Discrete Event Dyn Syst (2012) 22:511–540
DOI 10.1007/s10626-012-0130-6

Application of supervisory control theory to theme park vehicles

Stefan T. J. Forschelen · Joanna M. van de Mortel-Fronczak · Rong Su · Jacobus E. Rooda

Received: 4 March 2011 / Accepted: 12 January 2012 / Published online: 19 February 2012

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.

Keywords Supervisory control · Distributed control · Control engineering · Control applications · Implementation

The work described was performed in 2009 in cooperation with Sioux Electronics (formerly NBG Industrial Automation B.V.), in the framework of Twins 05004 project under the European Community's EUREKA cluster program ITEA 2.

S. T. J. Forschelen · J. M. van de Mortel-Fronczak (✉) · R. Su · J. E. Rooda
Department of Mechanical Engineering, Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
e-mail: j.m.v.d.mortel@tue.nl

Present Address:
S. T. J. Forschelen
Quintiq Applications, P.O. Box 264, 5201 AG 's-Hertogenbosch, The Netherlands

Present Address:
R. Su
Division of Control and Instrumentation, School of Electrical and Electronic Engineering,
Nanyang Technological University, 50 Nanyang Avenue, Singapore 639798, Singapore

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 Ramadge and Wonham (1987b) 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.

To design and build systems in a structured way, engineering processes have been introduced under the common denominators systems architecting (Rechtin and Maier 1997) and systems engineering (Martin 1996). Such a process starts with a global definition of the design, usually partitioned into subsystems or modules, based on the requirements that the system should fulfill. The global design is used to set up requirements for the modules, for which again designs are defined and then built. Every module is tested separately to check that the requirements are satisfied. Subsequently, the modules are integrated and the complete system is also tested with respect to its requirements. The V-model of Rook (1986) is often used in the context of software projects. Traditionally, most of the designs and requirements are captured in documents. More recently, engineers started to use executable software models to test the designs before they are actually built. An advantage of such models is that they can be used not only to analyse designed system behaviour but also to investigate how components or modules that are already built interact with the rest of the system that is not yet built. One can think of, for instance, checking if the system is nonblocking or estimating system performance. In Braspenning et al. (2011), evidence is provided that executable models can help in improving system quality

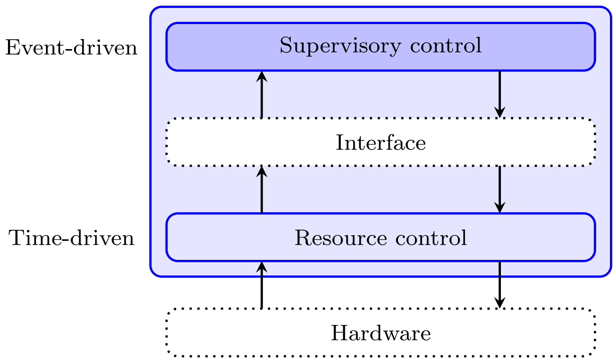


Fig. 1 Positioning supervisory control

and in decreasing time to market. Additionally, models support evolvability, that is they can easily be adapted to a similar system. The model-based engineering process incorporating formal models, as proposed in Braspenning (2008), enables the use of various model-based analysis techniques and tools to support system development. Several model-based systems engineering methodologies are commercially available, such as IBM Telelogic Harmony-SE, INCOSE Object-Oriented Systems Engineering Method (OOSEM), IBM Rational Unified Process for Systems Engineering (RUP SE) for Model-Driven Systems Development (MDSD) and Vitech Model-Based Systems Engineering (MBSE) Methodology. All of them include processes, methods and tools supporting development of systems and all can be characterized as approaches supporting manual design. Valuable as they are for many organizations in achieving a paradigm shift from traditional document-based approach to model-based approach, with respect to control software design they focus on design model formulation instead of derivation. Supervisory control synthesis allows for a more automated design approach and can be integrated in this setting. It has the potential to reduce human errors and also to speed up the development cycle.

Integration of executable models and supervisory control synthesis in engineering processes is rendered in the scheme of Fig. 2, introduced in Schiffelers et al. (2009). 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. The scheme illustrates important elements and activities in the framework described above. 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 R_S related to supervisory control and R_P related to the (uncontrolled) plant 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

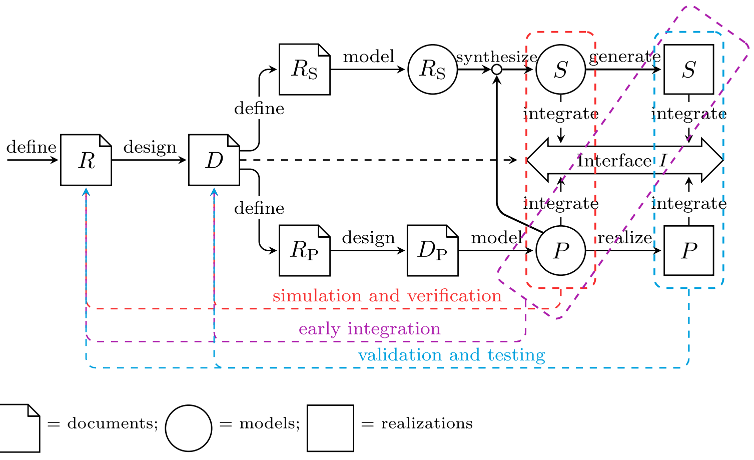


Fig. 2 The engineering process with supervisory control synthesis

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 and validation are discussed. In Section 6, a prototype implementation is presented. Finally, Section 7 concludes the paper by showing the impact supervisory control synthesis can have on a product development process. This paper is an extended version of Forschelen et al. (2010).

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. Ramadge and Wonham (1987b). Unfortunately, centralized approaches cannot overcome the state-space explosion phenomenon. To deal with this computational complexity issue, several advanced synthesis techniques have been introduced recently. For example, in Ma and Wonham (2006) a new state-based synthesis approach is proposed, which utilizes state-tree structures plus binary decision diagrams to encode states so that reachability search can be done efficiently. The approach can handle fairly large systems even though it is centralized. In Leduc et al. (2005) an interface-based hierarchical synthesis approach is presented, which fully exploits the benefit of interface invariance in system decoupling so that synthesis is carried out only in local components. One potential drawback of this approach is that the conditions for interface invariance may be too strong for many practical applications. In Flordal et al. (2007) and Malik and Flordal (2008) compositional synthesis approaches are provided which aggregate component models one after another hoping that "bad" behaviors can be dropped out during the process of aggregation, instead of being kept to the last stage which usually results in extremely high complexity. Notable attentions should also be paid to some recent distributed synthesis approaches presented in e.g., Su and Thistle (2006), Feng and Wonham (2006), Hill et al. (2008) and Su et al. (2009, 2010a), which utilize (language-based or automaton-based) model abstraction techniques when dealing with local synthesis. These distributed approaches 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 illustrated in Fig. 3.

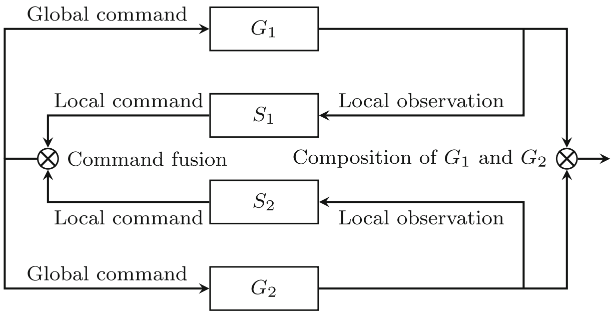
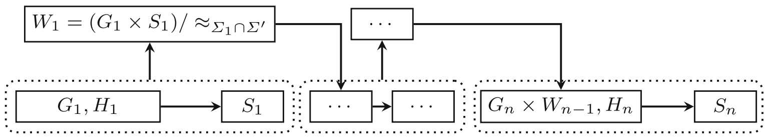


Fig. 3 A distributed control architecture

Two local components G_1 and G_2 are composed through parallel composition. 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 S_1 and S_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 (Rudie and Wonham 1992), or the disjunctive rule which defines the global control command as the union of local commands (Yoo and Lafortune 2002), or the combination of both conjunctive and disjunctive rules (Yoo and Lafortune 2002). To make control more effective, communication may be allowed among local supervisors (see e.g., Rudie et al. 2003), and to reduce potential costs of maintaining sensor readings in a partially observed system sensors may be activated only when necessary (see e.g., Wang et al. 2010).

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 Ramadge and Wonham (1987a) and Wonham and Ramadge (1988) with a focus on STS symbolic technique introduced in Ma and Wonham (2006). 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 Markovski et al. (2010) that can precisely specify both state-based and event-based requirements. In event-based synthesis, we apply the aggregative distributed synthesis technique from Su et al. (2010a) and the coordinated distributed synthesis technique from Su et al. (2009). The theoretical details of individual synthesis techniques can be found in the publications mentioned above, here we only provide intuitive explanations supported by simple pictorial illustrations.

Figure 4 illustrates the aggregative distributed synthesis. To explain this approach, suppose we have three components G_1, G_2 and G_3, whose alphabets are Σ_1, Σ_2 and Σ_3 respectively. We need to order them first according to some structural consideration, as described in Su et al. (2010a). For the illustration purpose we assume that the ordering is G_1, G_2, G_3. Suppose there are three specifications H_1, H_2, H_3, whose alphabets are Δ_1, Δ_2 and Δ_3 respectively. We assume that 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 $\Delta_3 \cap \Sigma_3 \neq \varnothing$. We use "$\times$" to denote the automaton synchronous product (see, e.g., Wonham 2011).



$W_1 = (G_1 \times S_1)/\approx_{\Sigma_1 \cap \Sigma'}$
$\cdots$
G_1, H_1
S_1
$\cdots$
$\cdots$
$G_n \times W_{n-1}, H_n$
S_n

Fig. 4 Synthesis of aggregative distributed supervisor

To perform aggregative synthesis, first, we compute the supremal nonblocking state-normal supervisor S_1 of G_1 under the specification H_1. Normality is used to deal with partial observation. (When there is no H_1 satisfying the conditions described, H_1 becomes the canonical recognizer of Σ_1^*. 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 abstraction process is essentially a quotient construction modulo the weak observation equivalence relation defined in Su et al. (2010b), which has the property that an automaton that is nonconflicting with an abstracted model is guaranteed to be nonconflicting with the original model. This important property ensures that supervisor synthesis can be done based on an abstracted model, which is usually much simpler than the original model. The alphabet Σ' of the abstraction 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 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. The system in Fig. 5 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.

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

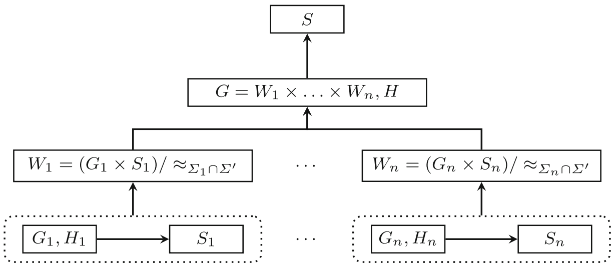

S
$G = W_1 \times \ldots \times W_n, H$
$W_1 = (G_1 \times S_1)/\approx_{\Sigma_1 \cap \Sigma'}$
$W_n = (G_n \times S_n)/\approx_{\Sigma_n \cap \Sigma'}$
G_1, H_1
S_1
G_n, H_n
S_n

Fig. 5 Synthesis of coordinated distributed supervisor

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.

3 Case study: a theme park vehicle

As mentioned in Section 1, current industrial practice shows that systems engineering processes are still mostly based on documents. This holds especially for SME's (Small and Medium-sized Enterprises), where usually the V-model is applied in the design of the control part and no formal models of the requirements are used. The purpose of the project reported in this paper was to investigate in an industrial setting the applicability of supervisor synthesis techniques integrated in a model-based engineering process. To this end, a real industrial product is chosen as a case study: a flexible vehicle that can be used in theme parks or museums, called the multimover. The multimover, as shown in Fig. 6a, is a battery-operated Automated

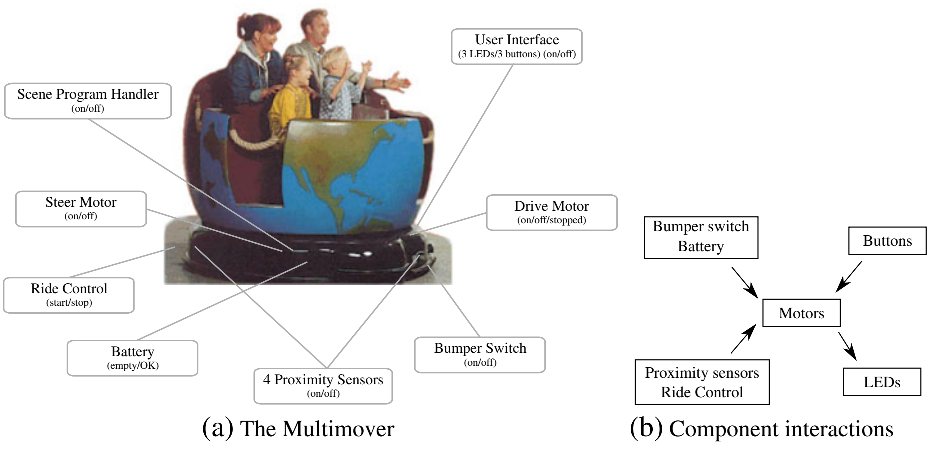


Fig. 6 Theme park vehicle

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).

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 Fig. 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 (I)** 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 (II)** 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 (III)** The user interface of the multimover contains three buttons that may only be used by the operator. 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.
- **Emergency and error handling (IV)** 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.
- **Proximity and Ride Control handling (V)** 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.

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. Figure 7

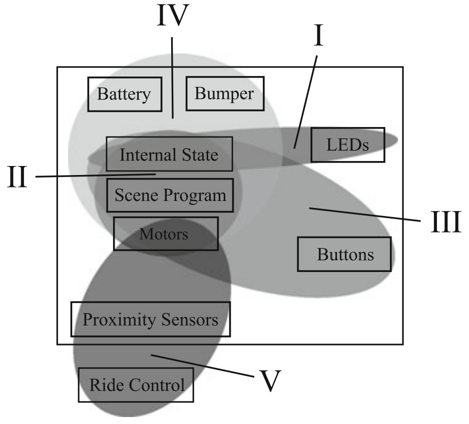


Fig. 7 Partitioning of the control problem

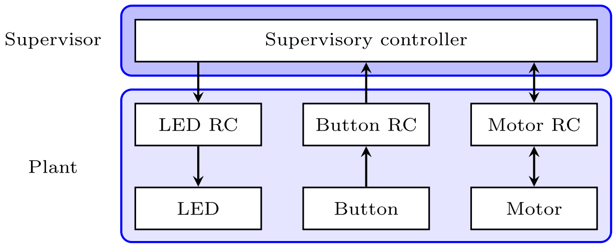


Fig. 8 The control architecture

shows a graphical representation of the partitioning of the multimover control problem. This forms the basis for distributed synthesis.

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 Fig. 8. The arrows represent the information flow between the components, the resource controllers and the supervisor.

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

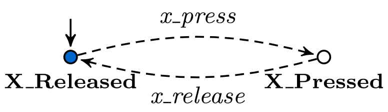


Fig. 9 Button automaton

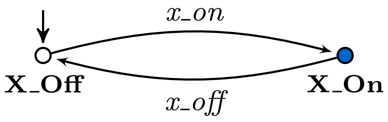


Fig. 10 LED automaton

an unconnected incoming arrow and marker states are denoted by filled vertices. Controllable and uncontrollable events are denoted by solid and dashed edges, respectively. The component models have disjoint alphabets.

The user interface of the multimover contains three buttons (the reset, the forward and the backward button) that are used to reset the vehicle and to deploy the vehicle into the ride and three LEDs (the reset, the forward and the backward LED) that are used to show its actual state. In Fig. 9, the automaton representing the buttons is depicted. It has two uncontrollable events: the event that represents the button being pressed (*x_press*) and the event that represents the button being released (*x_release*). For the reset button, X (*x*) should be replaced by RB (*rb*), for the forward button by FB (*fb*) and for the backward button by BB (*bb*). In Fig. 10, the automaton representing the forward and backward LEDs is depicted. The controllable events *x_on* and *x_off* represent the function call to switch the LED on and off, respectively. As for the buttons, for the forward LED, X (*x*) should be replaced by FL (*fl*) and for the backward LED by BL (*bl*). As shown in Fig. 11, in the model of the reset LED, the initial and marker states are different.

All sensors are modelled by automata having the same structure, as depicted in Fig. 12 for a sensor. The sensor can generate two events: *x_active* and *x_inactive*. As explained in Section 3, the multimover is equipped with two long-range proximity sensors, FLP and BLP, two short-range proximity sensors, FSP and BSP, and the bumper switch, BS, a sensor mounted on the bumper that can detect physical contact with an object. Again, for individual sensor models, X and *x* should be substituted by the respective sensor names.

A different kind of sensor measures the battery level. If the battery level is below a certain limit, an uncontrollable event *ba_empty* is sent. If the vehicle is charged, *ba_ok* is sent. The automaton representing this functionality is depicted in Fig. 13.

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 Fig. 14. Note that the events mentioned above are uncontrollable, since the supervisor cannot disable them.

The Scene Program Handler shown in Fig. 15, reads the scene programs provided by the customer and sends certain commands to the rotation device, drive motor, steer motor and audio player. Since only starting (*sh_enable*) and stopping (*sh_disable*) the reading of the scene program is relevant for our supervisor, only these events are modelled. Because a scene program could contain a command that

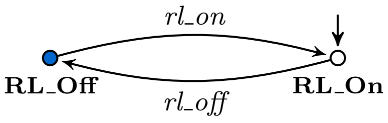


Fig. 11 Reset LED automaton

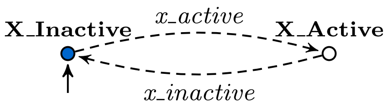


Fig. 12 Sensor automaton

the multimover should start driving in the opposite direction, the uncontrollable event *sh_chdir* is modelled. If the scene program file contains a parse error, the multimover should stop moving and enter the emergency mode. If a parse error is read, the uncontrollable event *sh_error* occurs.

In Fig. 16a, 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**.

In Fig. 16b, 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*.

The multimover itself can also be in three states, namely **MM_Emergency**, **MM_Reset** and **MM_Active**, see Fig. 17. **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.

Marker states are used to describe completed tasks. As stated in Malik (2003), they represent states that we always want to be reachable by any behaviour. 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.

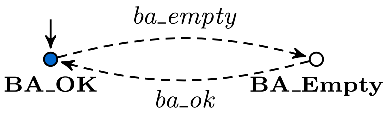


Fig. 13 Automaton of the battery sensor

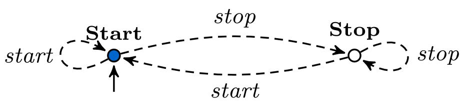


Fig. 14 Automaton of ride control

We have chosen for component models representing the uncontrolled behaviour. As shown in Fig. 8 and explained above, the models of plant components used for supervisory control problem definition describe the component behaviour in terms of discrete commands (turning on and off, setting the speed level and direction) to and messages (failure notification, sensor values) from the resource controllers. This means that an abstraction is made from physical variables like velocity and displacement. With respect to velocity, only the distinction is made between two predefined speed values, namely normal and slow. With respect to displacement, only the direction (backwards and forwards) and rotation are taken into account. In this way, the plant models are obtained that match exactly the behaviour of the interface of the components defined at the resource-control level. 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.

All requirements described in Section 3 are formally specified. The automata depicted in Fig. 18 specify the control requirements of the emergency and error handling control module. Figure 18a and b 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 Fig. 18c, 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 requirements for the remaining modules can be found in the Appendix.

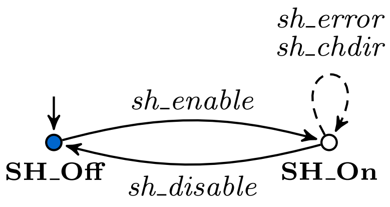
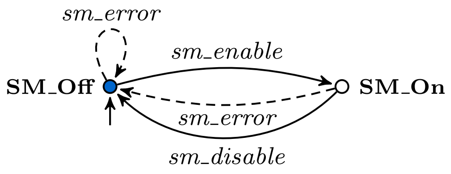
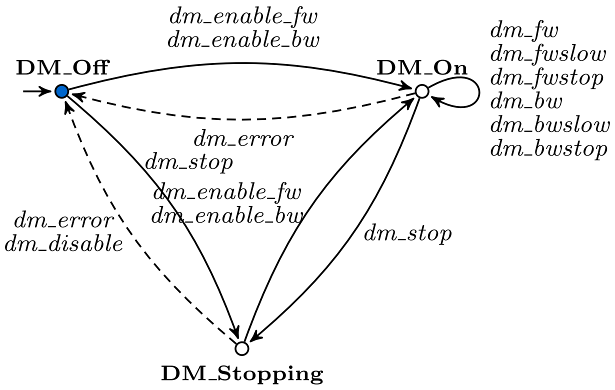


Fig. 15 Automaton of the Scene Program Handler

Fig. 16 Automata of motors



(a) Automaton of the steer motor

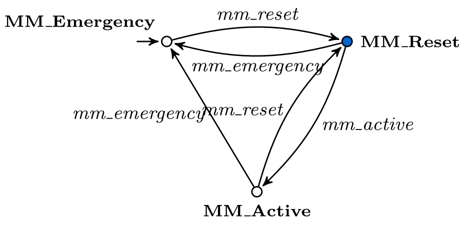


(b) Automaton of the drive motor

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 Markovski et al. (2010), deriving the suitable state-based expressions can be an error-prone and tedious task. To avoid this inconvenience, logical 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 Fig. 18a and b 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

Fig. 17 Model of the multimover



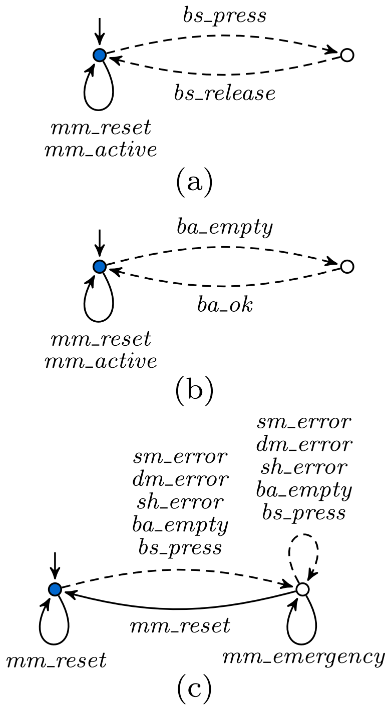


Fig. 18 Requirement models of the emergency module

of the battery is sufficient (**BA_OK**). Using the syntax introduced in Markovski et al. (2010), this logical expression reads: $\rightarrow$ { *mm_reset, mm_active* } $\Rightarrow$ **BS_Released** $\downarrow$ $\wedge$ **BA_OK** $\downarrow$.

To explain the syntax for the logical expressions of Markovski et al. (2010), we take unique state names as a starting point. In the expressions, 1 and 0 as truth values of propositional logic, negation $\neg$, conjunction $\wedge$, disjunction $\vee$ and implication $\Rightarrow$ as standard logical operators, state predicates $\mathbf{s}\downarrow$ and event predicates $\rightarrow E$ can be used. The state predicate $\mathbf{s}\downarrow$ expresses that a component is in state **s**. The event predicate $\rightarrow E$ expresses that an event from set E is enabled by the supervisor. The logical expressions used in this paper are of the form:

$$ST ::= \ \rightarrow E \Rightarrow MS$$

where $MS ::= 1 \mid \mathbf{s}\downarrow \ \mid \neg MS \mid MS \ op \ MS$ and $op \in \{\wedge, \vee, \Rightarrow\}$.

Logical expression defined above can be used instead of automata. However, not every requirement can be specified as such a logical expression, which is illustrated in the Appendix. As described in the next section, the same collection of requirements in terms of logical expressions and automata is used as basis for state-based and event-based synthesis.

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

Table 2 Distributed coordinated supervisors

Module	# states	# transitions
LED actuation	25	77
Motor actuation	41	222
Button handling	193	1,541
Emergency handling	181	2,149
Proximity handling	481	4,513

5 Supervisor synthesis and validation

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 Ma and Wonham (2006) based on state tree structures. The 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 Su et al. (2009) and the event-based aggregative approach of Su et al. (2010a). 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.

For both approaches, the same multimover component models are used. The state tree structure needed for the state-based approach applied consists of one AND superstate consisting of 17 OR component superstates with which the corresponding automata are associated. In both cases, also the same requirements (the collection of logical expressions and automata specified in the Appendix) are used as a starting point. The logical expressions are translated to the state-based expression format required by the NBC tool supporting the Ma and Wonham (2006) approach. The synthesis step takes only a few seconds. 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 Forschelen (2010), a detailed description of the results is provided.

Table 3 Distributed aggregative supervisors depending on synthesis order

Module	Order	# states	# transitions	Order	# states	# transitions
LED actuation	1	25	77	5	41	125
Motor actuation	2	41	222	2	257	1,428
Button handling	3	465	3,477	4	177	765
Emergency handling	4	89	626	3	118	609
Proximity handling	5	225	1,953	1	481	4,513

For the application of the coordinated distributed technique, first local supervisors are synthesized for modules defined in Section 3 and associated requirements (see Appendix). As all modules under control of their supervisors are nonconflicting, no coordinator is needed and, hence, no automaton abstraction needs to be calculated.

For the application of the aggregated distributed technique, first the local supervisor is synthesized for the LED actuation module and its requirements (see Appendix). An automaton abstraction of the module under control of its supervisor is calculated and in combination with the motor actuation module used for synthesis of the second local supervisor, etc. The sizes of the local supervisors synthesized according to this procedure, in terms of numbers of states and transitions, are stated in the left part of Table 3. In the right part of the table, the sizes of the local supervisors synthesized in a different order are listed. In general, the existence of the solution, when using aggregated procedure, may depend on the order in which modules are used for synthesis. In the multimover case, several solutions are possible.

The necessary steps of both distributed synthesis techniques are performed using the SuSyNA package, see SE Group, Eindhoven University of Technology (2010). The synthesis and automaton abstraction steps take only a few seconds. The nonconflicting check requires a longer calculation time, approximately one hour.

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. All scenario's related to pressing the buttons and activating sensors are evaluated, so that every local supervisor is activated, including the supervisor for the emergency and error handling module. The CIF-toolset described in van Beek et al. (2008), in which also synthesis tools are integrated, is used for discrete-event simulation.

6 Supervisor implementation

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., Balemi (1992). 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 Dietrich et al. (2002). 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. Simulation-based analysis of the synthesized supervisors indicated that there are no infinite loops of controllable events. This means that each execution of the selected controllable event brings the system closer to the desired final state. In this sense, although the supervisory controller still can choose between two control actions, it functionally resembles a directed controller of Huang and Kumar (2008).

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 Fig. 19, 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 Malik (2003). 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

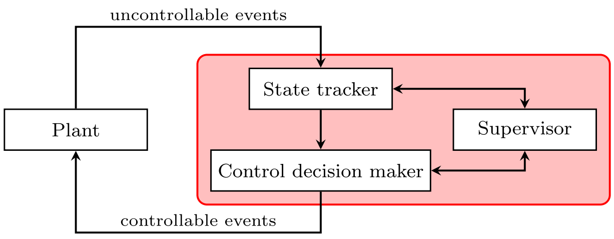


Fig. 19 A supervisory controller

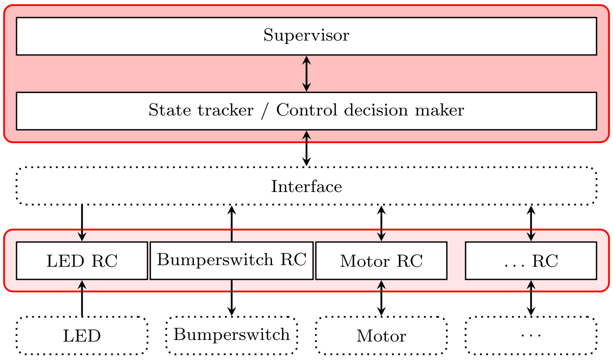


Fig. 20 Control architecture of the implementation

architecture of the multimover with the supervisory controller is given in Fig. 20. The implemented supervisory controller replaces the existing control software that has the same functionality.

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 used to implement distributed event handling systems. Libraries supporting this paradigm are available in the C platform used for our case study. The resource control of each component can publish messages of a certain topic and can subscribe to a certain topic, which means that it receives 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 Fig. 20 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”). This buffer is emptied, one by one, by the state tracker by taking and removing its first element ($E \leftarrow$ pop(list)). 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

Algorithm 1 Supervisory controller implementation

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

(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 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 Fig. 20 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 Forschelen (2010).

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 on a test rack and, then, on a real vehicle. For testing, the same scenarios were used as in simulation-based analysis of the synthesized supervisors 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. We conclude that both types of supervisors, state-based and event-based, are equally well suited for the system under investigation. Testing indicated that the event-based supervisor is faster, but the state-based supervisor needs less memory.

As mentioned previously, the purpose of the project described in this paper was to show that our model-based engineering process incorporating supervisor synthesis can reduce the time-to-market and development costs. To this end, we compared the time needed for implementation of changes associated with integration of one additional sensor and introduction of a few new requirements in the actual company project and in our set up. The company project leader indicated that for this change, approximately two man-days were needed to design, implement and test the new control software, and subsequently to integrate it in the existing control system and test the integration, according to the V-model. In our set up, approximately four hours were needed. However, we observed that as long as synthesis tools are not incorporated in a commercially available model-based engineering set up, the threshold for applying supervisor synthesis in industrial projects is still too high.

7 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 the synthesis-based approach described in this paper, 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, the remaining requirements are defined and explained per module. Most requirements are in the form of logical expressions. Automata are used for requirements that cannot be defined as logical expressions. This collection of logical expressions and automata is used for synthesis as explained in Section 5.

LED actuation

The reset LED may only be switched off if the status of the multimover is active or reset.
$\rightarrow \{ \textit{rl\_off} \} \Rightarrow (\textbf{MM\_Active} \downarrow \vee \textbf{MM\_Reset} \downarrow)$

The reset LED may only be switched on if the status of the multimover is emergency.
$\rightarrow \{ \textit{rl\_on} \} \Rightarrow \textbf{MM\_Emergency} \downarrow$

The forward LED may only be switched on if the status of the multimover is reset.
$\rightarrow \{ \textit{fl\_on} \} \Rightarrow \textbf{MM\_Reset} \downarrow$

The forward LED may only be switched off if the status of the multimover is active or emergency.
$\rightarrow \{ \textit{fl\_off} \} \Rightarrow (\textbf{MM\_Active} \downarrow \vee \textbf{MM\_Emergency} \downarrow)$

The backward LED may only be switched on if the status of the multimover is reset.
$\rightarrow \{ \textit{bl\_on} \} \Rightarrow \textbf{MM\_Reset} \downarrow$

The backward LED may only be switched off if the status of the multimover is active or emergency.
$\rightarrow \{ \textit{bl\_off} \} \Rightarrow (\textbf{MM\_Active} \downarrow \vee \textbf{MM\_Emergency} \downarrow)$

Motor actuation

The Scene Program Handler may only be switched off only if the status of the multimover is reset or emergency.
$\rightarrow \{ \textit{sh\_disable} \} \Rightarrow (\textbf{MM\_Reset} \downarrow \vee \textbf{MM\_Emergency} \downarrow)$

The Drive Motor may only be stopped if the status of the multimover is reset or emergency and the Scene Program Handler is off.
$\rightarrow \{ \textit{dm\_stop} \} \Rightarrow ((\textbf{MM\_Reset} \downarrow \vee \textbf{MM\_Emergency} \downarrow) \wedge \textbf{SH\_Off} \downarrow)$

The Steer Motor may only be switched off if the status of the multimover is reset or emergency and the Drive Motor is off.
$\rightarrow \{ \textit{sm\_disable} \} \Rightarrow ((\textbf{MM\_Reset} \downarrow \vee \textbf{MM\_Emergency} \downarrow) \wedge \textbf{DM\_Off} \downarrow)$

The Steer motor may only be switched on if the status of the multimover is Active.
$\rightarrow \{ \textit{sm\_enable} \} \Rightarrow \textbf{MM\_Active} \downarrow$

The Drive Motor may only be switched on if the status of the multimover is Active and the Steer Motor is on.
$\rightarrow \{ \textit{dm\_enable\_fw, dm\_enable\_bw} \} \Rightarrow (\textbf{MM\_Active} \downarrow \wedge \textbf{SM\_On} \downarrow)$

The Scene Program Handler may only be switched on if the status of the multimover is Active, the Steer Motor is on and the Drive Motor is on.
$\rightarrow \{ \textit{sh\_enable\_on} \} \Rightarrow (\textbf{MM\_Active} \downarrow \wedge \textbf{SM\_On} \downarrow \wedge \textbf{DM\_On} \downarrow)$

The Drive Motor may only execute another drive command if the multimover is Active.
$\rightarrow \{ \textit{dm\_enable\_fw, dm\_enable\_bw, dm\_fw, dm\_fwslow, dm\_fwstop, dm\_bw, dm\_bwslow, dm\_bwstop} \}$
$\Rightarrow \textbf{MM\_Active} \downarrow$

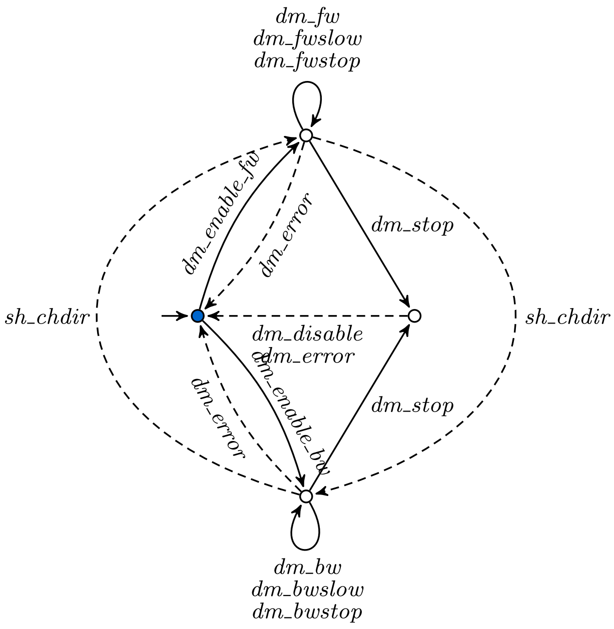


Fig. 21 Requirement of the motor actuation module

The automaton depicted in Fig. 21 specifies the relationship between the Scene Program Handler and the drive motor. If the Scene Program Handler receives a command to change the direction *sh_chdir*, the active state of the drive motor is changed. This requirement cannot be specified as a logical expression.

Button handling

The multimover may only switch to Active if the forward button or the backward button (not both) is pressed and the reset button is not pressed.
$\rightarrow \{\, mm\_active \,\} \Rightarrow (((\mathbf{FB\_Pressed} \downarrow \wedge \mathbf{BB\_Released} \downarrow) \vee (\mathbf{BB\_Pressed} \downarrow \wedge \mathbf{FB\_Released} \downarrow)) \wedge \neg\, \mathbf{RB\_Pressed} \downarrow)$

The multimover may only switch to Reset if the reset button is pressed.
$\rightarrow \{\, mm\_reset \,\} \Rightarrow \mathbf{RB\_Pressed} \downarrow$

The automata depicted in Fig. 22a and b determine the occurrence of the events *dm_enable_fw* and *dm_enable_bw*. Both events are only allowed if first the corresponding interface button is pressed (*fb_press* or *bb_press*) and then the multimover has become active (*mm_active*). Note that in this case, the occurrence of an event depends on a specific order of other events which must by specified by automata.

Proximity and ride control handling

The multimover must stop driving in the forward direction only if the status of Ride Control is Stop or the FSP sensor is active.
$\rightarrow \{\, dm\_fwstop \,\} \Rightarrow (\mathbf{RC\_Stop} \downarrow \vee \mathbf{FSP\_Active} \downarrow)$

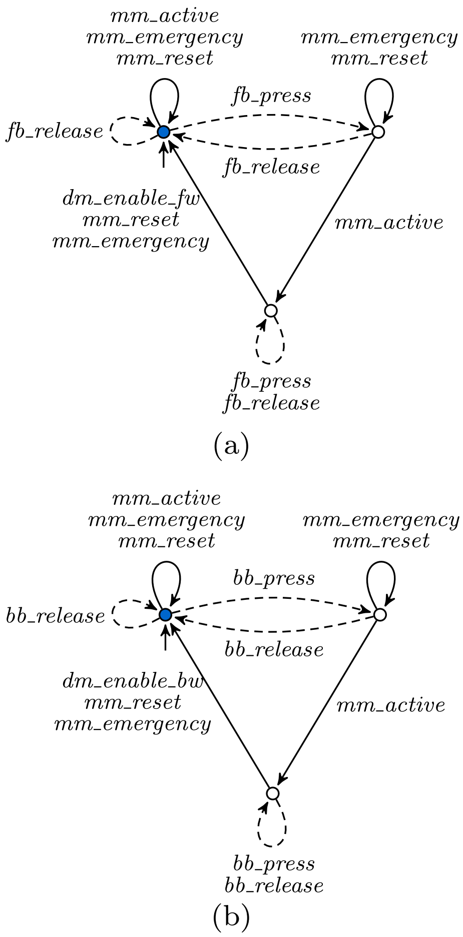


Fig. 22 Requirements of the button handling module

The multimover must stop driving in the backward direction only if the status of Ride Control is Stop or the BSP sensor is active.
$\rightarrow$ { *dm_bwstop* } $\Rightarrow$ (**RC_Stop** $\downarrow \vee$ **BSP_Active** $\downarrow$)

The multimover must continue driving in the forward direction only if the status of Ride Control is Start and the FSP sensor is inactive.
$\rightarrow$ { *dm_fwslow, dm_fw* } $\Rightarrow$ (**RC_Start** $\downarrow \wedge$ **FSP_Inactive** $\downarrow$)

The multimover must continue driving in the backward direction only if the status of Ride Control is Start and the BSP sensor in the backward direction is inactive.
$\rightarrow$ { *dm_bwslow, dm_bw* } $\Rightarrow$ (**RC_Start** $\downarrow \wedge$ **BSP_Inactive** $\downarrow$)

The multimover must slow down in the forward direction only if the FLP sensor is active.
$\rightarrow$ { *dm_fwslow* } $\Rightarrow$ **FLP_Active** $\downarrow$

The multimover must drive at regular speed in the forward direction only if the FLP sensor is inactive.
$\rightarrow$ { *dm_fw* } $\Rightarrow$ **FLP_Inactive** $\downarrow$

The multimover must slow down in the backward direction only if the BLP sensor is active.

$\rightarrow \{$ *dm_bwslow* $\} \Rightarrow$ **BLP_Active** $\downarrow$

The multimover must drive at regular speed in the backward direction only if the BLP sensor is inactive.

$\rightarrow \{$ *dm_bw* $\} \Rightarrow$ **BLP_Inactive** $\downarrow$

The proximity module contains one requirement specified by an automaton. Since this automaton is too large to depict here, only a description is given. This requirement specifies the occurrence of the events *dm_fw*, *dm_fwslow*, *dm_fwstop*, *dm_bw*, *dm_bwslow*, *dm_bwstop* and *dm_stop*. Each of these events except *dm_stop* is not allowed to take place twice without the occurrence of another event in between. The automaton has seven states, S_0 through S_6. S_0 is the initial and marker state. From every state including S_0, event *dm_stop* goes to S_0. From every state except S_1, event *dm_fw* goes to S_1. From every state except S_2, event *dm_fwslow* goes to S_2. From every state except S_3, event *dm_fwstop* goes to S_3. From every state except S_4, event *dm_bw* goes to S_4. From every state except S_5, event *dm_bwslow* goes to S_5. Finally, from every state except S_6, event *dm_fwstop* goes to S_6.

Emergency and error handling

All requirements of this module are defined in Section 4.

References

Balemi S (1992) Control of discrete event systems: theory and application. PhD thesis, Swiss Federal Institute of Technology Zurich

van Beek D, Reniers M, Rooda J, Schiffelers R (2008) Concrete syntax and semantics of the compositional interchange format for hybrid systems. In: Proc. of the 17th IFAC World Congress

Braspenning N (2008) Model-based integration and testing of high-tech multi-disciplinary systems. PhD thesis, Eindhoven University of Technology

Braspenning N, Boumen R, van de Mortel-Fronczak J, Rooda J (2011) Estimating and quantifying the impact of using models for integration and testing. Comput Ind 62(1):65–77

Dietrich P, Malik R, Wonham W, Brandin B (2002) Implementation considerations in supervisory control. In: Synthesis and control of discrete event systems. Kluwer Academic Publishers, pp 185–201

Feng L, Wonham W (2006) Computationally efficient supervisor design: abstraction and modularity. In: Proceedings of WODES 2006, pp 3–8

Flordal H, Malik R, Fabian M, Akesson K (2007) Compositional synthesis of maximally permissive supervisors using supervisor equivalence. Discrete Event Dyn Syst 17(4):475–504

Forschelen S (2010) Supervisory control of theme park vehicles. MSc thesis, Eindhoven University of Technology

Forschelen S, van de Mortel-Fronczak J, Su R, Rooda J (2010) Application of supervisory control theory to theme park vehicles. In: Proceedings of WODES 2010, pp 303–309

Hill R, Tilbury D, Lafortune S (2008) Modular supervisory control with equivalence-based conflict resolution. In: Proceedings of ACC 2008, pp 491–498

Huang J, Kumar R (2008) Directed control of discrete event systems for safety and nonblocking. IEEE Trans Autom Sci Eng 5(4):620–629

Leduc R, Lawford M, Wonham W (2005) Hierarchical interface-based supervisory control—part II: parallel case. IEEE Trans Automat Contr 50(9):1336–1348

Ma C, Wonham W (2006) Nonblocking supervisory control of state tree structures. IEEE Trans Automat Contr 51(5):782–793

Malik P (2003) From supervisory control to nonblocking controllers for discrete event systems. PhD thesis, University of Kaiserslautern

Malik R, Flordal H (2008) Yet another approach to compositional synthesis of discrete event systems. In: Proceedings of WODES 2008, pp 16–21
Markovski J, Jacobs K, van Beek D, Somers L, Rooda J (2010) Coordination of resources using generalized state-based requirements. In: Proceedings of WODES 2010, pp 297–302
Martin J (1996) Systems engineering guidebook. CRC Press
Ramadge P, Wonham W (1987a) Modular feedback logic for discrete event systems. SIAM J Control Optim 25(5):1202–1218
Ramadge P, Wonham W (1987b) Supervisory control of a class of discrete event processes. SIAM J Control Optim 25(1):206–230
Rechtin E, Maier M (1997) The art of systems architecting. CRC Press
Rook P (1986) Controlling software projects. Softw Eng J 1(1):7–16
Rudie K, Wonham W (1992) Think globally, act locally: decentralized supervisory control. IEEE Trans Automat Contr 37(11):1692–1708
Rudie K, Lafortune S, Lin F (2003) Minimum communication in a distributed discrete-event system. IEEE Trans Automat Contr 48(6):957–975
Schiffelers R, Theunissen R, van Beek D, Rooda J (2009) Model-based engineering of supervisory controller using cif. Electronic Communications of the EASST 21(9):1–10
SE Group, Eindhoven University of Technology (2010) SuSyNA package. http://se.wtb.tue.nl/sewiki/supcon/susyna
Su R, Thistle J (2006) A distributed supervisor synthesis approach based on weak bisimulation. In: Proceedings of WODES 2006, pp 64–69
Su R, van Schuppen J, Rooda J (2009) Synthesize nonblocking distributed supervisors with coordinators. In: Proceedings of MED 2009, pp 1108–1113
Su R, van Schuppen J, Rooda J (2010a) Aggregative synthesis of distributed supervisors based on automaton abstraction. IEEE Trans Automat Contr 55(7):1627–1640
Su R, van Schuppen J, Rooda J (2010b) Model abstraction of nondeterministic finite state automata in supervisor synthesis. IEEE Trans Automat Contr 55(11):2527–2541
Wang W, Lafortune S, Lin F, Girard A (2010) Minimization of dynamic sensor activation in discrete event systems for the purpose of control. IEEE Trans Automat Contr 55(11):2447–2461
Wonham W (2011) Supervisory control of discrete-event systems. Tech. rep., University of Toronto, Toronto
Wonham W, Ramadge P (1988) Modular supervisory control of discrete event systems. Math Control Signals Syst 1(1):13–30
Yoo T, Lafortune S (2002) A general architecture for decentralized supervisory control of discrete-event systems. Discrete Event Dyn Syst 12(3):335–377

Stefan T. J. Forschelen received the M.Sc. degree in mechanical engineering from the Eindhoven University of Technology, Eindhoven, The Netherlands, in 2010.

Since 2010, he is with Quintiq Applications, ’s-Hertogenbosch, The Netherlands.

Joanna M. van de Mortel-Fronczak received the M.SC. degree in computer science from the AGH University of Science and Technology, Cracow, Poland, in 1982 and the Ph.D. degree in computer science from the Eindhoven University of Technology, Eindhoven, The Netherlands, in 1993.

Since 1997, she has been with the Department of Mechanical Engineering, Eindhoven University of Technology, where she is currently an Assistant Professor. Her research interests include model-based engineering and synthesis of supervisory machine control systems.

Rong Su received the M.A.Sc. and Ph.D. degrees both in electrical engineering from the University of Toronto, Toronto, Canada, in 2000 and 2004, respectively.

He is affiliated with the School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore. His research interests include modeling, fault diagnosis and supervisory control of discreteevent dynamic systems.

Dr. Su has been a member of IFAC technical committee on discrete event and hybrid systems (TC 1.3) since 2005.

Jacobus E. Rooda received the M.Sc. degree from Wageningen University of Agriculture Engineering, Wageningen, The Netherlands, and the Ph.D. degree from Twente University, Twente, The Netherlands.

Since 1985, he has been a Professor of (Manufacturing) Systems Engineering at the Department of Mechanical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands. His research fields of interest are modeling and analysis of manufacturing systems. His interest is especially in control of (high-tech) manufacturing lines and in supervisory control of high-tech (manufacturing) machines.