PLC Implementation of Symbolic, Modular Supervisory Controllers

Laurin Prenzel [*] **Julien Provost** [*]

[*] *Technical University of Munich, Germany (laurin.prenzel@tum.de, julien.provost@tum.de).*

Abstract: The Supervisory Control Theory was introduced in 1987 by Ramadge and Wonham (1987), and industrial applications are still scarce. This paper outlines an automatic code generation framework to construct the control logic for a programmable logic controller (PLC) from a symbolic, modular supervisor. The modeling and supervisor generation is performed with Supremica (Malik et al. (2017)), and the supervisor is implemented with the IEC 61131-3 programming language Structured Text. This framework is used to teach model-based approaches and SCT to mechanical engineering students.

Keywords: Discrete Event Systems (DES) in Manufacturing, Supervisory Control Theory (SCT), Programmable Logic Controller, Automatic Code Generation

1. INTRODUCTION

There are multiple approaches to generate PLC code from supervisors created according to the Supervisory Control Theory (SCT). Notable recent examples include Leal et al. (2012), Junior and Leal (2012) and Vieira et al. (2017). Most commonly, Ladder Diagram is used to implement monolithic or modular supervisors. Monolithic supervisors suffer from state space explosion as the models grow. Modular supervisors require on-the-fly synchronization of the plant and specification models.

This paper introduces a framework to automatically generate PLC code (Structured Text, IEC 61131-3 (IEC (2014))) from a symbolic, modular supervisor, which is synchronized during the execution.

This framework is used in teaching model-based approaches and the SCT to mechanical engineering students. The modeling and supervisor generation is performed with Supremica (Malik et al. (2017)), and the Soft-PLC executing the automatically generated code can be executed in parallel on the same machine. The Soft PLCs are connected to a didactic platform consisting of 14 modules with a total of 130 Boolean inputs and 150 Boolean outputs, which will be explained in more detail in Section 5. Supremica allows the generation of guards that restrict the plant and specification models in the same way a monolithic supervisor would without having to synchronize the models. With all tools integrated in one platform, the students can conveniently switch between modeling and testing their systems.

Common issues encountered in the implementation of supervisory controllers are examined in Section 2. The main contribution of this paper is the automatically-generated Structured Text implementation of a supervisory controller, presented in Section 3. How the previously mentioned issues are solved in this implementation is discussed in Section 4.

2. BACKGROUND

The Supervisory Control Theory (SCT) was initially published by Ramadge and Wonham (1987). It is a method to automatically synthesize a supervisor from a set of plant and specification models. The SCT recognizes two types of events: controllable and uncontrollable events (in this paper marked by `!` and `?`, respectively). In theory, both types are spontaneously generated by the plant, but the supervisor can only disable the controllable events to prevent violation of the specification.

2.1 Implementation of Supervisory Controllers

The industrial implementation of supervisory controllers is an active field of research. Zaytoon and Riera (2017) give a recent overview of the synthesis and implementation of logical controllers and common implementation issues. This section presents a subset of these problems.

Event generation Whereas SCT is solely event-based, industrial PLC systems are operating with signals. Consequently, the signals have to be converted to events. A common approach is to detect the rising and falling edges of the signal. This requires a PLC cycle fast enough to capture all edges.

Avalanche effect The event-triggered transitions are used to calculate the new active state set. Every event should only be evaluated once in a cycle and not trigger an avalanche where multiple states are skipped in the same execution (Fabian and Hellgren (1998)).

Interleave insensitivity In event-based theory, two events cannot occur at the same time and thus all events can be evaluated in the order they arrive in. In practice, because events are typically generated cyclically based on the input signals, this order cannot be reliably captured, as multiple events can be generated in between two scan cycles of the controller (Zaytoon and Riera (2017)). Fabian and Hellgren (1998) define "interleave insensitivity" as the requirement that after any interleaving of two plant-generated strings, the supervisor should generate the same event.

Choice of controllable events In theory, the plant spontaneously generates controllable events for the supervisor to enable or disable. In reality, the plant does not generate events on its own, but the controller has to decide. Although the supervisor is nonblocking by design, this property cannot necessarily be guaranteed for any controller generated from this supervisor, because the controller might only allow a (blocking) subset of the supervisor.

A minimally restrictive specification leaves the choice to the controller (Fabian and Hellgren (1998)). If the controller only implements a subset of the supervisor, the supervisor has to be designed in a way that any controller taken from this supervisor will be nonblocking (Dietrich et al. (2002)). If the controller does not limit the supervisor, it has to include methods to decide between multiple available controllable events and guarantee fairness.

Inexact synchronization The cyclical execution of the PLC results in a delay between the physical system and the controller. The inputs are scanned in the beginning of the cycle, and a change can only be detected in the next cycle. Typical cycle times range between 1 ms and 100 ms.

The supervisor should be well-posed with respect to time delay (Li and Wonham (1987)). This can be checked beforehand by introducing the concept of "delay insensitivity", which states that any delay in reading a response must be tolerated by the controllers (Balemi (1992)). This definition only assumes delays of length one.

State space explosion The problem of large state spaces in supervisory controllers is amplified in the implementation on a PLC equipped with less memory and computing power than common workstations. In addition, the execution has to be performed with strict timing constraints. Miremadi et al. (2011) propose an approach of representing the supervisor as a set of guards that can be introduced into the plant and specification models instead of synchronizing a monolithic supervisor. The models must then be synchronized during the execution.

3. ST-CODE IMPLEMENTATION

In the implementation presented in this paper, the Structured Text code according to the IEC 61131-3 (IEC (2014)) is separated into multiple files. Two main types of files have to be distinguished: Global variable lists (`GVL`) and Programming Organization Units (`POU`) (Table 1).

Table 1. List of `GVL`'s and `POU`'s

GVL	POU
input_signals	MAIN
output_signals	reset
user_var	errormode
	localcontroller
	virtualsensor
	synthcontrol

The `GVL`s hold the global variables accessible by the different `POU`s. `input_signals` and `output_signals` are linked to the physical inputs and outputs of the plant. `user_var` are user-defined global variables shared with multiple `POU`s, such as local controllers and virtual sensors. The `POU`s can be programmed with the languages according to the IEC 61131-3 (IEC (2014)). Each `POU` has its own local variable declaration.

The `MAIN POU` is executed cyclically. It schedules the execution of the other `POU`s. In this implementation, `MAIN` switches between the reset phase, when the plant is initialized, and the control phase, where the control logic from all other `POU`s is executed.

The `reset POU` is a manually implemented function to move the physical plant into an initial state. This could be implemented in the SCT models, but it unnecessarily inflates the models in the beginning of the modeling process. By using the `reset` function, the initial states for the controller can be assumed to be known. This simplifies the synchronization of the physical plant and the controller.

`localcontroller` and `virtualsensor` contain user-defined functions to implement behaviors that cannot be expressed easily with the set of controllable and uncontrollable events alone, or would require the introduction of timed models. Local controllers are triggered with a controllable event, but once it is enabled the supervisor cannot interrupt

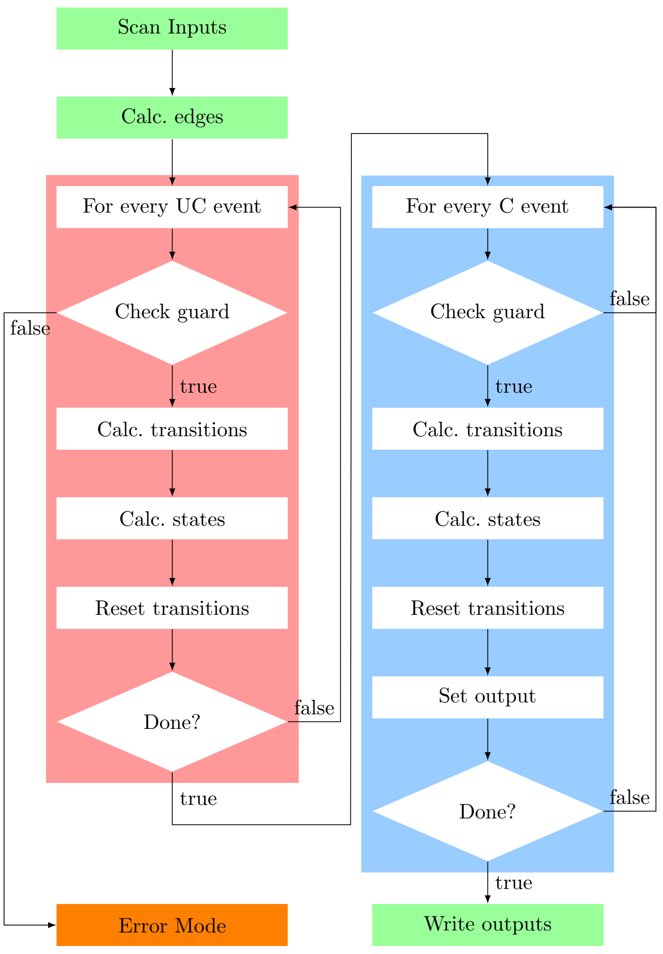


Fig. 1. Flowchart of the SCT controller (`synthcontrol`)

the actions performed by the local controller. They can administer low-level tasks, legacy code and automatic control functions. Virtual sensors are uncontrollable events triggered by manually implemented functions. Most commonly, virtual sensors signal when a timer or a local controller has finished.

`synthcontrol` (Figure 1) holds the modular control implementation and is the topic of the next sections.

3.1 Variable Declaration

There are seven types of local variables to be declared: *uncontrollable events*, *controllable events*, *sensor variables*, *virtual sensors*, *states*, *transitions*, and a set of *system variables*. The input and output signals are stored in the respective `GVL`s.

Uncontrollable and *controllable events* are declared as rising and falling edges according to the SCT model. They carry the prefix '`RE`' and '`FE`', respectively. The *sensor variables* are evaluated to detect the rising and falling edges. For every input in the `input_signals GVL`, two local variables hold the current and previous value. *Virtual sensors* follow the same principle, although only the rising edge, denoted by the prefix '`VS`', is detected.

States and *transitions* are implemented as binary variables. To unambiguously distinguish each *state*, they are declared as a concatenation of an 'X', the model name, and the state name. This is a compromise between readability during debugging and compactness. *Transitions* consist of a 'T', the model name and a counter. In contrast to the states, transitions are not as relevant for debugging, because they are only active for less than one cycle ($<$ 100 ms).

3.2 Edge Detection

The plant and specification models are event-based while the PLC is running a signal-based cyclical execution. Therefore, the events have to be extracted from the signals.

After the inputs are made available on the `input_signals GVL`, the new values are compared to the previous values to detect rising and falling edges. These events are stored in local variables and used during the execution.

3.3 Uncontrollable Transition Loop

The *Uncontrollable Transition Loop* (red block in Figure 1) evaluates one uncontrollable event after another. In the first step the synchronization of the plant models is checked. If the event is not expected in one of the individual models having it in their alphabet, the controller transitions into the error mode and will not escape until it is restarted, as the plant models are inconsistent with the behavior just observed. At this point the event could as well be ignored and discarded, but because the plant model was just proven to be faulty, the safety and synchrony of the application cannot be guaranteed.

This implementation of synchronization only checks if the next event is currently enabled, whereas a full synchronization would have to check all `N` currently fireable, uncontrollable events in all `N!` configurations. If both events

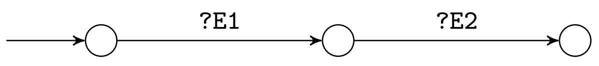


Fig. 2. Order of two uncontrollable events

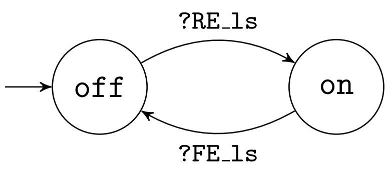


Fig. 3. Example transition and state model (`?` for uncontrollable events, `!` for controllable events)

`E1` and `E2` from Figure 2 are fireable, whether or not the error mode is triggered depends on the evaluation order.

If the error mode is not triggered, the available transitions in the active state set are calculated, the new active state set is calculated, and the transitions are reset. An example model and the resulting transition and state calculations are depicted in Figure 3 and Listing 1.

Listing 1. Transition and active state calculation for `RE_ls`

```
TRE_ls := Xoff;
Xoff   := TFE_ls OR (Xoff AND NOT TRE_ls);
Xon    := TRE_ls OR (Xon  AND NOT TFE_ls);
TRE_ls := 0;
```

The evaluation order in the uncontrollable transition loop cannot be modified by the user. Ideally, the cycle time is fast enough to capture only one uncontrollable event every cycle. Since this cannot be guaranteed for arbitrary events, the user has to ensure the order of uncontrollable events that can appear in close proximity does not invalidate the controller.

3.4 Controllable Transition Loop

After the uncontrollable events are handled, the controller can evaluate the available control actions. The *Controllable Transition Loop* (blue block in Figure 1) is similar to the *Uncontrollable Transition Loop*, but with some additions. There are two types of guards: guards to synchronize the plant and specification models, and guards generated by the symbolic supervisor. Instead of triggering an error mode, the event is simply disabled in this cycle.

The evaluation order of the controllable events cannot be influenced by the user. The implementation will choose the first fireable controllable event it encounters, and every event will only be evaluated once in the PLC-cycle. Afterwards, the fireable transitions and the new active state set are calculated. Finally, the controllable events are converted back to signals.

New uncontrollable events will not be recognized during this loop. Initially, it was considered to limit the execution to only allow a single controllable action in every cycle. As a result, the reaction time would increase substantially when multiple controllable events are available, and thus the decision was made to allow every controllable event to be taken in every cycle.

4. DISCUSSION

This section illustrates how the issues described in Section 2 are solved or at least limited, and what issues remain.

4.1 Event Generation

In this implementation, events are generated as rising and falling edges of the input signals. A sufficiently fast cycle time guarantees all edges can be detected. The optimal cycle time for the plant is between 10ms and 40ms.

If the cycle time is too long (e.g. above 60ms), a rising and falling edge may be skipped, whereas a short cycle time will cause the detection of the bouncing behavior of some sensors (e.g. light sensors). This could be solved by filtering critical sensors.

4.2 Avalanche Effect

Two types of avalanche effects can be distinguished. First, a single event should not fire consecutive transitions. This is dealt with by separating the calculation of transitions and states. For every event, the available transitions are only calculated once in a PLC cycle, whereas the state space is recalculated repeatedly.

Second, ideally, multiple controllable events should not be taken without the opportunity to receive an uncontrollable event. Consequently, the inputs would have to be scanned again after a controllable action has been taken. If many controllable events are enabled, only allowing one in every cycle means it will take many cycles until all of them are taken. Furthermore, this increases the relevance of the evaluation order.

In this implementation, every controllable event can be taken in every PLC cycle. The maximum time between the detection and reaction to an input event is one cycle. As a result, it is possible to skip an uncontrollable transition between two controllable actions. This should be accounted for in the models.

4.3 Interleave insensitivity

In this implementation the models are not tested to be interleave insensitive. The online synchronization will catch all unexpected strings and transition into the error mode if necessary. Depending on the evaluation order of the uncontrollable events, the controller can select different controllable events to be executed. The user should take the cycle-based nature of the controller into account, and built the plant and specification models accordingly.

4.4 Choice of controllable events

Whereas the supervisor is supposed to be maximally permissive, the controller has to deterministically choose between the enabled controllable events. This choice algorithm has to be implemented in the controller.

There are two ways of solving the indeterminism of the supervisor: Choosing the order of events dynamically each cycle (round-robin, random, ...), or adapting the supervisor in a way that every possible controller selected from the supervisor has the required properties. Leal et al. (2012) implement a random decision when multiple possible controllable events are encountered and thus obtain a fair, but indeterministic, controller.

In this implementation, the controller evaluates the fireable controllable events in a static order, i.e. the order is deterministic every cycle. It is thus possible to get stuck in a live-lock if this is part of the supervisor. On the other hand, this ensures that the controller will always react the same given a set of inputs. The synthesized supervisor should not contain infinite paths that do not lead to an accepted state.

4.5 Inexact synchronization

The issue of inexact synchronization is inherent to all control applications. It is physically impossible to prevent the arrival of new uncontrollable events during the execution cycle of the PLC. Balemi (1992) presents a definition to check for "delay insensitivity" when only a single controllable event is taken every cycle. When multiple events are enabled every cycle, the solution is more complicated, as more paths have to be considered.

Delay insensitivity is a property of the supervisor. The controller in this implementation can take multiple controllable events in one cycle and thus poses stricter requirements on the supervisor. Without a framework to test delay insensitivity of the supervisor for multiple events, it is a good rule of thumb to not pose unrealistic expectations on the specification models, i.e. the order of events that can happen in close proximity should not invalidate the control action taken.

4.6 State space explosion

In the previous solution (Jordan et al. (2017)), where a monolithic supervisory controller was generated using an existing Matlab script, the computational limit was quickly reached because of the synthesis algorithm.

After the switch to Supremica and a Structured Text implementation the synthesis was no longer the bottleneck, and large monolithic supervisors could be synthesized quickly. The ST implementation of the monolithic supervisor constituted the new limit, as the number of lines of code grew linearly with the number of transitions. The supervisor was thus limited to a couple thousand states, before the large files crashed the Soft PLC environment.

Switching to a symbolic modular synthesis algorithm allows the implementation of large supervisors. A medium-sized module with 10 inputs and 20 outputs can be implemented with 20 plant and 20 specification models, resulting in a modular supervisor with 10^{12} states. Since this is implemented modularly, the ST code has only 2000 lines, and the typical execution time was estimated experimentally to be below 500 μs. This execution time can vary with the number of available controllable and uncontrollable transitions.

4.7 Model Synchronization

Because of the modular implementation, the synchronization has to be performed on the fly by checking if the event is disabled in the active state set, i.e. evaluating a set of Boolean guards generated from the states restricting this event. For the controllable events, the synchronization determines whether this event will be taken or not. If an uncontrollable event is detected in a state where the synchronization disables it, the controller exits to the error mode where it will stay until the plant is restarted.

5. CASE STUDY

The framework presented in this paper is used to teach SCT to student groups. A didactic platform is separated into subsystems and each group is in charge of implementing the controller for their subsystem. This course is based on a previously held course described by Jordan et al. (2017). Since then the physical plant remained unchanged, but the control architecture has been renewed.

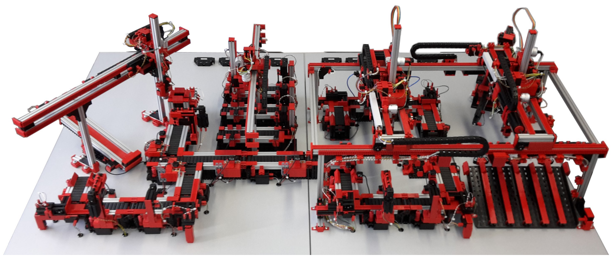


Fig. 4. Didactic Platform

5.1 Didactic Platform

The didactic platform (Figure 4) consists of two separable parts. The first part is a combination of assembly lines serviced by two 2-axis portal cranes. The portal cranes transport workpieces between different stations, while the assembly lines move the workpieces with conveyor belts and use a variety of tools on them.

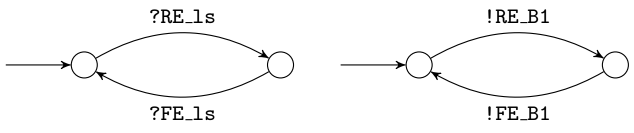
The second part revolves around two 3-axis portal cranes with overlapping domains, transporting workpieces between two assembly lines, a welding robot, and a storage.

Modularization of the platform The two parts are further divided into subsystems to simplify the modeling and to allow students to work independently. Each subsystem with 10 to 40 Boolean inputs and outputs is controlled by a remote input/output module, connected via EtherCAT to a Laptop running a Beckhoff TwinCAT Soft PLC.

Subsystem interaction The subsystems have interfaces where workpieces are transported from one subsystem to another. These interfaces require the cooperation of multiple student groups and thus cause the most difficulties. This high-level control is implemented by two global controllers, interacting with the subsystems through electrically connected inputs and outputs. The communication protocols usually require a subsystem to request clearance from the global controller.

5.2 Modeling Procedure

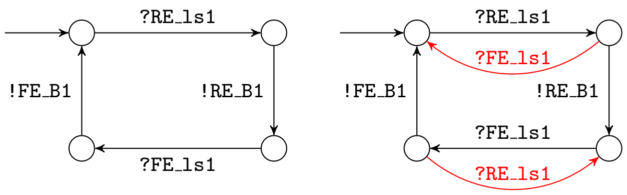
Each controller is synthesized based on a set of plant and specification models. They are implemented in Supremica (Malik et al. (2017)) and the symbolic, modular supervisor is generated with the built-in functions.



(a) Lightsensor (b) Conveyor Belt

Fig. 5. Example plant models

Plant models Plant models describe the behavior of the physical plant by restricting the language to what is physically possible. These models can be of varying detail. A common restriction for sensors and actuators is, for example, that rising and falling edges must occur alternatingly. Figure 5 displays a set of plant models for a light sensor (`ls`) and a conveyor belt (`B1`).



(a) (b)

Fig. 6. Possible conveyor belt specifications

Specification models The specification models restrict the capabilities of the plant to fit the requirements of the user. For example, a requirement might be that after an uncontrollable event occurs, an action is performed. Figure 6 displays two different specifications for a common cyclical execution: After a rising edge is detected, a belt is started. When the falling edge occurs, the belt is stopped. The plant models in Figure 5 can be used for this example.

When the students are modeling their first specifications, they usually model them similarly to the specification in Figure 6a and thus run into controllability issues because of disabled uncontrollable events. The specification in Figure 6b avoids these issues and recovers controllability. On the other hand, it might not be desirable to allow the removal of a piece before the belt is started. In this case, the plant model has to be modified to further restrict the occurrence of `RE_ls1` and `FE_ls1`.

Correctly modeling the plant behavior on the first try is hard. There has to be a balance between permissiveness, detail and solution independence. A very unrestrictive and solution-independent plant model will allow unrealistic event combinations and impede specification design. Plant models with too much detail will favor a specific solution and may conflict with the real behavior.

Local controllers and virtual sensors Local controllers are used to control outputs that should not be interrupted by the supervisor. When a combination of controllable actions should be executed in a predefined order and with strict timing constraints, the synthesized controller can not be trusted to take the right decisions. In this case, local controllers allow the implementation of arbitrary functions that are enabled as a controllable event. In addition, *local controllers* could also be used to call legacy code or automatic control functions.

Listing 2. Local controller implementation

```
IF LC_M1 THEN
        ton_M1 (IN:=TRUE, PT := T#1000MS);
        output.M1 := TRUE;
END_IF
ton_M1(IN:=timer_ls23.IN);
IF ton_M1.Q = TRUE THEN
        ton_M1(IN := FALSE);
        output.M1 := FALSE;
        VS_M1done := TRUE;
END_IF
```

Listing 2 depicts an example where the output `M1` should be active for one second. After this time, an uncontrollable virtual sensor event is emitted to notify the supervisor about the finished task.

Listing 3. Virtual sensor implementation

```
IF FE_ls THEN
        ton_vs(IN:=TRUE, PT := T#1000MS);
END_IF
ton_vs(IN:=ton_vs.IN);
IF ton_vs.Q = TRUE THEN
        ton_vs(IN := FALSE);
        virtualsensor.VS_ls := TRUE;
END_IF
```

Virtual sensors allow the user to emit uncontrollable events at predefined conditions. The conditions can be arbitrarily defined, a common implementation is depicted in Listing 3. In this case, the virtual sensor is emitted one second after the occurrence of `FE_ls`.

6. CONCLUSION

This paper has outlined a framework to control a physical plant using PLC code automatically generated from a symbolic, modular supervisor. It is used, coupled with a didactic plant, as a teaching platform to introduce students to model-based approaches and SCT. During this course the implementation proved to be reliable, and the integration of modeling and execution on the same platform enables fast and frequent modifications. In contrast to the previously-used implementation, larger supervisors and more complex systems can be considered.

There are still two major issues for the implementation of supervisory controllers on PLC:

- PLCs are signal-based, whereas the SCT is event-based. Events must be generated from signals and afterwards converted back to signals. This introduces a delay between plant and controller.
- The SCT assumes controllable events to be spontaneously generated. In practice, the controller has to implement a choice algorithm and guarantee fairness and/or determinism.

In the future, this framework will be extended and potentially adapted for an industrial-sized problem. An interesting extension would be to adopt a signal-interpreted approach, as introduced by Fouquet and Provost (2017).

The Python executable for the automatic code generation can be found on the chair website: `www.ses.mw.tum.de` .

REFERENCES

Balemi, S. (1992). *Control of discrete event systems: theory and application*. Ph.D. thesis, Automatic Control Laboratory, Swiss Federal Insitute of Technology.

Dietrich, P., Malik, R., Wonham, W.M., and Brandin, B.A. (2002). Implementation considerations in supervisory control. In *Synthesis and Control of Discrete Event Systems*, 185–201. Springer, Boston, MA.

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*, volume 3, 3305–3310 vol.3.

Fouquet, K. and Provost, J. (2017). A Signal-Interpreted approach to the supervisory control theory problem. *IFAC-PapersOnLine*, 50(1), 12351–12358.

IEC (2014). Programmable controllers – part 3: Programming languages. Technical Report IEC 61131-3:2013.

Jordan, C., Ma, C., and Provost, J. (2017). An educational toolbox on supervisory control theory using MATLAB simulink stateflow: From theory to practice in one week. In *2017 IEEE Global Engineering Education Conference (EDUCON)*, 632–639.

Junior, M. and Leal, A.B. (2012). Modelling and implementation of supervisory control systems using state machines with outputs. *Manufacturing System*.

Leal, A.B., da Cruz, D., and Hounsell, M.S. (2012). PLC-based implementation of local modular supervisory control for manufacturing systems. *Manufacturing System*.

Li, Y. and Wonham, W.M. (1987). On supervisory control of Real-Time Discrete-Event systems. In *1987 American Control Conference*, 1715–1720. ieeexplore.ieee.org.

Malik, R., Åkesson, K., Flordal, H., and Fabian, M. (2017). Supremica–An efficient tool for Large-Scale discrete event systems. In *The 20th World Congress of the International Federation of Automatic Control, Toulouse, France, 9-14 July 2017*.

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

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

Vieira, A.D., Santos, E.A.P., de Queiroz, M.H., Leal, A.B., de Paula Neto, A.D., and Cury, J.E.R. (2017). A method for PLC implementation of supervisory control of discrete event systems. *IEEE Transactions on Control Systems Technology*, 25(1), 175–191.

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