Preprints of the 19th World Congress
The International Federation of Automatic Control
Cape Town, South Africa. August 24-29, 2014

Robust Manual Control of a Manufacturing System using Supervisory Control Theory

Brian Bonafilia, Pontus Carlsson, Sebastian Nilsson, Martin Fabian

Chalmers University of Technology, Göteborg, Sweden
{bbrian, cpontus, sebnil}@student.chalmers.se, fabian@chalmers.se

Abstract: There are many situations where manual intervention in automated systems, such as flexible manufacturing systems, is inevitable. Commonly the control of a manufacturing system is implemented in terms of *operations*, and sequences thereof, and these operations can typically be run manually. However, running the operations in arbitrary order may lead to situations such as blocking or collisions, from which it its hard or even impossible to recover or to resume automatic execution. This paper describes an implementation of an operator interface for robust manual control of a manufacturing system, where the operator is aided not to manually drive the system into a state that breaches the system requirements. Hence, blocking and collisions are avoided, and automatic mode can always be resumed. From a model of the manufacturing system based on *self-contained operations*, each of which is identified in terms of events with preconditions, a *supervisor* is calculated by use of the Supervisory Control Theory framework. From this supervisor additional preconditions are extracted for each operation. The operations with the extended preconditions are then ported to an operator interface which allows manual control of the production cell by dynamically guiding the operator to only those operations for which the extended preconditions are satisfied.

Keywords: supervisory control theory, operator interface, manual control, manufacturing systems, extended finite state machines, human-machine interface

1. INTRODUCTION

Even in the most highly automated industrial system, manual intervention is inevitable. Fault recovery is an obvious case, but many other situations, such as training new operators, may arise where an operator drives the system manually. Typically the control system is implemented in terms of *operations* and sequences thereof, which lends itself to straightforward manual execution, especially if the operations are *self-contained* (Bengtsson (2012)), meaning that they only concern local execution restrictions. However, running the operations in arbitrary order may lead to situations such as blocking or collisions, from which it is hard or even impossible to recover or to resume automatic execution. A means to guide the operator to avoid manually driving the system into such situations would be beneficial.

SCT (Supervisory Control Theory, Ramadge and Wonham (1987)) is a model-based approach to automatically generate a *supervisor* for a discrete event *plant* so that the closed-loop system fulfills a given required behavior, the *specification*. The supervisor is a control function that dynamically disables as few controllable plant events as possible so that the closed-loop system remains safe, never breaching the specification, and non-blocking, always able to perform some desired task. Thus, if the system was manually operated under such supervision, the operator would be aided in avoiding bad situations while given the maximum amount of freedom.

Miremadi et al. (2011) present a method of generating a decentralized system of controllers by extracting from a synthesized supervisor a set of preconditions, known as *guards*, for each event. By inserting these preconditions into the local execution restrictions of the operations, this method allows a simple controller scheme to be implemented that fulfills the given specifications with reasonable memory and computation requirements.

This paper describes an implemented HMI (Human Machine Interface), which allows for robust manual control of a manufacturing system. By imposing the guards extracted from the synthesized supervisor on each operation, the HMI guarantees that an operator cannot manually violate the system specification, such as drive the system into a blocking state. This implementation also eliminates the necessity of hard-coding the guards directly into the controller that manages the system, replacing a time-consuming manual process with an automated one.

Section 2 presents the background of the SCT and the self-contained operations. Section 3 covers the process of modeling and guard extraction. Section 4 discusses the implementation of the HMI and automatic generation of the control code from the supervisor. Results from implementation on a physical robot cell are given in Section 5. Conclusions and future work are discussed in Section 6

2. PRELIMINARIES

The SCT is typically presented in a FA (Finite Automata, Ramadge and Wonham (1987)) setting. However, EFA (Extended Finite Automata, Chen and Lin (2000)) is a more compact modeling formalism, though with equivalent expressive power. As shown by Miremadi et al. (2011),

EFA can be transformed into equivalent FA, on which supervisor synthesis can be performed and from which guards can be extracted. These guards can then be added to the original EFA to implement the supervisor. The self-contained operations have straightforward interpretations as EFA.

2.1 Supervisor Synthesis

An FA is a 5-tuple $\langle Q, \Sigma, \delta, q_i, Q_m \rangle$, where Q is the finite set of states, Σ the finite set of events, $\delta : Q \times \Sigma \to Q$ a partial function describing the state transitions, Q_m is the set of marked states, and $q_i \in Q$ the initial state. The marked states represent states desirable to reach, and the supervisor must be such that at every state in the closed-loop system some marked state is always reachable. The supervisor does not have full influence over the events generated by the plant, though; some events are regarded as *uncontrollable*. Thus, while controlling the plant, the supervisor must be such that it never tries to disable an uncontrollable event.

Interaction between automata, as for instance the closed-loop system of plant and supervisor, is modeled by *synchronous composition*, where common events are enabled by both systems or not at all. Formally we have that for FA A and B, their synchronous composition is defined as $A \parallel B = \langle Q, \Sigma^A \cup \Sigma^B, \delta, \langle q_i^A, q_i^B \rangle, Q_m \rangle$, with $Q \subseteq Q^A \times Q^B$, $Q_m \subseteq Q_m^A \times Q_m^B$, and

$$\delta(\langle q^A, q^B \rangle, \sigma) = \begin{cases} \delta^A(q^A, \sigma) \times \delta^B(q^B, \sigma) & \sigma \in \Sigma^A \cap \Sigma^B \\ \delta^A(q^A, \sigma) \times \{q^B\} & \sigma \in \Sigma^A \setminus \Sigma^B \\ \{q^A\} \times \delta^B(q^B, \sigma) & \sigma \in \Sigma^B \setminus \Sigma^A \end{cases} \tag{1}$$

Synchronous composition is trivially extended to simultaneous composition of any finite number of FA.

The (monolithic) supervisor synthesis process is a formal method by which the plants and specifications are first synchronized into a single automaton. From this single automaton is then iteratively removed *bad states*, which are states from which no marked state can be reached or from which uncontrollable events lead to bad states. It can be shown (Ramadge and Wonham (1987)) that this iterative removal of bad states will terminate at a fix-point, the *minimally restrictive, controllable and non-blocking supervisor*, which when controlling the plant will allow the plant the greatest possible freedom within the specification while avoiding bad states and guaranteeing that some marked state may always be reached.

2.2 Extended Finite Automata

EFA are automata extended with bounded discrete variables, and guard and action functions over these variables.

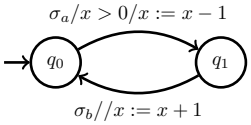

Fig. 1. Sample EFA with events, guards, and actions.

In Fig. 1, the event σ_a is disabled unless the guard $x > 0$ is true. When the transition is executed, the action $x := x - 1$ is performed. The event σ_b is always enabled due to lack of a guard, and the transition performs the action $x := x + 1$. An EFA is described as a 6-tuple: $\langle Q \times V, \Sigma, G, A, \to, (q_0, v_0) \rangle$, where the Q is the set of locations with q_0 the initial location, V is the set of variables with initial values v_0, Σ is the set of events, G and A are the sets of guards and actions, respectively, and $\to$ is the transition relation. Note that the state of an EFA, such as the initial state (q_0, v_0) above, is a tuple of a location and the current values of the variables, Note also that synchronous composition can be defined for EFA similarly to FA (Miremadi et al. (2011)).

2.3 Self-Contained Operations

Bengtsson (2012) shows how to model a manufacturing system as a set of self-contained operations. These operations store only locally relevant data, such as conditions for initiation and current state of operation. No information relating to any particular sequence of operations is stored within or assumed by the operation itself; any sequencing is managed by a controller that initiates the operations in the correct order.

A self-contained operation can be represented as an EFA, as operation O_k in Fig. 2. When the pre-condition represented by $C_k^{\uparrow}$ is satisfied the start event $O_k^{\uparrow}$ is enabled from the initial location O_k^i and the operation can start. When started the operation enters the execution location O_k^e, from where the finishing event $O_k^{\downarrow}$ is enabled when the post-condition represented by $C_k^{\downarrow}$ is fulfilled. Note that pre- and post-conditions can include guards as well as actions.

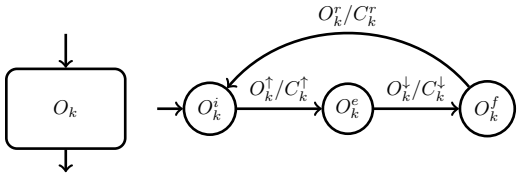

Fig. 2. Self-contained operation block and corresponding EFA

This representation also allows for an optional reset event O_k^r which returns the operation back to its initial state O_k^i. Many resource operations can be reset, for example, a robot may move back and forth as many times as necessary, but most product operations can not.

As mentioned, each operation stores only locally relevant data. The conditions for starting are found in the precondition $C_k^{\uparrow}$ and the outcome of the operation are in the post-condition $C_k^{\downarrow}$. No global sequence is enforced by the operation. However, in order to make sure that no arbitrary sequences of events are executed, a control scheme is generated based on synthesis between the operations (as plants) and procedural specifications. For example, it may be necessary to place one workpiece in a fixture before another. Such a specification is introduced in order to guarantee that the workpieces are placed in the correct order.

SCT is used to synthesize a controllable, non-blocking supervisor for the system, such that all specifications for the manufacturing process are satisfied.

3. PROCESS MODELING AND GUARD EXTRACTION

A laboratory robot cell was used as an application for implementation of the robust manual control. The cell consists of two industrial robots that can take pieces from a storage and place them in a fixture in order to assemble a model car.

3.1 Operations Modeling

The cell has almost 100 implemented operations, thus, an FA representing the monolithic system has tens of billions of discrete states. Though the synthesis approach of Miremadi et al. (2011) can handle huge state-spaces, it is sensitive to the number of variables in the model. Locations are handled as binary variables, so replacing locations with a variable of larger domain is beneficial.
To do this, abstraction techniques of Mohajerani et al. (2011) can be applied. With careful modeling $O_k^{\downarrow}$ can be made *local*, meaning that it is only used by operation O_k. Furthermore, $O_k^{\downarrow}$ is naturally regarded as uncontrollable. From Mohajerani et al. (2011) we then know that the O_k^e and O_k^f states can be merged so that $O_k^{\downarrow}$ can be abstracted away. Even more, careful modeling can also make the completion of the reset event O_k^r be implied by the post-condition $C_k^{\downarrow}$. Then the reset event transition can also be abstracted away and the event $O_k^{\uparrow}$ can be viewed as a self-loop from the initial state, as shown in Fig. 3. Though very simple, this model is useful in synthesizing a supervisor for the example system.

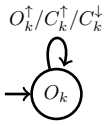

Fig. 3. Self-contained operation in abstracted form

Note that the resulting $C_k^{\uparrow}$ and $C_k^{\downarrow}$ now include variables with conditions to model location changes of the EFA. This does not affect the synthesized supervisor, though. A variable x_i representing a location can be modeled as automata sharing the event $O_k^{\uparrow}$ with the self-contained operations which include a change in x_i as part of the post-condition action $C_k^{\downarrow}$. Using these models synthesizes the same supervisor as would using the original models.

3.2 Guard Extraction

Using the operation models of Fig. 3 as plants P together with specifications for sequences and safety, K, the algorithms of Miremadi et al. (2011) calculates a supervisor S from $P||K$. This supervisor will have the state-space $Q_S \subseteq Q_{P||K}$. From S, for each controllable event $\sigma \in \Sigma_c$ guards are then extracted.
Let Q^{σ} be the set of states in $P||K$ that enable σ,

$$Q^{\sigma} = \{q \in Q_{P||K} | \exists q' \in Q_{P||K}, \sigma \in \Sigma_c, \delta(q,\sigma) = q'\}. \quad (2)$$

Similarly, let Q_S^{σ} be set of states where S enables σ,

$$Q_S^{\sigma} = \{q \in Q_S | \exists q' \in Q_S, \sigma \in \Sigma_c, \delta(q,\sigma) = q'\}. \quad (3)$$

Let $G^{\sigma} : Q^{\sigma} \to \{true, false\}$ be a guard such that

$$G^{\sigma}(q) = \begin{cases} true & q \in Q_S^{\sigma} \\ false & q \in Q^{\sigma} \setminus Q_S^{\sigma} \end{cases} \quad (4)$$

On each transition labeled by σ, G^{σ} is conjuncted with the existing guard, to form the *extended guard* $\tilde{C}_k^{\uparrow}$ that defines when σ is enabled under control. Note that the states outside of Q^{σ} can be used to reduce G^{σ}, details are given by Miremadi et al. (2011). Important here is that each operation can now be considered individually, looking only at the extended guard to determine if the initiating event $O_k^{\uparrow}$ (denoted as σ above), and hence the operation, is enabled.

4. IMPLEMENTATION

The HMI for robust manual control based on an operations model of the plant with guards extracted from a supervisor was implemented on a physical robot cell in the Production Systems Laboratory at Chalmers University of Technology. Here, only a brief description will be given, for details see Bengtsson (2012).

4.1 Previous Work

The cell consists of two robots, a fixture, a "wagon" acting as workpiece magazine, and a PLC (Programmable Logic Controller) that checks for safety conditions and sends commands to the robots. The PLC is set to receive inputs from and send signals to an OPC server (Leitner and Mahnke (2006)). The robots, the OPC server, and the PLC were programmed by undergraduate students (Börjesson et al. (2013)) and implement all operations in the cell. The PLC restricts operations based only on safety (e.g. avoids collision) and physical limitations (e.g. parts cannot be picked if the wrong tool is equipped); it does not guarantee absence of blocking.

4.2 Interface Structure

The structure of the interface is shown in Fig. 4. The work behind this paper focused on the generation of a supervisor for the system and the programming of the interface client and server. The web-server is programmed using the programming language Scala, and the interface uses HTML5 and AngularJS (JavaScript) and is shown in Fig. 5. The interface generates a list of possible operations from the information loaded in with the supervisor data, consisting of the list of operations and associated guard conditions. The interface contains a local OPC client which continuously fetches the current state from the PLC via the OPC server. The state is defined by the variables in the system.
The PLC works with two types of variables. Internal variables are set and stored in the PLC while external variables are sensor inputs read from the production cell. Examples of external variables are locking conditions of the fixtures and wagon, while internal variables include part positions that do not have associated sensors, such as the count of pieces available on the wagon or the position of the robots. Both types are sent to the OPC server from the PLC, so the interface treats different types of system variables the same.
The extended guards associated with each event are passed as an XML file to the HMI. Standard tools are used to parse the XML file to generate the pre-conditions as logical statements comparing the variable set from the OPC server with the states where each operation is enabled. The HMI client then filters the available operations based on whether their pre-conditions are fulfilled. In normal (non-debug) mode, the operator is presented with and can

choose from only the currently enabled operations. The operator chooses to execute an operation by clicking the "Go!" button, see Fig. 5.

Information about which operation was manually chosen to be executed is sent to the server, which then sends a command to the PLC via the OPC server. Finally the PLC activates the operation at the hardware level. When the operation is complete, the set of sensor data coming from the cell will have changed. Thus, the PLC will send values of the variables via the OPC server to the interface. The system variables, the list of operations, and their guards are transmitted to the client where the operator is then presented with a new set of enabled operations based on the current state.

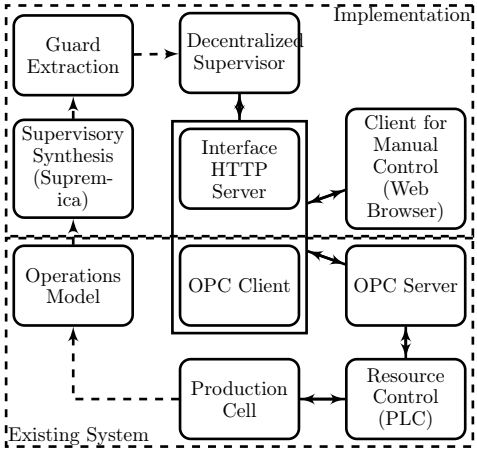

Fig. 4. Implementation and information chain

4.3 Supervisor implementation

The resource operations and physical limitations are extracted and programmed into the software package *Supremica* (Supremica (2013)) in their abstracted form, similar to Fig. 3. Supremica allows for conditions and actions to be set using variable expressions. From the variable expressions, corresponding automata are generated (see Fig. 6) and SCT is used on the entire system based upon the general algorithms and definitions of Section 2.

Supremica then generates extended guards for the preconditions for each operation as discussed in Section 3. Thus, there is no need to express the supervisor as a single monolithic automaton.

4.4 Performance of the HMI

The implemented HMI fetches all variables to evaluate the guards once per second. Thus, the requests from the client to the web-server including any calculations on the client needs to be done in less than one second. Different factors affect the calculation time such as client hardware, number of variables and operations, and size of guard expression. For an ordinary laptop, evaluation of 3675 if-statements corresponding to guard expressions of up to 250 terms required roughly 35 ms, scaling linearly with the number of expressions and operations.

The PLC implements only hardware safety functions and does not guarantee non-blocking. Interacting directly with the PLC, the operator can thus manually run the system into a blocking state. Interacting through the HMI, though, the system is non-blocking; the extended guards prevent the operator from choosing operations in an order that leads to blocking, or otherwise forbidden states. At the same time it is guaranteed that it is always possible to properly assemble a car. Also, the system is minimally restrictive, so the operator is never unnecessarily hindered from running a specific operation.

As an example, in addition to preventing operation sequences that would attempt the robots to build the car in the incorrect order, the guards prevent sequences of operations where the robots would pick workpieces in the incorrect order, as there are no operations programmed into the robots to put parts back once they have been picked.

5. APPLICATION

As mentioned above, the application for this implementation is a model car factory where two robots, R1 and R2, assemble a car from a number of different parts. The interface was programmed with a complete set of guards for assembling the whole body of the model car. For illustration though, the implementation of extended guards for a subset of the complete system will be focused on here, Table 1. These operations place the floor on the fixture ($fix_4.Ri$, with $i = 1, 2$), locks the fixture, ($lock$), and then mounts the roof ($pos_{14}.Ri$).

The operations to place the floor on the fixture, and to mount the roof, are made up of sub-operations, as illustrated by Fig. 7. First, either robot has to pick up ($pickUp_4.Ri$) and place ($place_4.Ri$) the floor-part on the fixtur. Then, after the fixture has been locked, either robot must pick up ($pickUp_1.Ri$) and place ($place_1.Ri$) the roof-part on the fixated floor.

Note that some of those sub-operations, marked in the "Pre" column in Table 1, are *pre-operations* that can occur before the condition of their respective operation is satisfied. For example, it is acceptable for R2 to pick up the roof before R1 has laid the floor. However, as one can easily imagine, this can lead to a deadlock if R1 has already picked up a roof in anticipation that R2 was going to lay the floor. In the original operations model in Bengtsson (2012), which presupposed an automated sequence that determined ahead of time which robot would do which tasks, there is no guard against this condition.

Table 1. List of Operations

Operation		Pre
$fix_4.R1$	Place the floor with R1	
$fix_4.R2$	Place the floor with R2	
$lock$	Lock the fixture	
$pos_{14}.R1$	Mount the roof with R1	
$pos_{14}.R2$	Mount the roof with R2	
$pickUp_4.R1$	Pickup the floor from the wagon	x
$place_4.R1$	Place the floor on the fixture	
$pickUp_4.R2$	Pickup the floor from the wagon	x
$place_4.R2$	Place the floor on the fixture	
$pickUp_1.R1$	Pickup the roof from the wagon	x
$place_1.R1$	Place the roof on the floor	
$pickUp_1.R2$	Pickup the roof from the wagon	x
$place_1.R2$	Place the roof on the floor	

When the $place_1.Ri$ operations are completed, a variable $Fixture.HAS\_FLOOR$ is updated to indicate that the fixture has had a floor added.

HMI for PSL by CC Cowboys (index.html)

Polling timer: 90102

Execution time [ms]: 0 , 28 , #Guards: 3675

Operations

Search: PreTrue: Don't care State: Executable: Don't care

Sort by: Name asc

Name	Pretrue	State	Executable	Guards	JS Guards	Eval	Execute	Reset
Fixture_close	false	0	true	Fixture.close == 0	$scope.plcVariables['Fixture.close'] == 0	true	Go!	Reset!
Fixture_open	false	0	false	Fixture.open == 0	$scope.plcVariables['Fixture.open'] == 0	false		Reset!
Operator_mount_hybrid	false	0	false					Reset!
R1_equip_gripper	false	0	false					Reset!
R1_equip_sucker	false	0	false					Reset!
R1_fixture_to_home	false	0	false	R1.POS_FIXTURE == 1	$scope.plcVariables['R1.POS_FIXTURE'] == 1	false		Reset!
R1_fixture_to_tool	false	0	false					Reset!
R1_fixture_to_wagon	false	0	false	R1.POS_FIXTURE == 1	$scope.plcVariables['R1.POS_FIXTURE'] == 1	false		Reset!
R1_home_to_fixture	false	0	false	R1.POS_HOME == 1	$scope.plcVariables['R1.POS_HOME'] == 1	false		Reset!
R1_home_to_get_floor_to_home	false	0	false					Reset!
R1_home_to_leave_floor_to_home	false	0	false					Reset!
R1_home_to_tool	false	0	false					Reset!
R1_home_to_wagon	false	0	false	R1.POS_HOME == 1	$scope.plcVariables['R1.POS_HOME'] == 1	false		Reset!

Fig. 5. The operator interface shown in debug mode. The extended guards are shown in the "Guards" column, while the "JS Guards" column shows their coding as the executed JavaScript.

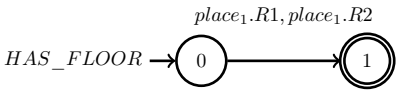

Fig. 6. Automata representing variable change through shared events.

The variable $Fixture.HAS\_FLOOR$ can be represented as an automaton which shares events with the $place_1.Ri$ operations, as in Fig. 6. By marking the value of '1' for this and all similar $Fixture.HAS...$ variables, a marked state representing the assembled car appears in the final supervisor. This allows the rejection of any states that cannot reach the final marked state of a completed car.
The guard conditions on the operation $pickUp_1.R1$ before the guard extraction are given by:

$$C^{\uparrow}_{pickUp_1.R1} = (R1.POS\_WAGON = 1 \wedge R1.HAS\_FLOOR = 0 \wedge R1.HAS\_ROOF = 0 \wedge Wagon\_HAS\_ROOF > 0)$$

$$C^{\downarrow}_{pickUp_1.R1} = (Wagon\_HAS\_ROOF -= 1; \; R1.HAS\_ROOF := 1)$$

Initially the conditions of the operations are basic physical requirements, the robot is at the wagon to do the picking, the robot has an empty tool, and the part is present on the wagon.

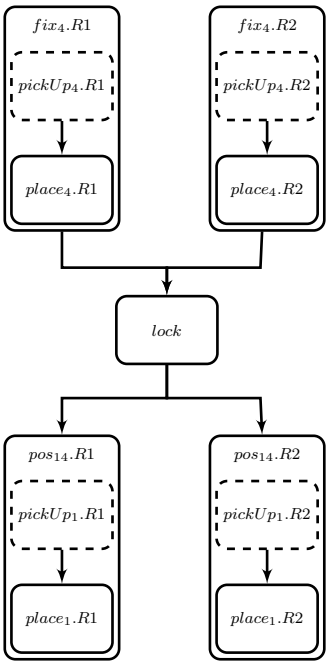

Fig. 7. Sequence of operations to assemble floor and roof. Hatched lined operations are pre-operations.

When $R1$ has picked the roof, the variables are updated by $C^{\downarrow}_{pickUp_1.R1}$, the wagon has one less roof part, which is now held by $R1$.

After guard extraction a more complicated condition emerges which dictates that the robot may not pick up the part if doing so will lead to a blocking state that will prevent the robots from reaching the marked state and thus completing the car. In this case the extended guard becomes:

$\tilde{C}^{\uparrow}_{pickUp_1.R1} =$
$(R1.POS\_WAGON = 1 \wedge R1.HAS\_FLOOR = 0 \wedge$
$R1.HAS\_ROOF = 0 \wedge Wagon\_HAS\_ROOF > 0 \wedge$
$(R2.HAS\_ROOF = 0 \vee Fixture.HAS\_FLOOR = 1))$

In addition to the previous physical requirements, the first robot may only pick up the roof if the second robot has not already done so ($R2.HAS\_ROOF = 0$) *or* the floor is already in place ($Fixture.HAS\_FLOOR = 1$). Of course, a similar extended guard will be generated for robot $R2$.

The 208 blocking states avoided make a rather trivial example that it is imaginable that an engineer could have foreseen when programming the PLC. But with larger, more complex systems, it is impossible to foresee all potential blocking situations or the conditions which may invariably lead to blocking. For the complete cell, the number of states reaches into the tens of billions with multiple unforeseeable blocking situations. Furthermore, the complexity of the extended guards that must be implemented to be both non-blocking and maximally permissive range into the hundreds of terms in each guard expression. Implementing this manually would be very time consuming, difficult to verify, and prone to errors.

6. DISCUSSION AND FUTURE WORK

This paper describes an application of the supervisory control theory where the supervisor is used to guide an operator that manually controls a manufacturing system, so that blocking and other breaches of the system requirements are avoided. The supervisor works as a safety device that observes the system and disables actions that would invariably lead to bad situations. In this way, this is a canonical use of a supervisor as originally described by Ramadge and Wonham (1987).

The implementation was successful for the completion of the body of the model car. As the complexity of the extended guards increases, their processing and conversion times also increases. For the entire production cell, including tool switching, internal part placement, and the available manual assembly options, the extended guard expressions become extremely large. While evaluation time is currently well within the one second window, for other systems of industrial scale the processing time may be problematic. Other methods of distributing the workload, such as several clients each devoted to one area of the production system, may need to be explored.

The PLC software was pre-programmed with the set of operations available to the cell. The structure of the operations in the PLC program was similar to the form presented in Bengtsson (2012), with state outputs, pre- and post-conditions, and start and reset events. The model in Supremica was forced to follow the same naming conventions and structure as in the PLC program. This presents two problems: (a) there is a potential for mismatch and (b) implementation of the code at the PLC level is not automatic. Addressing both of these issues may be worthwhile.

Furthermore, the filtration of enabled operations was done at the client level in this implementation. For systems in which multiple clients communicate with the same server it may be more desirable to have the operation set filtered at the server level and then only enabled operations sent to the client, in order to prevent overlap of any sort. Also, multiple operator interfaces of different priorities could be imagined, where higher priority operators have a larger, or different, set of operations available for manual control. This would probably require adjusting the existing SCT approaches.

Supervisory control of this nature operates on the assumption that the supervisor and the physical system function synchronously. The delay between operation initiation and retrieval of a new state set is less than a second with the current implementation. Potential problems are avoided by reducing the set of enabled operations to an empty set immediately after an operation has been initiated and then waiting until a new set of variables has been received. This keeps the list of enabled operations "current". However, this does not address the possibility that the PLC internal variables become out of sync with the physical system. For truly robust manual control, the interface would need to include additional guarantees that the production cell is in the state expected by the supervisor. Research is ongoing in this direction. However, this is outside the scope of this implementation.

REFERENCES

Bengtsson, K. (2012). *Flexible design of operation behavior using modeling and visualization.* Ph.D. thesis, Chalmers University of Technology.

Börjesson, T., Kjerstadius, S., Eliasson Lilja, C., Larsson, A., Grönbäck, M., and Noresson, O. (2013). Design och styrning av tidseffektiv tillverkningscell. Chalmers Tekniska Högskola.

Chen, Y.L. and Lin, F. (2000). Modeling of discrete event systems using finite state machines with parameters. In *Control Applications, 2000. Proceedings of the 2000 IEEE International Conference on*, 941–946.

Leitner, S.H. and Mahnke, W. (2006). OPC UA–service-oriented architecture for industrial applications. *Softwaretechnik-Trends*, 26(4). URL `http://pi.informatik.uni-siegen.de/stt/26_4`.

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

Mohajerani, S., Malik, R., Ware, S., and Fabian, M. (2011). Compositional synthesis of discrete event systems using synthesis abstraction. In *Chinese Control and Decision Conference CCDC*, 1549–1554. IEEE.

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

Supremica (2013). A tool for verification and synthesis of discrete event supervisors. URL `www.supremica.org`.