# Applied Supervisory Control for a Flexible Manufacturing System

**Thomas Moor** * **Klaus Schmidt** ** **Sebastian Perk** ***

* *Lehrstuhl für Regelungstechnik, Universität Erlangen-Nürnberg (e-mail: thomas.moor@rt.eei.uni-erlangen.de)*
** *Department of Electronic and Communication Engineering, Çankaya University, Ankara (e-mail: schmidt@cankaya.edu.tr)*
*** *HYDROMETER GmbH (e-mail: sebastian.perk@hydrometer.de)*

**Abstract:** This paper presents a case study in the design and implementation of a discrete event system (DES) of real-world complexity. Our DES plant is a flexible manufacturing system (FMS) laboratory model that consists of 29 interacting components and is controlled via 107 digital signals. Regarding controller design, we apply a hierarchical and decentralised synthesis method from earlier work in order to achieve nonblocking and safe closed-loop behaviour. Regarding implementation, we discuss how digital signals translate to discrete events from a practical point of view, including timing issues. The paper demonstrates how both, design and implementation, are supported by the open-source software tool `libFAUDES`.

*Keywords:* Discrete event systems, supervisory control, manufacturing system, implementation.

## 1. INTRODUCTION

In this paper, we report on the design and the implementation of a controller for a laboratory model of a flexible manufacturing system (FMS). By *design* we refer to the application of a methodology that, given the plant dynamics, provides a way to compute controller dynamics such that the closed-loop provably fulfils a formal specification. By *implementation* we refer to the organisation of hardware and software required to actually run the controller on the physical plant.

Our laboratory model consists of 1 stack feeder (SF), 16 conveyor belts (C), 4 rotary tables (T), 2 rail-transport units (R), 2 processing stations (M), 2 pushers (P) and 2 roll-conveyors (RC); see Fig. 1. All in all, 57 digital output and 50 digital input signals are connected to a standard PC via two digital IO boards. Considering the clearly observable gap between available methodology and engineering practice, we employ this example system to discuss how modern methods from discrete event systems theory can be applied to real world scenarios.

In order to employ an automata-based controller synthesis technique, we first discuss how *digital signals* can be mapped to and from *sequences of asynchronous events*, and how the actuator/sensor paradigm translates to the controllability attribute of events in our setting. As a result, this step allows us to determine formal models for the individual plant components in the form of one finite automaton per component. In this context, it has to be noted that an overall plant model has an estimated number of $10^{24}$ states, where the estimate is based on the product of component models. For the controller design, we apply a hierarchical and decentralized approach developed in our previous work (Schmidt et al., 2008). Based on individual models per plant component, a set of supervisors is

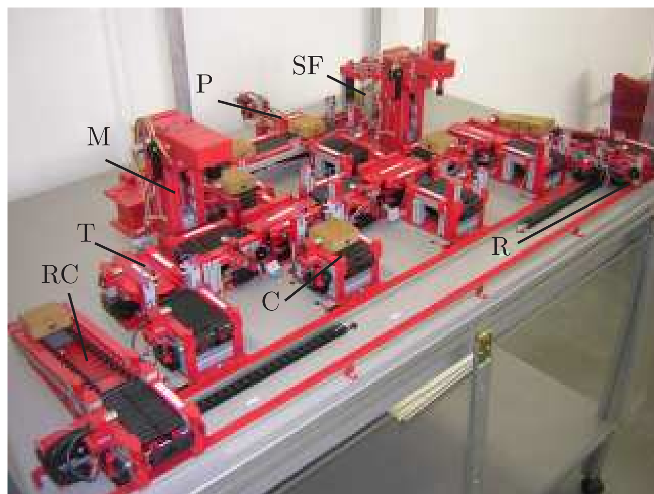


Fig. 1. Flexible manufacturing system model.

computed that leads to safe and nonblocking closed-loop behaviour. For the example at hand, this method results in 39 supervisors with an average number of about 100 states each, leading to an overall closed-loop in the order of $10^{30}$ states. The controller synthesis method is supported by the C++-library `libFAUDES` that is continuously developed at the Lehrstuhl für Regelungstechnik, Universität Erlangen-Nürnberg, and is available under terms of the *lesser GNU public license (LGPL)*; see (libFAUDES, 2006–2010; Moor et al., 2008).

`libFAUDES` also supports the physical realisation of the controller by providing the specialized *simulator* plug-in for simulation of the supervisor automata in synchronous composition with the physical plant (so called *hardware-in-the-loop simulation*). In this context, it is pointed out that the common model of discrete time is based on event

ordering and has to be synchronized with physical time, including situations in which multiple events occur at the same physical time. We address these issues by defining appropriate execution semantics, similar to those proposed in (Fabian and Hellgren, 1998; Basile and Chiacchio, 2007).

In summary, our work demonstrates that the application of the supervisory control theory to practical examples is by all means feasible, whereby it has to be acknowledged that profound expertise in the supervisory control of DES and software support for the entire work-flow are essential. This observation conforms with related research efforts that investigate various aspects of the design and implementation of DES supervisors (Leduc, 1996; Chandra et al., 2003; Vyatkin et al., 2006; Ljungkrantz et al., 2007).

The paper is organized as follows. In Section 2, we discuss how the physical plant interface is mapped to an abstract event-based interface. In Section 3, we develop automata models for individual plant components as a basis for the controller design reported in Section 4. The implementation of the resulting controller to the physical plant is presented in Section 5. Section 6 gives conclusions.

## 2. PHYSICAL PLANT

The laboratory setup implements a closed-loop configuration where a *physical plant* interacts with a *physical controller* [1]; see Fig. 2. Both systems interact via digital input signals and digital output signals. In this context, a digital signal is seen as a function with a Boolean range defined on the continuous time axis. An arbitrary input signal can be applied to the plant, which in turn produces a particular output signal. As a physical system, the relationship between input and output signals is causal, and, when appropriately modelled, deterministic. The physical perspective contrasts the perspective commonly taken in an automata based controller design, where component interaction is modelled by the synchronisation of an abstract sequence of events. In this section, we report on the hardware and software infrastructure that maps the signal-based interface to an event-based interface.

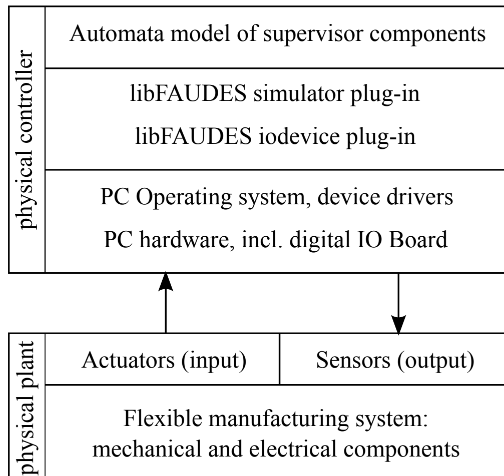


Fig. 2. Schematic of the laboratory setup.

[1] We use the terminology *physical* plant/controller to refer to an actual piece of equipment that is part of the laboratory setup, in contrast to a plant/controller *model* used during the design process.

The plant under consideration consists of electrical and mechanical components that resemble a flexible production line; see Fig. 1. An overall number of 25 DC motors serve as actuators, controlled by two digital signals each that correspond to clockwise and counterclockwise operation, respectively. Thus, the plant is controlled by 50 digital input signals. Relevant positions of the various mechanical components are sensed by 22 switch-keys. In addition, the plant is equipped with 35 sensors that indicate whether or not a workpiece is at a certain position. Thus, the plant provides 57 digital output signals for feedback.

The plant is connected to a standard PC equipped with two digital I/O boards, adequately wired with the plant input- and output-signals. A natural translation from a plant output signal to an event sequence is to generate *sensor events* whenever an edge has been detected. Vice versa, the execution of an *actuator event* shall clear or set the level of a plant input signal and thereby impose an edge. This scheme has been implemented in the `libFAUDES` iodevice plug-in. At the time of writing, the plug-in accesses digital signals via the Comedi open-source device drivers (Comedi, 2008), that support a wide range of hardware devices. The iodevice plug-in is configured by defining the correspondence between (a) digital signals addressed by a pin address, (b) polarity of edges and (c) symbolic event names.

Note that, independent of the particular software implementation, there is a principle issue with simultaneous sensor events: if two edges occur simultaneously, the event sequence to generate is not uniquely defined. One way circumvent this issue is to require the plant model to recognise any logical ordering of events that may occur simultaneously w.r.t. physical time. A well designed supervisor will then handle any particular ordering faithfully. The current version of the iodevice plug-in implements edge detection by polling the signals at a configurable sampling rate and writing detected events to a FIFO buffer. Thus, in addition to the general issue with simultaneous events, we require that a supervisor must accept an arbitrary ordering of any two events that follow each other within less than the sample period. Again, this can be guaranteed by using an adequate plant model in the controller design process.

The below Listing 1 shows an example configuration addressing a rotary table component (T); see Fig. 3. The rotary table has two defined positions (x-position and y-position) that are indicated by switch-keys. A DC motor

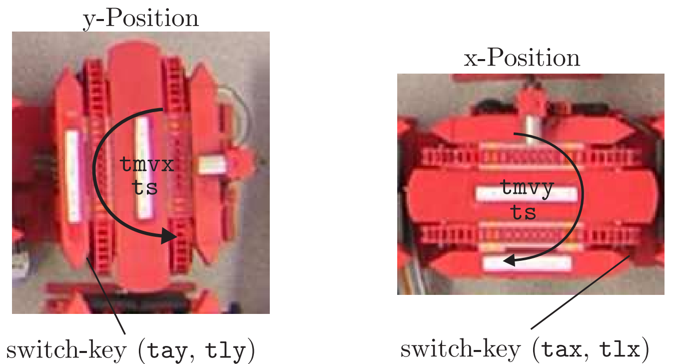


Fig. 3. Two defined positions of the rotary table (T).

can be activated via two distinct digital signals for either clockwise or counter-clockwise operation, directing the table towards the y- and x-position, respectively. The device configuration defines the actuator events `tmvx` (“move to the x-position”), `tmvy` (“move to the y-position”), and `ts` (“stop”) to set and clear the two signals accordingly. The sensor events `tax`/`tlx`/`tay`/`tly` are defined to indicate the table to `arive`/`leave` the `x`- or `y`-position, respectively.

Fig. 4 illustrates the signal levels and event generations of the rotary table when moving from the y-position to the x-position. Note that the illustration is idealised in that the stop event `ts` is executed instantaneously, i.e. $t_2 = t_3$, where $t_k$ denotes the physical time of the $k$-th event. In practice, a delay is expected and it is crucial that this delay is well below the time it would take until `tly` occurred to indicate that the table had overrun the switch-key. We come back to this issue in the following section.

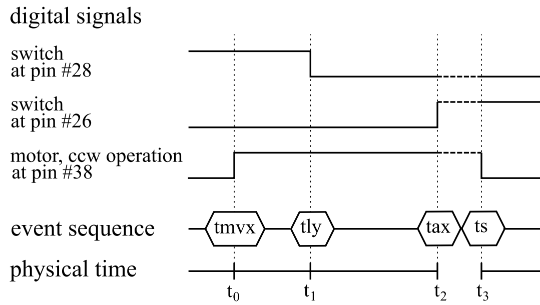


Fig. 4. Mapping signals to events

Listing 1. Example of a device configuration

```
<DeviceContainer>
"LrtLabSignalIO"
<Devices>

<ComediDevice>
"LrtLabInputDevice"  "/dev/comedi0"
<EventConfiguration>
% rotary table arriving in x position
"tax"
<Sensor>
<Triggers> 26 +PosEdge+ </Triggers>
</Sensor>
% rotary table leaving x position
"tlx"
<Sensor>
<Triggers> 26 +NegEdge+ </Triggers>
</Sensor>
[... more sensor events ...]
</EventConfiguration>
</ComediDevice>

<ComediDevice>
"LrtLabOutputDevice"  "/dev/comedi1"
<EventConfiguration>
% rotate table clockwise
"tmvy"
<Actuator>
<Actions> 38 +Clr+ 36 +Set+ </Actions>
</Actuator>
% rotate table counter-clockwise
"tmvx"
<Actuator>
<Actions> 36 +Clr+ 38 +Set+ </Actions>
</Actuator>
% stop rotation
"ts"
<Actuator>
<Actions> 36 +Clr+ 38 +Clr+ </Actions>
</Actuator>
[... more actuator events ...]
</EventConfiguration>
</ComediDevice>

</Devices>
</DeviceContainer>
```

## 3. PLANT MODELING

In order to preserve the modular structure of the flexible manufacturing system, we propose to determine a separate model for the behaviour of each component. Each model refers to the corresponding sensor- and actuator events of the respective component. Optionally, component models may introduce additional events to simplify the modelling of component interaction and to support efficient system abstraction. These additional events are called *logical events*. In this section, we continue the rotary table (T) as an example to present a detailed component model. For the remaining components we give statistical information.

The automaton $G_{\mathrm{T}}$ in Fig. 5 describes the dynamic behaviour of T with the relevant sensor- and actuator events `tax`, `tlx`, `tay`, `tly`, `tmvx`, `tmvy`, and `ts`, introduced in the previous section. Actuator events are considered controllable, sensor events as uncontrollable. Furthermore, it is assumed that T initially holds the y-position, and the marking captures that it is always desired to reach one of the defined positions.

The additional event $\mathtt{t}_{\mathrm{T}}$ represents the passage of time, and `tf` indicates the system failure that happens if the rotary table overruns one of the switch-keys. Both events are logical events and $\mathtt{t}_{\mathrm{T}}$ is considered to be controllable. In this regard, the model represents the fact, that a physical controller can prevent an overrun by immediately stopping the rotation when the respective position is reached [2] . This imposes a timing constraint on the implementation of the controller in that the immediate execution of `ts` after `tax` or `tay` must not have a delay longer than the time represented by $\mathtt{t}_{\mathrm{T}}$.

The logical events `txy` and `tyx` are supposed to initiate the motion of T directed to the x- and y-position, respectively. Both events are considered controllable and serve as an interface to the next higher level in a hierarchical controller design: a high-level controller shall be able to prevent a change of position independent of low level realisation details.

Regarding the software support with `libFAUDES`, $G_{\mathrm{T}}$ is stored in a specific data structure, and can either be input via the graphical user interface `DESTool` or directly in an XML-based file format; see also (Moor et al., 2008).

The remaining plant components are modeled in an analogous way. For the entire FMS, we end up with models for 29 components that have a sum of 692 states, 114/100/104

[2] This modelling technique can be interpreted as a particular simple variant of so called *forcible events*; see (Brandin and Wonham, 1994).

sensor/actuator/logical events and an estimated overall state space in the order of $10^{24}$. For the sake of conciseness, we only list the respective state and event counts in Table 1, complete models can be found on the webpage (libFAUDES, 2006–2010).

| comp. | # states | # sensors | # actuators | # logical ev. |
|---|---|---|---|---|
| SF | 14 | 4 | 2 | 1 |
| C | 28 | 3 | 3 | 4 |
| T | 14 | 4 | 3 | 2 |
| R | 34 | 10 | 3 | 10 |
| M | 34 | 4 | 5 | 2 |
| P | 16 | 5 | 3 | 2 |
| RC | 3 | 2 | 0 | 1 |

Table 1. Properties of the component models.

In course of our experiments we have experienced that deriving an adequate discrete event model of a physical phenomen is by no means trivial. While the choice of actuator and sensor events appears quite natural, the introduction of additional logical events at the stage of modelling is arbirary to some extent. However, the success of the synthesis method used in Section 4 crucially depends on which logical events were introduced. Without profound expertise in the synthesis method used, one is unlikely to succeed.

## 4. HIERARCHICAL SUPERVISOR SYNTHESIS

After obtaining a comprehensive plant model, we now address the supervisor computation. In this paper, we employ the hierarchical and decentralized control approach by Schmidt et al. (2008). For illustration, we apply this method to the *interconnection subsystem* (ICS) in Fig. 6 as a representative part of the FMS.

The ICS consists of the 7 components C5, C8, C9, C12, C13, T2 and T3 that are modeled analogous to the description in the previous section and with the statistical data listed in Table 1. Hence, the overall state space of the ICS comprises an order of $10^9$ states. From the functional perspective, the ICS allows the transport of products among the different parts of the FMS. In particular, 4 desired paths of products are shown in Fig. 6. In order to realize this system behaviour, we follow the hierarchical and decentralized supervisor synthesis in (Schmidt et al., 2008; Schmidt and Breindl, 2008) for the synthesis of nonblocking and maximally permissive supervisors. In the first step, local supervisors are designed for the respective plant components. We illustrate this procedure by the rotary table T2. The plant model $G_{\mathrm{T2}}$ conforms to $G_{\mathrm{T}}$ in Fig. 5 ("t" is simply replaced by "t2") and the specification

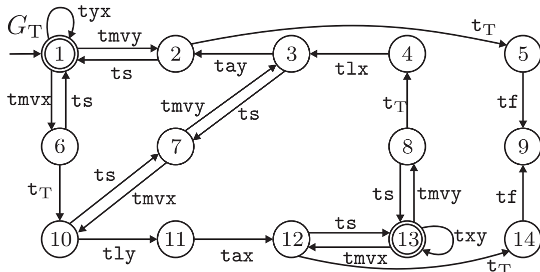


Fig. 5. Model of the rotary table.

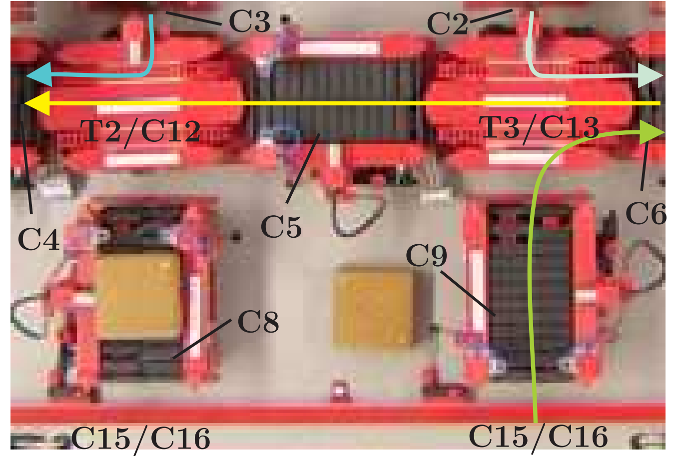


Fig. 6. Interconnection subsystem (ICS).

$C_{\mathrm{T2}}$ is shown in Fig. 7. It states that T2 has to move between its defined positions without stopping or changing the direction of motion. Then, $R_{\mathrm{T2}}$ in Fig. 7 represents the nonblocking and maximally permissive closed-loop for $G_{\mathrm{T2}}$ and $C_{\mathrm{T2}}$.

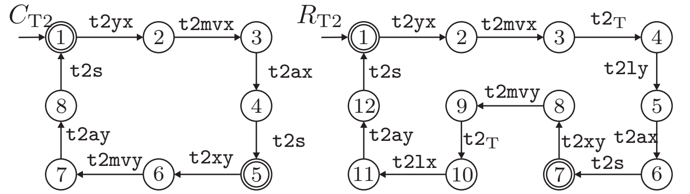


Fig. 7. Specification and supervisor for T2.

In the next step, we abstract the obtained closed-loop for a subsequent supervisor synthesis on the next hierarchical level (level 1). We use a natural projection $p_{\mathrm{T2}} : \Sigma_{\mathrm{T2}}^* \to (\Sigma_{\mathrm{T2}}^{(1)})^*$ from the original alphabet of $G_{\mathrm{T2}}$ to the abstraction alphabet $\Sigma_{\mathrm{T2}}^{(1)} = \{\mathtt{tyx}, \mathtt{txy}, \mathtt{ts}\}$, resulting in $G_{\mathrm{T2}}^{(1)}$ in Fig. 8. It has to be noted that $p_{\mathrm{T2}}$ fulfills both the *marked-string-accepting (msa)-observer* condition (Schmidt et al., 2008) and *local control consistency* (Schmidt and Breindl, 2008) in order to guarantee nonblocking and maximally permissive control.

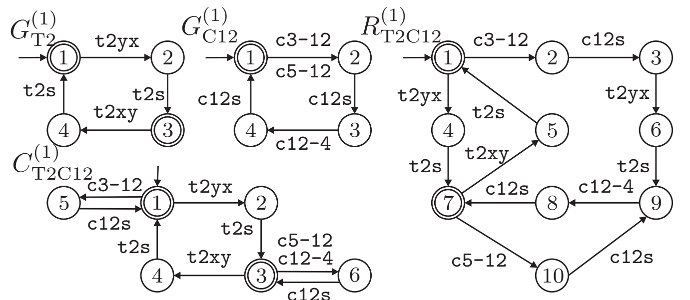


Fig. 8. Hierarchical synthesis for T2 and C12.

Next, we perform a joint supervisor synthesis of T2 and the conveyor belt C12 that is mounted on T2 on level 1 of the supervisor hierarchy. Fig. 8 already shows the abstracted model $G_{\mathrm{C12}}^{(1)}$ of C12 that operates according to the desired paths in Fig. 6. Here, the logical events `c3-12`, `c5-12` and `c12-4` characterize the transport of products from/to neighboring plant components, while the actuator

event `c12s` indicates that C12 stops. The joint specification $C^{(1)}_{\mathrm{T2C12}}$ ensures that T2 and C12 do not move at the same time, and the respective events in C12 can only happen if T2 is in the correct position. The resulting closed loop on level 1 is captured by $R^{(1)}_{\mathrm{T2C12}}$ in Fig. 8. Then, according to (Schmidt et al., 2008; Schmidt and Breindl, 2008), the overall nonblocking and maximally permissive closed loop for T2 and C12 is given by $R_{\mathrm{T2}}||R_{\mathrm{C12}}||R^{(1)}_{\mathrm{T2C12}}$. The corresponding supervisor hierarchy is depicted by the shaded box in Fig. 9. Moreover, an analogous synthesis for the remaining components of the ICS results in the overall supervisor hierarchy in Fig. 9. Here, the numbers indicate the state count of the respective closed-loop.

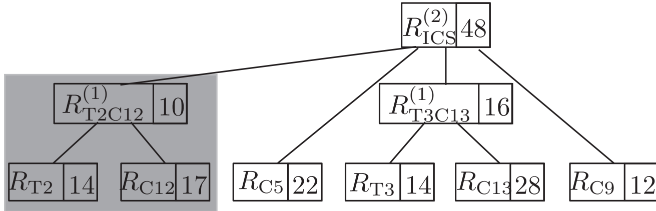


Fig. 9. Supervisor hierarchy for the ICS.

Regarding the software support, all required operations are implemented in `libFAUDES` and can be accessed via the scripting lanuguage `luafaudes` or the graphical interface DESTool. The function `SupConNB` performs the nonblocking supervisor synthesis, `Project` evaluates the natural projection, `IsMarkedStateAccepting` and `IsLocallyControlConsistent` verify the respective properties for natural projections, and the computation of appropriate projections for the hierarchical supervisor synthesis is done by `MSAObserverLcc`. The following `luafaudes` code performs the supervisor synthesis of T2 and C12.

Listing 2. Example Lua code

```
-- C12: local supervisor computation
plant = System("c12[0].gen")
spec = Generator("c12[0]_spec.gen")
supC12 = System()
SupConNB(plant,spec,supC12)

-- C12: MSA-observer computation with LCC
alph = EventSet("c12[1]_orig.alph")
highC12 = System()
MsaObserverLcc(sup,sup:ControllableEvents(),alph)
Project(supC12,alph,highC12)

-- T2: local supervisor computation
plant = System("t2[0].gen")
spec = Generator("t2[0]_spec.gen")
supT2 = System()
SupConNB(plant,spec,supT2)

-- T2: MSA-observer computation with LCC
alph = EventSet("t2[1]_orig.alph")
highC12 = System()
MsaObserverLcc(sup,sup:ControllableEvents(),alph)
Project(supT2,alph,highT2)

-- supervisor computation C12-T2 on level 1
Parallel(highC12,highT2,plant)
spec = Generator("t2c12[1]_spec.gen")
supC12T2 = System()
SupConNB(plant,spec,supC12T2)
```

Following the description in (Schmidt et al., 2008), a similar synthesis is performed for the remaining plant components. As with the DES models themselfs, the particular grouping of components and the particular choice of hierarchical abstraction levels requires some intuition on how the plant should be operated and is by no means trivial. For our laboratory setup, 39 supervisors on 5 hierarchical levels are computed, while all sufficient conditions for nonblocking and maximally permissive control are fulfilled. The average state size of the resulting supervisors is 100 in contrast to an overall closed loop with an estimated number of $10^{30}$ states. A complete description of the performed synthesis and the required data and source code can be found on the `libFAUDES` webpage (libFAUDES, 2006–2010).

## 5. SUPERVISOR IMPLEMENTATION

In order to apply the supervisor design from Section 4 to the physical plant, we propose the physical controller to simulate the supervisor dynamics, i.e., the parallel composition of the 39 supervisor components. Here, the simulation is required to synchronize actuator events and sensor events with the physical plant by means of the iodevice plug-in as presented in Section 2. The overall configuration can be interpreted as a *hardware-in-the-loop simulation* which, for our laboratory setup, is realised by the `libFAUDES` simulator plug-in.

Simulation of a set of automata models in the common semantics of the parallel composition is straightforward and can be done without explicit enumeration of the overall state set. Considering synchronization with the physical plant, we propose to map the paradigm of controllable and uncontrollable events to the actual situation of *actuator events*, *sensor events* and *logical events*. The mapping implemented by `libFAUDES` is built on two observations. On the one hand, sensor events are generated spontaneously and must be accepted as they occur. By the design of our supervisor and the imposed controllability property, it is guaranteed, that any sensor event that actually occurs will not be disabled by the supervisor at the time of its occurrence and can hence be executed by the simulator. On the other hand actuator events are exclusively controlled by the simulator and may be executed at any time. Since the execution of an actuator event amounts to changing signal levels of signals that are wired to actuators, actuator events are indeed accepted by the physical plant at any time. Additionally, in order to obtain a deterministic behaviour, the simulator imposes restrictions on the execution semantics based on priorities [3] and event type:

1. if a sensor event is available from the FIFO buffer for detected events and if it can be executed, do so instantly and continue with 1.;
2. if one or more actuator or logical events can be executed, execute the one with the highest priority instantly and continue with 1.;

[3] For the hierarchical synthesis method (Schmidt et al., 2008) used in our experiment, priorities on a per event basis are purely cosmetic. Other methods may or may not rely on event priorites.

3. if a sensor event is available from the FIFO buffer for detected events that could not be executed in 1. report a synchronisation error;
4. wait until the next sensor event is reported.

Repeated execution of steps 1. and 2. amounts to executing all enabled events, where sensor events are executed with priority while enabled. In this sense, events are put into order optimistically. Step 3 detects the error case in which the supervisor finally can not accept a sensor event that has been detected in the physical plant. Again, from the controllability property imposed by our design this error should not occur. Once execution has reached step 4, only sensor events are enabled, and hence the controller must wait until such an event occurs.

The only step that allows time to pass is step 4, all other steps are meant to take no physical time. However, in practice a delay is expected. As we have pointed out by the example of the rotary table (T), physical plant components may in particular states impose a constraint on the tolerable amount of delay. If the implementation violates this constraint by taking too much time to execute `ts` after e.g. `tay`, the plant will issue the sensor event `tly` at a time at which the latter is disabled by the supervisor. This situation is sensed in step 3. and the simulator will report a synchronisation error. Thus, it is important to record the tolerable delay during the modelling process and to carefully evaluate the performance of the controller hardware and software.

For the overall laboratory experiment, the hardware-in-the-loop simulation was configured to implement the 39 supervisors computed according to Section 4. In our experiments, the system behaved as desired, i.e., neither blocking nor violations of the safety specification could be observed. A video of the closed-loop system behaviour is available at http://www.rt.eei.uni-erlangen.de/FGdes/productionline. To monitor the controller performance, the current implementation of the simulator plug-in takes time stamps on entry and exit of step 4 to report statistical data on the actual delay. For the configuration at hand, the delay turns out far below the acceptable maximum and hence is of no particular concern. In the context of an industrial application, however, this topic should be considered more rigidly. Options include to compile rather than to interpret the supervisor dynamics, followed by a thorough performance analysis.

## 6. CONCLUSION

In this paper, we used a realistic example to investigate the tasks required for the application of supervisory control for discrete event systems to a physical plant. In a first step, the signal-based behaviour of the physical plant is translated to an event-based behaviour that can be modeled by finite automata. Then, we apply a hierarchical supervisory control method in order to obtain supervisors on small state spaces. The lattter have to be executed in parallel in order to control the overall plant. We introduce appropriate execution semantics for the generated events and their synchronization with the signal-based physical plant. In our laboratory setup, the software environment `libFAUDES` supports these tasks, and thus helps to demonstrate the principle applicability of modern synthesis methods to real-world systems. Here, it has to be noted that profound knowledge of DES theory and the use of a suitable software tool are essential in the overall work-flow. Future work will build on this experience and investigate how the identified tasks can be integrated in an industrial application context.

## REFERENCES


Basile, F. and Chiacchio, P. (2007). On the implementation of supervised control of discrete event systems. *IEEE Transactions on Control Systems Technology*, 15, 725–739.

Brandin, B.A. and Wonham, W.M. (1994). Supervisory control of timed discrete-event systems. *IEEE Transactions on Automatic Control*, 39, 329–342.

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

Comedi (2008). Comedi: The control and measurement device interface handbook. URL `www.comedi.org`.

Fabian, M. and Hellgren, A. (1998). Plc-based implementation of supervisory control for discrete event systems. In *Proceedings of the 37th IEEE Conference on Decision and Control*, 3305–3310.

Leduc, R.J. (1996). *PLC Implementation of a DES Supervisor for a Manufacturing Testbed: An Implementation Perspective*. M.Sc. Thesis, Dept. of Elec. & Comp. Engrg., Univ. of Toronto.

libFAUDES (2006–2010). libFAUDES software library for discrete event systems. URL `www.rt.eei.uni-erlangen.de/FGdes/faudes`.

Ljungkrantz, O., Akesson, K., Richardsson, J., and Andersson, K. (2007). Implementing a control system framework for automatic generation of manufacturing cell controllers. In *IEEE International Conference on Robotics and Automation*.

Moor, T., Schmidt, K., and Perk, S. (2008). libFAUDES - an open source C++ library for discrete event systems. *9th Int. Workshop on Discrete Event Systems*, 125–130.

Schmidt, K. and Breindl, C. (2008). On maximal permissiveness of hierarchical and modular supervisory control approaches for discrete event systems. In *9th Int. Workshop on Discrete Event Systems*, 462–467.

Schmidt, K., Moor, T., and Perk, S. (2008). Nonblocking hierarchical control of decentralized discrete event systems. *Automatic Control, IEEE Transactions on*, 53(10), 2252–2265.

Vyatkin, V., Hirsch, M., and Hanisch, H.M. (2006). Systematic design and implementation of distributed controllers in industrial automation. 633 –640.