. How to compute $G/\approx_{\Sigma'}$ imposes a computational challenge. To overcome it a sequential abstraction algorithm is presented in [25]. Next, we discuss how to use automaton abstraction in distributed synthesis.

Problem of automaton-based distributed synthesis

Given $G = (X, \Sigma, \xi, x_0, X_m) \in \phi(\Sigma)$ let

$$B(G) = \{s \in \Sigma^* | (\exists x \in \eta(x_0, s))(\forall s' \in \Sigma^*)\, \xi(x, s') \cap X_m = \varnothing\}$$

Any string $s \in B(G)$ can lead to a state x, from which no marker state is reachable, i.e. for any $s' \in \Sigma^*$, $\xi(x, s') \cap X_m = \varnothing$. Such a state x is called a *blocking state* of G, and we call $B(G)$ the *blocking set*. G is *nonblocking* if $B(G) = \varnothing$. For each $x \in X$ let

$$E_G : X \to 2^\Sigma : x \mapsto E_G(x) = \{\sigma \in \Sigma | \xi(x, \sigma) \neq \varnothing\}$$

Thus, $E_G(x)$ is simply the set of all events allowable at x in G.

Given $G_i = (X_i, \Sigma_i, \xi_i, x_{i,0}, X_{i,m})$ $(i = 1, 2)$, the *product* of G_1 and G_2 is an automaton

$$G_1 \times G_2 = (X_1 \times X_2, \Sigma_1 \cup \Sigma_2, \xi_1 \times \xi_2, (x_{1,0}, x_{2,0}), X_{1,m} \times X_{2,m})$$

where $\xi_1 \times \xi_2 : X_1 \times X_2 \times (\Sigma_1 \cup \Sigma_2) \to 2^{X_1 \times X_2}$ is defined as follows,

$$(\xi_1 \times \xi_2)((x_1, x_2), \sigma) = \begin{cases} \xi_1(x_1, \sigma) \times \{x_2\} & \text{if } \sigma \in \Sigma_1 - \Sigma_2 \\ \{x_1\} \times \xi_2(x_2, \sigma) & \text{if } \sigma \in \Sigma_2 - \Sigma_1 \\ \xi_1(x_1, \sigma) \times \xi_2(x_2, \sigma) & \text{if } \sigma \in \Sigma_1 \cap \Sigma_2 \end{cases}$$

Clearly, $\times$ is commutative and associative. By a slight abuse of notations, from now on we use $G_1 \times G_2$ to denote its *reachable* part, which contains all states reachable from $(x_{1,0}, x_{2,0})$ by $\xi_1 \times \xi_2$ and transitions among these states. $\xi_1 \times \xi_2$ can be extended to $X_1 \times X_2 \times (\Sigma_1 \cup \Sigma_2)^*$.

Definition 6.7. [23] Given $G = (X, \Sigma, \xi, x_0, X_m) \in \phi(\Sigma)$ and $\Sigma' \subseteq \Sigma$, let $A = (Y, \Sigma', \eta, y_0, Y_m) \in \phi(\Sigma')$ and $P : \Sigma^* \to \Sigma'^*$ be the natural projection. A is *state-controllable* with respect to G and Σ_{uc} if for all $s \in L(G \times A)$ and all $x \in \xi(x_0, s)$ and $y \in \eta(y_0, P(s))$, we have $E_G(x) \cap \Sigma_{uc} \cap \Sigma' \subseteq E_A(y)$. □

We can check that, A is state controllable implies that $L(G \times A)\Sigma_{uc} \cap L(G) \subseteq L(G \times A)$. Thus, it is always true that state controllability implies controllability of $L(G \times A)$ with respect to G defined in [18]. But the inverse statement is not true unless both A and G are deterministic. We now introduce the concept of *state observability*.

Definition 6.8. [23] Given $G = (X, \Sigma, \xi, x_0, X_m) \in \phi(\Sigma)$ and $\Sigma' \subseteq \Sigma$, let $A = (Y, \Sigma', \eta, y_0, Y_m) \in \phi(\Sigma')$. A is *state-observable* with respect to G and P_o if for all $s, s' \in L(G \times A)$ with $P_o(s) = P_o(s')$, and for all $(x, y) \in \xi \times \eta((x_0, y_0), s)$ and $(x', y') \in \xi \times \eta((x_0, y_0), s')$,

$$E_{G \times A}(x, y) \cap E_G(x') \cap \Sigma' \subseteq E_A(y')$$

□

If A is state observable then for any two states (x, y) and (x', y') in $G \times A$ reachable by two strings s and s' having the same projected image (i.e. $P(s) = P(s')$), any event σ allowed at (x, y) and x' must be allowed at y' as well. We can check that, if A is state-observable then for all $s, s' \in L(G \times A)$ and $\sigma \in \Sigma$,

$$P_o(s) = P_o(s') \wedge s\sigma \in L(G \times A) \wedge s'\sigma \in L(G) \Rightarrow s'\sigma \in L(G \times A)$$

Thus, state observability implies observability of $L(G \times A)$ with respect to G and P_o defined in [11]. But the inverse statement is not always true unless both A and G are deterministic. Notice that, if $\Sigma_o = \Sigma$, namely every event is observable, A may still not be state-observable, owing to nondeterminism. In many applications we are interested in an even stronger observability property called *state normality* which is defined as follows.

Definition 6.9. [23] Given $G = (X, \Sigma, \xi, x_0, X_m) \in \phi(\Sigma)$ and $\Sigma' \subseteq \Sigma$, let $A = (Y, \Sigma', \eta, y_0, Y_m) \in \phi(\Sigma')$ and $P : \Sigma^* \to \Sigma'^*$ be the natural projection. A is *state-normal* with respect to G and P_o if for all $s \in L(G \times A)$ and $s' \in \overline{P_o^{-1}(P_o(s))} \cap L(G \times A)$, we have that, for all $(x, y) \in \xi \times \eta((x_0, y_0), s')$ and $s'' \in \Sigma^*$,

$$P_o(s's'') = P_o(s) \wedge \xi(x, s'') \neq \varnothing \Rightarrow \eta(y, P(s'')) \neq \varnothing$$

□

We can check that, if A is state-normal with respect to G and P_o, then

$$L(G) \cap P_o^{-1}(P_o(L(G \times A))) \subseteq L(G \times A)$$

which means $L(G \times A)$ is prefix normal with respect to G and P_o as defined in Def. 6.2. The inverse statement is not true unless both A and G are deterministic. Furthermore, we can check that state normality implies state observability. But the inverse statement is not true. We now introduce the concept of supervisor.

Definition 6.10. [23] Given $G \in \phi(\Sigma)$ and $H \in \phi(\Delta)$ with $\Delta \subseteq \Sigma' \subseteq \Sigma$, an automaton $S \in \phi(\Sigma')$ is a *nonblocking supervisor* of G under H, if S is deterministic and the following conditions hold:

1. $N(G \times S) \subseteq N(G \times H)$
2. $B(G \times S) = \varnothing$
3. S is state-controllable with respect to G and Σ_{uc}
4. S is state-observable with respect to G and P_o □

The first condition of Def. 6.10 says that the closed-loop system $G \times S$ complies with the specification H in terms of language inclusion. Because of this condition we only consider H to be deterministic. Later we will use the term 'nonblocking state-normal supervisor' (NSNS), when we want to emphasize that S is state-normal with respect to G and P_o. It has been shown in [24] that the set

$$\mathcal{CN}(G, H) = \{S \in \phi(\Sigma) | S \text{ is a NSNS of } G \text{ w.r.t. } H \wedge L(S) \subseteq L(G)\}$$

contains a unique element $\hat{S}$ such that for all $S \in \mathcal{CN}(G, H)$, we have $N(S) \subseteq N(\hat{S})$. We call $\hat{S}$ the *supremal nonblocking state-normal supervisor* of G under H with respect to Σ'. In [23] an algorithm is provided to compute such a supremal NSN supervisor.

Definition 6.11. A *distributed system* with respect to $\{\Sigma_i | i \in I\}$ is a collection of nondeterministic finite-state automata $\mathcal{G} = \{G_i = (X_i, \Sigma_i, \xi_i, x_{i,0}, X_{i,m}) \in \phi(\Sigma_i) | i \in I\}$. Each G_i ($i \in I$) is called the i^{th} *component* of $\mathcal{G}$, and $\Sigma_i = \Sigma_{i,c} \cup \Sigma_{i,uc} = \Sigma_{i,o} \cup \Sigma_{i,uo}$, where disjoint subsets $\Sigma_{i,c}$ and $\Sigma_{i,uc}$ comprise respectively the *controllable* events and *uncontrollable* events, and disjoint subsets $\Sigma_{i,o}$ and $\Sigma_{i,uo}$ for the *observable* events and *unobservable* events respectively.□

Distributed Supervisor Synthesis Problem:

Given a distributed system $\mathcal{G} = \{G_i \in \phi(\Sigma_i) | i \in I\}$ and a set of specifications $\mathcal{H} = \{H_j \in \phi(\Delta_j) | \Delta_j \subseteq \cup_{i \in I} \Sigma_i \wedge j \in J\}$, where J is an index set and each H_j is a deterministic automaton, synthesize a collection of deterministic finite-state automata $\mathcal{S} = \{S_k \in \phi(\Gamma_k) | \Gamma_k \subseteq \cup_{i \in I} \Sigma_i \wedge k \in K\}$, where K is an index set, such that the following conditions hold,

1. $N(G \times S) \subseteq N(G \times H)$
2. $B(G \times S) = \varnothing$
3. S is state-controllable with respect to G and $\Sigma_{uc} = \cup_{i \in I} \Sigma_{uc}$
4. S is state-normal with respect to G and $P_o : (\cup_{i \in I} \Sigma_i)^* \rightarrow (\cup_{i \in I} \Sigma_{i,o})^*$

where $G = \times_{i\in I} G_i$, $H = \times_{j\in J} H_j$ and $S = \times_{k\in K} S_k$. If such a collection $\mathcal{S}$ exists, then it is called a *nonblocking distributed supervisor* of $\mathcal{G}$ under $\mathcal{H}$, where each S_k is a *local supervisor*. □

To solve this problem we present an aggregative approach and a coordinated approach.

Aggregative distributed synthesis

Without loss of generality, suppose $I = \{1, 2, \cdots, n\}$. We present the following algorithm.

Aggregative Synthesis of Distributed Supervisor (ASDS)

1. Inputs:
   - plant model $\mathcal{G} = \{G_i \in \phi(\Sigma_i) | i \in I\}$
   - requirement mopdel $\mathcal{H} = \{H_j \in \phi(\Delta_j) | j \in J\}$.
   - suppose the ordering is: $(G_1, \cdots, G_n)$
2. Initially,
$$W_1 := G_1, J_1 := \{j \in J | \Delta_j \subseteq \Sigma_1\}, Q_1 := J_1, T_1 := \Sigma_1$$
3. Iterates on $k = 1, \cdots, n$,
   (a) If $J_k \neq \varnothing$, let $V_k := \times_{j\in J_k} H_j$. Otherwise, set V_k as the canonical recognizer of $(\Sigma_k)^*$.
   (b) Synthesize the supremal NSN supervisor S_k of W_k under V_k.
   (c) Terminate when S_k is empty or $k = n$. Otherwise, perform the following updates.
   (d) Set
$$\begin{aligned} I_{k+1} &:= \{i \in I | k+1 \leq i \leq n\} \\ \Sigma_{I_{k+1}} &:= \cup_{i\in I_{k+1}} \Sigma_i \\ \Theta_{k+1} &:= \cup_{j\in J-Q_k} \Delta_j \end{aligned}$$
   (e) Pick $\Sigma_{A_k} \subseteq T_k$ with $(\Sigma_{I_{k+1}} \cup \Theta_{k+1}) \cap T_k \subseteq \Sigma_{A_k}$. Let $A_k := (W_k \times S_k)/\approx_{\Sigma_{A_k}}$.
   (f) updates:
$$\begin{aligned} W_{k+1} &:= A_k \times G_{k+1} \\ Q_{k+1} &:= \{j \in J | \Delta_j \subseteq \cup_{i=1}^{k+1} \Sigma_i\} \\ J_{k+1} &:= Q_{k+1} - Q_k \\ T_{k+1} &:= \Sigma_{A_k} \cup \Sigma_{k+1} \end{aligned}$$
4. When terminate upon k, output $\mathcal{S} = \{S_1, S_2, \cdots, S_k\}$. □

Theorem 6.12. Let $\mathcal{S} = \{S_1, \cdots, S_n\}$ be computed by ASDS, where none of S_k ($k = 1, 2, \cdots, n$) is empty. Then $\mathcal{S}$ is a nonblocking distributed supervisor of $\mathcal{G}$ under $\mathcal{H}$. □

Although during the above construction $\mathcal{S}$ contains the same number of local supervisors as that of local components, some of them may not impose any control on the system. This can be checked whenever a local supervisor S_k is computed, and S_k imposes no control if and only if $L(S_k) = L(W_k)$. In that case we simply remove those local supervisors from $\mathcal{S}$ during online supervisory control. Clearly, the ordering is important not only for the computational complexity purpose but also for the existence of a distributed supervisor. Given a distributed

system $\mathcal{G}$, some ordering of local components may yield empty distributed supervisory control under ASDS. How to choose a good ordering is an interesting and important problem. Currently, we adopt a heuristic ordering procedure, which says that, *for any two components next to each other in an ordering, they must share events*. The rationality of this heuristics is that strongly coupled components (in terms of interactions through event sharing) should always be ordered close to each other. For example, suppose we have three components: a motor, a conveyor belt and a robot, where the motor drives the conveyor belt to move goods which are picked up by the robot. Given two orderings: (1) the motor, the conveyor belt and the robot; (2) the motor, the robot and the conveyor belt, it seems more reasonable for us to prefer ordering (1) to ordering (2) because there is no direct connection between the motor and the robot. This heuristics is used in our case studies. We are still searching for other heuristic procedures that may work better than this simple one.

By imposing an ordering over local components we may also attain a limited power of reusing local supervisors when some local component is added to the target system or dropped out of it, as often encountered in system reconfiguration. For example, suppose we have a distributed supervisor $\{S_1, \cdots, S_n\}$ with respect to an ordering $(G_1, \cdots, G_n)$. If we change or remove G_k ($1 \leq k \leq n$), we only need to redesign local supervisors $\{S_k, \cdots, S_n\}$. If we add some component $\hat{G}$ after G_k and before G_{k+1}, then we only need to redesign local supervisors associated with $\{\hat{G}, G_{k+1}, \cdots, G_n\}$. Thus, a certain degree of implementation flexibility is achieved.

Notice that in ASDS all relevant automata are required to be standardized for the sake of using automaton abstraction. It is of our interest to know how to synthesize a nonblocking distributed supervisor for a distributed system modeled by non-standardized automata. Fortunately this can be easily coped with by using the concepts of *standardization* and *destandardization*, as described in [23]. It turns out that introducing the notion of τ and the concept of standardized automata, which are crucially important for automaton abstraction, does not impose any restriction on supervisor synthesis.

Coordinated distributed synthesis

Given a distributed system $\mathcal{G} = \{G_i \in \phi(\Sigma_i) | i \in I\}$, suppose each local component G_i ($i \in I$) has its deterministic local specification $H_i \in \phi(\Delta_i)$, where $\Delta_i \subseteq \Sigma_i$. Furthermore, there is one deterministic specification $H \in \phi(\Delta)$, where $\Delta \subseteq \cup_{i \in I} \Sigma_i$. We would like to synthesize a nonblocking distributed supervisor $\mathcal{S}$ of $\mathcal{G}$ under $\{H, H_i | i \in I\}$. We have the following result.

Theorem 6.13. [22] Suppose for each G_i we have a nonblocking supervisor $S_i \in \phi(\Sigma_i)$ under H_i. Let $\Sigma' \subseteq \cup_{i \in I} \Sigma_i$ such that $\cup_{i,j:i \neq j} \Sigma_i \cap \Sigma_j \subseteq \Sigma'$ and $\Delta \subseteq \Sigma'$. Let $S \in \phi(\Sigma')$ be a nonblocking supervisor of $\times_{i \in I}((G_i \times S_i)/\approx_{\Sigma_i \cap \Sigma'})$ under H. Then $S \times_{i \in I} S_i$ is a nonblocking supervisor of $\times_{i \in I} G_i$ under $\times_{i \in I} H_i \times H$. □

We first synthesize a local supervisor S_i for each component G_i so that the local specification H_i can be enforced. Then we compute an abstraction so that we can synthesize a local supervisor to take care of H. In practical applications sometimes a specification, say H_i, may cover several local components, say $\{G_{il} \in \phi(\Sigma_{il}) | l = 1, \cdots, r\}$, in the sense that, $\Delta_i \subseteq \cup_{l=1}^{r} \Sigma_{ij}$. In this case, we can compute $G_i = \times_{l=1}^{r} G_{il}$ and treat it as a local component so that H_i is defined for G_i. Thus, the setup in Theorem 6.13 is general enough. In Theorem 6.13 we call each S_i a *local supervisor* of $\mathcal{G}$ and S a *coordinator* of $\mathcal{G}$, which is mainly used to coordinate local supervisors $\{S_i | i \in I\}$ to avoid conflict. The existence of S gives rise to the term *coordinated distributed supervisor*. Of course, S itself is a supervisor, which enforces the specification H. Theorem 6.13 allows us to synthesize a multiple-level multiple-coordinator distributed supervisor. For example, the system in Theorem 6.13 may be only a single module

of a large system. Thus, after obtaining $\{S_i | i \in I\} \cup \{S\}$, we can compute an appropriate abstraction of $\times_{i \in I}(G_i \times S_i) \times S$ so that high-level local supervisors and/or coordinators can be synthesized.

Discussion

If we carefully examine their individual control structures, we can see that the aggregative synthesis is a special case of the coordinated synthesis in the sense that at the initial stage there is only one local supervisor S_1, and all the rest of the local supervisors are essentially (multiple-level) coordinators. The main feature of the aggregative synthesis is that at each synthesis step there is always one local plant created from the product of one abstracted model and one local component model, and only one local supervisor is synthesized. Thus, with a good ordering of local components we can efficiently handle the complexity issue. Nevertheless, finding a good ordering of components is a practical challenge, which requires knowledge of component interactions in a system. As a contrast, the coordinated synthesis does not require any ordering of components, although knowledge of a system's architecture is certainly helpful for a user to partition the system into modules. But the coordinated synthesis has a potential drawback, namely we need to form a product of abstracted models of all relevant modules before we can synthesize a coordinator. If there are many modules under consideration, then their product may impose a computational challenge. For this reason, in a complex system with many modules we may need to use both aggregative and coordinated synthesis approaches to cope with the complexity issue, where coordinated synthesis is used to deal with low-level local supervisors and coordinators, and aggregative synthesis is used for high-level coordinators.

Bibliography

[1] S. Balemi. *Control of discrete event systems: theory and application*. PhD thesis, Swiss Federal Institute of Technology Zurich, 1992.

[2] N.C.W.M. Braspenning. *Model-based integration and testing of high-tech multi-disciplinary systems*. PhD thesis, Eindhoven University of Technology, 2008.

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

[4] L. Feng and W.M. Wonham. Computationally efficient supervisor design: abstraction and modularity. In *Proceedings of the 8th International Workshop on Discrete Event Systems (WODES06)*, pages 3–8, 2006.

[5] H. Flordal, R. Malik, M. Fabian, and K. Akesson. Compositional synthesis of maximally permissive supervisors using supervisor equivalence. *Discrete Event Dynamic Systems*, 17(4):475–504, 2007.

[6] S. Forschelen, J.M. van de Mortel-Fronczak, R. Su, and J.E. Rooda. Application of supervisory control theory to theme park vehicles. In *Proceedings of WODES, Berlin, Germany*, pages 303–309, 2010.

[7] S.T.J. Forschelen. Supervisory control of theme park vehicles. MSc thesis, Eindhoven University of Technology, 2010.

[8] R.C. Hill, D.M. Tilbury, and S. Lafortune. Modular supervisory control with equivalence-based conflict resolution. In *Proceedings of the 27th American Control Conference (ACC08)*, pages 491–498, 2008.

[9] J. Huang and R. Kumar. Directed control of discrete event systems for safety and nonblocking. *IEEE Transactions on Automation Science and Engineering*, 5(4):620–629, 2008.

[10] R.J. Leduc, M. Lawford, and W.M. Wonham. Hierarchical interface-based supervisory control — part II: parallel case. *IEEE Transactions on Automatic Control*, 50(9):1336–1348, 2005.

[11] F. Lin and W.M. Wonham. On observability of discrete-event systems. *Information Sciences*, 44(2):173–198, 1988.

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

[13] P. Malik. *From supervisory control to nonblocking controllers for discrete event systems*. PhD thesis, University of Kaiserslautern, 2003.

[14] R. Malik and H. Flordal. Yet another approach to compositional synthesis of discrete event systems. In *Proceedings of the 9th International Workshop on Discrete Event Systems (WODES08)*, pages 16–21, 2008.

[15] J. Markovski, K.G.M. Jacobs, D.A. van Beek, L.J.A.M. Somers, and J.E. Rooda. Coordination of resources using generalized state-based requirements. In *Proceedings of WODES, Berlin, Germany*, pages 297–302, 2010.

[16] R. Milner. Operational and algebraic semantics of concurrent processes. In *Handbook of theoretical computer science (vol. B): formal models and semantics*, pages 1201–1242. MIT Press, 1990.

[17] P.J. Ramadge and W.M. Wonham. Modular feedback logic for discrete event systems. *SIAM J. Control and Optimization*, 25(5):1202–1218, 1987.

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

[19] K. Rudie and W.M. Wonham. Think globally, act locally: decentralized supervisory control. *IEEE Transactions on Automatic Control*, 37(11):1692–1708, 1992.

[20] R.R.H. Schiffelers, R.J.M. Theunissen, D.A. van Beek, and J.E. Rooda. Model-based engineering of supervisory controller using cif. *Electronic Communication of the European Association of Software Science and Technology*, 21(9):1–10, 2009.

[21] R. Su and J.G. Thistle. A distributed supervisor synthesis approach based on weak bisimulation. In *Proceedings of the 8th International Workshop on Discrete Event Systems (WODES06)*, pages 64–69, 2006.

[22] R. Su, J. van Schuppen, and J. Rooda. Synthesize nonblocking distributed supervisors with coordinators. In *Proceedings of the 17th Mediterranean Conference on Control and Automation*, pages 1108–1113, 2009.

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

[24] R. Su, J.H. van Schuppen, and J.E. Rooda. Model abstraction of nondeterministic finite state automata in supervisor synthesis. *IEEE Transactions on Automatic Control*, 55(11):2527–2541, 2010.

[25] R. Su, J.H. van Schuppen, J.E. Rooda, and A.T. Hofkamp. Nonconflict check by using sequential automaton abstractions based on weak observation equivalence. *Automatica*, 46(6):968–978, 2010.

[26] D.A. van Beek, M.A. Reniers, J.E. Rooda, and R.R.H. Schiffelers. Concrete syntax and semantics of the compositional interchange format for hybrid systems. In *Proc. of the 17th IFAC World Congress*, 2008.

[27] W.M. Wonham and P.J. Ramadge. Modular supervisory control of discrete event systems. *Maths. of Control, Signal & Systems*, 1(1):13–30, 1988.

[28] T.S. Yoo and S. Lafortune. A general architecture for decentralized supervisory control of discrete-event systems. *Discrete Event Dynamic Systems*, 12(3):335–377, 2002.