Proceedings of the 37th IEEE
Conference on Decision & Control
Tampa, Florida USA • December 1998

PLC-based Implementation of Supervisory Control for Discrete Event Systems

M. Fabian and **A. Hellgren**

Control Engineering, Department of Signals and Systems
Chalmers University of Technology, SE-412 96 Göteborg, SWEDEN
{fabian,ah}@control.chalmers.se

Abstract. The supervisory control theory is a general theory for automatic synthesis of controllers (supervisors) for discrete event systems, given a plant model and a specification for the controlled behavior. Though the theory has for over a decade received substantial attention in academia, still very few industrial applications exist. The main reason for this seems to be a discrepancy between the abstract supervisor and its physical implementation. This is specifically noticeable when the implementation is supposed to be based on programmable logic controllers (PLCs), as is the case with many manufacturing systems. The asynchronous event-driven nature of the supervisor is not straightforwardly implemented in the synchronous signal-based PLC. We point out the main problems of supervisor implementation on a PLC, and suggest procedures to alleviate the problems.

Keywords. Supervisory Control Theory, Flexible Manufacturing Systems, Programmable Logic Controller.

1. Introduction

Discrete event systems (DESs) are a useful modeling abstraction for certain, mainly man-made, systems, such as manufacturing systems. The main characteristic of DESs is that at each time instant they occupy a discrete symbolic-valued *state*, and perform state-changes on the occurrence of *events*. Events occur asynchronously and instantaneously at discrete intervals of time. Thus, a DESs behavior is described by the sequences of events that occur, and the sequences of states visited due to these.

The supervisory control theory (SCT) [1] is a general approach to the synthesis of control systems for DESs. Given a DES describing the uncontrolled behavior, the *plant*, and a *specification* for the controlled behavior, a *supervisor* can be automatically synthesized to control the plant to stay within the specification.

Though the SCT has received a wide acceptance within academia, industrial applications are scarce. [2] and [3] among others describe applications, but industry has by no means accepted formal methods in general, and the SCT in particular, on a larger scale. As [2] points out, this is mainly due to the problem of physical implementation. There are very few guidelines for how to implement the controller, once the abstract supervisor model has been calculated. Typically, finite-state automata describe the plant, specification and supervisor, and the step to a physical implementation is not necessarily straightforward. In the special case of manufacturing systems, where PLC-control is of great importance, the gap between the event-based asynchronous automata world and the synchronous signal-based PLC-world has to be bridged.

The problems can be dissected into the following main problem-classes.

Signals and events. The SCT consider events that have symbolic values and occur asynchronously at discrete instances of time. PLCs handle boolean valued signals (in our case), the values of which are updated synchronously. Merely regarding the rising and falling edges of the signals as the events does not resolve all problems. Furthermore, in the synchronous signal-based PLC-world several events can occur simultaneously, violating a fundamental assumption of events.

Causality. The SCT assumes that the plant generates all events, and that the supervisor dynamically disables events that the plant might otherwise have generated. This abstraction considerably simplifies the theory. However, in practice, the signals in the plant change values as responses to signal-changes initiated by the PLC. Thus, the concept of causality cannot be avoided in the implementation; we have to answer the question "who generates what?"

Choice. The supervisor is generally required to be *minimally restrictive* [1]. Thus, the supervisor includes alternative paths that the plant is supposed to choose between. However, when the supervisor generates some signal-changes, it may have to choose between alternatives in a single state. Depending on the order of the implementation internally in the PLC, different choices will be made, for each state with alternatives the same choice will always be made. Therefore, those choices have to be made explicitly by the programmer. Deciding the choice to make is by no means trivial, and beyond the scope of this paper. The point we are making is that if we do not choose, the sequential execution of the program within the PLC will make the choice for us.

Inexact synchronization. The supervisor-plant interaction can be modeled by *synchronization*, a useful abstraction that significantly simplifies the SCT. In real life, though, and especially in the PLC-world, exact synchronization does not exist. Time delays due to the periodic updating of the input signals, mean that a supervisor implemented in a PLC cannot execute in exact synchrony with the plant.

These problems will be addressed in Section 3, after a brief preliminary introduction to DESs, the SCT and PLCs in Section 2.

0-7803-4394-8/98 $10.00 © 1998 IEEE

2. Preliminaries

In this section we present the necessary background and define the notation that we will use. Note that we regard the supervisor as exercising control over the plant by the synchronous composition.

2.1. Discrete Event Systems

Discrete event systems (DESs) evolve on the occurrence of *events* that occur asynchronously and at discrete instances of time. Let Σ be a finite set of event-labels, the *alphabet*. Define the concatenation of event-labels as $\Sigma^2 = \Sigma\Sigma$ and $\Sigma^{n+1} = \Sigma\Sigma^n$. Then $\Sigma^* = \bigcup_{i=0}^{\infty} \Sigma^i$ is the set of all finite concatenations of labels over Σ. An element of Σ^* is called a *string*. Let $\Sigma^0 = \varepsilon$ be the empty string. For a string $s \in \Sigma^*$ the *prefixes* of s, denoted $\bar{s}$, is the set of all initial sub-strings of s; that is, $\bar{s} = \{s' \in \Sigma^* | \exists t \in \Sigma^* \text{ s.t. } s = s't\}$. Note that both ε and s itself, are prefixes of s. When s' is a prefix of s, t such that $s = s't$ is said to be the *remaining suffix*.

A subset $L \subseteq \Sigma^*$ is called a *language* over Σ. The *prefix-closure* of a language L, denoted $\bar{L}$, is the union of the prefixes of all its strings, that is, $\bar{L} = \bigcup_{s \in L} \bar{s}$. A language such that $\bar{L} = L$ is said to be (prefix) *closed*, otherwise it is *marked*. Note that it always holds that $L \subseteq \bar{L}$.

An *interleaving* [4] of two strings, $s_1, s_2 \in \Sigma^*$, is a string obtained by concatenating a prefix of one of the strings with a prefix of the other concatenated with the interleaving of the remaining suffixes. All interleavings of $s_1, s_2 \in \Sigma^*$ is a language denoted $s_1 ||| s_2$. Note that $s_1 s_2 \in s_1 ||| s_2$.

A DES A will be described by two languages $L(A)$ and $L_m(A)$, the (prefix) *closed* and the *marked* languages, respectively. We will always assume that $L_m(A) \subset L(A)$.

When two DESs, A and B, are brought together to evolve concurrently, the usual intention is that they are to interact. Interaction is modeled by *synchronization*, which requires mutually labeled events to occur simultaneously in both automata for the occurrence in the composed automaton. Thus, an event occurs in the resulting DES if and only if it can simultaneously occur in both DESs. Letting $\|$ denote the synchronous composition, we get for the composition, $L(A\|B) = L(A) \cap L(B)$ and $L_m(A\|B) = L_m(A) \cap L_m(B)$.

2.2. Supervisory Control

Given a DES plant P and a specification Sp, the supervisory control theory (SCT) is concerned with automatically synthesizing a supervisor S such that the controlled system as modeled by $P\|S$ satisfies Sp. The specification languages are assumed to be sublanguages of the plant languages. The closed-loop system satisfies the specification if its languages are sub-languages of the specification languages, and the closed-loop system is such that $\overline{L_m(P\|S)} = L(P\|S)$.

Not all events of the plant are subject to influence by the supervisor. Some events are *uncontrollable* to the supervisor. Thus, the alphabet Σ is divided into two disjoint sets, Σ_c the controllable and Σ_u the uncontrollable events. P can generate uncontrollable events when in a state to do so, whether S can follow this event or not. This is contrary to the definition of the synchronous composition above. Thus, for the synchronous composition to model the closed-loop system correctly, S must be such that it can always follow those uncontrollable events that it permits P to generate under control; S has to be *controllable* [1].

[1] showed that a supervisor S such that $P\|S$ satisfies the specification exists if and only if there exists controllable sublanguages $K \subset L(Sp)$ and $K_m \subset L_m(Sp)$ such that $K = \overline{K_m}$, K is controllable and $K_m = \overline{K_m} \cap L_m(P)$. This has been proved, in various formulations, by [1], [5] and others. The proof also tells us how to choose that supervisor, namely so that $L(S) = K$ and $L_m(S) = K_m$.

The SCT originally regarded the plant as generating all events, with the supervisor a passive device merely observing the event generation and disabling certain events in certain states. [2] observed the fact that for most real systems the plant events are not generated spontaneously, but only as *responses* to given *commands*. This led Balemi [2] to take an *input/output perspective* on control of discrete event systems. In this interpretation the supervisor generates some events, the *commands*, and the plant generates other events, the *responses*. [2] equates the commands with the controllable events, and the responses with the uncontrollable events. Then, just as the supervisor cannot prevent the generation of the responses, the plant cannot prevent the generation of the commands. Thus, the supervisor must be such that it never generates any command that the plant cannot follow. This is the dual to controllability. [2] shows that this will always be met when the supervisor languages are sublanguages of the plant languages; which will always be the case when the supervisor is synthesized from a specification for which this holds.

In the following we will therefore denote by Σ_S the events generated by the supervisor and by Σ_P the events generated by the plant. Note also that some uncontrollable events do not have intuitive "response" interpretations. Uncontrollable events are, in general, simply a matter of saying "control, but don't touch these". This observation means that it is not necessarily so that $\Sigma_P = \Sigma_u$ and $\Sigma_S = \Sigma_c$.

2.3. Programmable Logic Controllers

Programmable logic controllers (PLCs) have been used in industrial applications since the early 70s. Originally, PLCs were designed to replace hard-wired relay-logic in applications where use of ordinary computers could not be

economically motivated. The PLCs proved to be very versatile, and their application areas have increased constantly, as have their ability to handle more complex control issues. However, their original heritage still influences both their behavior and programming.

To simulate the parallelism inherent in the wired relay-logic, a PLC executes cyclically, reading and storing the inputs, executing the user-program and finally writing the outputs. This read-execute-write cycle, called a *scan cycle*, effectively simulates parallel behavior from an input-output point of view. For the outside viewer, and specifically the plant, the output-signals change their state simultaneously in response to the input-signals, given that the scan cycle time is short with respect to the time constants of the plant.

A PLC-program consists of a set of, in our case, boolean expressions. These are all evaluated during one scan cycle, typically sequentially, one by one, with their results stored in intermediate memory. When all expressions have been evaluated, the part of the intermediate memory that corresponds to the output signals is copied to the outputs and thus shown to the outside world. By using internal memory, sequential behavior can also be accomplished.

PLCs are traditionally programmed in Ladder Diagrams (LDs) for which there now exists an international standard, [6]. A LD consists of graphic symbols, representing for instance contacts and coils, laid out in networks similar to a *rung* of a relay logic diagram. The contacts and coils are connected by links; see Table 1.

The standard for PLC programming languages [6], lists a number of languages. Of these, Sequential Function Chart (SFC) is the most important. SFC allows a PLC program to be organized into a set of *steps* and *transitions* connected by directed links. Associated with each step is a set of actions, and with each transition a transition condition. A SFC evolves as steps are activated and inactivated in accordance with the fulfillment of the transition conditions of the active steps. When a step is active, its associated actions are executed. The SFC programming language can be seen as a restricted variant of safe Petri nets, [7], and would seem to be ideal for implementing supervisor state automata. However, SFCs are most often converted into LD before execution, so that the following problems still apply.

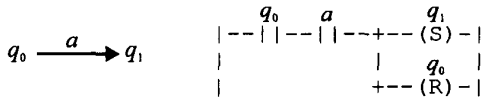

Fig. 1. A simple state machine (left) and its straightforward LD implementation (right).

3. Implementing Supervisors

A supervisor is typically described by a finite state machine, so that supervisor implementation is a matter of making the PLC behave as a state machine. However, there are a number of problems associated with this, both with regard to the underlying assumptions on the supervisor and concerning the asynchronous nature of an event-driven state machine.

The straightforward way to implement a state machine in LD is to represent each state and each event with internal boolean variables, and let the transitions be represented by a boolean AND between the state variable and the event variable. See Fig. 1 and [3]. When the transition occurs the next state is set and the previous state is reset. The latched coils are ideal for this. Note that this is generally the same way that SFCs are converted into LD.

Initialization of the initial state, and correct handling of multiple transitions to and from a state, are straightforward. However, a number of more intricate problems remain.

3.1. Events and Signals

State machines evolve on the occurrence of events, whereas a PLC handles boolean valued signals. Signals always exist but with different values, whereas events only exist momentarily. Intuitively the events could be associated with the rising and falling edges of the signals. Rising edges are detected by comparing the signal between two consecutive scan cycles; if the signal was low the previous scan cycle and is now high, then a rising edge has been detected.

However, when associating events with rising and falling edges of signals, care must be taken not to skip over an arbitrary number of states during the same scan cycle. This is called the *avalanche effect* and is a consequence of the sequential evaluation of the boolean expressions.

In Fig. 2 the state machine transits on the a event from state q_0 to q_1, where it is supposed to transit on the occurrence of

Normally open contact	A --| |--	The state of the left link is copied to the right link if the state of the boolean variable A is true. Else, the state of the right link is false.
Normally closed contact	A --|/|--	The state of the left link is copied to the right link if the state of the boolean variable A is false. Else, the state of the right link is true.
Positive transition sensing contact	A --|P|--	The state of the right link is true for one scan cycle when the state of the left link is true, and a rising edge of the boolean variable A is detected.
Coil	Q --( )--	The state of the left link is copied to the boolean variable Q, and to the right link.
Negative coil	Q --(/)--	The state of the left link is copied to the right link, and the negation of the left link is copied to the boolean variable Q.
Latched coil, Set	Q --(S)--	The boolean variable Q is set true, when the left link is true, else Q is not affected.
Latched coil, Reset	Q --(R)--	The boolean variable Q is set false when the left link is true, else Q is not affected.
Positive transition sensing coil	Q --(P)--	The boolean variable Q is set true for one scan cycle each time the left link goes from false to true.

Table 1. Contacts and coils of the IEC 1131 LD language.

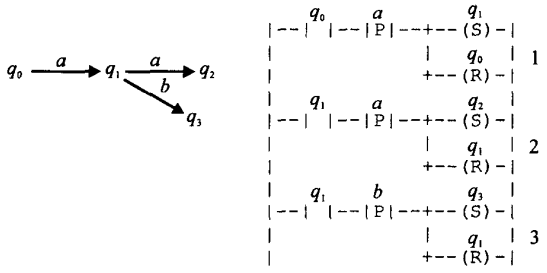

Fig. 2. The state machine (left) represents two strings *aa* and *ab*, whereas the LD implementation (right) represents only the occurrence of a single *a*.

a *new a* event to q_2 or, if a *b* occurs before another *a*, to q_3. The LD implementation to the right in Fig. 2, does *not* exhibit this behavior. Instead, the avalanche effect introduces a direct transition from q_0 to q_3 on the same rising edge of the *a* signal. Thus, even if the *b* event (the rising edge of the *b* signal) occurred, it would have no effect on the evolution of the LD.

One way to avoid the avalanche effect is to arrange the rungs in reverse order. Exchanging the order of rungs 1 and 2 of Fig. 2 will guarantee that a falling edge occurs before the transition from q_1 to q_2 takes place. Assume that q_0 is set and a positive edge is detected on *a*. When rung 2 is scanned before rung 1, then q_2 is *not* set since q_1 is false. On the other hand, when rung 1 is scanned (after rung 2) q_0 is set so that q_1 will be set and q_0 reset. This effectuates the transition from q_0 to q_1. During the next scan cycle, q_1 is set but now there is no positive edge on *a*, so that the transition from q_1 to q_2 cannot occur. Not until *a* has been reset can a positive edge be seen again, which means that at least two scan cycles must pass before the transition to q_2 can occur.

Note that, placing rung 3 of Fig. 2 before rungs 2 and 1 in the execution order will cause no harm. On the contrary, it would guarantee that the supervisor occupied each state at least during one scan cycle. This is important if the control decision is made last in the execution order, as mentioned below. Consequently, the PLC implementation of state machines benefit from sorting the rungs in reverse order, whenever this is well defined. In other cases, the loops have to be broken at "intelligent" places. What this actually means is still an open question. Of course, the sorting order greatly influences the execution times.

Another problem when moving from an event-based into signal-based world is simultaneity. Events are typically assumed to occur non-simultaneously. Signals, on the other hand, can overlap in time. Associating events with signal edges would allow us to still assume the non-overlapping of events, *provided we could guarantee the detection of the rising edge when it occurred*. This is, however, not possible due to the cyclic execution of the PLC.

Regard for instance the state machine of Fig. 3, which attempts to distinguish between the interleaving of the a_1 and a_2 events. If a_1 occurs before a_2 a b_1 is required, and if a_2 occurs before a_1 a b_2 is required. A problem arises if the rising edges of the a_1 and a_2 signals occur between the same scan cycles. Then, the PLC will detect two rising edges in the same scan-cycle, two simultaneously occurring events.

This problem cannot be remedied at all by clever programming, as it is a consequence of the synchronous nature of the PLC execution. To remove the problem the supervisor, would have to be such that the control decision does not depend on different interleavings of the same events. This is captured by the following definition.

Definition 1. Interleave insensitivity

A supervisor S is interleave insensitive *with respect to a plant P if for* $s_1, s_2 \in \Sigma_P^*$ *and* $\sigma \in \Sigma_S$

$$ss_1s_2\sigma \in L(P\|S) \Rightarrow s(s_1|||s_2)\sigma \subset L(P\|S). \quad (1)$$

This definition captures the requirement that after *any* interleaving of the plant generated strings s_1 and s_2 the supervisor generates the same event. This may have benefits with regard to the number of supervisor states as only one of the interleavings need be present in the supervisor. In general, the supervisor (nor the specification) is not interleave insensitive. For instance, it would make perfect sense to allow certain work-parts in a flexible production system not to visit some resources before other work-parts.

3.2. Causality

When controlled by a PLC, the plant and the controlling PLC interact through boolean valued signals. These signals have to be actively set and reset, and this is done either by the controller or by the plant. Thus, the input/output perspective seems a natural approach. At first glance, it would seem to definitely answer the question of causality; "who generates what"? However, the behavior of the controller may change radically with different implementational decisions.

Assume for instance that we have the supervisor shown in Fig. 4. Here the supervisor (to the left) generates *b* after a_1, either before or after a_2. The PLC implementation, though, *always* generates *b* directly after a_1, when the transition to q_1 has been effectuated. This, it does regardless of whether a_2 has occurred or not. Therefore, rung 1 is unnecessary. Note though, that the generation of *b* in q_1 invalidates the reception of a_2, which may lead to another problem, described in Section 3.4, below.

3.3. Choice

The supervisor generated by the SCT is generally required to be minimally restrictive; that is, it allows the plant the greatest possible freedom while still satisfying the specification. This means that alternatives may occur in the

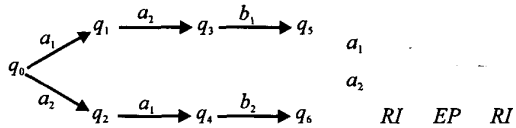

Fig. 3. A state machine (left) that attempts to distinguish between the interleaving of the a_1 and a_2 events. If the signals rising edges occurred between two scan cycles (right), the PLC program would have no knowledge about which event occurred before the other. *RI*: "read input", *EP*: "execute program".

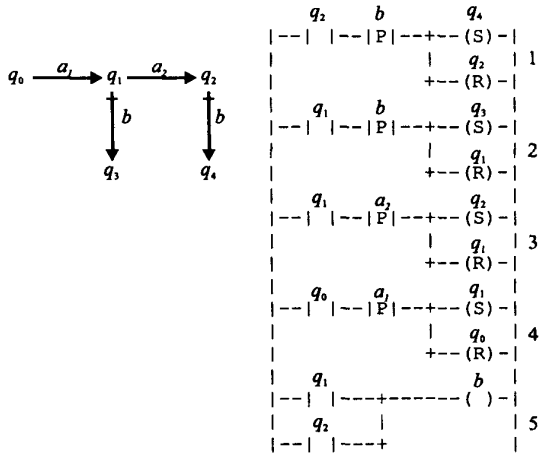

Fig. 4. A supervisor (left) and its PLC implementation (right). The b event is controllable, depicted by the small "handle".

supervisor that the plant is assumed to resolve by generating one of the events. However, when the controller generates some events, only one of those must be generated. Generating several may be contradictory and catastrophic.

In Fig. 5 is shown a supervisor with two alternative implementations. This example is a small part of the supervisor presented by [3]. Assume that q_4 is active and that a positive edge on b_2 is detected. Then the middle implementation of Fig. 5 transits to q_8 in rung 3. This sets q_8 which, in rungs 4 and 5, sets (that is, *generates*) r_2 and s_1, both. This is certainly not the intended behavior. It may even be that generating one invalidates the generation of the other. Furthermore, in the next scan cycle the implementation performs the transition to q_1, thereby disabling q_8. Thus, it behaves as if only the r_2 event was generated. This behavior may even be more catastrophic, since the plant may have ignored the r_2 event altogether when both r_2 and s_1 was set. In that case, the supervisor and the plant are entirely out of synch and it is no longer guaranteed that the supervisor, as implemented, can control the system satisfactorily.

To resolve this problem the implementation must simultaneously choose *and* transit; and only a single event must be chosen. This can be done by implementing exactly as stated in the previous sentence, and as shown in Fig. 5, right. Now, after the transition to q_8 has been made, the PLC in the next

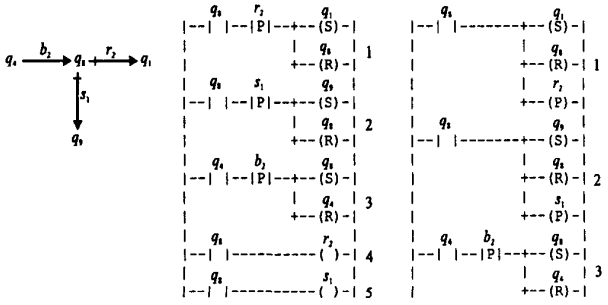

Fig. 5. Example of supervisor (left) and an implementation (middle) that generates two events, r_2 and s_1, but transits only on one of them, the first in execution order. To the right is an implementation that resolves this by choosing and transiting simultaneously.

scan evaluates rung 1; sets q_1, resets q_8 and sets r_2. This effectively performs the transition from q_8 to q_1 while generating r_2. In rung 2 q_8 is reset and no transition to q_9 nor generation of s_1 will take place. We have coded the event generation as a positive transition sensing coil, that generates a one scan cycle wide pulse as it senses q_8's rising edge.

Note that rung 2 in the right implementation of Fig. 5 is superfluous, since when it is executed q_8 has already been reset. Thus, the choice is made at "compile-time". It seems very difficult to implement in LD the "non-determinism" represented by the supervisor of Fig. 5, left. In particular, if the implementor does not explicitly make the choice, the PLC itself will make the choice, determined by the ordering of the rungs.

The "best" choice is a complicated problem, beyond the scope of this paper, that is solved with scheduling techniques, see for instance [8].

In some cases, there may be a "choice" as to whether generate an output event or follow an input event from the same state. Obviously, in those cases it is essential that the ordering of the rungs be such that the "uncontrollable" input event is followed. Otherwise, the state of the supervisor does not replicate the state of the plant. Thus, the rung that transits on the input event must be executed before the rung that generates the output event. This works well when the input event is seen at the right time. However, this is not guaranteed, due to the problem of *inexact synchronization*.

3.4. Inexact Synchronization

While the PLC executes the program, no observing of the plant takes place. The scan cycle is supposed to be short in relation to the time constants of the plant, but there is no guarantee that the two are in phase. A signal-change may occur in the plant while the program is executed. The problem here is that the changed signal may invalidate the choice made by the program. The supervisor can only have knowledge about the commands and responses issued so far.

Consider the supervisor shown in Fig. 6. Assume we start in q_0 and detect a rising edge on a_1. Then, rungs 1 and 2 do nothing since q_1 is not set. Rung 3 on the other hand resets q_0 and sets q_1. In the case that a rising edge is detected on a_2 in the next scan cycle, rung 1 performs the transition to q_2. Otherwise, b is generated in rung 2 with the transition to q_3. However, while the PLC executes the program, the plant

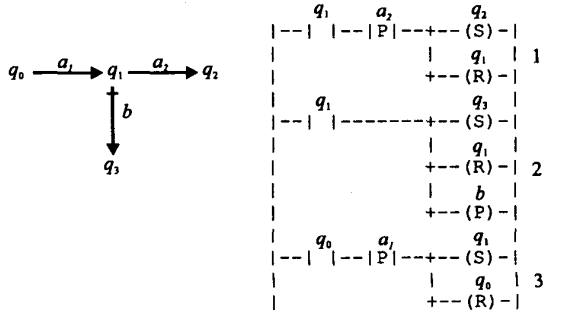

Fig. 6. A supervisor that may become out of synch with the plant if the a_2 is generated while the user program is executed.

may generate a_2, which invalidates the generation of b. That is, the plant cannot accept b after it has generated a_2. Thus, in the case that the rising edge of a_2 is detected in the scan cycle *after* the supervisor has generated b, the plant and the supervisor are out of synch.

This problem can be modeled as a communication delay, as was noted by [2], who therefore introduced the notion of *delay insensitive languages*.

Definition 2. Delay insensitive language [2]

A language K is said to be delay insensitive *if, for* $s \in \overline{K}$, $\sigma_c \in \Sigma_S$, $\sigma_u \in \Sigma_P$

$$s\sigma_c, s\sigma_u \in \overline{K} \Rightarrow s\sigma_u\sigma_c, s\sigma_c\sigma_u \in \overline{K}. \quad (2)$$

Two things can be noted about the above definition. First, it captures delays in one direction only; from the plant to the supervisor. Balemi's motivation for this is that we can always control the plant so that only a single command is generated between the responses. With a PLC implementation, this is natural, since the delay is due to the cyclic execution of the PLC. In the direction from the PLC to the plant, there are typically no delays since all communication is achieved through digital I/O.

Second, the definition concerns only delays of length *one*. This is assumed by [2] in order to simplify the problem. However, again this is inherent in the communication between plant and supervisor. The delay is due to something happening in the plant while the PLC is executing its program and not updating its inputs. The next scan cycle, though, starts with an update of the signals, and thus, the event is seen, assuming that the scan cycle time is short compared with the time constants of the plant.

The definition of delay insensitivity means that for the supervisor in the example above to be delay insensitive, it would have to be able to generate b in q_2 as well as be able to accept a_2 in q_3.

Note, that the definition of delay insensitivity does not capture the "simultaneous events" problem described in Section 3.1. Interleave insensitivity is not a "sub-problem" of delay insensitivity, as it stems from the problem of not being able to distinguish between the order of two (or more) input events. Delay insensitivity, on the other hand, resolves the problem of not observing while choosing. Thus, it concerns the order of input and output events.

4. Conclusions

Supervisor implementation in a PLC-based control system, raises a number of problems. We have described them and suggested implementational details to solve the problems.

First, there is the problem of moving from an asynchronous event-based state-machine world into the synchronous signal-based PLC-world. Two main problems are associated with this, avoiding the avalanche effect and handling the simultaneous event problem. The avalanche effect occurs when several states are directly stepped through on the same event, although the intent was to count the occurring events. Sorting the ladder diagram rungs in an "intelligent" order can solve this. Exact details for how to do this are still an open research matter.

Due to the cyclic execution of the PLC, with a *read-execute-write* scan cycle, events in the plant can appear to occur simultaneously. Thus, if the control decision depends on the order of those events, havoc will reign. The notion of an *interleave insensitive* supervisor captures the requirement that the control decision is not dependent on the order of "simultaneous" events. Again, more research is necessary.

Furthermore, three related problems have been discussed.

Causality – someone must generate the events that are not naturally generated by the plant, and that someone can only be the PLC. Thus, the PLC implementation must include the event generation, and be such that several events are not generated simultaneously.

Choice – the generator must choose. To guarantee that only one choice is made, the choice, that is the generation of the event, must be made simultaneously with the transition. In LD however, the chosen transition will always be the same in a particular state according to the ordering of the rungs.

Inexact synchronization – while the generator chooses, something may occur that invalidates the choice. This problem was addressed by [2] and resolved by the introduction of *delay insensitive* languages. The supervisor is delay insensitive if the choice is not invalidated by a plant event that occurs during the time of making the decision.

There is a strong indication that a correct PLC-implementation of a supervisor can exist if and only if the supervisor has a delay and interleave insensitive language.

5. Acknowledgments

This work received financial support from the Swedish National Board for Industrial and Technical Development (NUTEK) under grant number 9304792.

6. References

[1]. Ramadge, P. J., W. M. Wonham, *Supervisory Control of a Class of Discrete Event Processes*, SIAM Journal of Control and Optimization, Vol. 25, No 1, 206-230, 1987.

[2]. Balemi, S., *Control of Discrete Event Systems: Theory and Application*, Ph.D. thesis, Swiss Federal Institute of Technology, Zürich, Switzerland, 1992.

[3]. Brandin, B., *The Real-Time Supervisory Control of an Experimental Manufacturing Cell*, IEEE Transactions on Robotics and Automation, Vol. 12, No. 1, February, 1996.

[4]. Hoare, C.A.R., *Communicating Sequential Processes*, Prentice Hall International, 1985.

[5]. Kumar, R., V. Garg, S.I. Marcus, *On Controllability and Normality of Discrete Event Dynamical Systems*, Systems & Control Letters, 17, 157-168, 1991.

[6]. ISO/IEC, *International Standard IEC 1131-3, Programmable Controllers – Part 3*, ISO/IEC, 1993.

[7]. Peterson, J.L., *Petri Net Theory and the Modeling of Systems*, Prentice-Hall Inc., 1981.

[8]. Liljenvall, T., *Event Driven Scheduling for Manufacturing Systems*, Lic. Thesis, in preparation.