Available online at www.sciencedirect.com
SCIENCE DIRECT®
Control Engineering Practice 13 (2005) 1283–1293
CONTROL ENGINEERING PRACTICE
www.elsevier.com/locate/conengprac

On the execution of sequential function charts

Anders Hellgren*, Martin Fabian, Bengt Lennartson

Control and Automation Laboratory, Department of Signals and Systems, Chalmers University of Technology, SE-412 96 Göteborg, Sweden

Received 28 January 2004; accepted 30 November 2004
Available online 13 January 2005

Abstract

The transition between the supervisory control theory (SCT) and its implementation in programmable logic controllers (PLCs) is not straightforward. This is mainly due to the fact that the SCT is stated in an event-based asynchronous setting whilst PLCs are signal-based, synchronous and sequential. Based on Petri nets, the PLC programming language sequential function chart (SFC) seems to be the ideal choice for this transition. However, the transition requires detailed knowledge of how the PLC and SFCs work. Different execution models for SFCs are presented and compared with the international standard IEC 61131-3. Often, the application of the SCT to a control problem results in a set of modules between which mutual exclusion and synchronisation are important aspects. The results indicate, however, that for modular SFC programs it seems generally impossible to simultaneously achieve both mutual exclusion between modules as well as synchronisation of modules without using workarounds that may be either manual or suffer from state space explosion. Nevertheless, one of the presented execution models, the immediate transit/immediate action model, together with a handshaking procedure is shown to give a simple solution to both mutual exclusion and synchronisation.
© 2004 Elsevier Ltd. All rights reserved.

Keywords: Discrete event systems; IEC 61131-3; Programmable logic controllers; Sequential function charts

1. Introduction

To meet international competition, including demands on rapid product changes, there is an industrial trend towards technically advanced but also customised products manufactured in small batch sizes. This asks for increased flexibility in modern production systems, where the concurrent development of new products and production processes is crucial to meet demands on reduced costs and shortened time-to-market. This typically lead to more complex control programs that must be known to be correct at the time the production system is put into operation.

Programmable logic controllers (PLCs) are arguably the most important tool for control of automated production systems. Though the programming languages for modern PLCs are highly standardised (ISO/IEC, 2001), these languages can be easily characterised as 'low level'. There is very little support for any high-level structuring, which obviously complicates the development of correct PLC programs.

One approach to raise the correctness-level of PLC programs may be called *virtual verification* (Richardsson & Fabian, 2003). The PLC program is developed in a virtual simulation of the system that it is to control. Manual implementation and testing aims to guarantee the correctness of the code. However, due to the complexity and the usually large size of the control programs, manual testing or simulation of all execution paths of the program is very time consuming, if at all possible. Another way to tackle the problem is to apply mathematically rigorous *formal methods*. Typically, such formal methods are defined in the domain of *discrete event systems* (DESs).

DESs are a useful modelling abstraction for certain, mainly man-made, systems, such as manufacturing

*Corresponding author. Tel.: +46 31 772 3720; fax: +46 31 772 1782.
E-mail address: andh@s2.chalmers.se (A. Hellgren).

0967-0661/$ - see front matter © 2004 Elsevier Ltd. All rights reserved.
doi:10.1016/j.conengprac.2004.11.011

systems. The main characteristic of DESs is that at each time instant they occupy a discrete symbolic-valued state. Events occur asynchronously and instantaneously at discrete instants of time and lead to a change of the state. Thus, a DESs behaviour is described by the sequences of events that occur, and the sequences of states visited due to these.

Two main approaches to the application of formal methods may be distinguished: *Top-down* (model to implementation) and *bottom-up* (implementation to model). Thus, a top-down approach involves first designing a discrete event model of the uncontrolled system as well as a specification of the intended controlled behaviour. The behaviour of the model can then be verified. In the case that the specification is not fulfilled, the system model must be redesigned. Here, the supervisory control theory (SCT) (Ramadge & Wonham, 1989) provides a general framework to the synthesis of control systems for DESs. Given a DES describing the uncontrolled behaviour, the plant, and a specification for the controlled behaviour, a supervisor can be automatically synthesised to control the plant to stay within the specification. Once a model that fulfils the specification is obtained, it remains to be implemented.

Though the SCT has received a wide acceptance within academe, industrial applications are scarce. As Balemi (1992) 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 or Petri nets model the plant, specification and supervisor, and the step to a physical implementation is not necessarily straightforward. In the special case of manufacturing systems, where PLCs are of great importance, the gap between the event-based automata world and the signal-based PLC-world has to be bridged.

In contrast to top-down approaches, a bottom-up approach concerns formal verification of a given PLC program. This typically requires a move to the DES world (Lampérière-Couffin & Lesage, 2000; L'Her, Parc, & Marcé, 1999; Lilius & Östergård, 1991; Treseler, Bauer, & Kowalewski, 2000). For both top-down and bottom-up approaches, though, it is important to know exactly how the PLC works, and in particular how the PLC program is executed. Otherwise, the implementation may not behave as expected or the verification may be misleading.

This was noticed by the authors during the implementation of a modular SFC-based control program for a small transportation system. Due to the way the used PLC system executes SFCs, certain design and structuring decisions had to be made. Examining two other PLC systems, it was clear that these systems executed SFCs differently. Due to these differences, the design and structuring of the control code for the original implementation would not result in correctly functioning control code if implemented in the other two PLC systems. Specifically, it was noticed that the behaviour of mutual exclusion and synchronisation of modules differed depending on whether user-defined boolean variables or the system-defined step activity variables were used.

So, what should one expect, and does it even matter? The answer is, as usual: It depends on the application. In general, there are a number of aspects that must be considered, such as the application being mainly combinatorial logic or sequencing, centralised or decentralised, modular or monolithic. The implementation language is of no less importance, since the languages defined by the international standard for PLC programming languages, IEC 61131-3 (ISO/IEC, 2001), are not mutually interchangeable (Lewis, 1998). Furthermore, the execution of sequential function charts (SFCs) is not unambiguously defined in IEC 61131-3. An SFC program often consists of several concurrently executing SFCs as well as simultaneous sequences within a single SFC. For the safety of such systems it is of great importance to know how such programs behave. As in all real-time systems programmers are, for instance, faced with racing conditions, such as shared variables changed by simultaneously executing code, that may influence the evolution of a program or the results of the execution. The focus in this paper is synchronisation and mutual exclusion of concurrently executing SFCs. Initial investigations on the execution of SFCs may be found in Hellgren, Fabian, and Lennartson (1999). The need for a well-defined execution model for SFCs has more recently also been identified in Bauer and Huuck (2002) and Bornot, Huuck, Lakhnech, and Lukoschus (2000). However, the mutual exclusion and synchronisation problems studied in Hellgren et al. (1999) are not treated in Bauer and Huuck (2002) and Bornot et al. (2000). Conversely, nested (hierarchical) SFCs, that is, actions defined by SFCs is not treated (explicitly) in this paper. It should here be noted that nested SFCs is not a required feature of IEC 61131-3. Note in particular that the PLC systems mentioned above do not support SFC nesting.

For sequential control programs, it is recommended to use SFCs (Lewis, 1998). However, the SFC imposes certain restrictions on the structures that can be (straightforwardly) implemented. More general structures may be implemented by using ladder diagrams (LDs) or structured text (ST), for instance. Indeed, automatic implementation of (coloured) Petri net models by means of LD is addressed in, among others Jiménez, López, and Ramírez (2001) and Uzam, Jones, and Ajlouni (1996), and by means of ST in Feldmann, Colombo, Schnur, and Stöckel (1999). Note, however, that in a modular setting, the problems experienced by

the authors for SFCs are present also in the case that LD or ST is used. Nevertheless, it may then be possible to choose an execution model that is suitable for a particular application. That decision is made by the PLC vendor in the case that SFCs are used. The SFCs shown in Figs. 4 and 5 may then be used to determine the execution model used by the PLC system.

The remainder of the paper is organised as follows. In Section 2, an introduction to PLCs will be given. SFCs will be introduced in Section 2.2. The execution of SFCs will be discussed in detail in Sections 3 and 4, where the execution models presented in Hellgren et al. (1999) are extended, algorithms given and their behaviour examined.

2. Programmable logic controllers

PLCs have been used in industrial applications since the early 1970s. 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 behaviour and programming.

To simulate the parallelism inherent in the wired relay-logic, a PLC executes cyclically, reading and storing the inputs, executing the entire user-program and finally writing the outputs. This read-execute-write cycle, called a *scan cycle*, effectively simulates parallel behaviour from an input–output point of view. For the outside viewer, and specifically the controlled 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.

The IEC 61131-3 standard for PLC programming languages (ISO/IEC, 2001), defines a number of languages. Of these, the SFC is used for sequential structuring of PLC programs. The SFC programming language evolved from a restricted variant of safe Petri nets (Lewis, 1998), suitable for implementation in PLCs. However, low-end PLCs may sometimes not offer SFC programming. Then, traditional languages such as LDs (ISO/IEC, 2001) may be used. An SFC program may be converted into LD as shown in Fig. 3.

2.1. Ladder diagrams

An LD consists of graphic symbols, representing for instance contacts and coils, laid out in a network similar to the rungs of a relay logic diagram. A simple LD is shown in Fig. 1a. The LD corresponds to the boolean functions in Fig. 1b. The input signals are denoted `Ix`

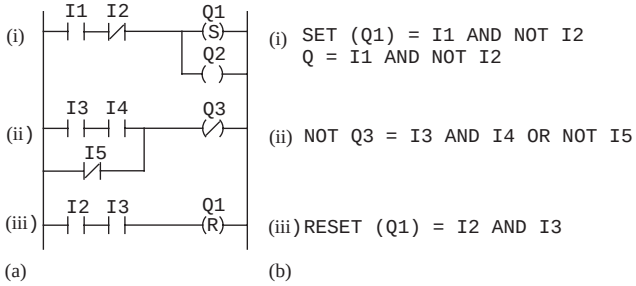

Fig. 1. A Ladder Diagram (a) and the corresponding boolean functions (b).

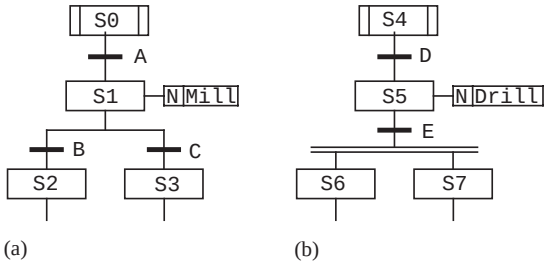

Fig. 2. Some basic SFC constructs.

and the output signals `Qx`. Note that it is considered an unsafe design if two rungs are allowed to influence the same output signal, unless the *Set* and *Reset* modifiers are used as for `Q1` in Fig. 1. Otherwise, the state of `Q1` would always be determined by rung (iii) because it is executed last, regardless of the values of `I1` and `I2`. The Set and Reset modifiers means that the output is only influenced in the case that the right-hand side expression is true. Thus, in the case that both `I1` and `I2` are false `Q1` retains its previous value whilst `Q2` will always be false. A more thorough treatment of LDs may be found in Lewis (1998).

2.2. Sequential function charts

As already mentioned, the SFC evolved from Petri nets to be used in industrial control systems. Similar to Grafcet (David & Alla, 1992; David, 1995), SFC allows a PLC program to be organised 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*. An SFC evolves as steps are activated and deactivated in accordance with the fulfilment of the transition conditions of the active steps. A transition is *active* when all its immediately preceding steps are active. If the transition condition of an active transition is true, the transition is *fireable*.

Some elementary characteristics of SFCs are illustrated in Fig. 2. The initial steps, `S0` and `S4`, respectively, are drawn with vertical bars. The SFC in Fig. 2a has diverging paths leading from step `S1`. For

such cases the standard requires that there is either a priority relation between the transitions or that the transition conditions are mutually exclusive. The default priority rule is a left to right precedence, but it is also possible for the user to explicitly define the precedence. In the following it will, when necessary, be assumed that diverging paths will be handled by the default priority rule. Thus, in the case that `S1` is active and both transition conditions `B` and `C` are true, the left transition has precedence, that is, `S2` would be activated.

Only one step of an SFC may be active at all times, unless the parallel construct shown in Fig. 2b is used. When `S5` is active and transition condition `E` is fulfilled, both `S6` and `S7` are activated. The two sequences thus initiated will evolve independently. Such simultaneous sequences can converge to a single transition using the same double bar construct. Note, however, that mixtures of parallel and diverging paths are not well defined in the standard. Some examples of unsafe and unreachable designs are given, such as a diverging path started within a parallel construct leaving this construct. It is, however, beyond the scope of this paper to classify safe and unsafe designs.

Steps `S1` and `S5` have actions associated with them. An *action block* consists normally of two fields. The left contains the *action qualifier*, whilst the action itself is given in the right field. In this case the action qualifiers are `N`. According to Table 1, this means that the actions are executed once every scan cycle when the corresponding steps are active. Furthermore, every step has two variables associated with them. A boolean that is set when the step is active, denoted `StepName.X` and a timer variable denoted `StepName.T` indicating how long the step has been active. It is also possible that a program contains many SFCs that execute in parallel. The two SFCs in Fig. 2 could for instance be part of the same program.

The execution of an action is determined by the state of its *activity condition*. The state of an activity condition is determined by the action qualifier and the state of the step that the action is associated with. The action qualifiers defined in the standard are shown in Table 1. The update of the activity conditions is defined by an *action control* function block with one input for each of the qualifiers defined in Table 1. However, it is only necessary to implement a functional equivalent using only the qualifiers relevant for each action. The activity condition consists of the *action flag* and the *activation flag*, and may be accessed via the variables `ActionName.Q` and `ActionName.A`, respectively. In general, the activation flag is set whenever the action flag is set and on the falling edge of the action flag. For the `P1` and `P0` qualifiers, though, the action flag is never set.

The simplest action is a boolean variable. The state of the boolean should be the state of its action flag. As can be seen in Table 1, a boolean action with a `P1` or `P0` qualifier will thus never be set. All other types of actions are executed whenever the activation flag is set. The state of the action flag is then used to determine whether it is the final execution of the current activation. This may be used for executing clean up code, if necessary.

Note, however, that vendors may choose to implement a simpler action control function block without the activation flag. In this case actions are executed whenever the action flag is set. Then the `P` and `P1` qualifiers are equivalent. Similarly, the action flag is set one scan cycle also for the `P0` qualifier. In the following it will nevertheless be assumed that both the action flag and the activation flag are implemented.

2.3. Ladder interpretation of SFCs

Sometimes SFCs are converted to LDs before execution. The straightforward and traditional way to make such a *ladder interpretation* is to separate the sequential part of the SFC from the actions. The sequence is achieved by using internal memory. Simultaneously fireable alternative transitions of an SFC are

Table 1
Action qualifiers defined in ISO/IEC (2001)

Qualifier	Description	# Scan cycles	
		Q set	A set
N	Non-stored, executes while associated step is active.	At least 1	At least 2
L	Limited, executes only a limited time while associated step is active. (A restriction of N.)	At least 1	At least 2
D	Delayed, starts executing after the associated step has been active a certain time. (A restriction of N.)	At least 1	At least 2
S	Stored, starts executing when the associated step is activated until reset.	At least 1	At least 2
R	Reset stored action.	N/A	N/A
SL	Stored and limited, executes the specified time. (A restriction of S.)	At least 1	At least 2
SD	Stored and delayed, executes starting from a specified time until reset. (A restriction of S.)	At least 1	At least 2
DS	Delayed and stored, executes until reset but only if the associated step is still active after the specified time.	At least 1	At least 2
P	Pulse, executes when the associated step is activated.	1	2
P1	Pulse, positive flank, executes once when the associated step is activated.	0	1
P0	Pulse, negative flank, executes once when the associated step is deactivated.	0	1

handled correctly by placing the rungs from top to bottom according to their left to right order in the SFC. When the rung of the second fireable transition is executed, the step is already deactivated and safe evolution is guaranteed, that is, only one (non-parallel) step is active at any time.

Similarly, in order to avoid *avalanches*, the rungs should be placed in *reverse order* as was suggested in Fabian and Hellgren (1998). Placing the rungs in reverse order is well defined, and hence straightforward, whenever the SFC has no loops. Otherwise, the loops have to be broken at appropriate places. One way to do this is to introduce a dummy action to one of the steps in the loop. This dummy action should then be part of the transition condition that must be fulfilled in order to deactivate the step.

The straightforward ladder interpretation of the SFCs in Fig. 2 is shown in Fig. 3. The initialisation of step `S0` and step `S4` is omitted for brevity. As can be seen in Fig. 3, the rungs are placed in reverse order so that no step can be activated and deactivated in the same scan cycle. Assume, for instance, that step `S0` is active, that is, the variable `S0` in Fig. 3a is set, and that the transition conditions `A`, `B` and `C` are all true. Then nothing happens when the first two rungs are evaluated because `S1` is false. However, since `S0` and `A` are true, `S1` is set and `S0` reset, that is, the transition between step `S0` and step `S1` is fired. Next, the `Mill` action is executed by the forth rung because `S1` is now true. Assuming that transition condition `B` is true also in the next scan cycle, `S2` will be set and `S1` reset regardless of the value of `C`. In the case that `B` is false and `C` is true, though, `S3` will be set. In either case, the `Mill` action will cease to execute since `S1` is no longer true.

3. IEC 61131-3 SFC execution model

The execution of SFCs requires a number of subtasks to be performed in a certain order in each scan cycle.

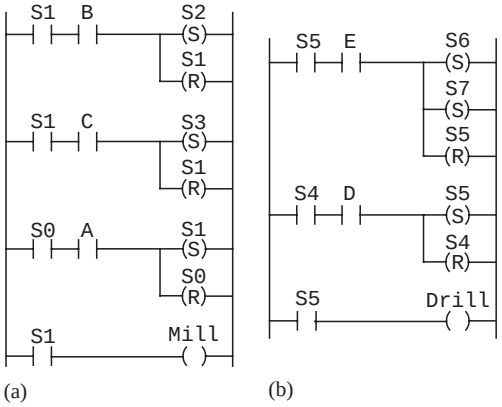

Fig. 3. Ladder interpretation of the SFCs in Fig. 2.

However, the IEC 61131-3 standard (ISO/IEC, 2001) is to some extent vague regarding the execution rules. A basic rule of evolution is that a step cannot be activated and deactivated in the same scan cycle, thus prohibiting avalanche effects. This is similar to the no-search-for-stability interpretation of Grafcet (David & Alla, 1992). Nevertheless, according to Lewis (1998) the standard defines the following tasks in execution order as in the following list (taken from Lewis (1998)):

(1) determine the set of active steps;
(2) evaluate all transitions associated with the active steps;
(3) execute:
  (a) actions with falling edge action flag one last time;
  (b) active actions;
(4) deactivate active steps that precede transition conditions that are true and activate the corresponding succeeding steps;
(5) update the activity conditions of the actions.

Note that task 2 mentions *evaluation of transitions*, whilst task 4 mentions *transition conditions that are true*. Of course, actions executed in task 3 may influence variables used in the transition conditions. Such changes of the state of transition conditions will in the following be disregarded, though. That is, it will be assumed that it in task 2 is decided which transitions to fire in task 4. Otherwise, task 2 would seem pointless.

3.1. Formal execution algorithm

In this section the execution of SFCs will be formalised. This first execution algorithm is based on the execution model given in Lewis (1998). For simplicity, only the steady-state execution will be discussed. In Section 4 alternative execution models will be presented. The following definitions are necessary before an algorithm can be formulated.

Definition 1. An SFC Program is a pair $P = \{SFC, A\}$, where

- SFC is a non-empty set of SFCs.
- A is a set of actions.

Definition 2. An SFC is a 6-tuple $Sfc = \{S, T, Pre, Post, A_c, s_I\}$, where

- S is the non-empty set of steps. For each $s \in S$, the boolean $s.X$ denotes whether the step is active or not.
- T is the non-empty set of transitions. To each $t \in T$ is associated a boolean-valued transition condition denoted $t.C$.
- $Pre \subseteq S \times T$ is the input incidence set. For each $t \in T$, $pre(t)$ is the preset of t, that is, the set of steps with arcs leading to t, the input steps. Presets for steps are

similarly defined.

- $Post \subseteq T \times S$ is the output incidence set. For each $t \in T$, $post(t)$ is the postset of t, that is, the set of steps with arcs leading from t, the output steps. Postsets for steps are similarly defined.
- $A_c \subseteq S \times 2^{A \times Q}$ is the set of action associations. Q is the set of action qualifiers defined in Table 1. For each $s \in S$, $A_c(s)$ is the set of actions associated with s. Similarly, for each $a \in A$, $A_c(a)$ is the set of steps to which a is associated.
- $s_I \in S$ is the initial step.

The presets $pre(\cdot)$, postsets $post(\cdot)$, and action association sets $A_c(\cdot)$ are naturally extended to sets of steps and transitions.

Definition 3. An action is a 3-tuple $a = \{a_{Id}, a.Q, a.A\}$, where

- a_{Id} is the action identity (the action name);
- $a.Q$ is the action flag;
- $a.A$ is the activation flag.

The steady-state execution, according to the interpretation of ISO/IEC (2001) given by Lewis (1998), of an SFC program P can now be formulated, see Algorithm 1. For brevity, the initialisation of the SFCs, that is, the activation of the initial steps and the execution of actions associated with the initial steps is omitted. Furthermore, the details of the update of the action and activation flags are omitted for notational ease. Finally, let $Steps = \cup_{SFC \in P} S$ be all steps and $Trans = \cup_{SFC \in P} S$ be all transitions, respectively, of all SFCs in program P.

Algorithm 1. *SFC execution*

(1) *Let* $ActSteps = ActTrans = \emptyset$.
(2) *Compute* $ActSteps = \{s \in Steps | s.X\}$, *the set of active steps.*
(3) *Compute* $ActTrans = \{t \in Trans | t.C \wedge pre(t) \subseteq ActSteps\}$, *the set of fireable transitions.*
(4) *Execute*:
  (*a*) *each* $a_i | \neg a_i.Q \wedge a_i.A$
  (*b*) *each* $a_i | a_i.Q \wedge a_i.A$.
(5) *For each* $t_i \in ActTrans$
  if $pre(t_i) \subseteq ActSteps$ *then*
  $\left.\begin{array}{l} \forall s_i \in pre(t_i), s_i.X = false \\ \forall s_j \in post(t_i), s_j.X = true \\ ActSteps = ActSteps - pre(t_i) \end{array}\right\}$ $fire(t_i)$
  endif
  endfor
(6) *For each* $a_i \in A$,
  update $a_i.Q$ *and* $a_i.A$
  endfor
(7) *Repeat from step* 2 *in next scan cycle.*

Remark 4. In general, whenever an SFC has diverging paths there may be simultaneously fireable alternative transitions. In the case that the transition conditions are non-mutually exclusive, this must be taken into account in either step 3 by not adding to *ActTrans* transitions that should not be fired according to the chosen priority relation, or in step 5 by checking that the preceding steps are still active before executing the transition. The latter option has been chosen above. Note that this has no practical impact on the following discussion. Note, also, that step 6 has been formulated for notational ease, not computational efficiency. Again, this has no practical impact on the following discussion.

3.2. Synchronisation and mutual exclusion

In this section the synchronisation and mutual exclusion properties of concurrently executing SFCs will be discussed. Synchronisation and mutual exclusion may be based on user-defined signals or on the activity of steps.

3.2.1. Synchronisation

Typically, synchronisation in an SFC is only well defined when synchronising subnets of the same SFC, called *rendezvous* in Lewis (1998). However, when modular implementations are desired (see Hellgren, Fabian, & Lennartson, 2001; Hellgren, 2002), this is not a feasible solution. The modular synchronisation problem is illustrated in Fig. 4 in two forms. In Fig. 4a, the synchronisation is based on the activity of steps, and in Fig. 4b the synchronisation is based on signals.

For the step-based problem, consider Fig. 4a. The SFCs are supposed to execute the transitions leading to steps `S2` and `S4` synchronously. Assume that steps `S1` and `S3` are active. The synchronisation will work in the case that both `S1.X` and `S3.X` are true during the evaluation of the transition conditions in step 3 of Algorithm 1. This is clearly the case in the situation depicted in Fig. 4a.

Consider now Fig. 4b for the signal-based synchronisation problem. The synchronisation will work if the boolean actions `Synch1` and `Synch2` are set during the evaluation of the corresponding transition conditions. In general, there will now be a two cycle synchronisation. Assume, for instance, that `S1` but not `S3` has been active for a few scan cycles. Then `Synch1` is set and `Synch2` is false. Assume now that `S3` is activated. Then,

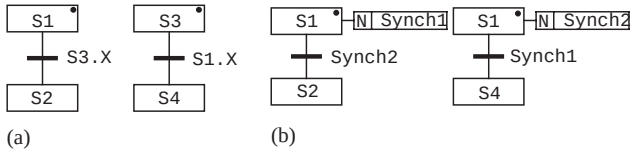

Fig. 4. Synchronisation based on steps (a) and signals (b).

at the beginning of the next scan cycle, Synch2 will still be false and the transition from S1 to S2 will not be found fireable. The transition from S3 to S4 is fireable, though, since Synch1 is set and S3 is active. Then, after this evaluation, Synch2 will be set in step 4 of Algorithm 1. This is followed by the activation of S4 and deactivation of S3. In the following scan cycle, Synch2 is set during the evaluation of the transitions and, consequently, S1 will be deactivated and S2 activated. Thus, both step- and signal-based modular synchronisation is supported by the IEC 61131-3 standard.

3.2.2. Mutual exclusion

For the mutual exclusion problem, consider the SFCs in Fig. 5. The step-based problem is illustrated in Fig. 5a. The two SFCs are supposed to mutually exclude each other from having steps S2 and S4 active simultaneously. Assume that steps S1 and S3 are active. Then, in step 3 of Algorithm 1, both transition conditions, NOT S4.X and NOT S2.X, are true since neither S2 nor S4 are active. Both transitions will thus be fired in step 5 of Algorithm 1, and the intended mutual exclusion will fail.

Step-based mutual exclusion is practical only for a small number of steps. Instead, signals may be used, as in Fig. 5b. Assuming that steps S1 and S3 are active it will in step 3 of Algorithm 1, again, be found that both transition conditions are true. Thus, both S2 and S4 will be activated and, again, the mutual exclusion fails.

3.3. Discussion

Clearly, it is unsatisfactory that the straightforward modular mutual exclusion does not work. For small systems, though, it is fairly easy to manually introduce priorities in the transition conditions. For instance, in Fig. 5a NOT S1.X AND NOT S2.X may be used to give the left SFC priority over the right SFC in the case that both S1 and S3 are active. For large-scale systems it is necessary to automate the introduction of such priorities. In general, though, an exhaustive search of the state space may be required to find all interdependencies between the relevant transition conditions. The question, then, is whether it is possible to reorder the different tasks of the SFC execution so that mutual exclusion works without priorities. Preferably, this reordering should not influence the synchronisation

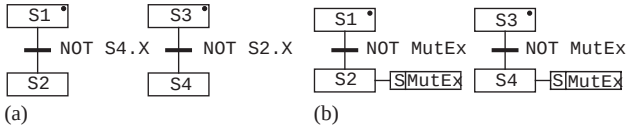

Fig. 5. Mutual exclusion based on (a) steps being active or inactive and (b) signals.

properties. This is discussed in the remainder of this paper. Alternatively, implementation schemes that circumvent these problems should be devised. One such scheme for the implementation of supervisory control is presented in Hellgren et al. (2001) and Hellgren (2002).

4. Alternative SFC execution models

As is shown above, the SFC execution defined by IEC 61131-3 is problematic in that it does not handle both synchronisation and mutual exclusion of simultaneously executing modular SFCs. It should also be noted that commercially available control systems are not necessarily fully standards compliant, nor do they implement the same execution model. One system from a European vendor examined by the authors, for instance, executes task 2 and task 4 interleaved as one. This is captured in the *immediate transit evolution model* below. Furthermore, in Section 3 it was assumed that each step of the execution algorithm was carried out for all SFCs before the next step of the algorithm was carried out. This is the case for the system mentioned above. There are other possibilities, though. For instance, another European system as well as a Japanese system execute each SFC individually. Interaction of different SFCs will then be consistent with the *immediate transit/immediate action execution model* below. Execution models for some different PLC makes have also been investigated in Bauer and Treseler (2001). Some different ways to reorder the tasks of the SFC execution will be examined next.

From Algorithm 1 it can be seen that there are two aspects of the execution of SFCs. First, there is the *evolution* of the SFC that pertains to the firing of the transitions, that is, steps 2, 3 and 5. Second, there is the *activity* model that pertains to the execution of actions, that is, steps 4 and 6. Combining the evolution and activity models, the *execution* model is obtained. The choice of execution model has a significant impact on the behaviour of concurrently executing SFCs. The exact execution model is therefore of utmost importance when verifying correctness of SFC-based programs.

4.1. Evolution models

There are two extremes of evolution models, called the *deferred transit* and the *immediate transit* evolution model, respectively.

Under the deferred transit evolution model (DT), the transition conditions of all transitions leading from active steps are evaluated first. Then, the transitions that were found fireable are executed one by one. Thus, the set of active steps is updated *after* all transition conditions have been evaluated. Hence, the deferred transit evolution model implements tasks 2 and 4 as two

separate tasks, and in that order. Note that this is exactly as specified by IEC 61131-3. In other words, the standard implements the deferred transit evolution model.

Under the immediate transit evolution model (IT), the transition conditions of transitions leading from active steps are evaluated one by one. In the case that a transition condition is true, that is, the corresponding transition is fireable, this transition is immediately fired. Thus, the set of active steps is updated during the evaluation of the transition conditions. In other words, tasks 2 and 4 of the SFC execution are interleaved. Unlike the deferred transit evolution model and the standard, the determination of active steps is crucial for avoiding avalanches in this case.

4.2. Activity models

Similar to the evolution models, there are two extremes of activity models, called the *deferred action* and *immediate action* activity model, respectively.

Under the deferred action activity model (DA), all actions of active steps are executed after the evolution of the SFCs is completed. That is, the set of active steps is updated *before* the actions of the active steps are executed. As a dual, actions associated with deactivated steps cease to execute after the completion of the evolution of the SFC. Thus, this activity model is different from IEC 61131-3 in that IEC 61131-3 executes the actions between the decision what transitions to fire, and the firing of those transitions. Furthermore, the deferred action activity model implements task 5 before task 3. The difference in behaviour this yields is studied in the next section. Also, note that this model does not say anything about the order in which the actions are executed, only that they are executed one by one after the update of active steps. This activity model is straightforwardly combined with both evolution models.

Under the immediate action activity model (IA), actions are executed as soon as the state of the steps to which they are associated are determined. Though conceptually clear, this model is not as straightforward to implement as is the case for the deferred action activity model. As will be shown in the next subsection, though, the combination with the immediate transit evolution model is important.

4.3. Behaviour of the execution models

The evolution and activity models presented above will now be combined to get a set of execution models. The different combinations of execution models are DT/DA, DT/IA, IT/DA and IT/IA. The DT/IA model, however, conflicts with the system state snapshot nature of the deferred transit model and will, therefore, not be discussed further.

The DT/DA, IT/DA and IT/IA execution models may be described as *transit then execute*. There is another possibility, though; *execute then transit*. This latter approach is taken in Bauer and Huuck (2002) and Bornot et al. (2000). However, it is the authors' opinion that it is more natural to first change state due to changed inputs. Intuitively, the state determines what actions to perform, rather than the other way around, as in Bauer and Huuck (2002) and Bornot et al. (2000). Changed inputs typically mean that actions different from the ones in the previous scan cycle should be executed. The approach taken in Bauer and Huuck (2002) and Bornot et al. (2000) may in such cases lead to unexpected results.

A major difference between the execution models presented in this section and IEC 61131-3 is that the standard executes the actions between the evaluation of the transitions and the firing of the transitions. According to IEC 61131-3, non-stored boolean actions are supposed to reflect the state of the steps they are associated with. However, as the authors understand the standard, there will be a one cycle delay. Actions are set the cycle *after* the corresponding steps have been activated and reset the cycle *after* the steps have been deactivated. Similarly for the execution model in Bauer and Huuck (2002) and Bornot et al. (2000). This delay is not present in the alternative execution models presented in this paper.

Algorithms for the deferred action models can be obtained from Algorithm 1 with some modifications, mostly a reordering of the steps. The DT/DA model is given in Algorithm 2, and the IT/DA model in Algorithm 3.

Algorithm 2. *DT/DA SFC execution*

(1) *Let* $ActSteps = ActTrans = \emptyset$.
(2) *Compute* $ActSteps = \{s \in Steps | s.X\}$, *the set of active steps.*
(3) *Compute* $ActTrans = \{t \in Trans | t.C \wedge pre(t) \subseteq ActSteps\}$, *the set of fireable transitions.*
(4) *For each* $t_i \in ActTrans$
  if $pre(t_i) \subseteq ActSteps$ *then*
    $fire(t_i)$
  endif
endfor
(5) *For each* $a_i \in A$,
  update $a_i.Q$ *and* $a_i.A$
endfor
(6) *Execute*:
  (*a*) *each* $a_i | \neg a_i.Q \wedge a_i.A$
  (*b*) *each* $a_i | a_i.Q \wedge a_i.A$.
(7) *Repeat from step* 2 *in next scan cycle.*

Algorithm 3. *IT/DA SFC execution*

(1) *Let* $ActSteps = ActTrans = \emptyset$.
(2) *Compute* $ActSteps = \{s \in Steps | s.X\}$, *the set of active steps.*
(3) *For each* $t_i \in Trans$
  if $pre(t_i) \subseteq ActSteps \wedge t_i.C$ *then /* t_i is fireable */*
    $fire(t_i)$
  endif
 endfor
(4) *For each* $a_i \in A$,
  update $a_i.Q$ *and* $a_i.A$
 endfor
(5) *Execute*:
 (*a*) *each* $a_i | \neg a_i.Q \wedge a_i.A$
 (*b*) *each* $a_i | a_i.Q \wedge a_i.A$.
(6) *Repeat from step* 2 *in next scan cycle.*

The IT/IA model, though, requires a more elaborate algorithm. The reason is that for this model one has to take into account that actions, in general, may be associated with more than one step. This means that the state of all steps to which an action is associated must be known before the action can be executed. The algorithm must also handle alternative transitions. Actions associated with a step cannot be executed until the transition condition of all transitions leading from that step have been evaluated. The functionality to handle these two situations is embedded in the *execute* function in Algorithm 4. The *update* function is similarly a shorthand for updating the action and activation flags of a set of actions. Note that it can always be assumed that a step has an output transition. In the case that there exist a step without output transition, a transition with an always false transition condition can be added. That is, all steps will be considered in the iteration over all transitions in step 3 of Algorithm 4.

Algorithm 4. *IT/IA SFC execution*

(1) *Let* $ActSteps = ActTrans = \emptyset$.
(2) *Compute* $ActSteps = \{s \in Steps | s.X\}$, *the set of active steps.*
(3) *For each* $t_i \in Trans$
  if $pre(t_i) \subseteq ActSteps \wedge t_i.C$ *then /* t_i is fireable */*
    $fire(t_i)$
    $ActTrans = ActTrans \cup \{t_i\}$
    / First-time execution */*
    update $A_c(pre(t_i))$
    execute $A_c(pre(t_i))$
  endif
  / Don't execute twice in the same scan-cycle. */*
  if $pre(pre(t_i)) \cap ActTrans = \emptyset$ *then*
    update $A_c(pre(t_i))$
    execute $A_c(pre(t_i))$
  endif
 endfor
(4) *Repeat from step* 2 *in next scan cycle.*

Next, the mutual exclusion and synchronisation properties of the execution models will be discussed.

4.3.1. Synchronisation

The step-based problem will be discussed first. Note that this problem is only related to the evolution model since no actions are involved. The SFCs in Fig. 4a are supposed to execute the transitions leading to steps `S2` and `S4` synchronously. Assume that steps `S1` and `S3` are active. Obviously, in the case that the DT model is used, the synchronisation works (see Section 3.2). In the case that the IT model is used, though, the synchronisation fails. Let for instance the left SFC be checked first. Then, because `S3.X` is true, `S1` is deactivated and `S2` activated. The transition condition `S1.X` is now false so that `S4` cannot be activated, thus preventing the intended synchronisation to occur.

Consider now Fig. 4b for the signal-based synchronisation problem. Assume that steps `S1` and `S3` are active. Again, the synchronisation will work if the boolean actions `Synch1` and `Synch2` are set during the evaluation of the transition conditions. This is the case for all DA models. IT/IA is, nonetheless, unable to achieve the synchronisation.

4.3.2. Mutual exclusion

The step-based problem is, again, discussed first. Consider, again, Fig. 5a. The two SFCs are supposed to mutually exclude each other from having steps `S2` and `S4` active simultaneously. Assume that steps `S1` and `S3` are active. First recall that the standard implements the DT model. It is thus clear that in the case that the DT model is used the mutual exclusion fails. In the case that the IT model is used, though, the mutual exclusion works. Let for instance the left SFC be checked first. Then, because `S3` is active, `NOT S4.X` is true and `S1` is deactivated and `S2` is activated. The transition condition `NOT S2.X` is now false so that `S4` is not activated, as intended.

To illustrate the signal-based mutual exclusion, consider Fig. 5b. Again, assume that steps `S1` and `S3` are active. From the discussion above it is obvious that DT/DA is not able to achieve the desired mutual exclusion because S2 and S4 are inactive during the evaluation of the transition conditions and, consequently, the actions of these steps have not yet been executed. The mutual exclusion fails also for IT/DA because actions are executed after the evolution is finished. It is only IT/IA that is able to achieve the mutual exclusion.

4.4. Discussion

To conclude, each model has advantages and disadvantages. However, regardless of the choice of execution model it is not possible to automatically achieve *both* mutual exclusion *and* synchronisation. In general, mutual exclusion requires an immediate update of the internal environment, as is emphasised by the IT/IA execution model. This effectively simulates the behaviour of concurrently executing automata models, for instance, where events are observed instantaneously on their occurrence. That is, changes take effect immediately. Conversely, synchronisation requires that a snapshot of the state of the program is taken, as is emphasised by the DT/DA execution model. This simulates that all relevant subsystems react simultaneously on the occurrence of events. It is stated in Lewis (1998) that from a programming point of view all transition firings should be considered simultaneous. The examples in Figs. 4 and 5 clearly show that such an assumption often cannot be made; at least not when any of the execution models presented here is used, including the one specified by IEC 61131-3.

A prototype implementation of the IEC 61131-3 standard is presented in Öhman, Johansson, and Årzén (1998). In their discussion regarding the execution of SFCs they propose an execution model similar to the DT/DA model presented above. However, given the choice, the authors of this paper maintain that the IT/IA model should be used. It is the authors' opinion that it is a better strategy to handle mutual exclusion automatically rather than synchronisation, since modular synchronisation may be easily achieved by *handshaking* as is shown in Fig. 6. Note specifically that handshaking does not require an exhaustive search of the state space. In other words, the IT/IA model is advantageous complexity wise.

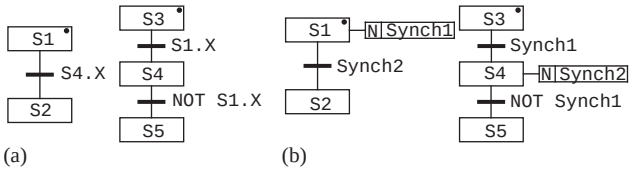

Fig. 6. Step based (a) and signal-based (b) synchronisation by means of handshaking.

5. Conclusions

The execution of sequential function charts (SFCs) has been discussed. There are two fundamental aspects of the execution of SFCs: The *evolution* and the *activity*. Different ways to combine models for the evolution and the activity have been presented and compared with the standard IEC 61131-3. The *execution* models, or *semantics*, thus obtained differ significantly in behaviour for modular SFC programs. Therefore, an SFC program that exhibits a certain behaviour in one PLC may behave differently in another. In other words, the semantic differences influence portability between PLCs. Specifically, the transportation system implementation discussed in the introduction would have to be structurally modified in order to function properly on the other PLCs examined by the authors.

Specific problems caused by these differences in behaviour are synchronisation and mutual exclusion. Synchronisation is best achieved by using the *deferred transit + deferred action* (DT/DA) model, whilst mutual exclusion requires the *immediate transit + immediate action* (IT/IA) model. Since synchronisation may be achieved by handshaking, it is the authors' opinion that, generally, IT/IA is a better model. In any case, the design and implementation as well as the verification of SFC programs requires a detailed knowledge of the execution model. An interesting future feature for PLCs and verification tools would be the possibility to choose execution model. Then, it may be shown that an SFC program would function correctly simply by changing the execution model, or that the execution model that requires the least amount of reprogramming can be used.

References

Balemi, S. (1992). *Control of discrete event processes: theory and application*. Ph.D. thesis, Swiss Federal Institute of Technology, Zürich, Switzerland.

Bauer, N., & Huuck, R. (2002). A parameterized semantics for sequential function charts. In *Proceedings of the semantic foundations of engineering design languages* 2002, Satellite Event of ETAPS 2002, April 2002.

Bauer, N., & Treseler, H. (2001). Vergleich der Semantik der Ablaufsprache nach IEC 61131-3 in unterschiedlicher Programmierwerkzeugen, VDI-Bericht 1608 (pp. 135–142). Dusseldorf: VDI-Verlag.

Bornot, S., Huuck, R., Lakhnech, Y., & Lukoschus, B. (2000). An abstract model for sequential function charts. In R. Boel, & G. Stremersch (Eds.), *Discrete event systems: analysis and control*. Dordrecht: Kluwer Academic Publishers.

David, R. (1995). Grafcet: a powerful tool for specification of logic controllers. *IEEE Transactions on Control Systems Technology*, *3*, 253–268.

David, R., & Alla, H. (1992). *Petri nets and grafcet: tools for modelling discrete event systems*. Englewood Cliffs, NJ: Prentice-Hall International.

Fabian, M., & Hellgren, A. (1998). PLC-based implementation of supervisory control for discrete event systems. In *Proceedings of the CDC'98*, Tampa, FL, USA, December 1998.

Feldmann, K., Colombo, A. W., Schnur, C., & Stöckel, T. (1999). Specification, design and implementation of logic controllers based on coloured petri net models and the standard IEC 1131 Part II: design and implementation. *IEEE Transactions on Control Systems Technology*, *7*, 666–674.

Hellgren, A. (2002). *On the implementation of discrete event supervisory control*. Ph.D. thesis, Technical report no. 422, Department of Signals and Systems, Chalmers University of Technology, Göteborg, Sweden.

Hellgren, A., Fabian, M., & Lennartson, B. (1999). Synchronised execution of discrete event models using sequential function charts. In *Proceedings of the 1999 IEEE conference on decision and control*, Phoenix, AZ, USA, December 1999.

Hellgren, A., Fabian, M., & Lennartson, B. (2001). Modular implementation of discrete event systems as sequential function charts applied to an assembly cell. In *Proceedings of the 2001 IEEE conference on control applications*, Mexico City, Mexico, September 2001.

ISO/IEC. (2001). *International standard IEC 61131-3* (2nd ed.). Programmable logic controllers—Part 3. ISO/IEC (final draft).

Jiménez, I., López, E., & Ramírez, A. (2001). Synthesis of ladder diagrams from petri net controller models. In *Proceedings of the 2001 IEEE international symposium on intelligent control*, Mexico City, Mexico, 2001.

Lampérière-Couffin, S., & Lesage, J. J. (2000). Formal verification of the sequential part of PLC programs. In R. Boel, & G. Stremersch (Eds.), *Discrete event systems: analysis and control*. Dordrecht: Kluwer Academic Publishers.

Lewis, R. W. (1998). *Programming industrial control systems using IEC 1131-3*. IEE control engineering series (Vol. 50) (Revised ed.). London, UK: The Institution of Electrical Engineers.

L'Her, D., Parc, P., & Marcé, L. (1999). Proving sequential function chart programs using automata. In J.-M. Champarnaud, D. Maurel, D. Ziadi (Eds.), *Lecture notes in computer science* (Vol. 1660) (pp. 149–163). Berlin: Springer.

Lilius, J., & Östergård, P. (1991). On the verification of programmable logic controller programs. In *Proceedings of the 12th international conference on application and theory of petri nets* (pp. 310–328). Aarhus 1991.

Öhman, M., Johansson, S., & Årzén, K. E. (1998). Implementation aspects of the PLC standard IEC 1131-3. *Control Engineering Practice*, *6*, 547–555.

Ramadge, P. J. G., & Wonham, W. M. (1989). The control of discrete event systems. *Proceedings of the IEEE*, *77*(1), 81–98.

Richardsson, J., & Fabian, M. (2003). Automatic generation of PLC programs for control of flexible manufacturing cells. In *Proceedings of the ninth IEEE international conference on emerging technologies and factory automation*, 16–19 September 2003, Lisbon, Portugal.

Treseler, H., Bauer, N., & Kowalewski, S. (2000). Verification of IL programs with an explicit model of their PLC execution. In R. Boel, & G. Stremersch (Eds.), *Discrete event systems: analysis and control*. Dordrecht: Kluwer Academic Publishers.

Uzam, M., Jones, A. H., & Ajlouni, N. (1996). Conversion of petri net controllers for manufacturing systems into ladder logic diagrams. In *Proceedings of the 1996 IEEE conference on emerging technologies and factory automation*, Kauai, HI, USA.