PLCopen®
for efficiency in automation

PLCopen Software Construction Guidelines:

Structuring with SFC: do’s and don’ts

PLCopen Technical Document

Version 1.0 Official Release

DISCLAIMER OF WARANTIES

The name ‘PLCopen®’ is a registered trade mark and together with the PLCopen logos owned by the association PLCopen.

This document is provided on an ‘as is’ basis and may be subject to future additions, modifications, or corrections. PLCopen hereby disclaims all warranties of any kind, express or implied, including any warranty of merchantability or fitness for a particular purpose, for this moment. In no event will PLCopen be responsible for any loss or damage arising out or resulting from any defect, error or omission in this document or from anyone’s use of or reliance on this document.

Date: Jul 3, 2018

The following paper

**PLCopen Software Construction Guidelines:
Structuring with SFC: do's and don'ts**

is a PLCopen training document.
It is made possible and based on sections of the book "IEC 61131-3 Programming Methodology: Software engineering methods for industrial automated systems" by Dr. Monari, Prof. Bonfatti and Dr. Sampiery.
It summarises the results of the Task Force *Structuring with SFC: do's and don'ts* and several meetings of PLCopen Promotional Committee Training, containing contributions of all its members.

The present specification was written thanks to the following members:

Name	***Company***
Bert van der Linden	ATS
Bernhard Werner	Codesys
Hiroshi Yoshida	Omron
Barry Buxton	Omron
Arnulf Meixner	Phoenix Contact
Andreas Weichelt	Phoenix Contact Software
Eelco van der Wal	PLCopen

Change Status List:

Version	**Content**
V 0.1	Sept. 11, 2014. Basic document provided by Mr. Bonfatti et al. and basic editing by EvdW
V 0.2	Oct. 2, 2014. With comments from Barry Buxton. Provided to the group
V 0.3	Nov 28 – incl. comments and objective from Andreas Weichelt and Hiroshi Yoshida
V 0.4	Dec.3 2014 – during the Face2Face meeting in Frankfurt
V 0.5	March 2017 – as result of meeting with Bert van der Linden in Jan. and addition of PackML
V 0.6	June 29, 2017 as result of the webmeeting on May 31 and June 29
V0.6a	July 31, 2017 as a result of the webmeeting on July 6
V 0.7	August 24, as result of the webmeeting and the addition of Ch 3.6 on final scan
V0.8	As result of the webmeeting January 25, 2018. Basis for V0.99 Release for Comments
V0.99	Release for Comments, published on Feb. 28, 2018
V 1.0	Official release

TABLE OF CONTENTS

1 Introduction to this document

This document explains the advantages of Sequential Function Chart, SFC. This is a very expressive graphic formalism of the IEC 61131-3 standard. It is not considered a programming language as it needs other languages to express transition conditions and actions.

SFC provides a means for partitioning a programmable controller program organization unit into a set of steps and transitions interconnected by directed links. Associated with each step is a set of actions, and with each transition is associated a transition condition. Since SFC elements require storage of state information, the only POUs which can be structured using these elements are function blocks and programs (not functions).

If any part of a program organization unit is partitioned into SFC elements, the entire program organization unit shall be so partitioned. If no SFC partitioning is given for a program organization unit, the entire program organization unit shall be considered to be a single action which executes under the control of the calling entity.

Whenever a sequential process shall be controlled, SFC shall be considered as most suitable for structuring the internal organization of a POU especially in the Functional Description:

- When the process consists of several steps to be executed sequentially in a time flow, for example an assembly process, SFC can be used to map the different phases of assembly as *steps* and structure the process as sequence of *steps*.
- When the process can be modelled as a state machine, these states can be mapped to *steps* and changing from one state to another can be structured by *transitions*.
- SFC structures the internal organization of a program, and helps to decompose a control problem into manageable parts, while maintaining the overview.

1.1 Software Modularity and the role of SFC

The main prerequisite to improve the current programming practice and create a higher productivity and better quality of the resulting product is software modularity.

Software modularity means that a software program has to be organised into weakly coupled parts and each of them should be developed and tested independently of the others.

For this the PLC software development process must rely on a proper methodology which, in our opinion, should be based on two fundamental assumptions:

- ■ *Focus on design*. The weight of the design phase should increase, so that the development phase starts only after a clear definition of the control program structure and the interactions between its parts.
- ■ *Focus on standard*. The potentialities of the IEC 61131 standard languages should be fully exploited in both the design and development phases, as they can cover most of the programmer needs.

SFC is a suitable language (in IEC 61131-3) to support the initial and more crucial phases of the PLC software development life cycle. Reasons include:

- *High expressive power*. The SFC language has the same expressive potential as state diagrams and, it is comparable to Petrinets in facing concurrent problems. State diagrams and Petrinets are considered the most appropriate tools to model dynamics and they are extensively used in many fields. Thus, we can say that the SFC language is intrinsically capable of modelling the behaviour of a system.

- *Graphic formalism*. SFC is not the only language with graphic primitives made available by the standard, but with respect to the other two graphic languages (LD and FBD) it offers higher level characteristics to describe system dynamics. Thanks to its graphic syntax, it is very easy to learn and use. Moreover, it results particularly suited to represent the process at different detail levels.
- *Support of preliminary design*. The SFC graphic formalism can be used right from the beginning, to give a first formal representation of the system behaviour. Therefore, it is a precious tool in the preliminary analysis and early design phases, when many aspects are still not well defined or even unknown to the designer. Using SFC we avoid adding the ambiguity of natural language description to the approximate specifications available to the designer. In this way, the number of misunderstandings between customer, designer and programmers is substantially reduced.
- *Support of detailed design*. The SFC scheme produced in the early design phase can be progressively specified and refined as new information becomes available. Thus, the desired level of detail is reached stepwise.
- *Natural connection with the other languages*. It is quite evident that the SFC language can be used in combination with the other languages of the standard, particularly suited to describe control details such as transition conditions and elementary actions. The possibility to use the right language at the right moment increases the global efficiency of software development and betters the performances of the resulting executable code.
- *Support of software partitioning*. Using the SFC language makes the partitioning of code into portions to be executed at different cycle scans easier. This is one of the possible techniques to reduce the maximum execution time of the cycle. The advantage given by SFC schemes is that they represent such partitions explicitly and, above all, they clarify which are the conditions affecting the execution order of the various software portions.

SFC is a good choice because:

- each state of the process can be clearly mapped to a *step*
- the *transition* from one state/*step* to different/next state/*step* can be expressed by its corresponding condition
- the relation of states/*steps* and the activity flow from one state to another is visualized/programmed (usually) in a graphical manner and easy to understand due to the underlying rules for evolution of the active states of steps. Consequently, the SFC chart mirrors the design of the process. This benefit applies to all phases of the PLC software development life cycle: planning, design (incl. coding, debug and test), commissioning and maintenance.
- the *actions* to be performed are created independently of the states/*steps* and combined with them using action blocks (and corresponding action qualifiers)

2 Introduction SFC

2.1 Steps and transitions

SFC provides a means for partitioning the POU into a set of steps and transitions interconnected by directed links. Associated with each step is a set of actions, and with each transition a transition condition.

A step represents a process situation. A step is either active or inactive. A step is represented graphically by a block containing a step name in the form of an identifier or textually by a STEP...END_STEP construction (see Figure 1).

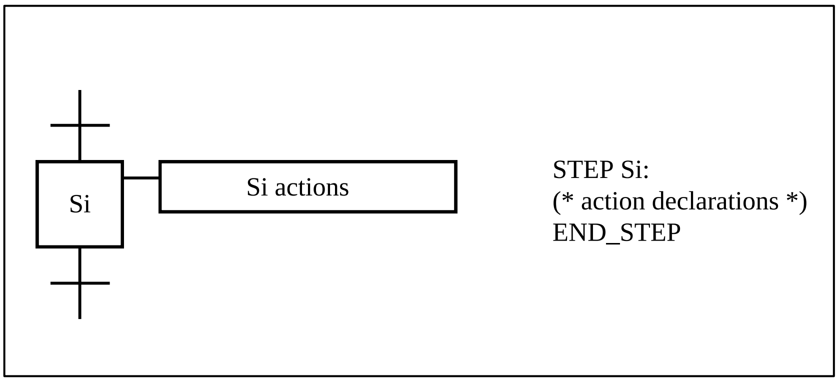


Figure 1 **Graphical and textual representation of a step**

It is advised for real programs to use more process related names like Mixing.

The step flag (representing in the system the active or inactive state of a step) is represented by the logic value of the Boolean variable S.X, where S is the step name. Similarly, the elapsed time S.T of the S step is defined as a TIME variable. When the step is deactivated the value of its time parameter remains at the value it had at the deactivation event, while it is reset to t#0s when the step is activated. The scope of step names, step flags, and step times is local to the POU wherein the step is used.

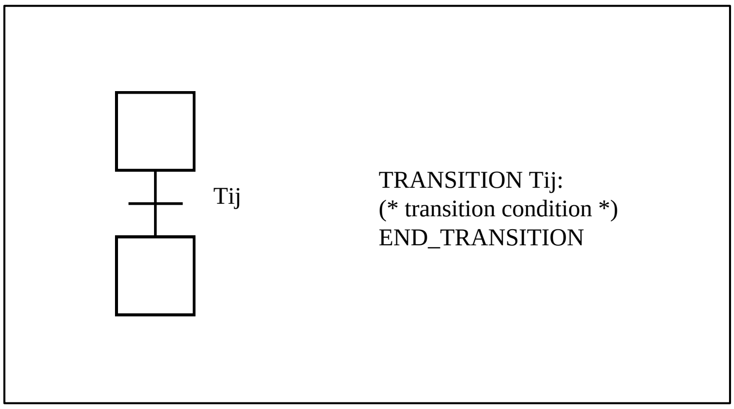
The initial state of the POU is represented by the initial values of its internal and output variables and by its set of initial steps, i.e., the steps which are initially active. Each SFC network, or its textual equivalent, has exactly one initial step, drawn graphically with double lines for the borders and with the keyword INITIAL_STEP in the textual representation. For system initialisation, the default initial step flag is FALSE for ordinary steps and TRUE for the initial steps.

A transition represents the condition whereby control passes from one or more preceding steps to one or more successor steps along the corresponding directed link. The processing order is from the bottom of the predecessor step(s) to the top of the successor step(s). Each transition has an associated transition condition which is the result of the evaluation of a single Boolean expression. A transition condition which is always true shall be represented by the keyword TRUE.

The links reaching and leaving the steps are represented by vertical lines (see Figure 2). A transition condition can be associated with a transition by one of the following means:

- an appropriate Boolean expression in ST
- a ladder diagram network in LD whose output intersects the vertical directed link
- a network in the FBD whose output intersects the vertical directed link
- a LD or FBD network whose output intersects the vertical directed link via a connector

- a TRANSITION...END_TRANSITION construction using ST, consisting of the keywords TRANSITION FROM followed by the predecessor step(s) name, the keyword TO followed by the successor step(s) name, the assignment operator followed by a Boolean expression specifying the condition, the terminating keyword END_TRANSITION
- a TRANSITION...END_TRANSITION construction using IL, consisting of the keywords TRANSITION FROM, followed by the predecessor step(s) name and by a colon, the keyword TO followed by the successor step(s) name, a list of instructions in the IL language determining the transition condition, the terminating keyword END_TRANSITION
- a transition name associated to the directed link, referred to a TRANSITION...END_TRANSITION construction, whose evaluation results in the assignment of a Boolean value to the variable denoted by the transition name and whose body is a network in the LD or FBD language, or a list of instructions in the IL language, or an assignment of a Boolean expression in the ST language.



```
TRANSITION Tij:
(* transition condition *)
END_TRANSITION
```

Figure 2 **Graphical and textual representation of a transition**

The scope of a transition name is local to the POU where the transition is located. No “side effects” (for instance the assignment of a value to a variable other than the transition name) can occur during the evaluation of a transition condition.

2.2 Actions

Zero, one or more actions may be associated with each step. A step with no associated actions is providing a WAIT function, which is, waiting for the following transition condition to become true. An action declaration consists of the action name (of type string) and the action body. The action body can be a Boolean variable, a collection of instructions in IL, a collection of statements in ST, a collection of rungs in LD, a collection of networks in FBD or a sequential function chart (SFC) in its turn. Actions are declared and associated with steps via textual step bodies or graphical action blocks. The scope of the declaration of an action is local to the POU containing the declaration.

IEC 61131-3 3rd edition:

Table 60 – Action control features

No. Description Reference

1 With final scan per Figure 22 a) and Figure 23 a)

2 Without final scan per Figure 22 b) and Figure 23 b)

These two features are mutually exclusive, i.e., only one of the two shall be supported in a given SFC implementation.

2.3 Qualifiers

The control of actions is defined by action qualifiers. Possible action qualifiers are those listed in the following table. Qualifiers specify the execution of actions at every execution cycle in relation to the states of their associated steps. Actions using the qualifier N or none are executed as long as their associated steps are active. The L, D, SD, DS, and SL qualifiers require an additional associated parameter of type TIME for the delay or the limitation. See 3.5 Qualifiers for further explanations of action qualifiers.

Qualifier	Definition	Effect on the action
None	Null qualifier	executes while the associated step is active
N	Non-stored	executes while the associated step is active
P	Pulse	executes when the associated step is activated
S	Set	executes until the related R qualifier is met
R	Reset	terminates the set (S) execution
L	time Limited	ends the execution after a given time
D	time Delayed	starts the execution after a given time
SD	Stored and time Delayed	starts set execution after a given time
DS	time Delayed and Stored	starts set execution if the step lasts a given time
SL	Stored and time Limited	starts set execution and ends after a given time

2.4 SFC evolution rules

The initial situation of an SFC network is characterized by the initial step which is in the active state upon initialization of the program or function block containing the network. The processing order of the active states of steps takes place along the directed links when caused by the clearing of one or more transitions. A transition is enabled when all the preceding steps, connected to the corresponding transition symbol by directed links, are active. The execution of a transition occurs when the transition is enabled and when the associated transition condition is true.

The clearing of a transition causes the deactivation of all directly preceding steps connected to the corresponding transition symbol by directed links, followed by the activation of all the directly following steps. The clearing time of a transition may theoretically be considered as short as one may wish but it can never be zero. In practice the clearing time will be imposed by the programmable controller implementation. For the same reason the duration of a step activity can never be considered to be zero. Several transitions which can be cleared simultaneously shall be actually cleared within the timing constraints of the particular programmable controller. Testing of the successor transition condition(s) of an active step is performed until the effects of the step activation have propagated throughout the POU where the step is declared.

2.5 Divergence and convergence

Divergence is a multiple connection link from one SFC symbol (step or transition) to two or more SFC symbols of the opposite type. Convergence is a multiple connection link from more than one SFC symbol of the same type to one symbol of the opposite type. Divergence and convergence can be alternate or parallel.

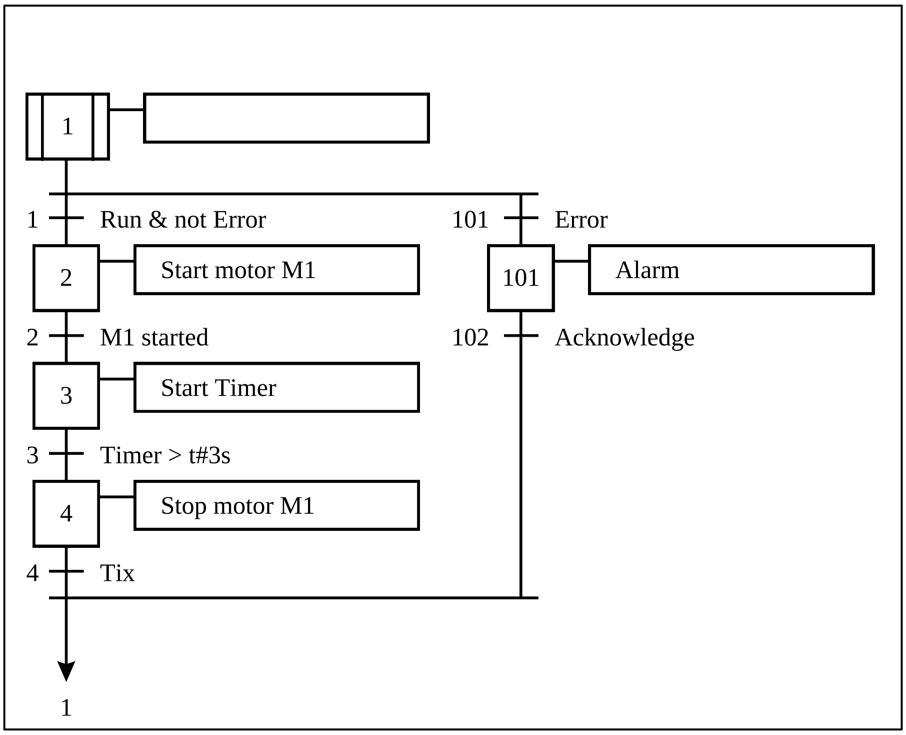


Figure 3 **SFC schema with divergence and convergence**

An alternate divergence is a multiple link from one step to many transitions. In the figure above only one transition becomes TRUE because they are exclusive (see also 4.3 Mutually exclusive transition conditions). Nevertheless, conditions attached to the different transitions at the beginning of a single divergence are not necessarily exclusive, thus exclusivity has to be ensured by fixing a priority among conflicting transitions (or by default, depending on the single implementation) so that only one transition is cleared. A convergence from an alternate sequence is a multiple link from many transitions to the same step. Such a convergence is generally used to group the SFC branches which were started on a single divergence.

Alternate divergence and convergence are represented by single horizontal lines, as in the example of Figure 3. A sequence skip is a special case of an alternate divergence where one or more of the branches contain no steps. A sequence loop is a special case of simple divergence and convergence where one or more of the branches return to a preceding step.

A parallel divergence (or simultaneous divergence) is a multiple link from one transition to many steps. It corresponds to parallel operations of the process, also called simultaneous sequences. The parallel divergence is executed when the preceding SFC step is active and the transition condition becomes true. After the divergence, all the simultaneous sequences have activated their first steps. A parallel (or simultaneous) convergence is a multiple link from many steps to the same transition. It is generally used to group the SFC branches started on a double divergence. The parallel convergence is executed when all the simultaneous steps preceding it are active and the following transition condition is true. After the convergence, the preceding active steps are deactivated and a single SFC schema step results again active.

Parallel divergence and convergence are represented by double horizontal lines as in the example of Figure 4. Criteria for a correct use of simultaneous sequences are proposed in 4.3 Mutually exclusive transition conditions, together with the analysis of the most frequent modelling errors.

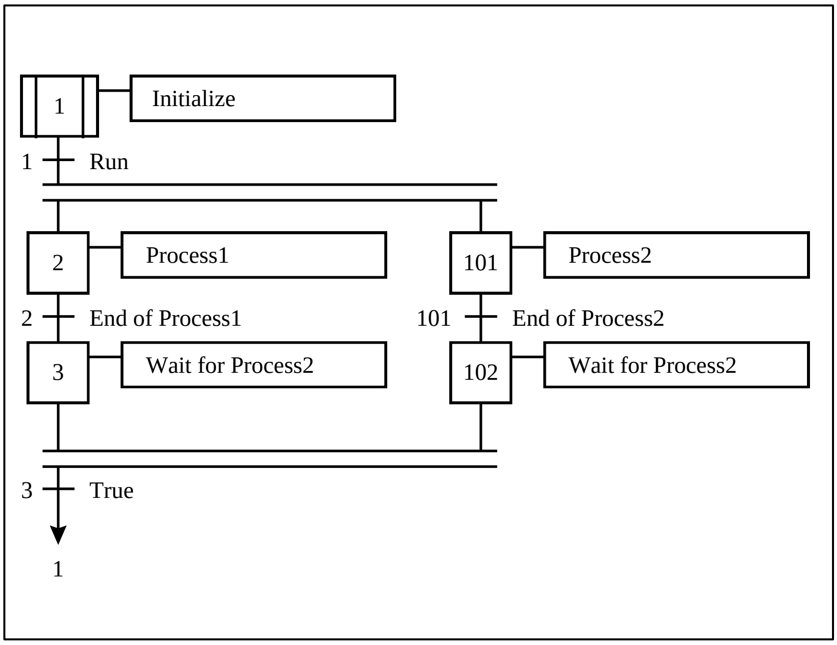
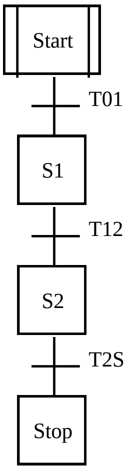


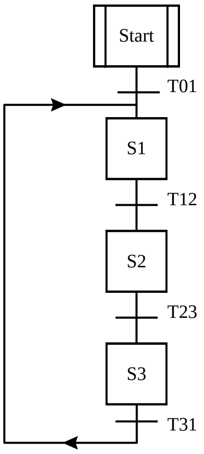
Figure 4 **SFC scheme with parallel divergence and convergence**

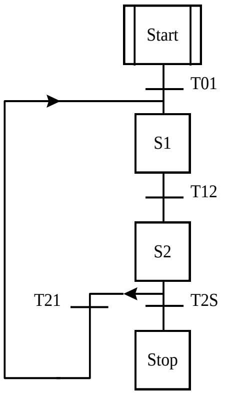
3 Structural properties of SFCs

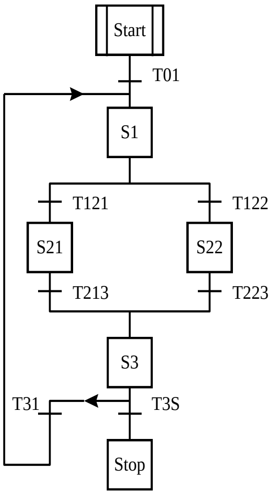
3.1 Process Structure

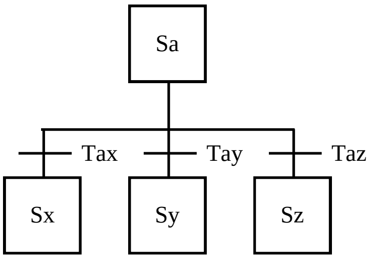
The simplest structure of a control program is that constituted by a simple sequence of states as in the example shown in hereunder. The initial step, represented by the double bar rectangle, is the one activated by the turn on of the system under control. Then, the other steps reachable by the system are visited in the given order following the diagram from top down.

Sequence of steps 	An SFC program in its simplest form: “one shot” behaviour. The system moves from one step to the other as soon as the triggering condition associated to the transition between the two states is verified. More precisely, when the condition associated to T01 becomes true, the system moves from the Start step, which consequently becomes inactive, to the S1 step, which becomes active. The same occurs for T12, closing S1 and starting S2, and for T2S that brings the system from S2 to the Stop step. Since there are no leaving paths from the final step, the system remains in that state until a warm or cold start happens.

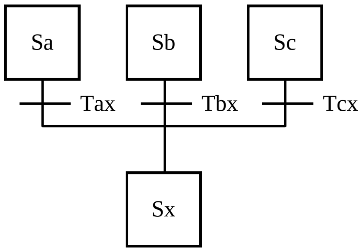
Cyclic sequence of steps 	The most common situation is that realising a cyclic execution of the control. In the scheme here we imagine to return to the S1 step after having reached S3 through S2. In this case we still have an operator turning on the system in step Start, but then the system keeps on running cyclically through steps S1, S2 and S3 until an external (and therefore not represented) event stops it. One can interpret this on-going cycle as part of an SFC scheme with an action associated to a step belonging to an upper level SFC scheme. This action can be stopped (that is, no more executed) by simply closing (making inactive) the upper level step it is associated to.

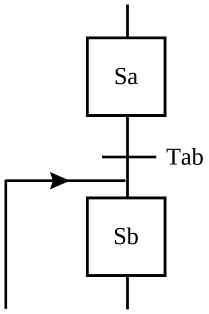
Cyclic sequence of steps with controlled termination  	In the scheme to the left a cyclic behaviour with a controlled exit condition is represented. This condition is associated to the transition T2S bringing from step S2 to step Stop, as an alternative to transition T21 bringing back the system to S1 from S2. In this situation a smooth conclusion of the execution is introduced. In practice, the PLC software includes the functions which are required to keep under control the system shut down sequence. This can represent for example, the behaviour of a machine tool which is turned on (Start), take a piece (S1), process the piece (S2) and then take another piece to process. The machine has a command that can stop it after the processing of the last piece has been terminated, but it cannot move directly from S1 to Stop.

Cyclic sequence of steps with divergence  Start T01 S1 T121 T122 S21 S22 T213 T223 S3 T31 T3S Stop	Divergence corresponds to operative situations where the behaviour of the controlled system should differ according to the conditions that occurred in conclusion of the step immediately preceding the divergence. The figure on the left shows a divergence after step S1 to either steps S21 or S22. After them, the system converges towards step S3 before reaching the final choice (again a divergence) of coming back to S1 or exiting to the Stop step. Concerning transitions, we observe that conditions T121 and T122 should be disjoined to determine unambiguously the choice between the two alternative steps S21 and S22. On the contrary, conditions T213 and T223, leaving respectively S21 and S22, are independent of each other: they are individually related to the steps they leave, thus, in general, each of them has no relation with the step which has not been chosen at the divergence.

Divergence and transition conditions  Divergence of sequence	This scheme shows a divergence of sequence case: it is equivalent to the representation of the three transitions linking step Sa respectively to steps Sx, Sy and Sz. Since the leaving transitions from Sa are three, there are three triggering conditions Tax, Tay and Taz, with necessary disjunction to obtain a deterministic algorithm.

Convergence and transition conditions	On the left a convergence case: it is equivalent to the

 Convergence of sequence	representation of the three transitions that link step Sx respectively to steps Sa, Sb and Sc. Also in this case there are as many triggering conditions as the number of transitions.

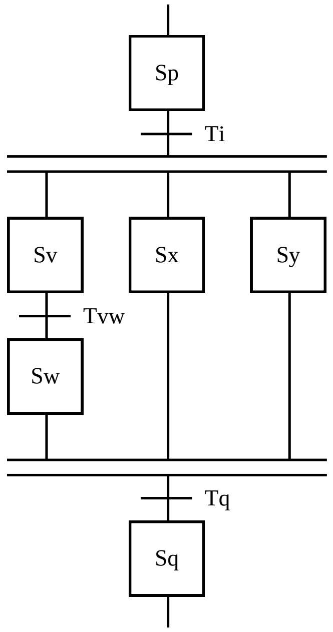
Sequence loop and transition conditions  Sequence loop	In the figure on the left, with the ending of a sequence loop, gives the representation of the (implicit) convergence, where a triggering condition is associated to each transition.

3.2 Simultaneous sequences

The SFC language allows the definition of parallel control sequences, that is, sequences executed contemporarily so that two or more steps are active at the same time. The parallel sequences are activated and deactivated by the simultaneous divergence and convergence constructs. As brief summary:

- The activation of the parallel sequences following a simultaneous divergence occurs when the previous step is active and the condition enabling the entrance transition is true. The simultaneously diverging steps, that is, the initial steps of the involved (parallel) sequence, are all activated and the previous step is deactivated.
- The parallel sequences preceding a simultaneous convergence are closed together when the final steps of all them are active and the condition associated to the leaving transition is true. The simultaneously converging steps are all deactivated and the following step becomes active.

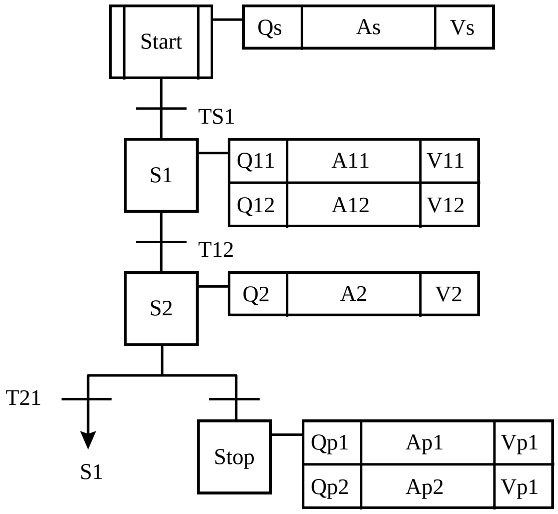
3.3 Parallel sequences

Parallel sequences 	In this example we have three parallel sequences formed, respectively, by steps Sv and Sw, step Sx and step Sy. If step Sp is active and the transition condition Ti becomes true, the simultaneous divergence activating such parallel sequences is applied. This means that step Sp is deactivated and the first steps of the parallel sequences, namely Sv, Sx and Sy, are activated. From this moment the three sequences proceed autonomously, apart from possible mutual influences caused by their actions and transitions. The divergence cannot end until the final states of the three parallel sequences, namely Sw, Sx and Sy, are all active (in this particular case steps Sx and Sy are always active while step Sw becomes active when the transition condition Tvw turns true). When finally the condition associated to Tq becomes true, the simultaneous convergence occurs, steps Sw, Sx and Sy are deactivated and step Sq is activated. The introduction of a simultaneous divergence means that, at a given time, more independent processes (possibly synchronized) may be activated in the described system. This behaviour can be seen as if the control was organised in levels. Whenever a simultaneous divergence is met, the system intended as a whole is put in a sort of state like "more active processes" and the control passes to the diverging partial processes. Each of them has its own local state, representing its evolution, and behaves in its turn as an automaton. The condition for leaving the simultaneous divergence synchronises the termination of the inner automata. Thus, the SFC scheme representing the whole system should be interpreted as a hierarchical state diagram. Consider that the hierarchical structure can be iterated, as parallel sequences may include further simultaneous divergence.

3.4 Action Blocks

3.4.1 Introduction

SFC scheme with action blocks



The complete description of a control program modelled as an SFC scheme requires making explicit the effects of step activation in terms of actions. Each effect is described by an action block associated to the step. The frequency of action execution is chosen by means of proper qualifiers. The example here shows an SFC scheme with four steps and six associated actions. If we refer to the generic i-th action, Qi represent the qualifier, Ai identifies the action, Vi is the name of an optional Boolean variable chosen to summarise the results of the action.

Some of the steps may have no associated actions. They represent waiting states, where no activity is performed. In such states the system simply waits for the occurrence of events, typically detected as variations of input variables, which turn true the exit condition

It is also allowed to associate the same action to more than one step, in such a way that it is executed in different phases of the control cycle. This possibility is often connected with a proper use of qualifiers.
Normally the action body is described aside, for obvious reasons of space and readability of the SFC scheme. However, if its text is reasonably short, it can be included into the action block. Examples of action blocks of this type are given below.
The action can be programmed using any of the standard languages.

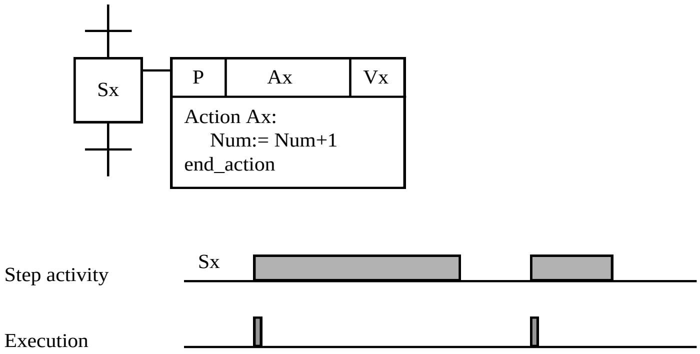
3.5 Qualifiers

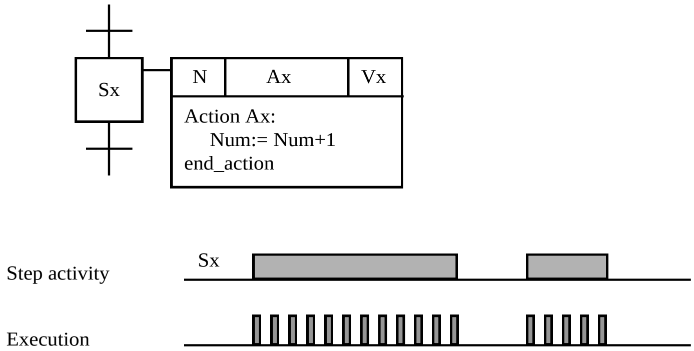
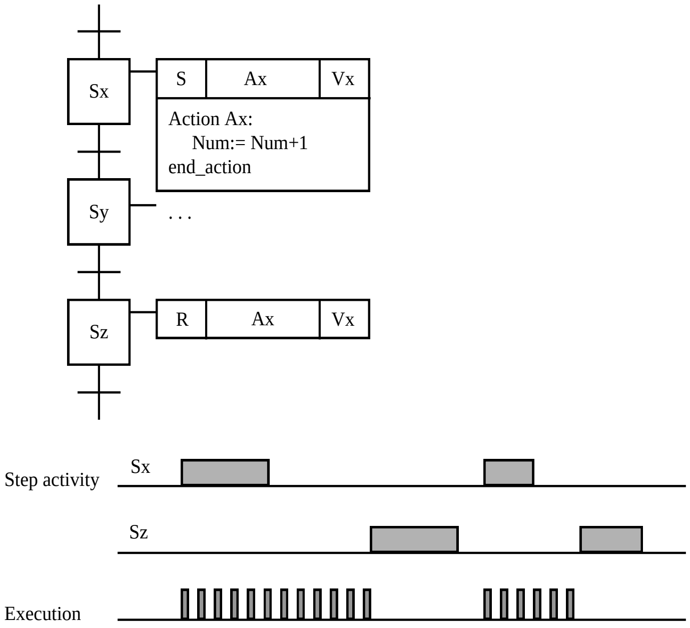
3.5.1 Introduction

The execution of an action block is related, through its qualifier, to the activation of the steps invoking it and the duration of their staying in an active condition. For this reason, two special variables are updated by implicit actions associated to all the steps. One is a Boolean variable, called step active flag (and denoted as stepname.X) which represents the activation state of the step: its value is 1 in every execution cycle where the step is active and 0 otherwise. The other is a TIME variable, representing the step elapsed time (and denoted as stepname.T): it is set to zero whenever the step becomes active. Both these variables may be tested in transition conditions and in actions associated to other active steps for synchronisation purposes. The standard refers to several qualifiers, representing alternative ways to associate actions to steps (Table 59 in IEC 61131-3):

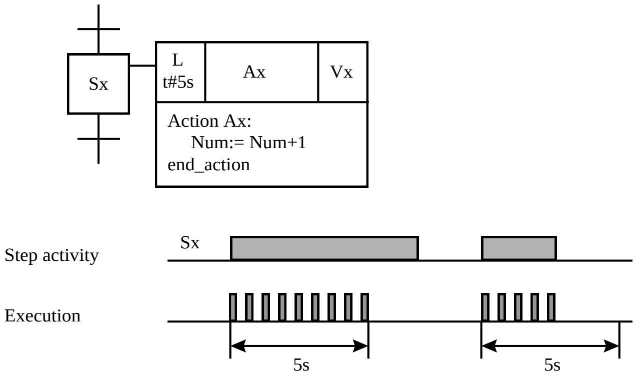
• None	Non-stored (null qualifier)
• N	**N**ormal, non-stored
• R	overriding **R**eset
• S	**S**et (Stored)
• L	time **L**imited
• D	time **D**elayed
• P	**P**ulse
• SD	**S**tored and time **D**elayed
• DS	**D**elayed and **S**tored
• SL	**S**tored and time **L**imited.
• P1	**P**ulse (rising edge)
• P0	**P**ulse (falling edge)

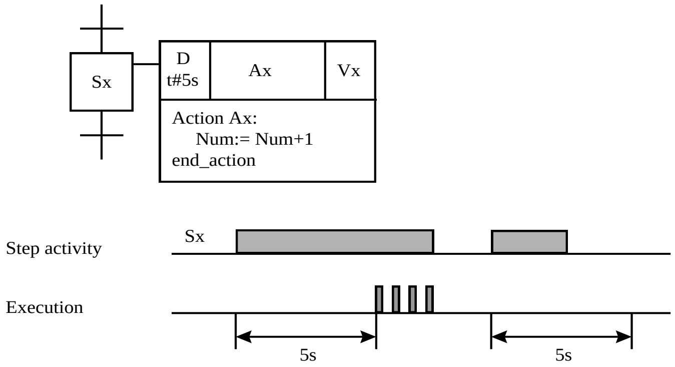
If the transition condition becomes TRUE there are 2 (implementation specific) options: with a final scan or without a final scan of the related ACTION block. For details see **Error! Reference source ot found.** Execution control.

Effect of the P qualifier  Sx P Ax Vx Action Ax: Num:= Num+1 end_action Step activity Sx Execution	The P (pulse) qualifier characterizes the actions that have to be executed only once, just when the step becomes active. The effect of a P qualifier is shown in the figure on the left, where the action associated to the step Sx increments the integer variable Num. The timing diagram shows the relation between step activity (i.e. the value of the Sx.X variable) and operations on the Num variable. According to the action definition, the Num value is incremented by one unit in correspondence of each rising edge of the Sx.X variable.
Effect of the N qualifier	The N (normal) qualifier characterises all the actions that have to be executed every time the execution cycle of the control program finds the associated steps active.

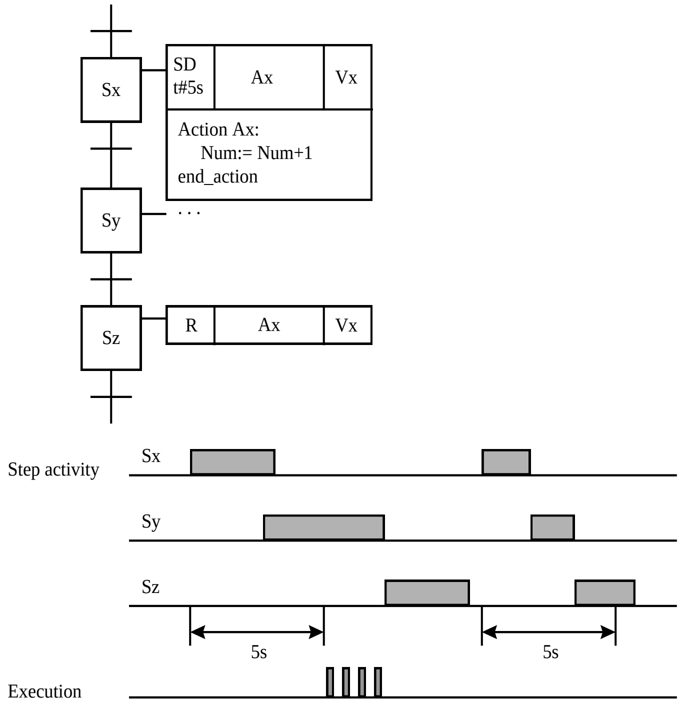
 Sx N Ax Vx Action Ax: Num:= Num+1 end_action Step activity Sx Execution	In other words, as soon as a step becomes active, all its N actions are executed and the execution is repeated in the following cycles until the step turns inactive. It may happen that, due to the particular execution mechanism adopted by the single PLC machine, the last execution of the N actions occurs immediately after the deactivation of the step. It is necessary that the programmer is aware of this behaviour to forecast precisely the operations that the code will execute. The figure presents the case of an action aimed at incrementing repeatedly the Num variable, once at every cycle while state Sx is active. As it can be seen from the diagram, the amount of the increment is proportional to the duration of the interval when the Sx.X variable stays true.
Effect of the S and R qualifiers  Sx S Ax Vx Action Ax: Num:= Num+1 end_action Sy . . . Sz R Ax Vx Step activity Sx Sz Execution	The S (set) and R (reset) qualifiers characterize the so-called stored actions. A stored action has to be executed at every cycle, from the moment when the step where it is qualified S becomes active to the moment when the step (usually different from the former) where it is qualified R becomes active. The figure shows a situation of this type. The S qualifier is on the action block associated to step Sx and the R qualifier is on the action block associated to step Sz (the intermediate steps make no reference to action Ax). The diagram describes this behavior relating the operations on the Num variable to the values of the Sx.X and Sz.X variables. It is worth observing that the same result can be obtained by associating the Ax action to all the steps from Sx to Sz (the last excluded) and qualifying them with N. More precisely this alternative solution requires that: • Sx has associated the action [N, Ax] • Sy has associated the action [N, Ax] (if other steps precede Sz, they also have associated the action [N, Ax]) • Sz does make no reference to action Ax.

<table>
<tr><td></td><td>Thus, the pair (S, R) is a way to shorten the SFC scheme, as it avoids duplicating the same action reference many times. Unfortunately, the resulting scheme is often less readable. If the distance between the S and R steps is considerable, or a divergence is put in between, the execution range of such an action becomes very difficult to understand. Besides, it may happen to define paths including the only S or R qualifiers. This causes serious problems during testing, maintenance and code upgrading.</td></tr>
<tr><td>Effect of the L qualifier


</td><td>The L (time Limited) qualifier is one of the five time-related qualifiers provided by the standard to delay or limit in time the execution of actions, according to specified time expressions. In particular, the L qualifier starts the execution of the relative action when the step becomes active, and keeps it running for a given time interval provided that the step remains active. In the figure on the left, action Ax execution should last 5 seconds or less. During the first period of activation of step Sx such limit is reached, while during the second period the step is deactivated before the time limit is reached.
To better understand the meaning of the L qualifier, it can be observed that the same result is obtained by qualifying the Ax action with N, and subordinating the increment of the Num variable to the value of the elapsed time Sx.T (depending on the implementation of the L qualifier, the relational operator could be "<=" or "<"):
<table>
<tr><th>N</th><th>Ax</th></tr>
<tr><td colspan="2">Action Ax:
if Sx.T <= t#5s then Num:=Num+1
end_if
end_action</td></tr>
</table></td></tr>
<tr><td>Effect of the D qualifier</td><td>The D (time Delayed) qualifier is also time-related, as it delays the beginning of action execution with respect to the step activation time. The figure on the left represents a situation where the Ax action</td></tr>
</table>

 Sx D t#5s Ax Vx Action Ax: Num:= Num+1 end_action Step activity Sx Execution 5s 5s	has to start 5 seconds after step Sx is activated. Since the first activity period of Sx is longer than 5 seconds the action can start, while in the second period this does not happen.

Effect of the N qualifier
See above
	To obtain the same time delayed result with a N qualifier it is sufficient to modify the Ax action so as to subordinate the increment of Num to the value of the elapsed time Sx.T (depending on the implementation of the D qualifier, the relational operator could be ">=" or ">"):
	N **Ax**
	Action Ax:
	if Sx.T >= t#5s then Num:=Num+1
	end_if
	end_action

Effect of the SD qualifier  Sx SD t#5s Ax Vx Action Ax: Num:= Num+1 end_action Sy . . . Sz R Ax Vx Step activity Sx Sy Sz 5s 5s Execution	The SD (Stored and time Delayed) qualifier, used in combination with the R (reset) qualifier, stores the action but delays its execution of a given time interval. Therefore, the action starting may occur when the step with the SD qualifier is no more active. More precisely, it may occur during the activation of any of the following steps, but before reaching that with the R qualifier. In the figure on the left, action Ax has to start 5 second after its storing. During the first execution of the sequence there is time enough to start the action (in particular, while Sy is active), but this does not hold during the second execution as the R qualifier is reached too soon. Note: To represent the same behaviour using only P and N qualifiers, it is necessary to remember that the starting of action Ax execution is conditioned by the elapsed time calculated from the activation time of step Sx. Unfortunately, the value of the Sx.T variable is not

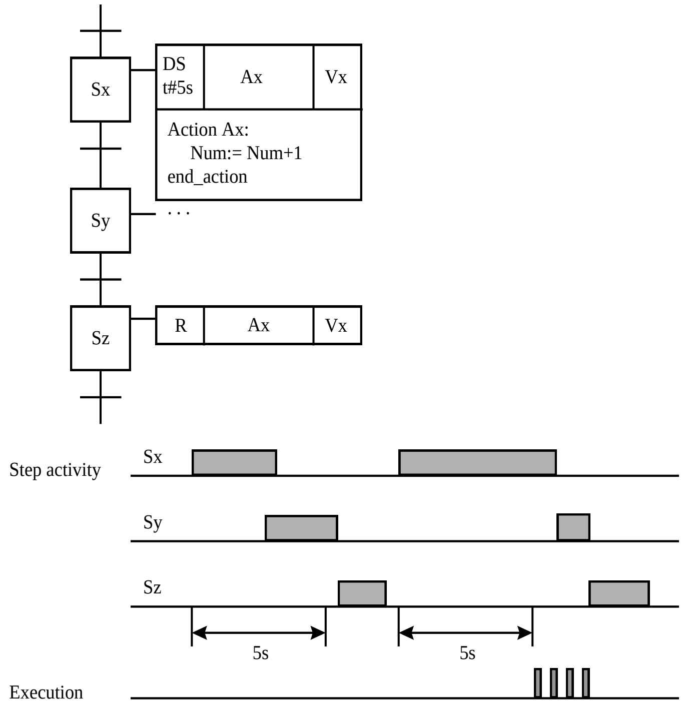
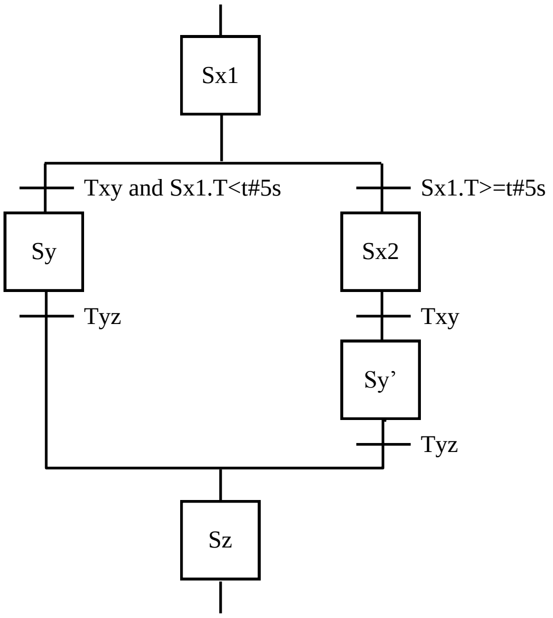
sufficient, as Ax can start in a moment when step Sx is no more active. Then, we need a timer (say, the time-on timer TONx) that should be initialised with a pulse action associated to the Sx step, like the following:

P	**Ax1**
Action Ax1: TONx (IN:=true, PT:=t#5s) end_action	

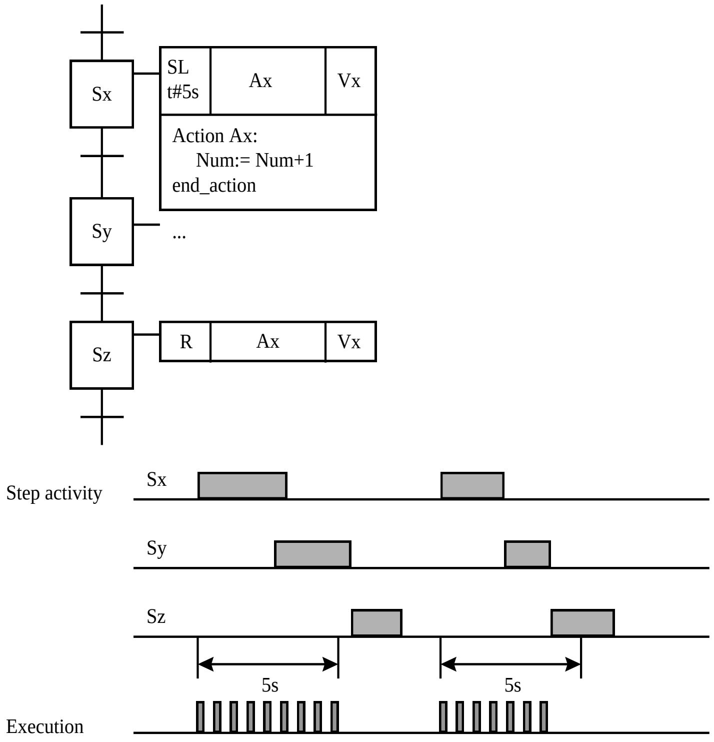
Now, each of the steps where the execution of Ax could start must have associated the conditioned version Ax2 of the action (once again, depending on the implementation of the SD qualifier, the relational operator could be ">=" or ">"):

N	**Ax2**

Action Ax2:
if TONx.Q then Num:=Num+1
end_if
end_action	

Effect of the DS qualifier  Sx DS t#5s Ax Vx Action Ax: Num:= Num+1 end_action Sy . . . Sz R Ax Vx Step activity Sx Sy Sz 5s 5s Execution	The DS (Delayed and Stored) qualifier, used in combination with the R (reset) qualifier, waits for a given time interval before storing the action. The action starts within the step with the SD qualifier, provided it stays active for a long enough time, or it does not. In the figure on the left, action Ax has to start 5 seconds after step Sx activation: such a temporal condition is satisfied during the second cycle, but not during the first one.
The same behaviour as DS but with only the P and N qualifiers  Sx1 Txy and Sx1.T<t#5s Sx1.T>=t#5s Sy Sx2 Tyz Txy Sy' Tyz Sz	Note: To get the same behaviour as DS using only the P and N qualifiers, it is necessary to split Sx into two steps, and use two versions of Sy (and of all the other steps possibly preceding Sz), as it is shown in the figure on the left (the relational operators depend on the particular implementation of the DS qualifier). The actions associated to the steps in this solution are so characterised: • Sx1 has associated all the P and N type actions of Sx, but not Ax • Sx2 has associated all the N type actions of Sx, and the action [N, Ax] • Sy remains as before (it has not associated the Ax action) • Sy' has associated the same actions as Sy, and the action [N, Ax] • (if other steps exist before Sz, they are distinguished as Sy and Sy') • Sz does not mention the Ax action.

Effect of the SL qualifier



Finally, the SL (Stored and time Limited) qualifier, used in combination with the R (reset) qualifier, stores the action and keeps it in execution for a given time interval. The ending of action execution may occur even if the step with the SL qualifier is no more active. More precisely, it may occur in each of the following steps, but before the step with the R qualifier is reached. In the figure on the left, the Ax action has to be executed for 5 seconds. During the first run of the sequence such a limit is reached, while during the second run the R qualifier is reached before.
Note: The representation of such a behaviour using only the P and N qualifiers is somehow close to that previously discussed for the SD qualifier. In fact, it is again necessary to measure the elapsed time from the activation of step Sx using a proper timer. Thus, the following pulse action must be associated to Sx:

P	**Ax1**
Action Ax1: TONx (IN:=true, PT:=t#5s) end_action	

Now, each of the steps where the action can stop must have associated the conditioned version Ax2 of the action (once again, depending on the implementation of the SL qualifier, the relational operator could be ">=" or ">"):

N	**Ax2**

Action Ax2:
if not TONx.Q then Num:=Num+1
end_if
end_action	

It is very important that the programmer puts much care in using the stored qualifiers, as they can cause effects difficult to see and control. The major problem with the stored qualifiers is their hidden association to the steps preceding the R declaration. A clear and strong principle should inspire the programmer activity: make coding as explicit as possible so as to obtain a more readable control software although that may be longer.

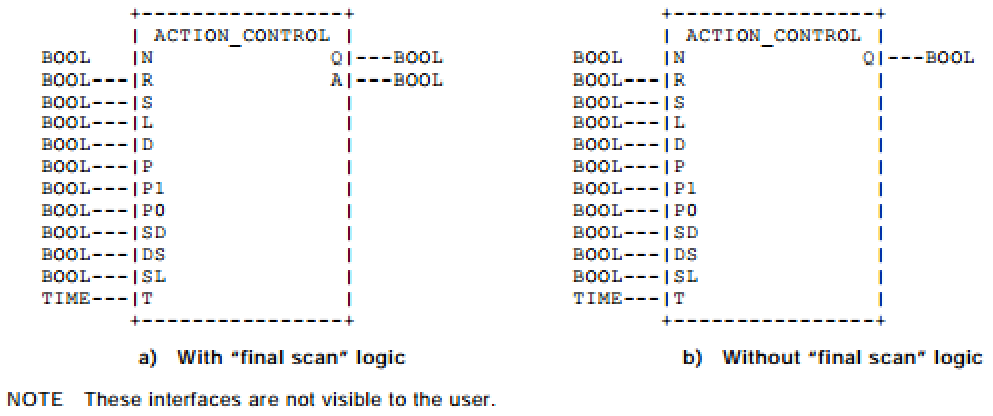
3.6 Execution control

Linked to the 3rd edition of the IEC 61131-3 standard section 6.7.4.6 Action Control.

3.6.1 ACTION_CONTROL function block

In order to understand the execution logic of an SFC it is necessary to keep in mind, that there is no direct link between the active state of a Step and the execution of an associated action.

Associated with each action is an implicit ACTION_CONTROL function block (or a functional equivalent), which is in control of the execution of the action. An active step sets an input of the ACTION_CONTROL block and the evaluation of the ACTION_CONTROL block controls whether or not the action is executed in the current cycle.



```
          +-----------------+                                   +-----------------+
          | ACTION_CONTROL  |                                   | ACTION_CONTROL  |
BOOL      |N               Q|---BOOL                  BOOL      |N               Q|---BOOL
BOOL------|R               A|---BOOL                  BOOL------|R                |
BOOL------|S                |                         BOOL------|S                |
BOOL------|L                |                         BOOL------|L                |
BOOL------|D                |                         BOOL------|D                |
BOOL------|P                |                         BOOL------|P                |
BOOL------|P1               |                         BOOL------|P1               |
BOOL------|P0               |                         BOOL------|P0               |
BOOL------|SD               |                         BOOL------|SD               |
BOOL------|DS               |                         BOOL------|DS               |
BOOL------|SL               |                         BOOL------|SL               |
TIME------|T                |                         TIME------|T                |
          +-----------------+                                   +-----------------+
```

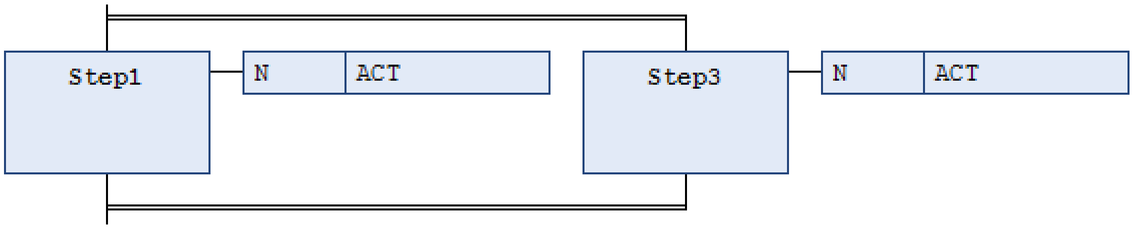
a) With "final scan" logic

b) Without "final scan" logic

NOTE These interfaces are not visible to the user.

This is important if an action is associated to different STEPs and even more, if the action is associated to these steps with different qualifiers.

For example, if an action is associated with two parallel steps like in the following figure, then the action will only be executed once in a cycle, even if both steps are active:



A typical execution of an SFC could be done in the following way:

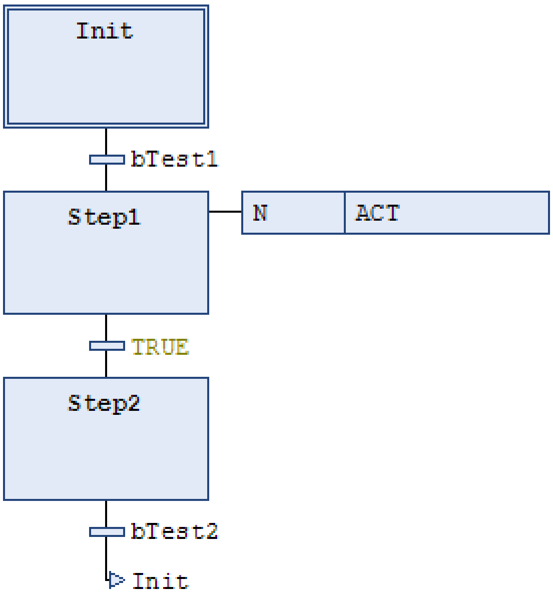
- all ACTION_CONTROL blocks of all actions are reset.
- the new active states of the steps are evaluated according to the old active states and the transition conditions
- the new active states write to the ACTION_CONTROL blocks of all actions

- the ACTION_CONTROL blocks are executed
- all actions are executed according to the result of the ACTION_CONTROL block (plus final scan, see next section).

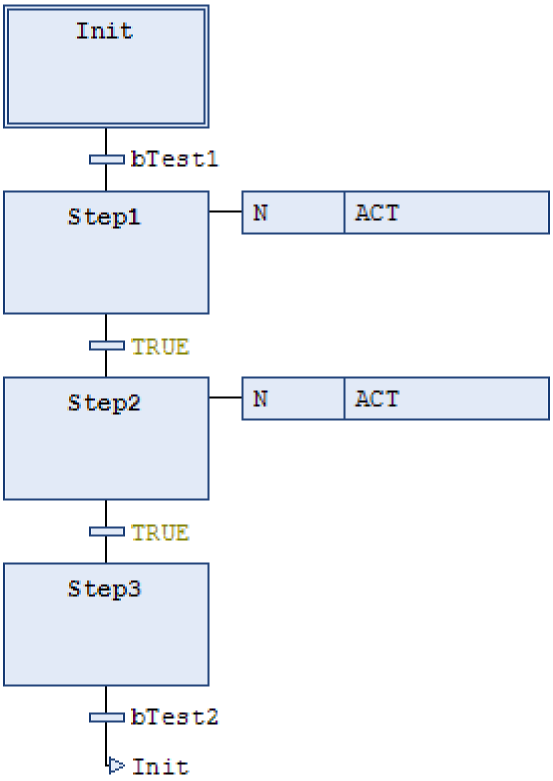
The specifics of this implementation may have significant impact on the semantics of the SFC.

3.6.2 Final Scan

The IEC-Standard defines a "final scan" as an implementation dependent feature of the Action. In simple words, the final scan means, that an Action is executed one more time after the transition condition becomes TRUE. If the associated ACTION_CONTROL block of an action produces a falling edge on its output, then the action is executed in this cycle.
This could be confusing, consider the following example:



If bTest1 is TRUE for one cycle, then ACT will be executed in the next two cycles, even if Step1 is active only in the next cycle.
Now once again it is important to remember, that the Action is not directly linked to the state of the Step, but to the state of the ACTION_CONTROL block.
Consider the following slightly different situation:



If bTest1 is TRUE for one cycle, then ACT will be executed in the next three cycles, there is no final scan between the active states of Step1 and Step2, since the associated ACTION_CONTROL block does not detect a falling edge.

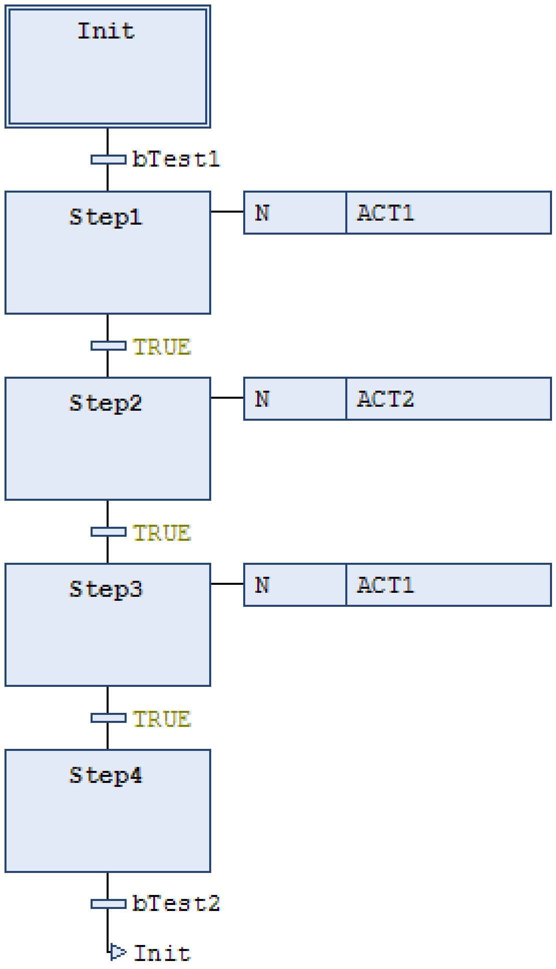
If an action is associated exactly to one step, and if it is associated with the N-qualifier, then the final scan can be identified by checking the Step-Flag of the associated step. The execution is in the final scan, if the Step is not active.

In the upper example, the Action ACT could be implemented in the following way:

```
ACTION ACT
    IF Step2.x OR Step1.x THEN
        // do cyclic things;
        ;
    ELSE
        // do final things;
        ;
    END_IF
END_ACTION
```

Some implementations of the SFC will offer in addition to the Step flags (and as an extension to the standard) Action flags that allow access to the state of the ACTION_CONTROL Block. Then the final scan can be determined independent from any step flags.

Due to the final scan, it is a common situation, that two actions are executed in the same cycle, even if there is no parallel branch in the graph. To discuss the problems that may arise due to the final scan, consider the following SFC graph:



The two Actions ACT1 and ACT2 shall have the following implementations:

```
ACTION ACT1
    x := x + 1;
END_ACTION

ACTION ACT2
    y := x;
END_ACTION
```

The following table shows the states of the upper Graph in the following cycles after bTest1 has a rising edge:

Cycle	*Step1*	*Step2*	*Step3*	*ACT1*	*ACT2*	*x*	*y*
1	Active	Inactive	Inactive	Executed	Not Executed	1	0
2	Inactive	Active	Inactive	Executed (final)	Executed	2	1 / 2
3	Inactive	Inactive	Active	Executed	Executed (final)	3	2 / 3
4	Inactive	Inactive	Inactive	Executed (final)	Not Executed	4	2 / 3

Thus, in cycle 2 and 3, both actions get executed, which may be unexpected. Note furthermore, that the standard does not require any specific order of execution for the actions. An order of execution according to the topology in the graph may be expected, but is not possible, since the action may be associated to different steps at several locations in the graph. ACT1 is above *and* below ACT2 in the same graph.

Therefore, after cycle 2 and 3 the value of y may be equal to x or equal to x-1, depending on the implementation of the SFC, and depending on the order of execution of the actions. If ACT2 is always executed after ACT1, the sequence of values for y would be 0, 2, 3, 3. If ACT2 is always executed before ACT1, the sequence of values for y would be 0, 1, 2, 2.

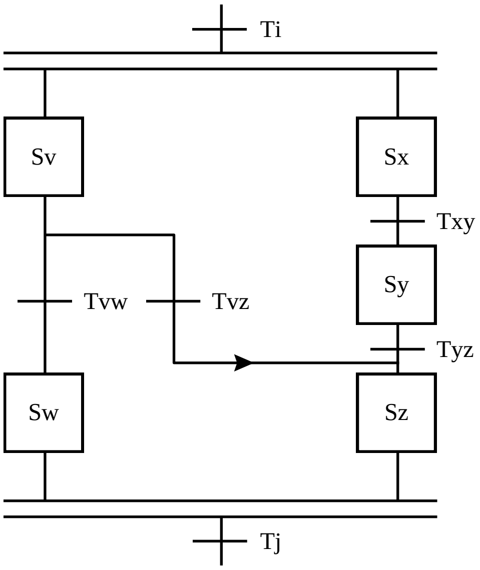
Some implementations of the SFC may execute all final scans before executing all other actions, but this is an implementation dependent feature and not required by the standard. In this case the order of execution of the two actions would be different in cycle 2 and 3. In cycle 2 ACT1 would be executed first, in cycle 3 ACT2 would be executed first. In this case, the sequence of values for y would be 0, 2, 2, 2.

4 Coding rules SFC

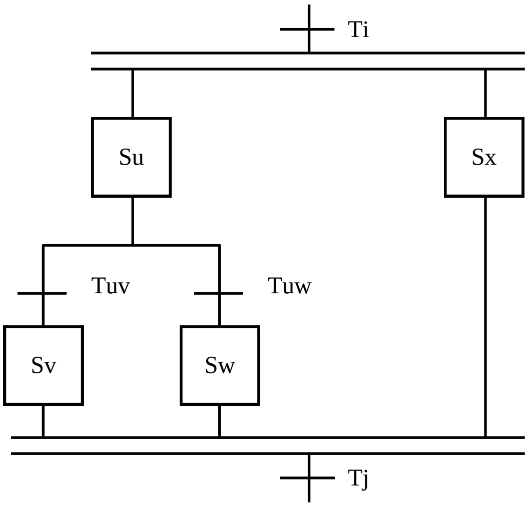
This chapter shows constraints for the usage of SFC.

4.1 Convergence and Divergence do's and don'ts

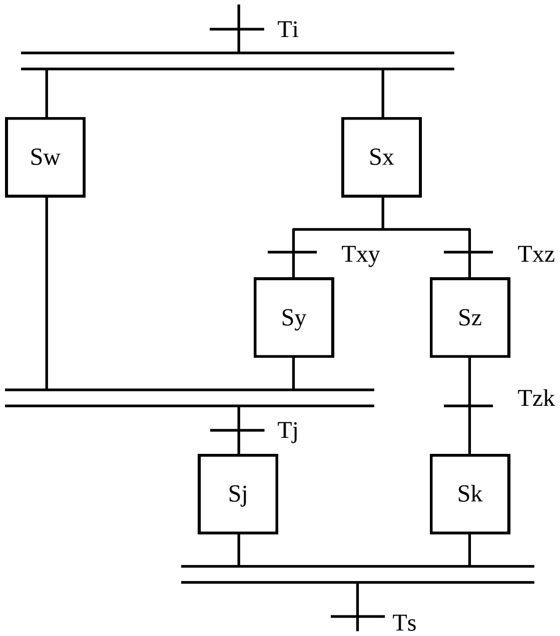
Parallel processes must be kept clearly distinct and separated

Wrong SFC scheme	Parallel processes must be kept clearly distinct and separated.
 Ti Sv Sx Txy Sy Tvw Tvz Tyz Sw Sz Tj	Therefor it is not allowed to link two parallel sequences. If this is done, as in the wrong example on the left, a block of the system control may occur. In fact, the simultaneous convergence is applied when the entering steps Sw and Sz are both active and the condition Tj controlling the leaving transition is true. Unfortunately, if the transition Tvz between the two sequences is activated, the Sw state will never become active and the control cannot leave this part of the SFC scheme. Also this situation may determine a variation in the number of the simultaneously active steps. At the divergence time two are the activated steps: Sv and Sx. If state Sv is left through the transition Tvw, the number of active steps does not change. Instead, if the transition Tvz is chosen, it is possible to obtain that two of the steps on the right-hand sequence are contemporarily active (Sx and Sz or Sy and Sz) or even that the number of globally active steps decreases to one (the only Sz step). The result is very confused and extremely difficult to interpret.

Every parallel sequence must have only one initial step and only one final step.

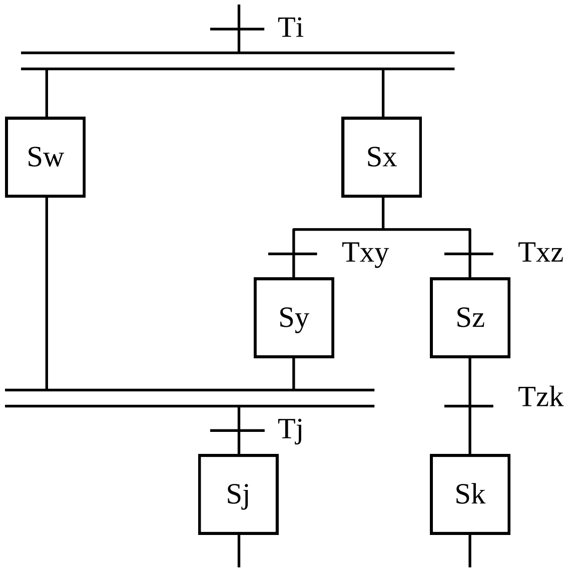
Wrong SFC scheme  	Every parallel sequence must have only one initial step and only one final step. As example shown on the left a wrong situation is when one of the parallel sequences included between a simultaneous divergence and the following convergence presents two or more final steps. Since only one step of such sequence is active at a time, the final steps cannot be all contemporarily active. It means that, in the example, the final steps Sv and Sw of the left-hand sequence are never both active, therefore the simultaneous convergence will never operate.

A sequence leaving a simultaneous divergence should never be directed towards two or more simultaneous convergence.

Wrong SFC scheme  Ti Sw Sx Txy Txz Sy Sz Tzk Tj Sj Sk Ts	A sequence leaving a simultaneous divergence should never be directed towards two or more simultaneous convergence. Combining a simultaneous convergence and divergence with a simple divergence, as in the example on the left, can introduce unreachable steps. The problem does not arise from having nested parallel sequences, as this is a natural use of the SFC language. Rather, the mistake is caused by the fact that the nesting does not respect the established rules. In the example, the simple divergence forces to choose between the execution of transitions Txy and Txz, so that either step Sy or step Sk can finally be activated, but not both. This means that one simultaneous convergence is never reached: the consequence is again a blocking of the control function.

A sequence leaving a simultaneous divergence can stop only at a simultaneous convergence.

Wrong SFC scheme

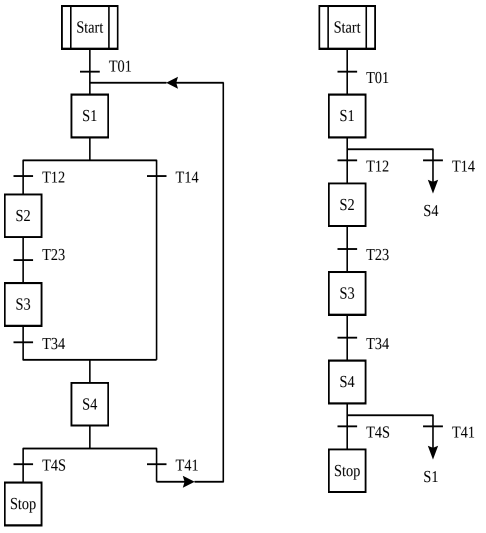


A sequence leaving a simultaneous divergence can stop only at a simultaneous convergence. This is a confused situation which can forget some active steps and is difficult to be interpreted. With reference to the example on the left, the step Sj is correctly reached if, at the branching between Txy and Txz, the former is chosen. In the other case, Sj will never be reached and, what is worse, the step Sw will stay active forever.

4.2 Linearization in SFC

In SFC schemes there are no arrows indicating the orientation of the transition. The implicit assumption is that the scheme is organized according to a sequential, descending logic. This means that each transition leaves the upper step to reach the lower one. The only exception is represented by the sequence loop that normally goes from bottom up.

Linearization of an SFC scheme



(a) regular SFC syntax (b) use of the *jump* primitive

As an extension to SFC some programming systems support the use of the *JumpStep* to stress the sequential structure of SFCs. The jump symbol (downward arrow) follows the transition condition and includes the indication of the destination state of the transition itself.
The figure (a) shows an SFC scheme with a sequence skip and a sequence loop, and (b) its simplification obtained by the introduction of two jumps. The use of jumps can help to improve the scheme readability. (Note: As SFC lines are not allowed to cross each other, sometimes JumpStep is the only alternative to express the intended flow. JumpStep is an extension to IEC 61131-3.).

4.3 Mutually exclusive transition conditions

In SFC schemes the transitions leaving a given step are activated by distinct enabling condition. It is advised that there is no overlap in the transition conditions, e.g. no ambiguity remains on the next step to be activated. The definition of overlapping could be either the risky choice of a good programmer or, more likely, the consequence of a mistake of a bad programmer. To avoid misunderstandings it is strongly recommended to use always a deterministic approach.

The IEC 61131-3 standard is deterministic: it ensures that in all cases only one branch in a divergence of sequence can become active. In absence of additional information it is implicitly assumed that the branch to activate is the leftmost one (among those which present in that moment a true condition). Otherwise, the programmer can assign an explicit priority value (the lower this value, the higher the priority) to the branches with overlapping conditions.

The IEC 61131 standard allows the overlapping of conditions associated to transitions leaving the same state. Nevertheless, even if more than one condition could be true at the same time, only one is executed and consequently we have only one step activated next. When the triggering event cannot determine alone the step swap, further information is required to decide. The SFC syntax suggests that the choice should be made according to a priority parameter, implicitly or explicitly defined.

In absence of additional information it is implicitly assumed that the branch to activate is the leftmost one (among those which present in that moment a true condition). Otherwise, the programmer can assign an explicit priority value (the lower this value, the higher the priority) to the branches with overlapping conditions.

This idea of priority is still ambiguous and can cause some mistakes. First of all, the word priority is misleading in itself as it could induce to think that all the overlapping transitions will be executed in the order given by the established priorities. Besides, the risk of mistakes is very high when the overlapping transition conditions affect many diverging branches, and various priority relations are defined on them. These are the reasons why many authors strongly suggest, as we do, to use mutually exclusive transition conditions only.

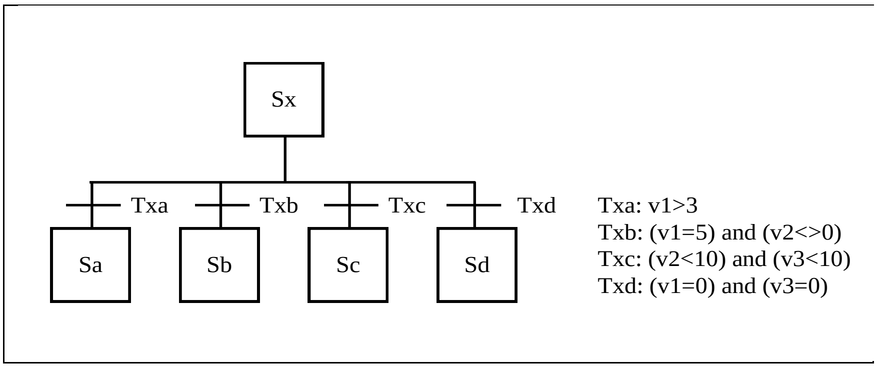


Figure 5 **Example for overlapping transition conditions**

For example, consider a control structure close to that shown in Figure 6 and suppose to establish the following priority relations (Tij denotes the condition of the transition leaving step Si and reaching step Sj):

```
Txa precedes Txb e Txc
Txb precedes Txc
Txb precedes Txd
```

This situation is conveniently represented by means of an oriented graph, with nodes corresponding to the transition conditions and branches starting from the lower priority node and ending to the higher priority node (see Figure 6a). If we move along the graph starting from the lowest priority nodes (Txd in this case) it is easy to assign the correct numerical values to priorities:

```
priority (Txd) = 4
priority (Txb) = 3
priority (Txc) = 2
priority (Txa) = 1
```

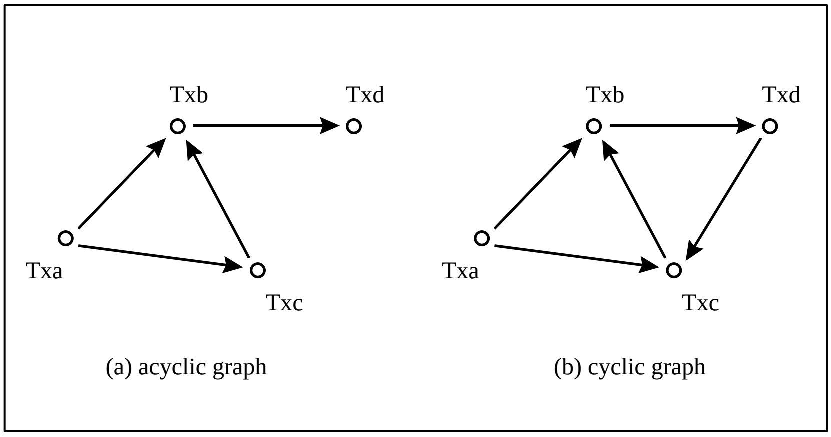


Figure 6 **Graph of transition precedence**

The oriented graph technique is particularly useful to find out those cases where priority numerical values cannot be used. The graph represented in Figure 6a is not cyclic, in that it is not possible to start from one node and reach the same node moving along the oriented branches. Obtaining an acyclic graph is a necessary and sufficient condition to assign numerical values to priorities. Instead, Figure 6b shows a cyclic graph, derived from the previous one by adding the further relation:

```
Txd precedes Txc
```

This condition generates the cycle involving nodes Txb, Txc e Txd. If the graph resulting from the given relations is cyclic it is not possible to define a set of priority values satisfying all the conditions. In that case, the only thing to do is modifying the transition conditions in such a way to remove overlaps. Better, let us see how to achieve the recommended solution constituted by mutually exclusive transition conditions starting from a non-deterministic SFC scheme.

We examine first a case with only two overlapping conditions Tx e Ty. Figure 7a shows, in form of a Venn diagram, the truth domains of the two conditions, where the common part is indicated as Txy. We can say that the Txy domain corresponds to the variable values that force true both conditions. If we want to disjoint the two conditions without losing the priority of, say, Ty over Tx (i.e. Ty precedes Tx), we have to redefine the truth domains as follows:

```
    domain (Tx') = domain (Tx) - domain (Txy)
    domain (Ty') = domain (Ty)
```

The result is shown in Figure 7. It is obtained by simply rewriting the Tx predicate:

```
    Tx' = Tx and not Ty
```

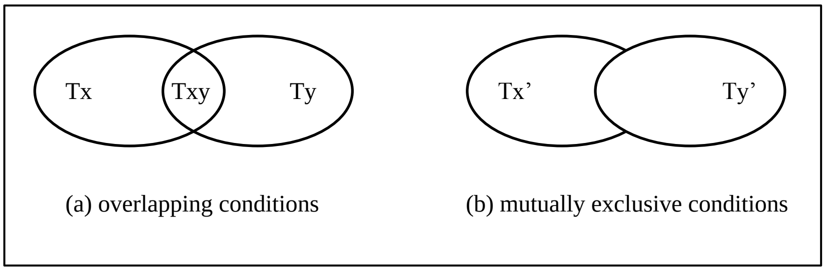


Figure 7 **Disjoining the truth domains of two overlapping conditions**

Cases with three or more overlapping conditions are solved in the same way, although their complexity increases very rapidly with the number of conflicting transitions. Figure 8a shows the Venn diagram for three non-exclusive conditions. In general, there are four overlapping domains, namely Txy, Txz, Tyz and Txyz. In order to ensure the following priorities:

```
    Ty precedes Tx and Tz
    Tx precedes Tz
```

and, nevertheless, remove the ambiguities it is necessary to redefine the transition domains according to the following expressions:

```
    domain (Tx') = domain (Tx) - domain (Txy) - domain (Txyz)
    domain (Ty') = domain (Ty)
    domain (Tz') = domain (Tz) - domain (Txz) - domain (Tyz) -
    domain (Txyz).
```

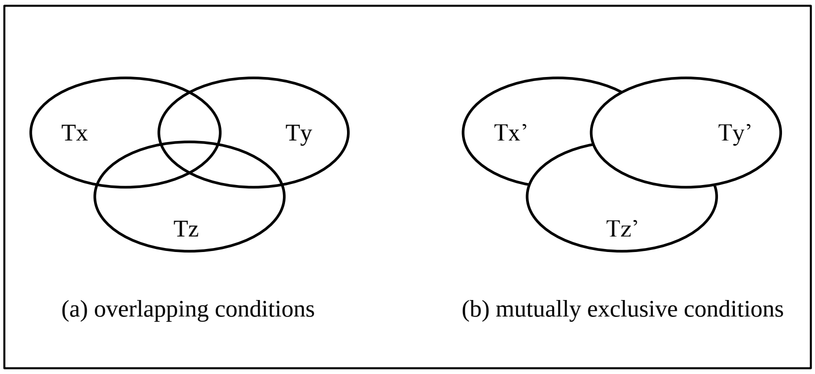


Figure 8 **Disjoining the truth domains of three overlapping conditions**

The result is shown in Figure 8b. In terms of transition conditions, the Tx and Tz predicates are thus rewritten:

```
    Tx' = Tx and not Ty
    Tz' = Tz and not (Tx or Ty
```

For the sake of completeness, let us consider again the non-deterministic SFC scheme reported in Figure 8. In order to make it deterministic and taking into account the assigned priorities reported on the scheme branches, the transition conditions must be modified as follows:

```
Txa' = Txa and not Txb
Txb' = Txb
Txc' = Txc and not (Txa or Txb or Txd)
Txd' = Txd and not (Txa or Txb)
```

that is:

```
Txa' =    (V1 > 3) and not ((V1 = 5) and (V2 <> 0))
Txb' =    (V1 = 5) and (V2 <> 0)
Txc' =    ((V2 < 10) or (V3 < 10)) and not ((V1 > 3) or
          ((V1 =5) and (V2 <> 0)) or ((V1 =0) and (V3 =0)))
Txd' =    (V1 = 0) and (V3 = 0) and not ((V1 > 3) or
          ((V1 = 5) and (V2 <> 0)))
```

4.4 Do not use priorities for the different transitions

Avoid using priorities for the different transitions. The risk of mistakes is very high when the overlapping transition conditions affect many diverging branches, and various priority relations are defined on them. These are the reasons why many authors strongly suggest, as we do, to use mutually exclusive transition conditions only.

4.5 Dependence on the previous state

The need for the actions associated to a given step to behave differently depending on which step was active before the current one (or, if we prefer, on the transition that led to the current step).

Up to now we have seen examples of actions to execute at the activation of a given step, or during its activity, or after a given time interval from its activation, or for a given interval of time. We have not yet considered the possibility that the action presents different behaviours according to which was the active step before the current one. This case is quite frequent.

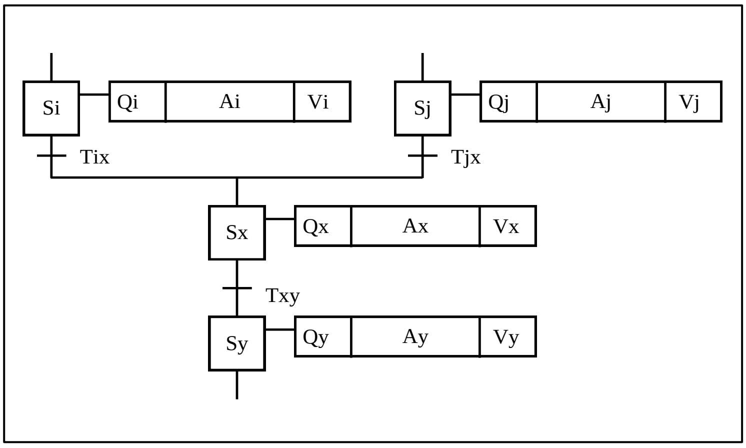
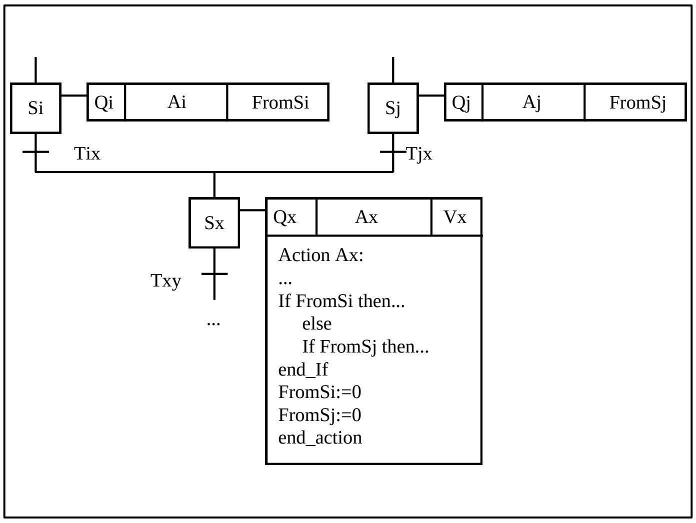


Figure 9 **Action dependence on previous steps, original scheme**

Figure 9 shows a hypothetical case of dependence from the preceding state. In fact, we can assume that in the text of action Ax, associated to step Sx, it is required to differentiate the code in consideration of the fact that the control comes from either step Si or step Sj.
If the differentiation is minimal, in that the most part of the code is shared in both cases, the easiest solution is that shown in Figure 10. The coming from steps Si or Sj is marked by the Boolean variables FromSi and FromSj which are turned true respectively from the actions Ai and Aj (both of them have been previously reset). So, the code of action Ax can be properly conditioned.



```
Action Ax:
...
If FromSi then...
    else
    If FromSj then...
end_If
FromSi:=0
FromSj:=0
end_action
```

Figure 10 **Action dependence using step variables**

If, on the contrary, the behaviour of Ax in the two cases is quite different, the best solution is that shown in Figure 11, where step Sx is split into two steps, namely Sxi and Sxj. The Axi and Axj actions are the versions of Ax for the left and the right branches, respectively. Note that this solution presents the absolutely general advantage of rendering explicit the dependence of the Sx behaviour from the reaching path. In this way the distinction appears clear simply examining the graph topology, while in the case of Figure 10 it can be seen only when going through the action code.

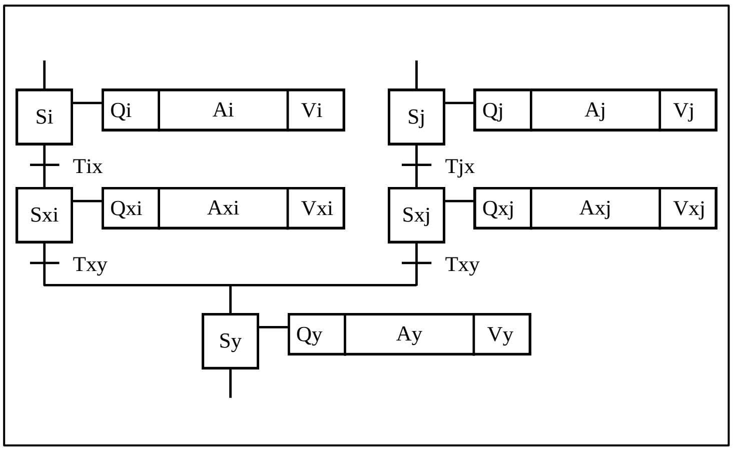


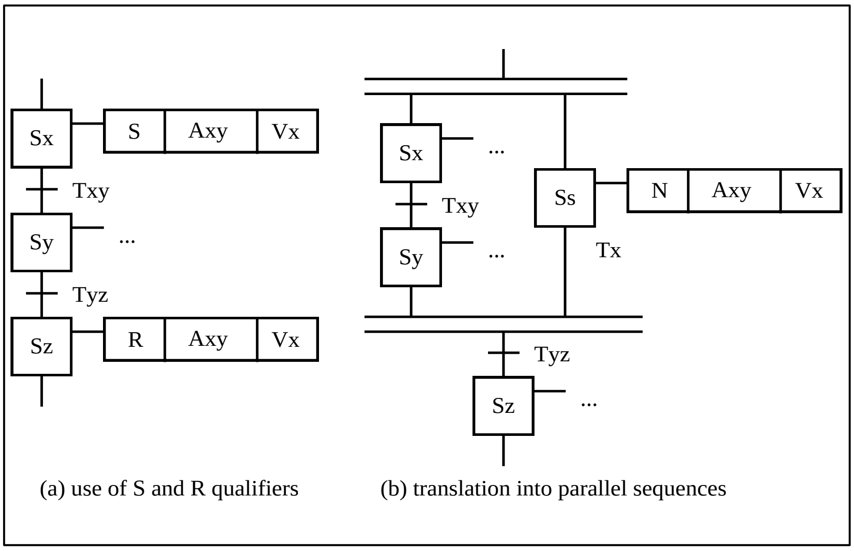
Figure 11 **Action dependence by step splitting**

4.6 Advanced use of parallel sequences

The possibility of using synchronisation steps to remove stored actions, or even to manage the interactions among parallel sequences.

Another situation where it is possible to achieve a better representation of the control organisation, thus improving code readability, is that involving actions characterised by the S (set) and R (reset) qualifiers. We have seen, in the previous Section, that it is possible to be more explicit by representing the same actions with the only N qualifier. The limit of this solution it the bore of replicating the actions as many times as the number of steps where they must be executed, and the consequent loss of conciseness. An alternative solution is obtained by transferring the problem to the structural level, with the use of parallel sequences.

Figure 12a shows an example of a stored action, the Axy action, to be executed while steps Sx and Sy are active. In general, we can assume that both Sx and Sy have other associated actions: if this was not the case, there would not be any need to take them distinct and it would be enough to associate the Axy action to a unique step (say Sxy) with the N qualifier. Instead, the scheme in Figure 12a says that, during the execution of the actions of Sx, and then of Sy, the execution of Axy has to be kept on running.



(a) use of S and R qualifiers (b) translation into parallel sequences

Figure 12 **Parallel sequences in place of stored actions**

The same result is obtained with the scheme of Figure 12b, where the parallelism between the actions of Sx and Sy and the Axy action (now associated to step Ss) is made evident by the definition of two sequences activated by a simultaneous divergence. Obviously, the exit condition from the divergence, that is, the simultaneous convergence condition, is the same that determined the transition from step Sy to step Sz in the previous scheme.

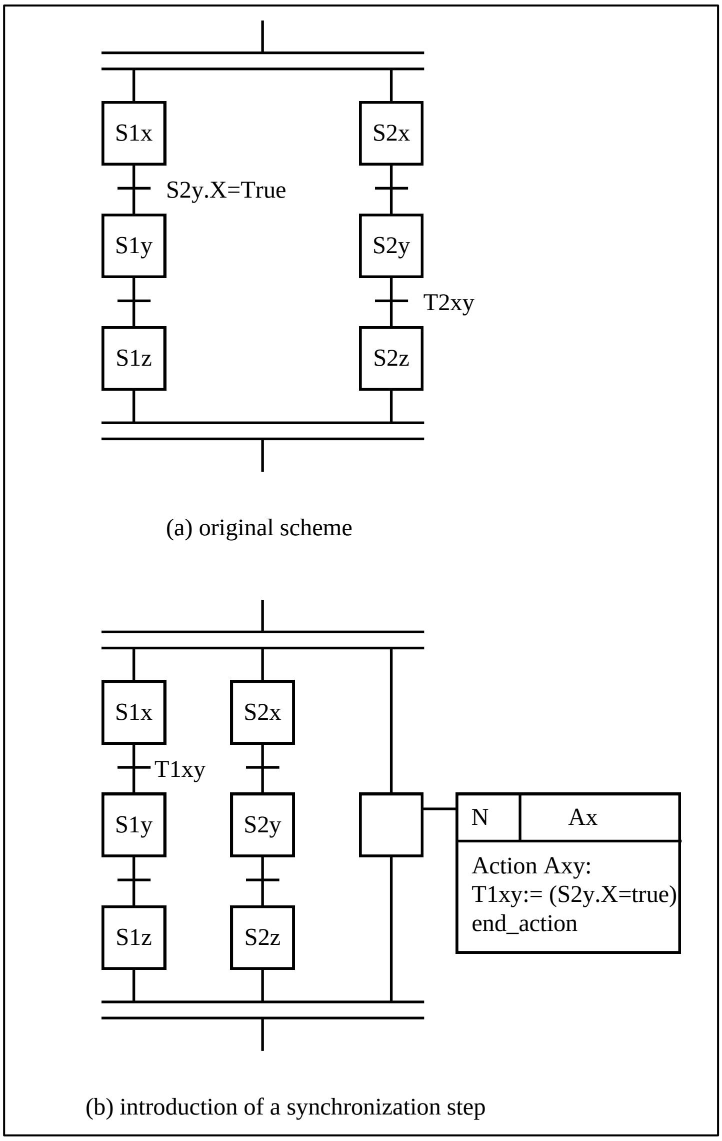
A similar approach permits to make explicit in structural terms the behaviours of temporally constrained stored actions, those qualified as SD (Stored and time Delayed), DS (time Delayed and Stored) and SL (Stored and time Limited). We have seen that a representation based only on N actions implies the use of a proper timer to measure the elapsed time from the activation of step Sx. Unfortunately, this timer has to be pre-set in Sx and then tested in the following steps. This fact introduces a strong coupling between these actions and, what is more important, such a coupling is hidden in their texts. As a consequence, the degree of reusability of the resulting code is decreased since the actions can be hardly separated one from the other.

Then, we observe that all the cases of temporally constrained stored actions generate the same structure of Figure 12b. The difference is represented by the way how the actions take into account the step elapsed times:

- *SD (Stored and time Delayed)*. At any scan cycle, action Axy compares the given time delay with the measure of the elapsed time from the activation of steps Sx and Ss. This can be done with a proper Axy timer or, more simply, by watching to the value of the Sx.T variable and, if necessary, to the value of the sum Sx.T + Sy.T. If the delay threshold is reached, the core action body begins to be executed.
- *DS (time Delayed and Stored)*. In this case, action Axy compares the given time delay only with the measure of the variable Sx.T. If Sx stays active for a time interval longer then the delay threshold, the core action body begins to be executed.
- *SL (Stored and time Limited)*. As in the SD case, the Axy action compares the given time delay with the measure of the elapsed time from the activation of steps Sx and Ss. The difference is that the core action body begins to be executed at the activation of the divergence, and the delay threshold determines the end of its execution.
- *Combined cases*. Besides the possibility offered by the standard qualifiers, it can be useful to define actions where the execution interval ranges between two time instants defined by the programmer. This is equivalent to combine the delay option with the time limit option. While this situation is difficult to realise in the framework given by the standard, it is immediately obtained with the approach described above. It can be noted that the scheme structure remains always the same represented in Figure 12b.

It is worth observing that step Ss plays a supervision role on the evolution of the control from Sx to Sy. Thus, each of these steps is no more required to be aware of the behaviour of the other, that is, the programmer can develop either step autonomously. This makes the resulting code much more reusable and the step Ss itself, as supervisor of the interactions between steps Sx and Sy, could become in its turn a reusable module.

A generalised use of a supervision (synchronisation) step is that shown in Figure 13.



(a) original scheme

(b) introduction of a synchronization step

Figure 13 **Synchronising parallel sequences**

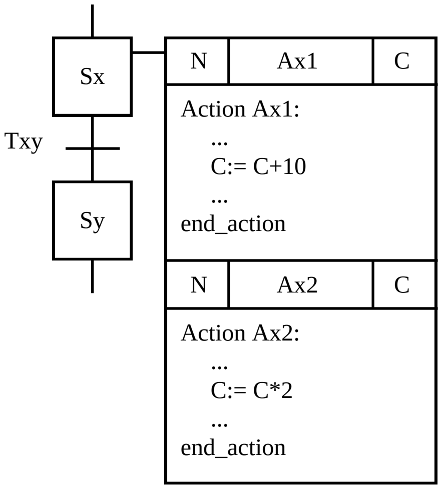
The scheme of Figure 13a presents two parallel sequences where the activation of one step of the former (S1y) is conditioned by the activation of one step of the latter (S2y). By introducing the synchronisation step Ss (see Figure 13b) the two parallel sequences no longer need to know the evolution of each other. In fact, action Axy performs the task of assigning the enabling value to the Boolean variable T1xy at the right moment.

4.7 Action independence

If the choice of splitting the control into steps is crucial for software readability and reusability, the care of coding actions so as to be sure of their effects is equally important. In this respect it is necessary to remember that, at every PLC scan cycle, more than one step can be found active and more than one action is often executed for each active step.

In order to be sure of the correct control program running, and to avoid undesired side effects, it must be ensured that all the actions that can be executed at a given scan cycle are mutually independent. It means that they should not use results coming from other simultaneous actions, nor they should modify variables that other actions use during the same cycle. The reason is that the

IEC standard gives no rule on the execution order of actions. In practice, every PLC environment adopts its own execution strategy which is often unknown to the programmer.



```
Action Ax1:
    ...
    C:= C+10
    ...
end_action
```

```
Action Ax2:
    ...
    C:= C*2
    ...
end_action
```

Figure 14 **Interdependent actions**

Therefore we have to imagine that all actions contemporarily active are executed in parallel. We have no mean to know if an operation precedes or follows another operation on the same variable in another action. This situation is reproduced in Figure 14 with the two actions Ax1 and Ax2 associated to the same step Sx.

Assuming that the current value of the C variable is 5, let us indicate respectively with C1 and C2 the results produced separately by Ax1 e Ax2:

```
C1 = C + 10 = 15
C2 = C * 2 = 10.
```

Thus, after one cycle the following different results could be obtained:

```
C = C1 = 15              if Ax1 is executed after Ax2
C = C2 = 10              if Ax2 is executed after Ax1
C = C1 * 2 = 30          if Ax2 uses the result of Ax1
C = C2 + 10 = 20         if Ax1 uses the result of Ax2,
```

and the difference increases in the subsequent cycles.

The case of Figure 14 is a typical symptom of the low communication level between the programmers that developed Ax1 and Ax2, as variable C is used without paying attention to its evolution in time. This kind of situation can be singled out by producing cross reference tables and tracing the changes of system variables. However, a more general and effective rule states that conflicts do not arise if the contemporarily active actions are mutually independent.

Let E be the set of the actions simultaneously executed during a given scan cycle. A sufficient condition to avoid conflicts among them is:

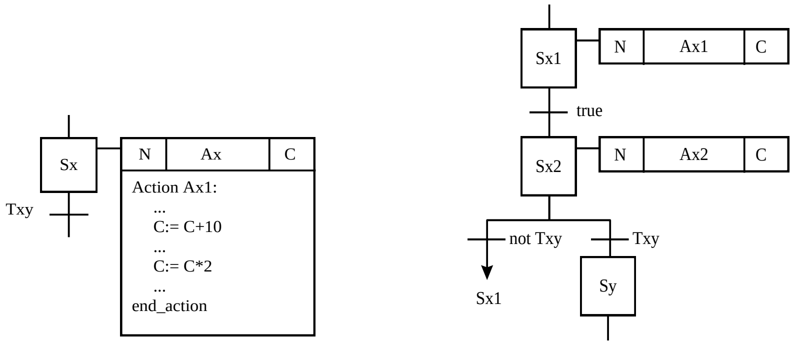
```
∀ Ai, Aj ∊ E :
     (i)     Mi ∩ Mj = ∅
     (ii)    Ui ∩ Mj = ∅
     (iii)   Uj ∩ Mi = ∅
```

where Ui and Mi are the sets of variables respectively used and modified by action Ai, and Uj and Mj the sets of variables respectively used and modified by action Aj. The condition is only sufficient since situations may be found where the condition is not true and nevertheless there are no conflicts (for example, in the cases when the used variables do not alter the computation of the modified variables).

The violation of condition (i) corresponds to the case where both actions try to modify the value of the same variable. If this is not a programming bug, it is the evidence of a low action cohesion. In fact, there is an aspect in the control, represented by the variable in conflict, whose management is in charge of two different actions. In this case we suggest to assemble all the operations dealing with strongly related variables in to a single action, and to code elsewhere the tasks that correspond to other control aspects.

A more frequent case is that causing the violation of conditions (ii) and (iii). Not considering code bugs, it can be interpreted as the programmer intention to modify a given variable many times, relying on a precise actions execution order. Since, as we have said, such an order is in many cases unpredictable, the result may be out of control. This case results in a strong action coupling, which can be corrected by making explicit the desired execution sequence.

Figure 15 shows two possible solutions to the problem described in Figure 14. In Figure 15a it is suggested to gather the actions in conflict (or, at least, the specific instructions in conflict) in a unique action fixing the execution order. In Figure 15b, actions Ax1 and Ax2 are associated to steps Sx1 and Sx2 resulting from splitting Sx. The jump instruction following Sx2 causes the iterated execution of the two actions in the established order, until the occurring of the exit condition Txy.



```
Action Ax1:
   ...
   C:= C+10
   ...
   C:= C*2
   ...
end_action
```

(a) unifying actions

(b) splitting steps

Figure 15 **Solution of action conflicts**

Obviously, the two solutions are not perfectly equivalent: the former guarantees that the two changes of variable C occur during the same cycle, but this is not true in the latter.

4.8 Rules for S/R qualifiers’ usage

The S (set) and R (reset) qualifiers characterize the so-called stored actions. The pair (S, R) is a way to shorten the SFC scheme, as it avoids duplicating the same action reference many times. Unfortunately, the resulting scheme is often less readable. If the distance between the S and R steps is considerable, or a divergence is put in between, the execution range of such an action becomes very difficult to understand. Besides, it may happen to define paths including the only S or R qualifiers. This causes serious problems during testing, maintenance and code upgrading.
Hence, it is suggested to limit the use of the S and R qualifiers and to ensure that the execution range of the stored action is a short sequence without a divergence.

Overall it is advised not to use S and R qualifiers due to less readable and less understandable code.

However, if an action block uses an action with the ‘S’ qualifier then another action block shall use the same action with the ‘R’ action qualifier (antivalent), or vice versa.

4.9 Rules for Step variables

Create a rule: Avoid using *stepname.X* and *stepName.T* variables in action. This interleaves the graph with the action code. Consequently after changing the association of action blocks to steps the programmer has to check and adjust the *stepName* in the actions. ??

4.10 Rules for Actions

Program actions in such a way to be “mutually independent” of each other, avoid dependencies between different actions. Avoid dependencies or assumptions about the execution order of actions within the same POU. See also page 20, chapter “Ad 3: Action independence”

5 Introduction State Diagrams

The SFC representation of a system control has all the characteristics of a *state diagram*. It is commonly accepted that state diagrams are effective formal tools particularly suitable in representing dynamics knowledge. The most important differences are:

- A state diagram has only one state active, while SFC can have parallel steps active
- The presence of actions, that is, the activities the control has to perform at each single SFC step.

State diagrams provide the graphic representation for a class of particularly simple and efficient algorithms, the so-called *finite state automata*. A finite state automaton is particularly easy to handle since it can be formally described in a non-procedural way. It is totally and uniquely defined by giving the following information:

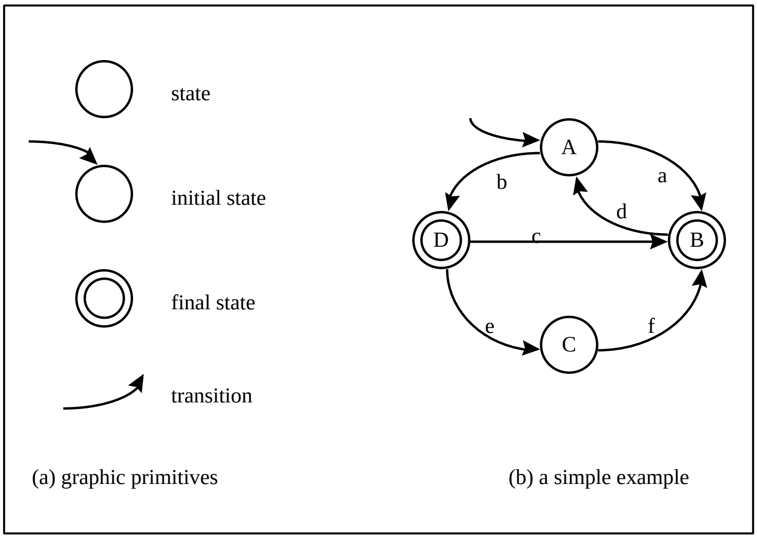
- *The finite set S of states*. Each state is the representation of a specific, significant situation characterising the phenomenon under description. When the automaton describes the dynamics of a system, the identification of its meaningful states is fundamental for a clear partitioning of the control.
- *The set E of events*. These are the events causing the transitions from one state to another. In an automated system they correspond to particular combinations of signals coming from sensors, commands given by the operator, and values reached by internal variables.
- *The initial state I*. The automaton execution starts from one of the states of the set S, the so-called initial state. The initial state is unique. The initial state of an automated system is the one corresponding to a correct and coherent initialisation phase.
- *The set F of final states*. The final states F constitute a subset of the set S of states. Each of them corresponds to one of the correct situations where the system should lie at the end of the execution. If the automaton stops in a state not belonging to F, it means that we are in presence of an abnormal termination.
- *The set T of transition rules*. A transition rule (function) is defined as the triple <si, eij, sj> that relates the current state si, the event eij enabling the transition to a new state, and the new reached state sj. The set T contains as many transition rules as the legal state transitions of the automaton.

The *execution model* of the automaton requires that it has always assigned a state value that changes time by time. At the beginning, the automaton is placed in the initial state, and there it remains until one of the events that cause a state transition occurs. The condition under which the transition takes place is the existence, in the set T, of the rule <si, eij, sj>, where si is the current state (at the beginning, the initial state) and eij is the occurred event. In such a case the automaton moves to the new state sj where it remains up to a new transition triggering event. If we think at the automaton as an algorithm, we see that it receives, as input, a sequence of events and generates, as output, the indication of the reached final state or that of an abnormal termination.

This execution model implies that a further condition, still not stated, should be satisfied: the transitions leaving each of the states are enabled by disjoint events, so as to guarantee that the new state to be reached is uniquely identified. In other words, the two transition rules:

```
<si, ex, sj> and <si, ex, sk>
```

cannot belong to the set T simultaneously, otherwise the same event ex could, leaving si, bring to either sj or sk. An automaton satisfying such a condition is called deterministic. A non-deterministic automaton is not necessarily wrong, but it requires a more complex and less efficient execution model.



(a) graphic primitives

(b) a simple example

Figure 16 **State diagram representation and example**

State diagrams represent finite state automata as oriented graphs, obtained starting from the graphical primitives shown in Figure 16a: states are circles, the initial state is a circle with the starting arrow, final states are double line circles, transitions are oriented (arrowed) branches going from the current state to the next one and carrying the indication of the enabling event. Figure 16b shows a simple diagram with four states, two of which are final, and six transition rules. Its formal definition is the following:

```
S  =     {A, B, C, D}
E  =     {a, b, c, d, e, f, g}
I  =     A
F  =     {B, D}
T  =     {<A, a, B>, <A, b, D>, <D, c, B>, <B, d, A>, <D,
          e, C>, <C, f, B>}
```

The first and most important aspect to consider in defining an automaton is the choice of the states that better represent the behaviour of the modelled system. In most cases it is not easy to select the significant states among all the possible states that the system can assume. For instance, suppose that the system behaviour is described by N variables, and that each of them can assume on the average M different values. The number of the possible combinations of these values is then MN. Even if some of the combinations cannot be reached, typically because of incompatibilities among variable values, the potential number of states still remains very high.

It is for this reason that we are interested to those states we consider significant. Let us imagine a shuttle moving from point to point in a warehouse. During its motion, the variable expressing its position assumes continuously changing values. However, if we observe the motion from the PLC viewpoint, the principal task is verifying that the path actually followed by the shuttle coincides with the desired one. This control operation has to be executed exactly in the same way at every cycle, in correspondence of each of the points reached by the shuttle. This means that we can resume the whole motion states into a unique system state (say, the shuttle moving state) whose associated action is the path control.

An effective way for identifying the significant states of a system is to start from detecting the events causing communications between the entities involved in the system control. Two of these entities are always the controlled system and its control software. Coming back to the shuttle example, we imagine that another entity could be the operator, who assigns missions to the shuttle.

In order to represent the events it is comfortable to use an event trace diagram, based on these simple rules:

- *entities* are drawn as vertical lines
- *communications between entities* are drawn as oriented horizontal lines leaving the sending entity and reaching the receiving entity
- *enabling events* are written on the corresponding communication lines
- *temporal order of events* is expressed, although not in a proportional scale, by the vertical drawing order of the communication lines.

Figure 17 shows the event trace diagram describing some of the possible events affecting the shuttle behaviour. At first, the operator sends to the system control a message specifying a new mission for the shuttle. The control, after executing possible correctness checks, gives the shuttle the command to move and, at the same time, sends an acknowledgement message to the operator. The shuttle executes the command and, once reached the new position, sends a confirmation message to the control. As a consequence, the control informs the operator that now it is possible to work on the material transported by the shuttle. As this activity terminates, the operator informs the control that the shuttle is free again, and the control commands the shuttle to go back to its stand-by position. Once the stand-by position is reached, the shuttle informs the control, and the control sends a message to the operator to confirm that the shuttle is ready carry out a new mission.

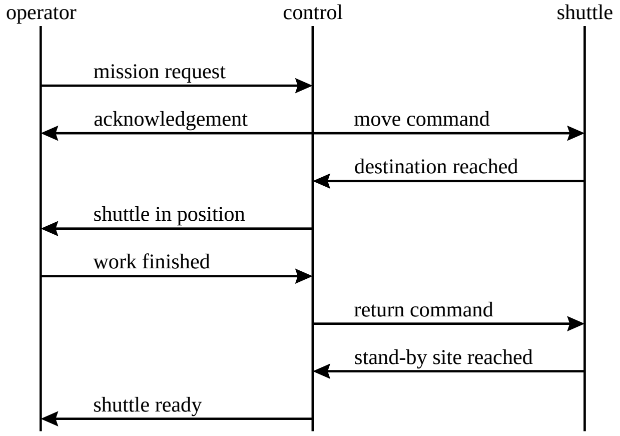


Figure 17 **Event trace diagram for a shuttle**

Note: Event trace diagrams are called Sequence Diagrams in UML, however use a somewhat different notation.

Starting from this representation, some of the significant states of the shuttle behaviour are easily identified, namely:

- *ready*, if it is available to perform a new mission (at the stand-by site)
- *moving*, while it is changing its position in the warehouse
- *arrived*, if it has reached the destination (other than the stand-by site)
- *working*, while the transported material is handled
- *free*, when it can go back to the stand-by site.

Other states, not considered in the example, could be those corresponding to the different error conditions (wrong positions along the trajectory, troubles on the motor, and so on).

The relationship to establish between system behaviour and control software functions is now likely easier to understand. The state diagram representing the system dynamics must be translated into a control algorithm, say an SFC scheme, with as many steps as the states of the diagram. Each step will include the actions (signal analysis, comparisons, computations, command issuing) to perform whenever the system is in the corresponding state.

Although state diagrams are very useful in the design of PLC software, as the corresponding SFC schemes present identical structures, it is important to separate the two concepts. On the one hand, we have a system changing its state as the consequence of internal events or commands sent by the control. On the other hand, we have a software program which keeps under continuous control the state of the system so as to react in the shortest possible time. Even though the state change is forced by the control, this cannot move to the next step until a confirmation of the occurred transition comes from the system.

The rules that the programmer has to follow to implement correct control algorithms arise from the comparison between state diagrams and SFC schemes:

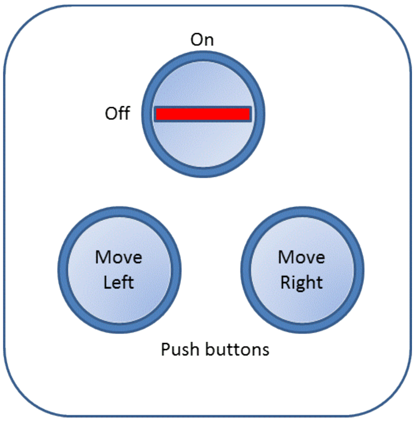
- steps and transitions are interleaved
- each transition has a unique origin step
- each transition has a unique destination step
- a step can be both origin and destination of a transition
- for each transition an enabling event is defined
- a single step can have more than one transition reaching it
- a single step can have more than one transition leaving it
- transitions leaving different steps may adopt the same enabling event
- transitions reaching a given step may adopt the same enabling event
- transitions leaving a given step must adopt disjoint enabling events.

The syntactical differences between state diagrams and SFC schemes can lead the non-expert programmer to some mistakes.

6 Examples with state diagrams

6.1 Example 1: Simple motor control

6.1.1 Introduction



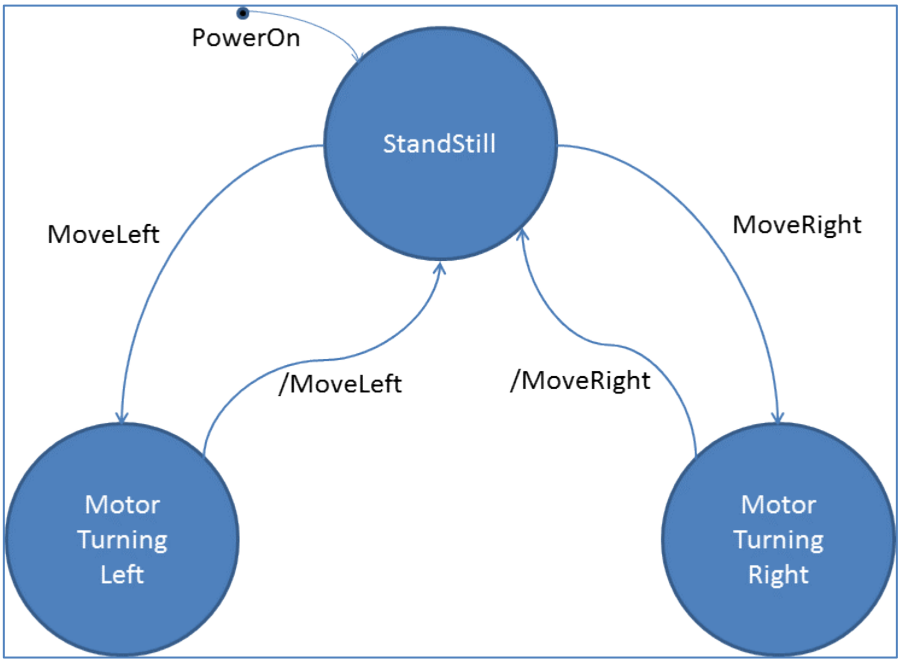
As a first example let us look to a simple motor control via a switch and 2 push buttons.
Overall there are 3 operational states:

1. Motor at StandStill and power on
2. Motor turning left
3. Motor turning right

The first state is reached by switching the motor on (*PowerOn*).
The second state is reached by pushing the button *MoveLeft*. Releasing the button brings it back to StandStill.
The third state is reached by pushing the button *MoveRight*. Releasing the button brings it back to StandStill.
Issuing the *Stop* command in the state StandStill does nothing, also not issuing an error flag in this case.

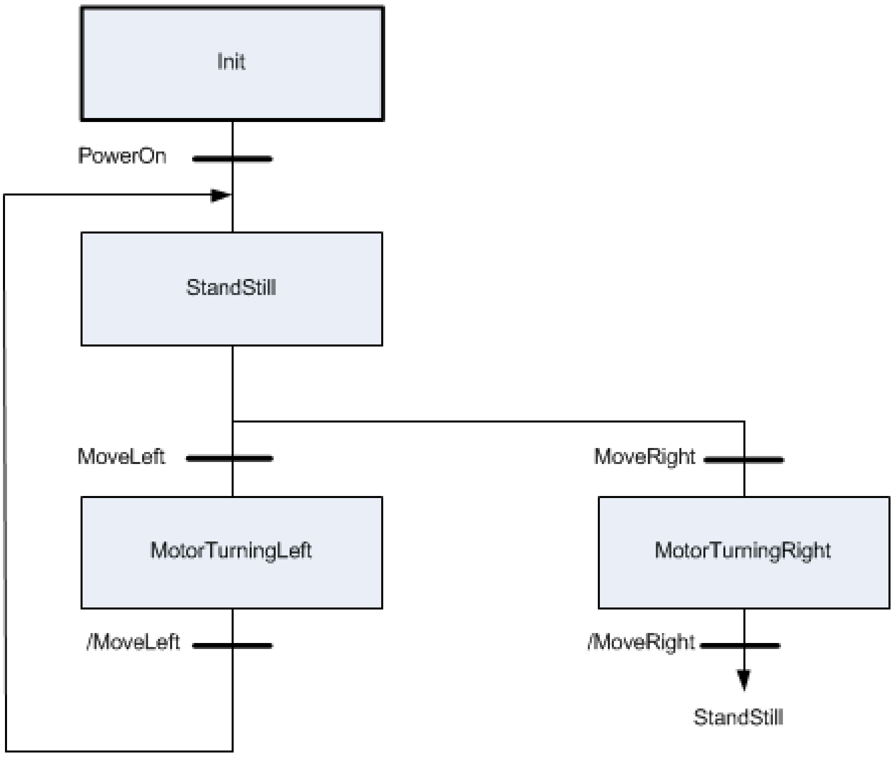
Note: in this example it is assumed that the control system is powered already.

6.1.2 States and Transitions:



6.1.3 Mapping to SFC

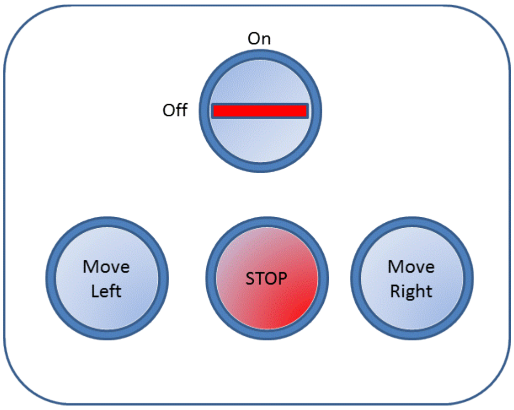
To map this state diagram to SFC, one can do the following:



Note: /MoveLeft and /MoveRight is the transition with releasing the button.
Couple to the states MotorTurningLeft and MotorTurningRight are action blocks that issue the commands to the underlying system. Such an action block can be calling an MC_MoveVelocity with the relevant direction and parameters. As an extension one can add the state *Error*, on the right side of MotorTurningRight and StandStill, which will be accessed from all the states in case of an error. A reset procedure for the error can be switching the power off and on.

6.2 Extended Example 1

6.2.1 Introduction

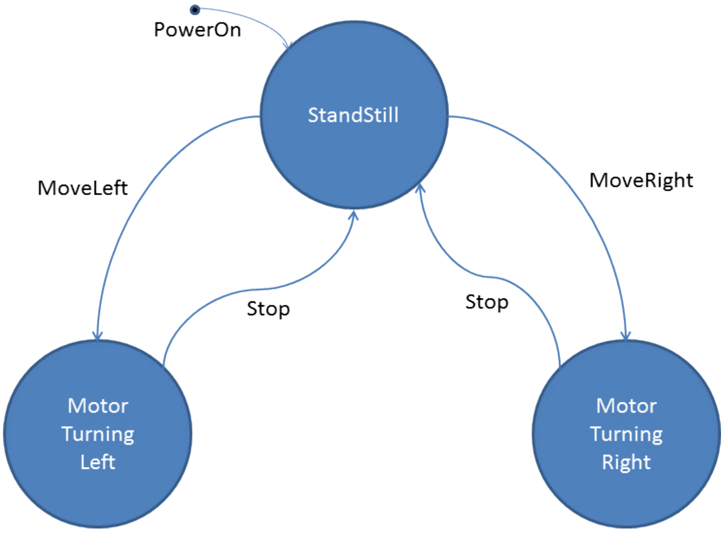


One can extend this example with continuous movements, so permanently left or right movement till the Stop button is pushed. In this case there are 3 operational states:

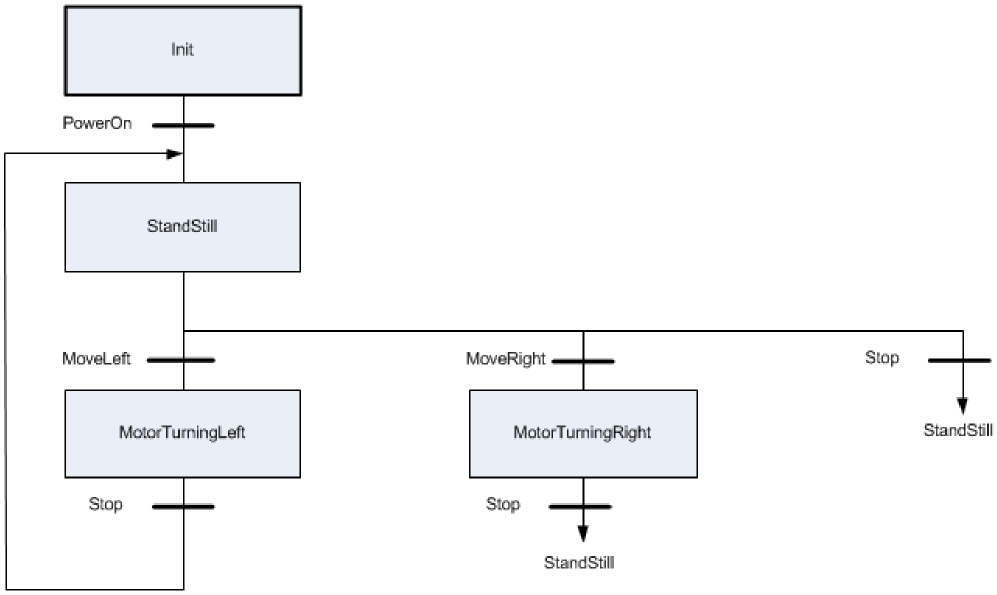
1. Motor at StandStill and power on
2. Motor turning left
3. Motor turning right

The first state is reached by switching the motor on (PowerOn).
The second state is reached by pushing the button *MoveLeft*. Pushing the *Stop* button brings it back to StandStill.
The third state is reached by pushing the button *MoveRight*. Pushing the *Stop* button brings it back to StandStill.
Pushing the *Stop* button in the state StandStill does nothing, not even issuing an error flag in this case.

6.2.2 States and Transitions



6.2.3 Mapping to SFC



The Error handling is not included in this example. How this can be handled is shown in 6.3.3 Error handling via the loops Stop and Abort.

6.3 Example 2: Mapping of the PackML state diagram to SFC

6.3.1 Introduction of PackML

For a machine, being either part of a production line or stand-alone, it makes sense to use a state diagram to harmonize the access to its functionality as well as measuring the Overall Equipment Effectiveness, OEE. In addition, using a state diagram helps to decompose the application software, making it more transparent, efficient and flexible, and less prone to errors.
As an example of how to use a state diagram, the state diagram Version 3.0 as defined by www.Make2Pack.org is used, as developed by the OMAC PackML group (see www.omac.org).

Basically PackML consists of 3 elements:

1. A state diagram, which is used here
2. A definition of a set of naming conventions incl. datatypes, called PackTags
3. A description of the different modes of operation

The OMAC State Diagram looks as follows:

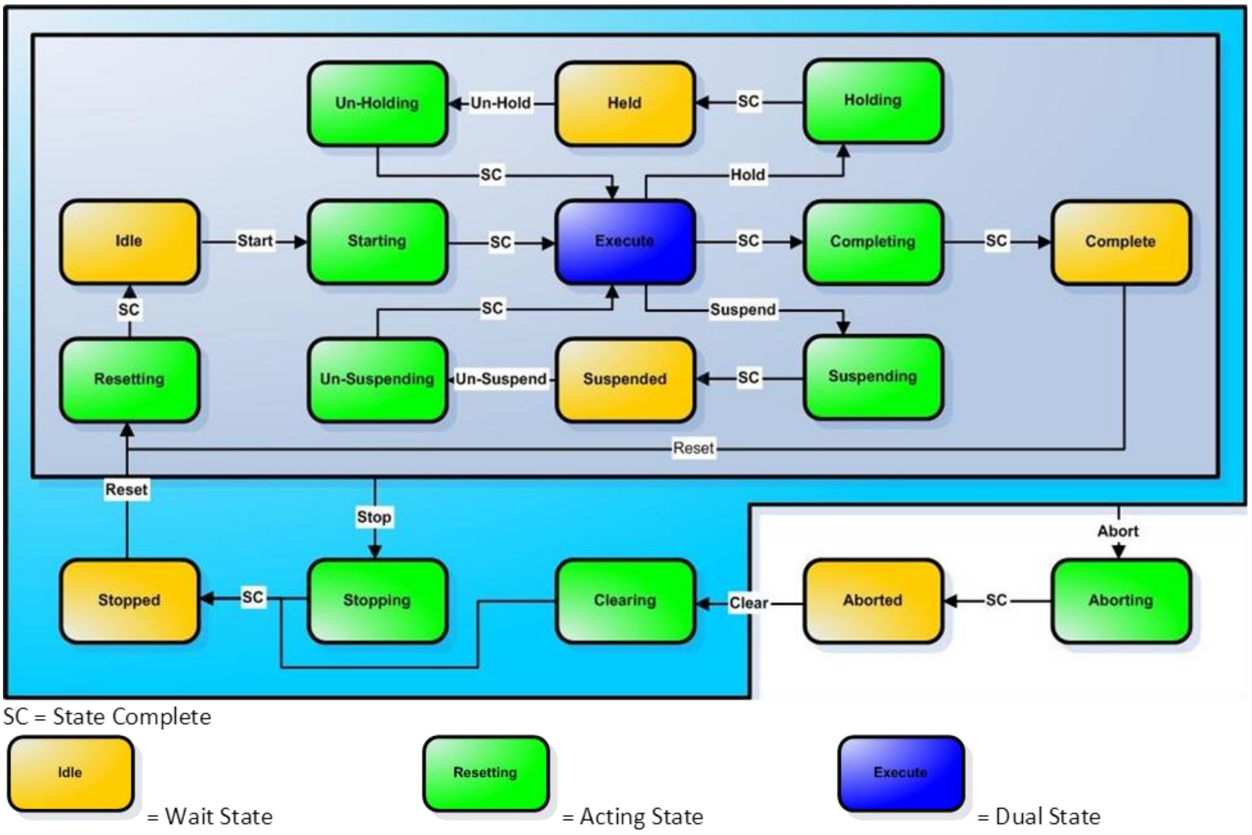


Figure 18 **PackML State Diagram.**

- A Wait State (Orange in picture) is used to identify that a machine has achieved a defined set of conditions.
- An Acting State (Green in picture) is one which represents some processing activity.
- Dual state (Blue) is defined as a machine actively executing in the chosen mode.

The states in orange and blue are stable states, meaning that they can be valid for a longer period of time.

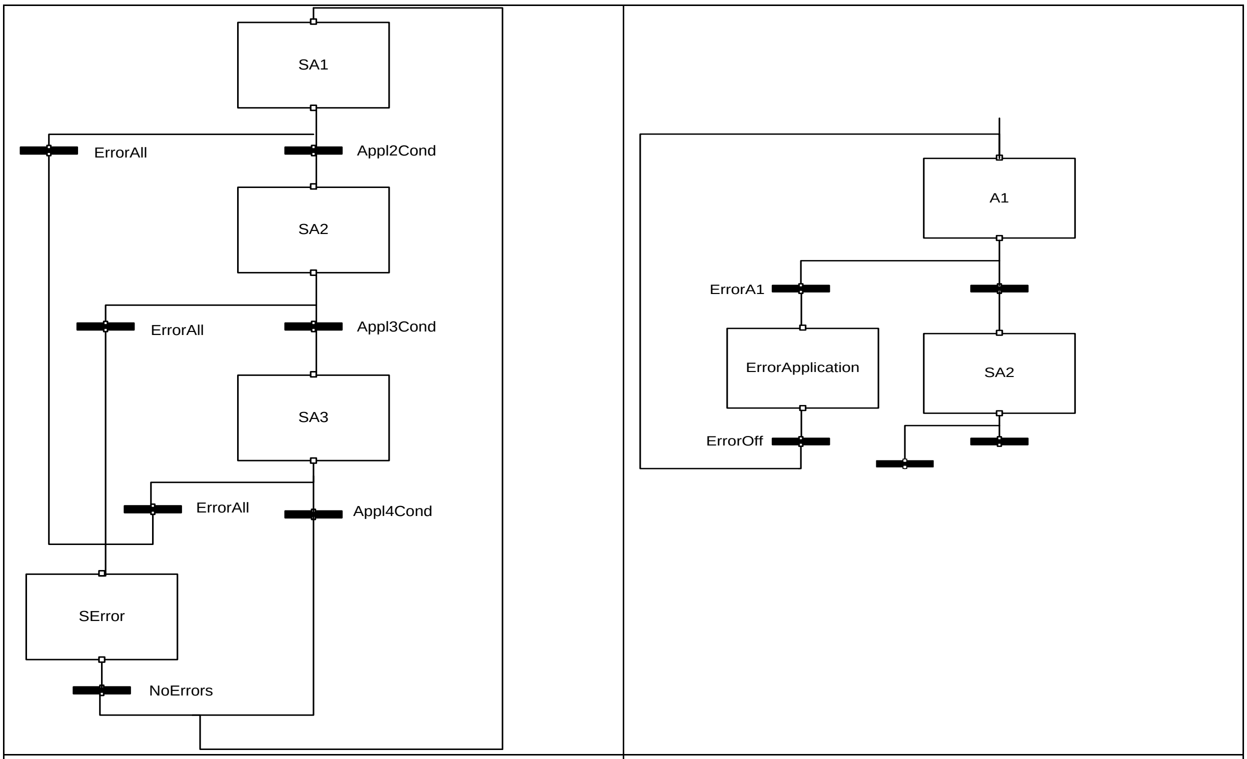
The states in green are states that are only valid for a certain period of time and transfer to the next state without intervention from an operator. The transition is automatically done if the state is complete (SC = State Complete).
Shown above is the full state diagram with the state Execute (in blue) the producing state. The loop underneath, via Suspended, is a waiting loop for material to be worked upon. The loop above, via Held, is the loop where the operator holds the system out of the producing state.
After all products are made, the producing state Execute is left via Complete, and ready for a new production order.
At power on, the state Stopped is valid. After a Reset it moves to the state Idle via Resetting.
Issuing 'Start' gets the unit to 'Execute' via 'Starting'.
The PackML state diagram leaves its normal loop via either Abort or Stop. The Abort is coupled to the error handling from every state. The Stop is for the operator interface.

6.3.2 Conversion of the State Diagram to SFC

A State Diagram should be reflected in the programming environment. One way to do this is to use Sequential Function Chart, SFC.
Because of its general structure, SFC provides also a communication tool, combining people of different backgrounds, departments or countries.
To map the PackML state diagram, we need to implement the following normal operation sequences: Stopped, Resetting, Idle (at specified pre-conditions), Starting and Execute. After Execute there are 3 alternative options: Complete, Hold, and Suspend, where the last 2 will continue via Execute.

6.3.3 Error handling via the loops Stop and Abort

All states can also be left via the Stop loops (in case a stop is commanded) or the Abort loop (in case of an error). Concerning this error handling, there are basically two ways of dealing with this:
1 Centralized – all errors in the SFC sequence are linked to one error related Step
2 De-centralized – for each step in the SFC sequence an error loop is defined



Centralized error handling | **De-centralized error handling example**

Figure 19 **Centralized versus Decentralized Error Handling**

The main states in PackML for the normal production process (executing) looks as follows:

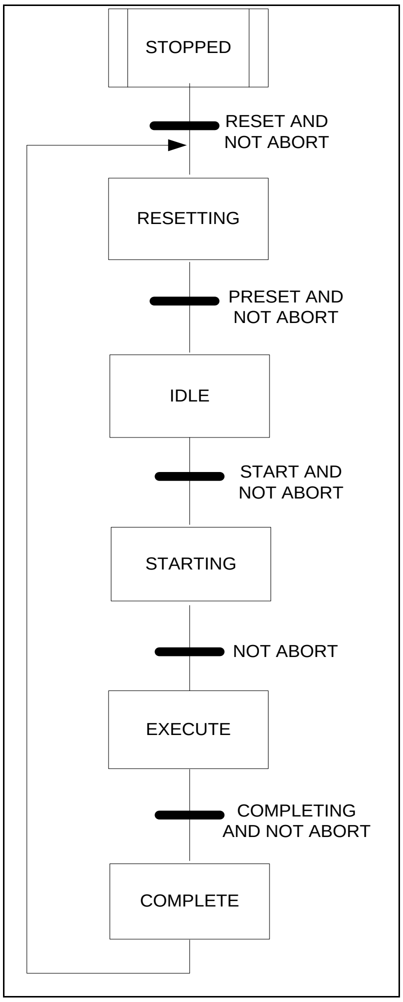


Figure 20 **Main states of PackML**

The next figure, Figure 21, shows a basic implementation of the full PackML state diagram, including the ‘Abort’ and ‘Stop’ loops, and on the lower right side the ‘Hold’ and ‘Suspend’ loops, which loop back to the ‘Execute’ state. For the error handling option 1 - centralized is in this case shown via the ‘Abort’ loop. In the top middle, the abort sequence is specified, with the ‘Abort’ entry point on top. All other abort loops refer to this starting point. Also the ‘Stop’ loop is identified there.

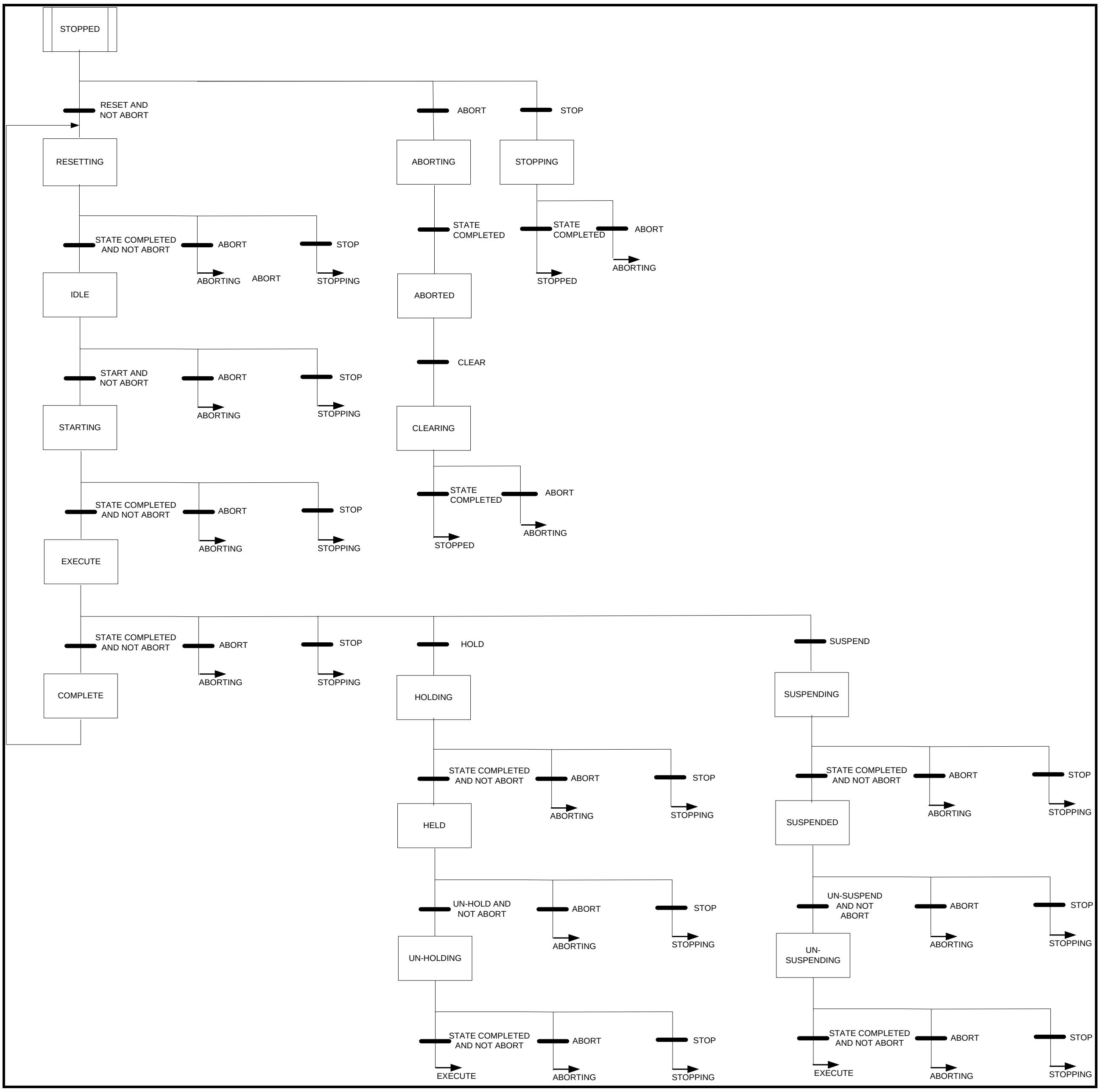


Figure 21 **SFC of PackML Sate Diagram**

Note 1: In the above drawing the graphical readability has been preferred versus implicit transition conditions [Start AND NOT Abort vs. left-to-right order (implicit priorities) in Aborting / Stopping / Reset] – because in that case the graphical representation runs out of the screen on the right side.

Concerning the transition conditions one has to note that the order also could be (from left to right) first Abort, then Stopping and then the transition condition in the normal production process loop. For readability this is changed in this example. Note that the conditions are mutually exclusive.

6.3.4 Multi-level approach – safety required

The PackML state diagram is valid for several modes, like Automatic (as used above), Semi-Automatic, and SetUp. Different modes use different states. In the producing mode all states are applicable, and no special safety precautions are involved. In the Semi-Automatic mode the holding loop is not made available, limiting the feed-in of products to on one product at a time only for checking purposes. In the Setup mode there is no production, so the ‘Execute’, ‘Suspending’, and ‘Holding’ loops are not available. This is coupled to the function blocks in the SFC program above. For instance, the Starting and Execute states are only accessible if the SetUpMode is not set, and the ‘Holding’ loop if SemiAutoModeSelected is not SET so the system is in Automatic mode.

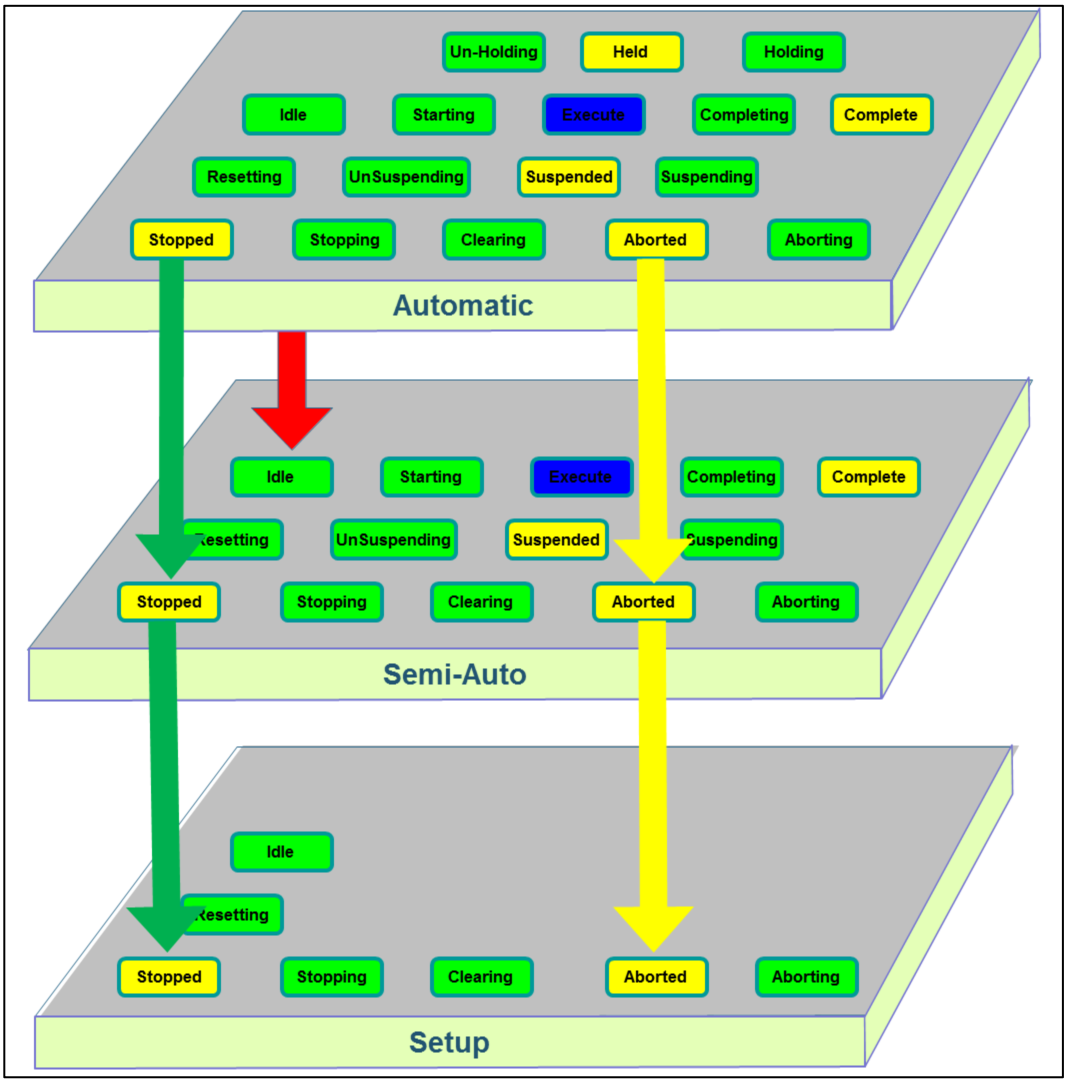


Figure 22 **Multilevel Sates**

The different states are linked, like shown in Figure 22.
In order to change between these modes, one has to fulfill the applicable safety requirements. For instance, a safety approved mode selector can be used, coupled to the safety requirements in the Setup mode. For this the PLCopen Safety Specification is intended. This specification fulfils also the requirements as specified in the *ANSI / PMMI B155.1-2006 Safety Requirements for Packaging and Packaging-Related Converting Machines*.
In this example, there is no difference in the safety requirements between the automatic and semi-automatic modes, unlike the mode Setup.
The safe ModeSelector takes care that unacceptable changes are avoided, like an inhibition from SetUp to Automatic, and like mode changes from the Automatic mode in the execute state without first stopping. This can be resolved in several ways:

1. The mode change is neglected / not accepted
2. The mode change is accepted, but via the STOP state
3. The mode change generates an ABORT

In parallel, the safety program takes care that in the setup mode the drives are in a safe state. For this the ‘Safely Limited Speed’ functionality can be used, combined with the ‘EnableSwitch’ functionality, with which the operator can move the machine at a reduced speed. The functional application program defines the safely limited speed, while the safety application checks that the limit set is not exceeded.

In addition, one can couple the emergency switch to different stop categories per motor.
A small overview of the applicable safety function blocks is shown in the next drawing. For details on the PLCopen Safety specification, check the website www.PLCopen.orgff under TC5 Safety.

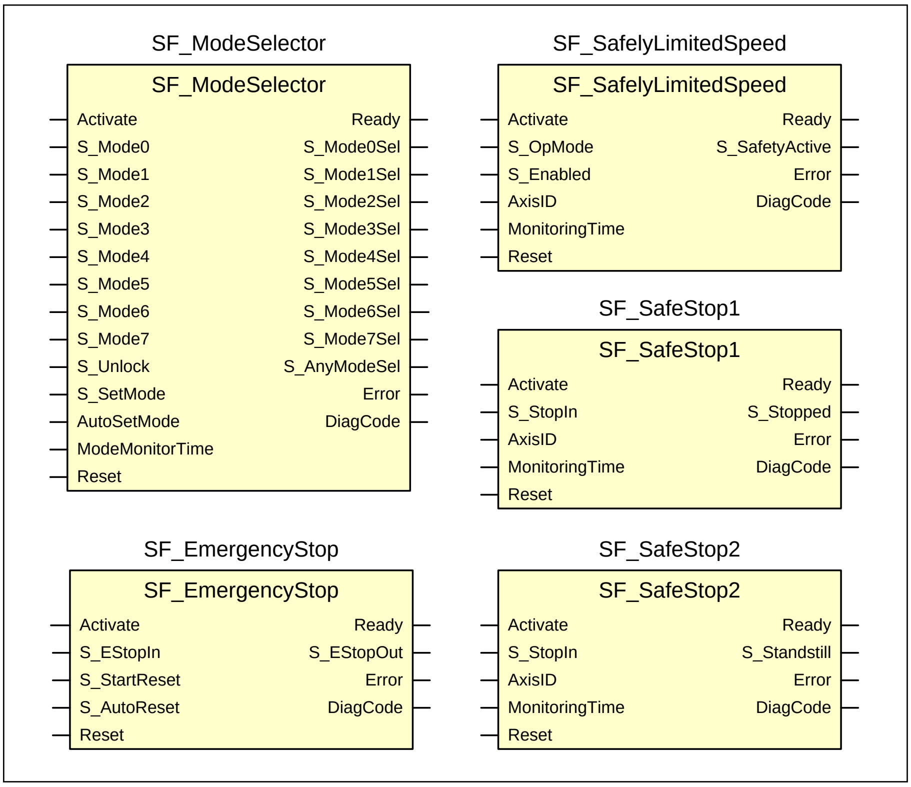


Figure 23 **Examples of PLCopen Safety Function Block**

6.4 SFC is not Petri Nets

The simultaneous presence, at run time, of two or more active steps suggests the possibility of establishing an analogy between SFC schemes (with parallel sequences) and Petri nets. Petri nets are a well-known and appreciated modelling tool, particularly suitable to represent the behaviour of concurrent systems. A Petri net provides a formalised description of the system that makes rigorous its analysis and the study of its behaviour.

In an automaton the system states are predefined, and the system passes from one state to the other in such a way that, at every time, only one state is active. On the contrary, in Petri nets state is a distributed concept and transition is a local concept: at every transition occurring in the net, the states of some places change while the others are not affected. For this reason Petri nets are suited to model asynchronous systems, where events impact on different points and do not occur according to a predefined sequence.

Also Petri nets substantial lack with respect to the SFC approach: not deterministic behaviour, absence of a primitive for defining the initialization step, places as passive repositories of objects, need of modelling the control logic in terms of flowing objects.

For these reasons do not consider SFC schemes as variants of Petri nets. Petri nets are worth to be known as they give interesting hints for modelling asynchronous dynamic systems. Nevertheless, forcing the analogy with the SFC language could be misleading and, after all, negative. Instead, we believe that designing the control architecture as a hierarchical state diagram is very useful to the PLC programmer.

6.5 Relation to Moore automata and the Mealy automata

Concerning the analogy between SFC schemes and state diagrams, it should be observed that two are the principal classes of finite state automata: the Moore automata and the Mealy automata.

- *Moore automata*. In these automata the response given by the system is associated to the state reached at every step, and not to the ways through which such state has been reached. In practice, they represent situations where the decisions to undertake are not affected by the memory of the preceding events. All the examples that we have seen so far pertain to this class of automata.
- *Mealy automata*. In these automata the response given by the system is, at least in some cases, a function of the transition leading to the new state. A typical example is that where a state concludes the execution of a process that might have been turned on, with some differences, in two or more alternative preceding states. It can be proved that such an automaton can always been converted into an equivalent Moore automaton.

This explains how to represent “Mealy automaton like” behavior in SFC which is designed to represent “Moore automaton”.