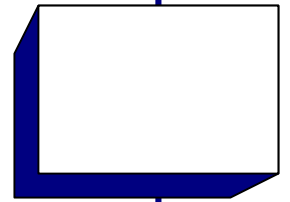


Grafcet Step-by-Step

Tutorial and
Reference Guide
to the
Grafcet Language

Paul Baracos, Ph.D., P.Eng.



Contents

1	Author's Preface	1
	Acknowledgments.....	1
2	Introduction	3
	Grafcet: An automation language	3
	The Standards.....	4
	Using Grafcet	4
3	Basic Concepts	7
	What is a control system?	7
	Types of control signals and control functions	8
	Boolean logic and relay ladder diagrams	8
	Timing Charts	11
	State machines and flow charts	12
	Petri nets.....	14
4	Grafcet Tutorial	17
	Example 1- Elements of Grafcet.....	18
	Example 2 - On/Off control	21
	Example 3 - On/Off parallel control	26
	Example 4 - On/Off multi-graph parallel control	28
	Example 5A - Sequential control.....	32
	Example 5B - Sequential control with fault handling.....	38
	Example 6 - Synchronizing parallel sequences	43
	Conclusion	50
5	Elements of Grafcet	51
	Organization and modularity	51
	Variables and nomenclature.....	53
	Steps.....	55
	Transitions.....	63
	Actions	78
	Links	89
6	Grafcet Structures	93
	The stand-alone action	93
	The permanent step	94
	Data initialization.....	95
	Data initialization and continuous calculation.....	95
	Non-looping sequence with a permanent final step.....	96
	Repeating sequence (loop).....	97
	Loop with initialization.....	98
	Degenerate unstable loop.....	99

Multiple initial steps	100
Sequence jumping	101
Sequence selection	102
Interpreted parallelism	103
Redundant sequence selection	104
Another sequence selection example	105
Parallelism versus multi-statement actions	106
The OR node or logic step	107
Parallel sequences	108
Sequence synchronization (the AND node).....	109
Structural versus interpreted synchronization.....	110
7 Cook Book Examples	113
Safety interlock	114
Permanent monitoring	116
Emergency shut-down	117
Event counting	118
Program structuring	121
8 Case Study - Substation Automation.....	125
Why Grafcet?	125
The transmission line switcher.....	126
Conclusions - Hydro-Québec's experience with Grafcet	133
9 Implementing Grafcet on the PLC	135
Basic Assumptions.....	135
Implementation models.....	136
Manual translation into PLC language	139
10 Glossary.....	149
Appendix A – Bibliography.....	A2
In English:	A2
In French:	A3
Appendix B– RLL Listing	B4

1 Author's Preface

In the spring of 1986 I had the opportunity to visit the automation lab at the Université de Nancy in France. In a visit noteworthy for its hospitality, wine, fine cuisine and radioactive rain (from Chernobyl), I spent three months getting acquainted with some local developments in automation techniques, especially a new method for specifying and implementing automatic control of industrial processes.

My host, Professor G. Morel, claimed that the Grafcet method could dramatically reduce the time required to design, code and maintain programmable logic controller programs. At first I was skeptical, but when I saw the method in its real-life applications I was immediately impressed. Its elegance and ease of use made simple work out of some very complex manufacturing and materials handling automation problems. Compared with the ladder-logic, Boolean and state-transition methods I was used to, Grafcet was a dream to work with. Since then, I've been using Grafcet in my own automation consulting business, and teaching it in industry as well as in technical schools and universities.

Grafcet has become the accepted standard for programmable automation. And its use has gotten even easier; no longer is it necessary to translate the Grafcet programs into machine code by hand, powerful computer programs now do that work for us. However, until now, none of the text books on Grafcet was in English. Since English is used by a large portion of the industrialized world, I hope that this work will help fill the gap.

Acknowledgments

The creation of this book was helped along by a number of my friends and colleagues. I gratefully acknowledge the contributions of Stephen Perron, Kim Gomery, Ray Lizée, Mario Tremblay and Joseph Mereb. Special thanks go to Famic Automation Inc. and its president, Alain Latry, for their support and encouragement.

2 Introduction

Grafcet: An automation language

Ever since control systems were introduced, people have also been inventing various tools for their representation and understanding. Boolean algebra, relay ladder diagrams, gate logic diagrams, state transition diagrams, flowcharts and other methods have all been used. However, most people found these methods difficult to use and communicate with.

What was needed was a complete method for specifying the behavior of a controller. It had to be a simple and easy-to-learn method that everyone involved in industrial automation could understand.

Grafcet was invented in Europe (like Prolog and Pascal) in 1979. A design team established by the French Association for Economical and Applied Cybernetics (AFCET) included members from both university and industry. The academic members made sure that the language is rigorous and powerful, while the industrial representative ensured its applicability to real-life problems.

GRAFCET (the acronym of *GR*Aphe *F*onctionnel de *C*ommande *E*tape/*T*ransition or, in English, Step Transition Function Charts) is a graphical method for specifying industrial automation. Simple syntax, graphical representation, powerful and concise commands — these are what make Grafcet easy to learn and use.

During the first half of the 1980's, Grafcet's popularity increased rapidly in Europe as people found that it helped them solve automation problems efficiently. Today, numerous large and small companies worldwide rely on Grafcet as the key to greater automation productivity. Grafcet is taught in European universities as an integral part of the regular engineering program.

Grafcet is based on sound mathematical foundations. In its most simple form, Grafcet includes the full power of Boolean logic. Grafcet's heritage is not limited to 19th century math; it is firmly grounded in the mathematics of 20th century automation theory concepts such as state machine theory and Petri nets. However, Grafcet is not an abstract mathematical notion. It is a method developed to help people like yourself, not just mathematicians, write real-life programs to solve real-life problems.

The Standards

The Grafcet method is supported by national and international standards. The original defining document is the French Standards Commission NFC-03-190. Under an alternate name, *Sequential Function Charts* (SFC), Grafcet has also been standardized by the International Electrotechnical Commission under IEC 848. The IEC has also released pre-standardization document DIS 1131-3 to provide guidance in the implementation of PLC programming languages. This means that, around the world, wherever people are using Grafcet, they are using the same language.

Using Grafcet

Grafcet is useful for specifying almost all kinds of automation. It is especially valuable for use with programmable controllers, because it may easily be translated into low level machine languages (ladder, list, assembler). Grafcet is useful for many kinds of automation, including:

- Factory automation
- Assembly operations
- Batch process automation and control where emphasis is given to:
 - continuous control
 - sequential control
 - interlocking

Standard Grafcet has the capability to handle the arithmetic used in many continuous functions. More complex functions (such as PID) have not been included in standard, but it is clear that Grafcet's instruction set was made to be extended as required. Commercially available implementations provide access to a variety of special functions.

Whatever your automation requirement, Grafcet offers numerous advantages:

- Compact and rapid automation programming. Compared to traditional methods of programming such as relay ladder logic, Grafcet is shorter and more concise. In most cases, a page of Grafcet may be equivalent to 5 to 15 pages of ladder logic.
- Complete automation specification. Grafcet lets you place related information on the same page, no more hunting through cross-references as with ladder. You can combine sequential, parallel, Boolean and numerical control actions with commenting on a single page.
- Improved design. Grafcet is a compact and clear specification tool that lets you spend more time improving designs. It encourages a top-down, structured approach, which in turn helps avoid costly errors.
- Shortened final software development and speedier start-ups.
- Better communication between all personnel. This means that people who are familiar with the system being automated can easily participate in the automation design process with little programming knowledge.
- CASE (Computer-Aided Software Engineering) tools. Well structured codes can be generated automatically on a personal computer. CASE tools also offer documentation packages and real-time on-line troubleshooting capabilities, as well as a variety of useful tools such as simulation of the controlled process.

3 Basic Concepts

In this section, we introduce some of the basic ideas of control theory that you need for understanding Grafcet.

What is a control system?

A control system consists of a controller, which issues control signals to a physical plant or process. The controller receives its inputs from sensors and sends its outputs as control signals to actuators which convert the signals into various kinds of physical actions (opening valves, starting motors, etc.). The controller may also exchange signals with an operator console. The typical controller-plant arrangement is shown in Figure 3.1.

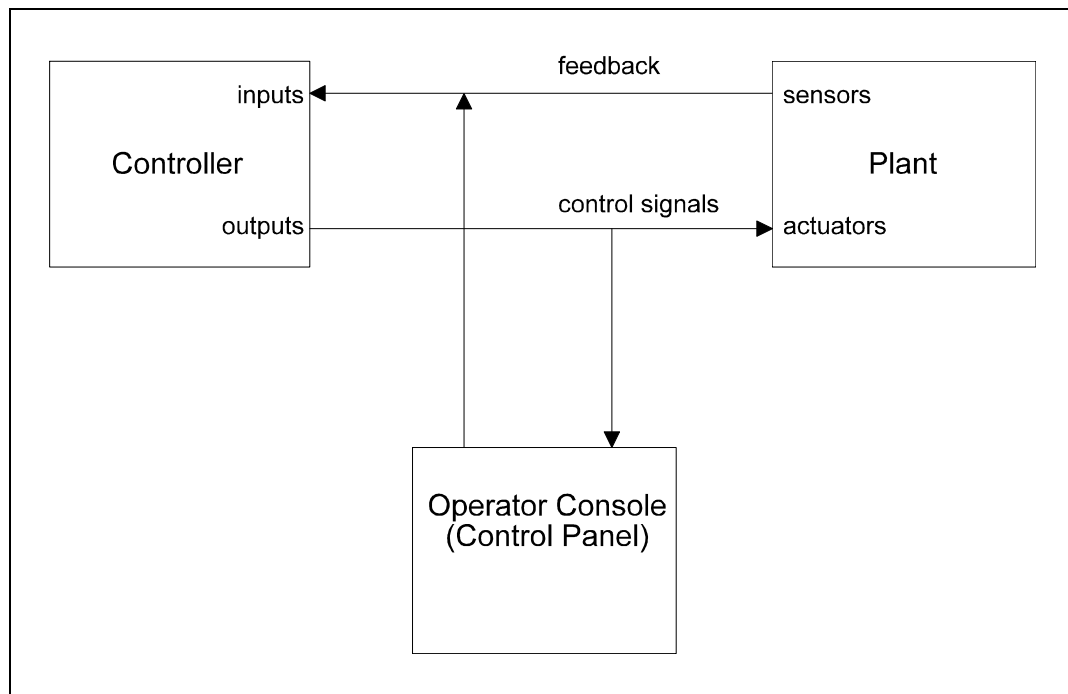


Figure 3.1

Types of control signals and control functions

Control signals fall into two general classes:

- Boolean (logical True/False)
- Numeric (usually from 8, 12 or 16 bit analog/digital converters)

Boolean signals can be either logical True or logical False. True is the *on* state while False is the *off* state. We commonly use **1** for True and **0** for False. Automation engineers often refer to Boolean control as *logic* or *combinational* control.

Numeric signals can take on any of a range of values. Most PLCs use a 16 bit integer to encode numeric values, so the range of possible values is from $-32,768$ to $32,767$. Numeric control is often called *regulation* or *analog* control.

A controller is a device (a computer in the broadest sense) that calculates a set of control signals, called a *control function*, based on a set of input signals. The control function depends on time, because its value is updated continuously in real time.

For On/Off control, when the control function depends only on the current state of the input field, we say that it is a combinational function. When it depends also on past values, then we say that it is a sequential function.

This classification can also be extended to numeric control. For instance, in proportional control, the control signal depends on the present state of the input, while in integral or derivative control, the control signal depends on the past values of the input.

Boolean logic and relay ladder diagrams

Boolean logic, named after the 19th century mathematician George Boole, is based on the AND, OR and NOT operators, whose operation is shown by the truth table in Figure 3.2. Note that values A and B in the truth table may refer to simple variables or to entire Boolean expressions. The operations correspond to the normal English usage of the words *and*, *or* and *not*. For example:

- The expression *A AND B* is only True when both A and B are both True.
- The expression *A OR B* is only True when A or B (or both, inclusively) is True. Watch out, the XOR operator is different from the OR operator. The expression *A XOR B* is True if either A or B (but not both) is True.
- The expression *NOT A* is only True when A is False.

A	B	A AND B	A OR B	NOT A	A XOR B
False	False	False	False	True	False
False	True	False	True	True	True
True	False	False	True	False	True
True	True	True	True	False	False

Figure 3.2 Truth table of basic Boolean operations.

In addition to the standard Boolean operations, most PLCs also provide two additional operators, the rising edge operator, \uparrow , and the falling edge operator, \downarrow . These are actually sequential operators rather than combinational operators, but they are so useful and they fit so well into Boolean expressions that it's worth bending the theory a little for the sake of practicality. The use of these operators is shown below in Figure 3.3.

$\uparrow A$	True only at the instant when A passes from False to True
$\downarrow A$	True only at the instant when A passes from True to False

Figure 3.3 Boolean edge operators

Figure 3.4 illustrates the Boolean operations by showing the equivalent relay ladder logic networks.




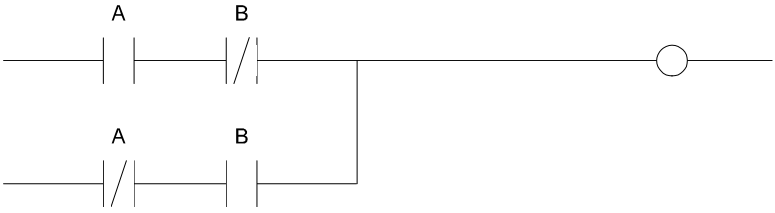
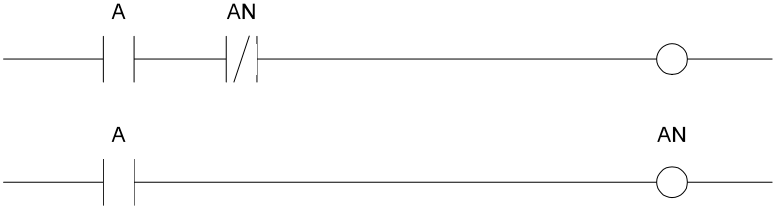
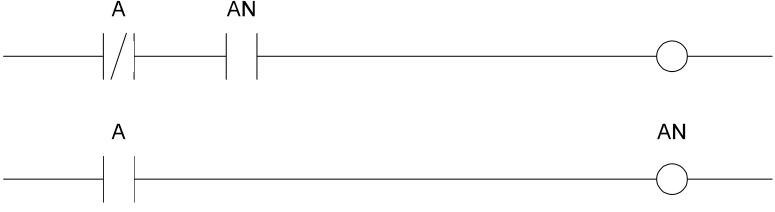
A AND B	
A OR B	
NOT A	
A XOR B	
$\uparrow A$	
$\downarrow A$	

Figure 3.4 Basic LADDER logic constructions

There are two reasons why Grafcet uses Boolean notation rather than the traditional relay ladder logic used by PLCs:

- Boolean logic is more compact.
- Boolean logic is more natural. Everyone knows the meanings of the words and, or and not. Except for electricians and PLC programmers, not every-one is familiar with ladder logic.

Timing Charts

Timing charts are used to show how one or more variables change with time. For example, in Figure 3.5, the variable A is initially False, at time t_0 it becomes True and at time t_1 it returns to False.

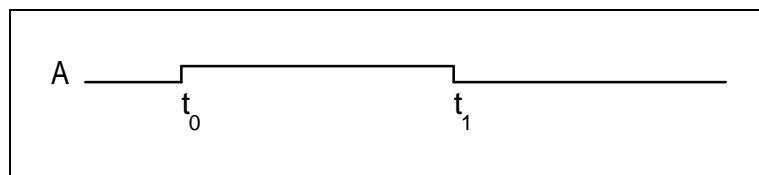


Figure 3.5

Figure 3.6 shows the timing charts of some common Boolean expressions. In this figure, variables A and B may be considered as given, and the other values are calculated based on A and B.

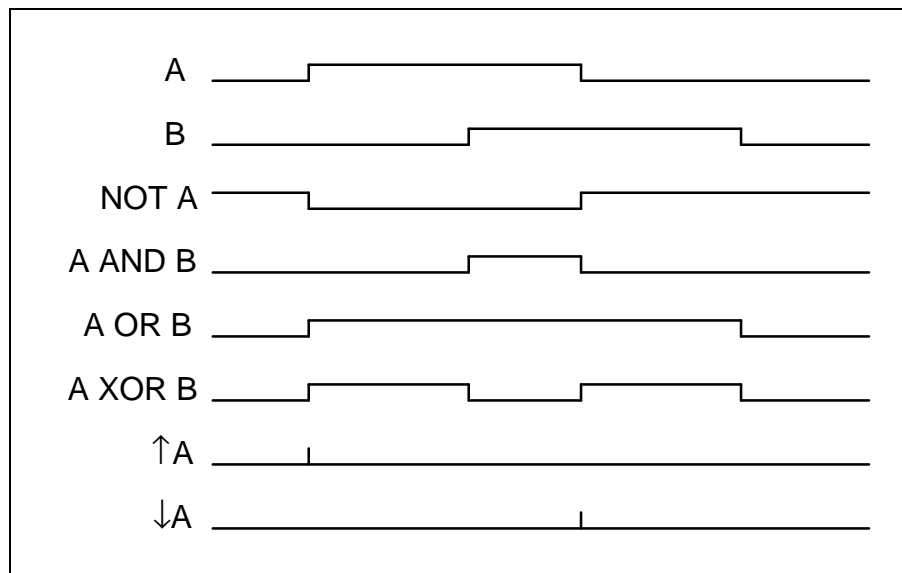


Figure 3.6

State machines and flow charts

Relay ladder diagrams have been used in the design of automatic systems since before the end of the 19th century and Boolean algebra has been used since C.E. Shannon's 1938 paper on switching network design. However, neither of these two equivalent models was an adequate tool for modern automation. The first major breakthrough came in the 1950s with the development of state machine theory, notably that of G.H. Mealy and E.F. Moore. The state machine model combined Boolean logic with memory cells to generate sequential control mechanisms. A block diagram of a Moore model state machine is shown in Figure 3.7.

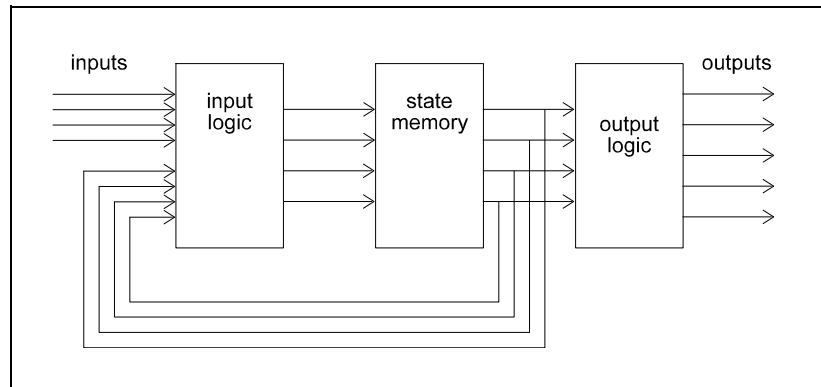


Figure 3.7 A Moore model state machine

Although a state machine can be implemented in Boolean logic or in relay logic, the state machine may be considered a more powerful tool, because it explicitly shows the memory circuits used to encode information about past events and a great body of mathematical and practical theory has grown up around state machines. Their disadvantage is that their use is usually not at all intuitive; we are still designing a hardware machine rather than working with a conceptual tool.

To get around this problem, most engineers use an equivalent representation of the state machine: the state transition diagram or flow chart. An example is shown below in Figure 3.8. This diagram shows a Moore state machine for automatically filling a tank.

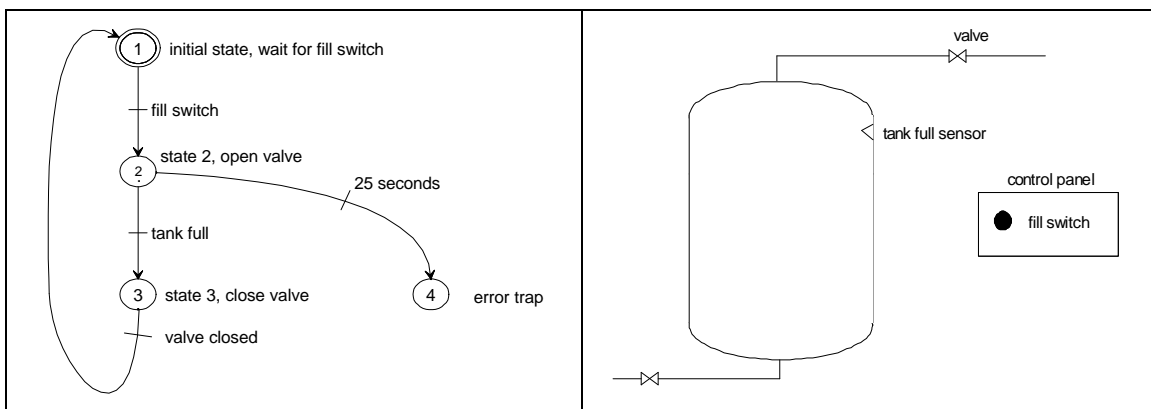


Figure 3.8 A state transition diagram

A state transition diagram is nothing more than a flow chart showing the sequence of a number of control states, usually represented by circles. Only one state can be active at a time, and the active state can be marked with a token. In Figure 3.8, state 2 is active and we would say that the controller is in state 2.

The first state, called the *initial state*, is shown by a double circle. The sequence may be conditional on input conditions. The condition for going from one state to another is written beside the arrow linking the two states.

In the Moore model, control actions are associated with the states, while in the Mealy model they are associated with the transition. Therefore, common flowcharts are simply Moore type state machines where the states are shown as boxes rather than as circles.

The main shortcoming of state machines and flow charts is that they do not provide an explicit way of specifying simultaneous control actions. A flow chart or state machine can only be in one state at a time. This is clearly inadequate for specifying parallel control.

Petri nets

The solution to the problem of specifying simultaneous or parallel control came in 1962 when C.A. Petri published the basic theory of the mathematical model that now bears his name.

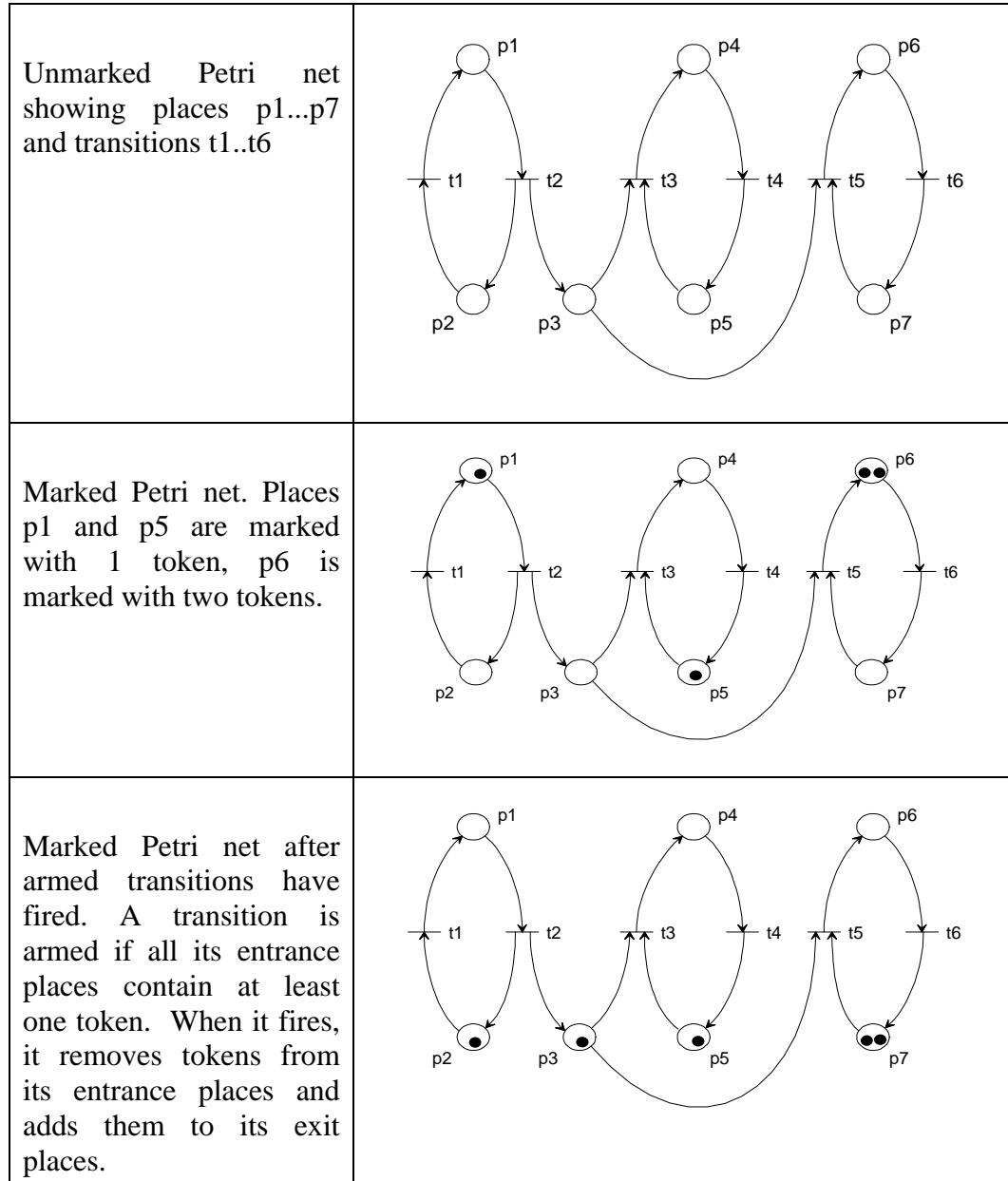


Figure 3.9

In a Petri net, the states are replaced by *places*, which may be linked by *transitions*. Unlike a state diagram, which has only a single token indicating which state it is in, a Petri net can have more than one token at a time, and may even have multiple tokens in the same place. This allows Petri nets to model queuing systems such as operating

systems for mainframe computers. Control actions may be associated with either the places or the transitions.

Like state machines, Petri nets have been the object of intense mathematical study and a body of knowledge has grown up around them. They provide an extremely powerful tool for analyzing complex automatic systems. In fact, for most production automation, they are excessively powerful and therefore are difficult to use. The developers of Grafcet liked the way Petri nets could handle parallelism, but wanted to simplify the theory so that everyone could use it, not just the most mathematically inclined. So, in 1979, a specialized form of Petri nets was first standardized under the name Grafcet.

Figure 3.10 shows the relationship between Petri nets and Grafcet. Basically, Grafcet is a form of Petri net drawn using straight lines instead of curves, and where the places are called *steps*. The new graphic format has the advantage that it could be drawn easily by hand using only a ruler and a T square. On the computer, it could be drawn easily and efficiently without recourse to splines.

The mathematical simplification involved in going from Petri nets to Grafcet is that tokens cannot accumulate in a Grafcet step the way they can in a Petri net place. Instead of representing a transaction waiting in a queue, the Grafcet token represents the activity of a step. A step can be active or inactive, and at most one token can be shown in a step. Once a step is activated, it can not become any more active.

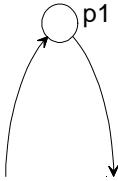
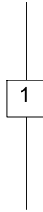
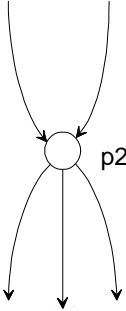
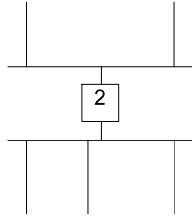
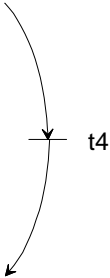
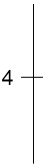
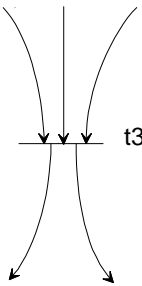
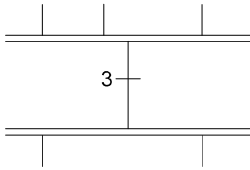
	Petri net symbol	Grafcet symbol
The Grafcet step replaces the Petri net place.		
Multiple links entering or leaving a Petri net place are replaced in Grafcet by OR convergences and OR divergences		
The Grafcet transition is drawn the same way as the Petri net transition		
Multiple links entering or leaving a Petri net place are replaced in Grafcet by AND Convergences and AND Divergences		

Figure 3.10

4 Grafcet Tutorial

Grafcet is a graphical language, so it's easy to learn. This tutorial introduces the basic elements of Grafcet and their usage through a series of sample control problems, starting with the most simple and working up to real-life complexity. Each example is designed to introduce some of Grafcet's ideas and concepts. Except for the very first example, they follow a standard format:

- First, a physical description of the plant to be controlled.
- Next, a statement of the control problem, i.e., the automation requirement.
- Finally, the Grafcet that implements an automation solution is presented and explained.

Just look them over carefully and you'll be ready to start applying Grafcet to your own control requirements.

Example 1- Elements of Grafcet

This first example is designed to provide an overview of Grafcet before getting down to details. Grafcet has four basic kinds of elements:

1. Steps, shown graphically as numbered boxes.
2. Links, shown as horizontal or vertical lines.
3. Transitions, shown as short numbered horizontal lines.
4. Statements, text instructions describing control actions and conditions.

A graph is a drawing of steps, transitions, links and statements. Typically, steps are arranged into a sequence corresponding to some series of control actions to be executed by the controller. The actions to be executed are written in a box to the right of each step. The transitions and steps are connected by links, but no two steps can be directly linked, the link must pass through a transition. The conditions each have a trigger condition which specifies when to stop executing one step and start executing the next. The graphic symbols used in Grafcet are shown in Figure 4.1.

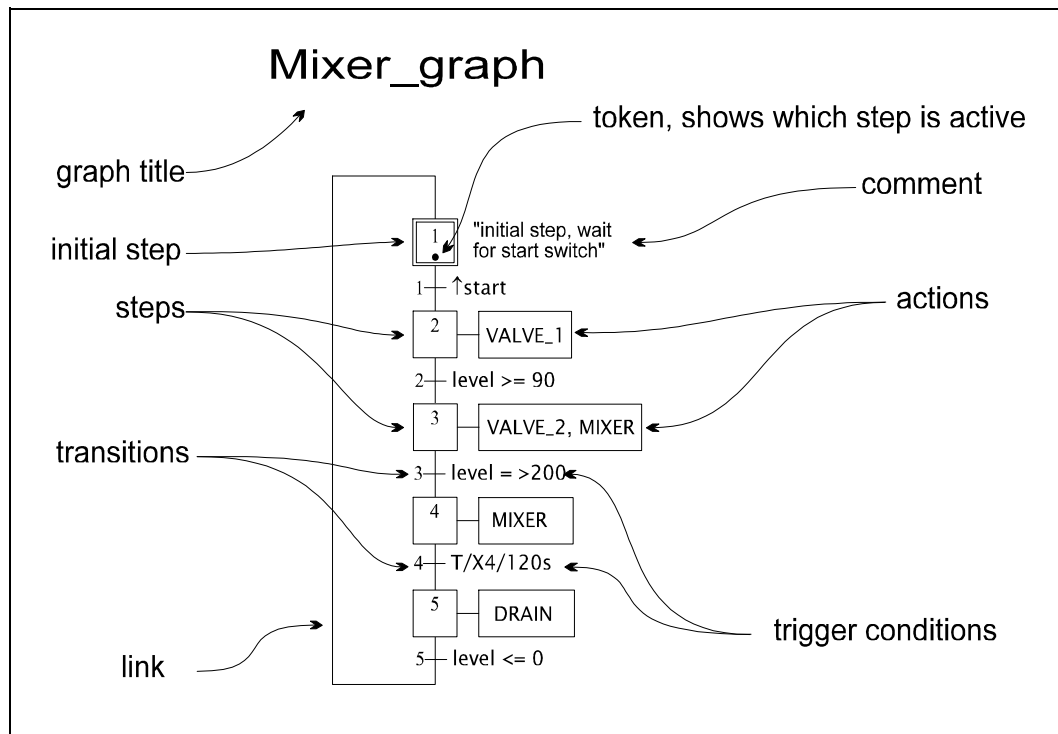


Figure 4.1

To understand the meaning of the graph, we must also examine the physical plant to be controlled. In this case the plant consists of the mixing tank shown in Figures 4.2. and 4.3. We use the convention that the tag names of PLC outputs are written in upper case while input tag names and internal variable names are in lower case. Two chemical products are fed through solenoid valves, VALVE_1 and VALVE_2, and the tank is emptied through valve DRAIN. The depth of product in the tank in cm, *level*, is given by a level sensor. The desired control action is as follows:

- When the operator pushes the start button, feed the first product to a depth of 90 cm.
- Start the mixer and feed the second product to a depth of 200 cm.
- Continue mixing for two minutes.
- Drain the tank.
- Repeat.

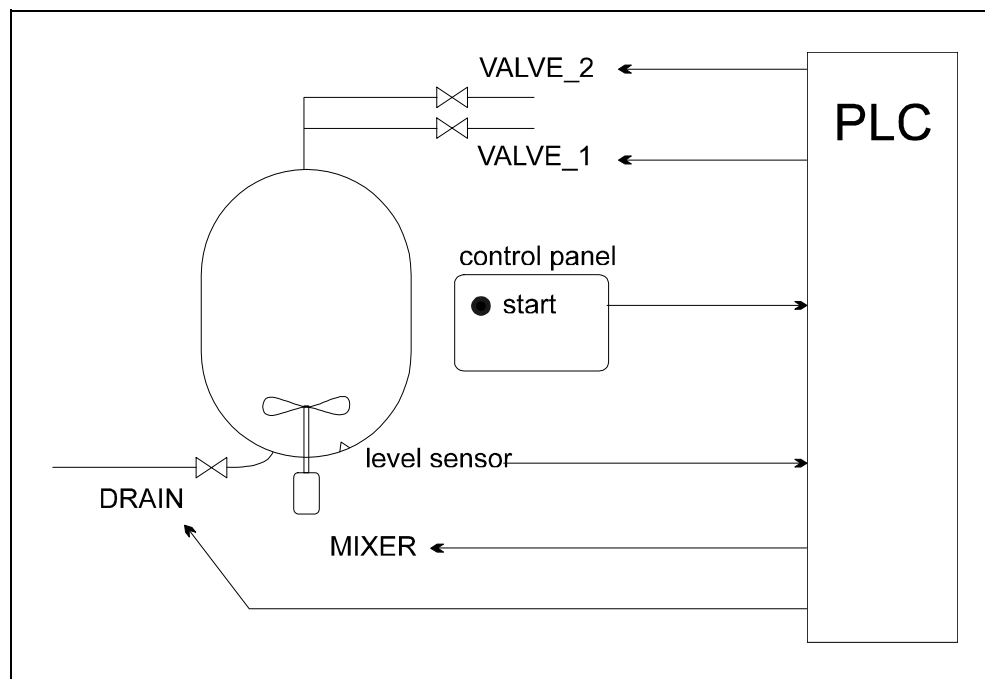


Figure 4.2

The graph that controls the mixer is a simple repeating sequence that mirrors exactly the control requirement. The timing chart shown in Figure 4.3 shows the behavior of the system over time. In this figure, X1, X2, X3, X4, and X5 are each a step activity variable. Each step variable is True when its corresponding step is active. For example, X1 is True when Step 1 is active.

In the diagram, we see that X1 is initially True because Step 1 is an initial step. At the rising edge of *start*, X1 falls, and X2 becomes True. A transition from one step to the

next occurs as soon as the intervening trigger condition becomes True. The control actions associated with each step are executed while the step is active. Output signals VALVE_1, VALVE_2 and DRAIN are True when X2, X3 and X5 are True, respectively. This is because these variables appear in the actions of the corresponding steps. The variable MIXER appears in the actions of steps 3 and 4, so it remains True while either of the two steps is active.

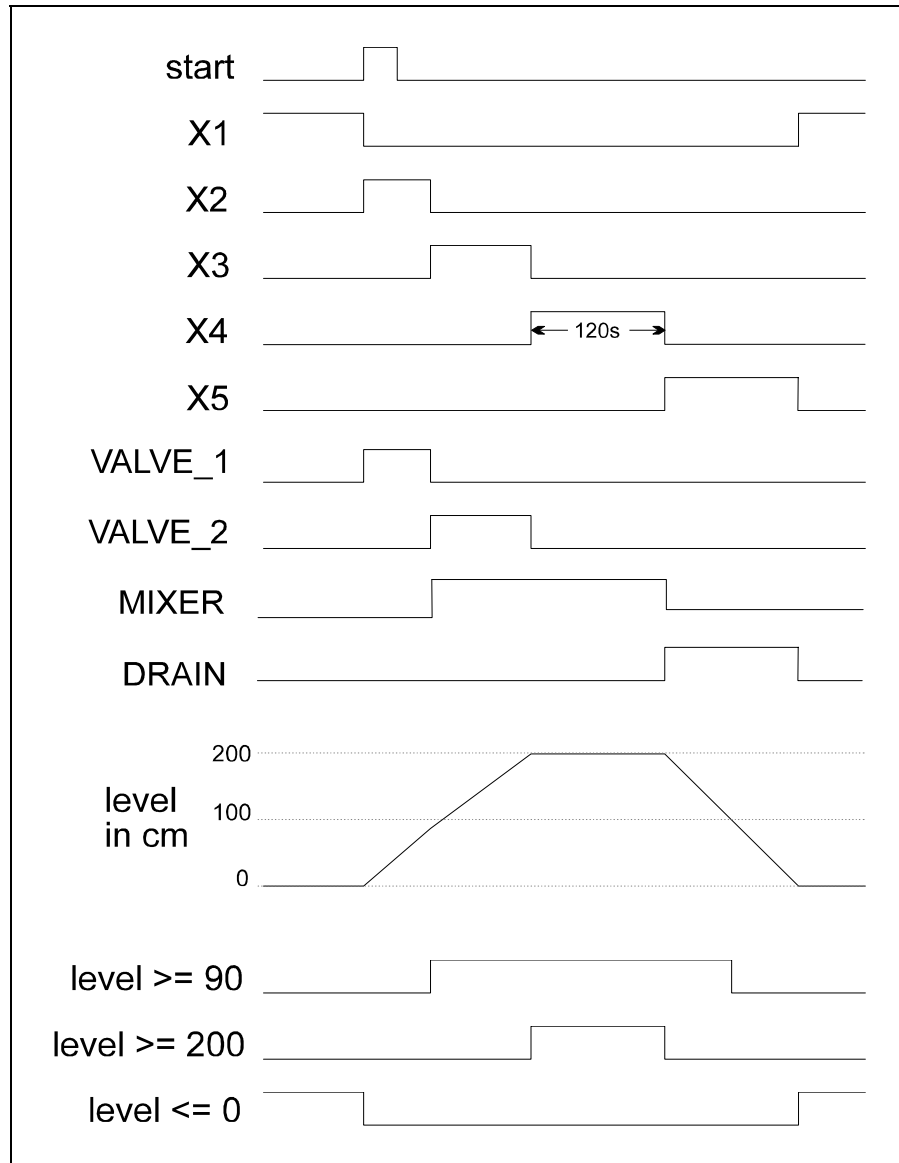


Figure 4.3

Example 2 - On/Off control

In this second example we combine the basic elements of Grafcet to solve the most basic control requirement: On/Off control.

Physical description

The physical plant shown in Figure 4.4 consists of a radial saw powered by an electric motor. The motor is controlled by start and stop buttons. When you push the start button, the saw should start, and when you push the stop button, the saw should stop. Assume that the plant is controlled by a programmable logic controller.

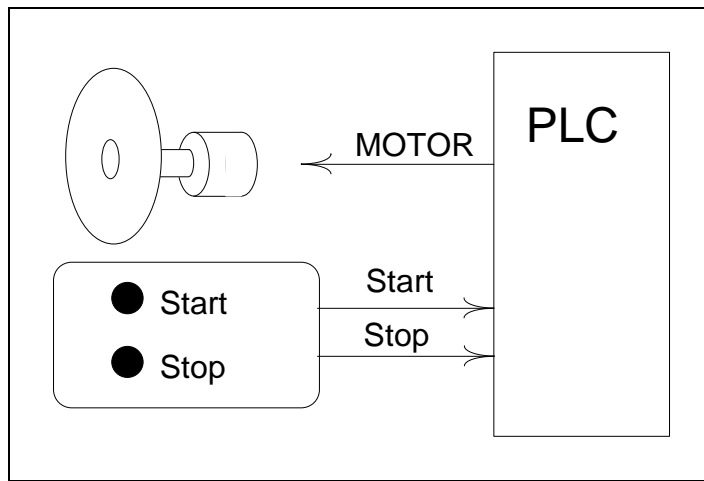


Figure 4.4

Control signal description

The inputs and outputs are as follows:

start	input signal from operator push button
stop	input signal from operator push button
MOTOR	output from programmable logic controller to activate motor

Automation requirement

As mentioned above, the saw has two push buttons, which control the saw's operation. In addition, for the sake of security, we insist also that:

- The stop button takes priority over the start button.
- If the start button is already depressed when the controller is initialized or powered up, the saw should remain stopped, i.e., the saw only comes on when the start button is pressed (upon a rising edge detection of the start signal).

Therefore, the logic expression for starting the saw can be stated in Boolean logic as:

$$\uparrow \text{start AND NOT stop}$$

which would read as *the rising edge of start and not stop*.

The Grafcet solution

Terminology and basic description

The Grafcet diagram that controls the radial saw is shown in Figure 4.5. This diagram is called a *graph* and its title is Graph1. The names of its various Grafcet elements are noted in the figure.

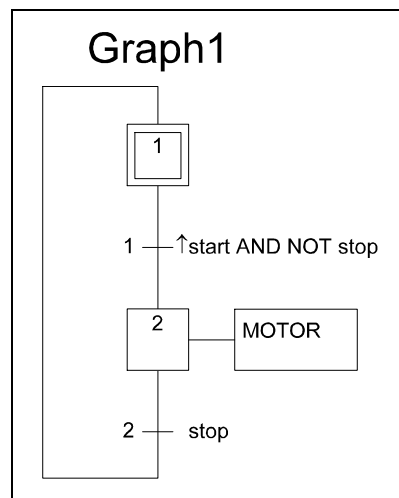


Figure 4.5

Graph1 contains two *steps*, which appear as the numbered boxes. In Grafcet, a step is simply a basic unit in a sequence of instructions. Steps are linked together by directed *links*, indicating that the second step follows the first, but the link must always pass through a *transition*. Links always enter the top of a step and exit from the bottom, so you don't have to draw an arrow head to know which way they go. The transitions appear in

Figure 4.5 as numbered horizontal tick marks. Each transition has a *trigger condition*, or *trigger* for short. The trigger represents the condition for passing from one step to the next.

Step 1 is shown as a double box; it is the initial step. This means that it will be executed as soon as the programmable logic controller starts running. We say that a step is active while it is being executed. When we want to show which step is active, we draw a dot, called a *control token*, inside the active step. Figure 4.6 shows Graph1 while Step 1 is active. Figure 4.7 shows Graph1 while Step 2 is active.

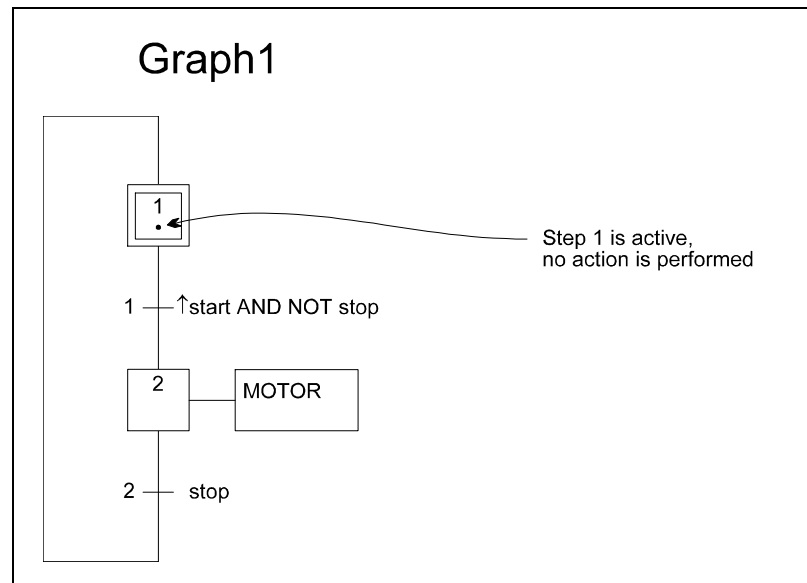


Figure 4.6

A step's action (a textual control statement written on the right-hand side of the step symbol) is executed only when the step is active. The most common type of control statement, *momentary action* statements, give the name of an output signal. For instance, in Figure 4.7, the output signal MOTOR is written beside Step 2. This means that MOTOR will be energized while Step 2 is active and will be de-energized when Step 2 is no longer active. Step 1 has no action statement, meaning that while Step 1 is active, no outputs are energized and the motor is off.

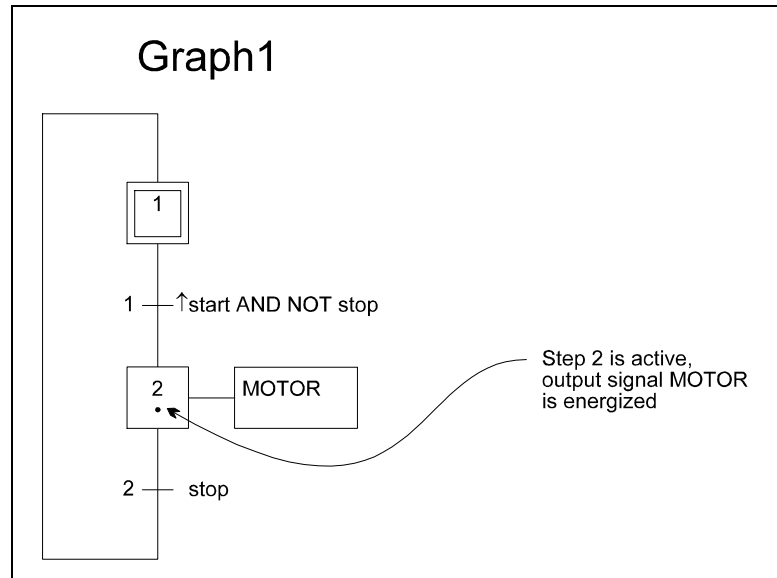


Figure 4.7

Operation of the graph

The step-by-step functioning of the graph is explained below:

Step 1

When the programmable logic controller starts to run, Step 1 is active. Since Step 1 executes no action statement, the motor remains stopped.

While Step 1 is active, we say that Transition 1 is armed. When the trigger expression

- start AND NOT stop

becomes True, Transition 1 fires, deactivating Step 1 and activating Step 2. This happens only if the stop button is not pressed when the operator pushes the start button. When Step 1 is inactive, Transition 1 is no longer armed so it can't fire again even if its trigger is True.

Step 2

While Step 2 is active, the motor starts as output MOTOR is energized, and Transition 2 is armed. As soon as the stop button is pressed, then trigger expression

stop

becomes True and Transition 2 fires. This deactivates Step 2 and activates Step 1 to complete the start/stop cycle. Note that when Step 2 is active, only the input signal *stop* has a possible effect on the state of the graph. The *start* signal is irrelevant to the evolution of the graph while Step 2 is active.

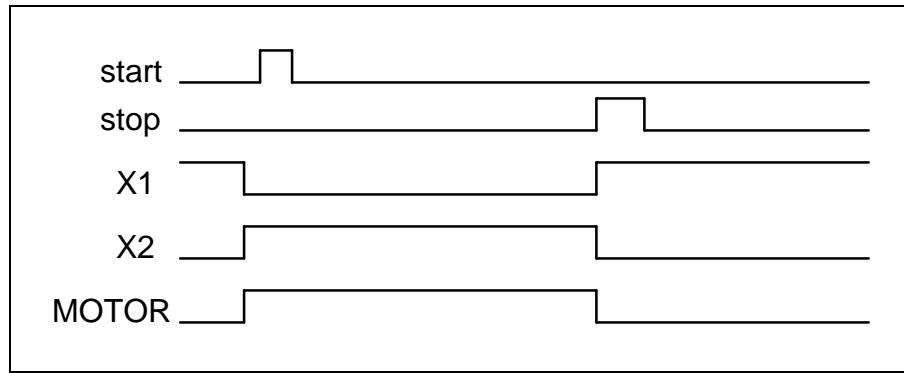


Figure 4.8

The timing chart of Graph1's operation is illustrated in Figure 4.8. We see that a simple two-step graph produces exactly the desired behavior.

Example 3 - On/Off parallel control

Here we build on the previous example to introduce the concept of parallel control within a single graph.

Physical description

The physical plant shown in Figure 4.9 is similar to that in Figure 4.4 except that it has two radial saws, each powered by an electric motor and controlled by its own start and stop buttons. From now on we won't show the PLC; just assume that it is still there.

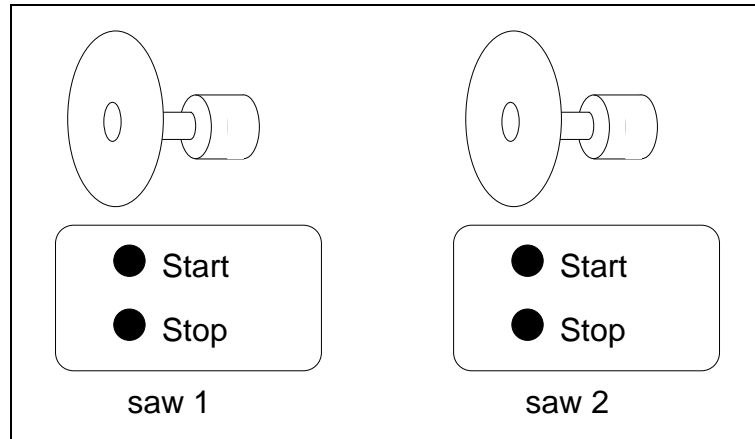


Figure 4.9

Control signal description

The programmable logic controller inputs and outputs are as follows:

start1	input signal from operator push button for motor 1
start2	input signal from operator push button for motor 2
stop1	input signal from operator push button for motor 1
stop2	input signal from operator push button for motor 2
MOTOR1	output from PLC to activate motor 1
MOTOR2	output from PLC to activate motor 2

Automation requirement

The automation requirement is the same as in the previous example except that each motor should be controlled independently by its own set of push buttons. Assume a single programmable logic controller for both saws.

A Grafcet solution

Terminology and basic description

The Grafcet diagram that controls the two radial saws, Graph2, is shown in Figure 4.10. It contains two sub-graphs, one for each motor. The two sub-graphs will evolve independently. This means that either Step 1 or Step 2 in sub-graph 1 may be active, regardless of which step is active in sub-graph 2.

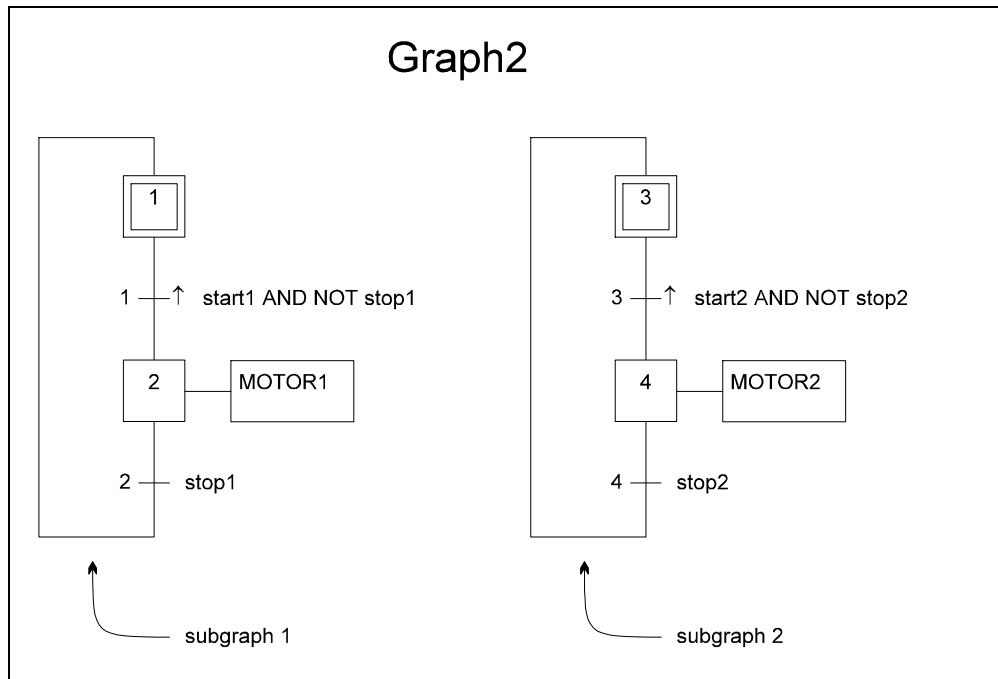


Figure 4.10

The *situation* of a graph tells which steps are active at any given time. If Step 1 and Step 3 are active, the situation is $\{1,3\}$. This situation is shown in Figure 4.11, where control tokens are drawn in Steps 1 and 3. Since both are initial steps, the graph is in its *initial situation*, and we can write:

$$S(\text{Graph2}) = \{1,3\} = \{\mathcal{I}\}, \text{ where } \{\mathcal{I}\} \text{ is the initial situation.}$$

There are a total of four possible situations:

- $\{1,3\}$ as shown in Figure 4.11
- $\{1,4\}$ as shown in Figure 4.12
- $\{2,3\}$ as shown in Figure 4.13
- $\{2,4\}$ as shown in Figure 4.14

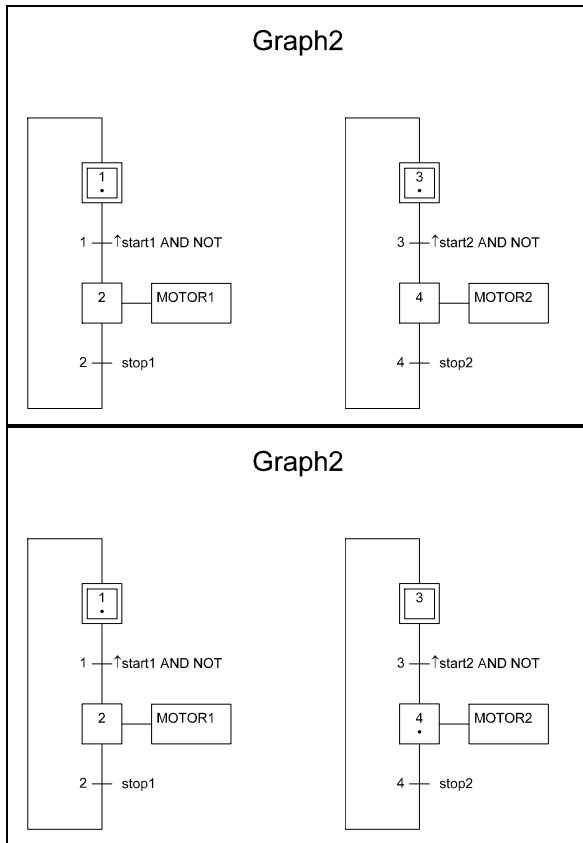


Figure 4.11

Figure 4.12

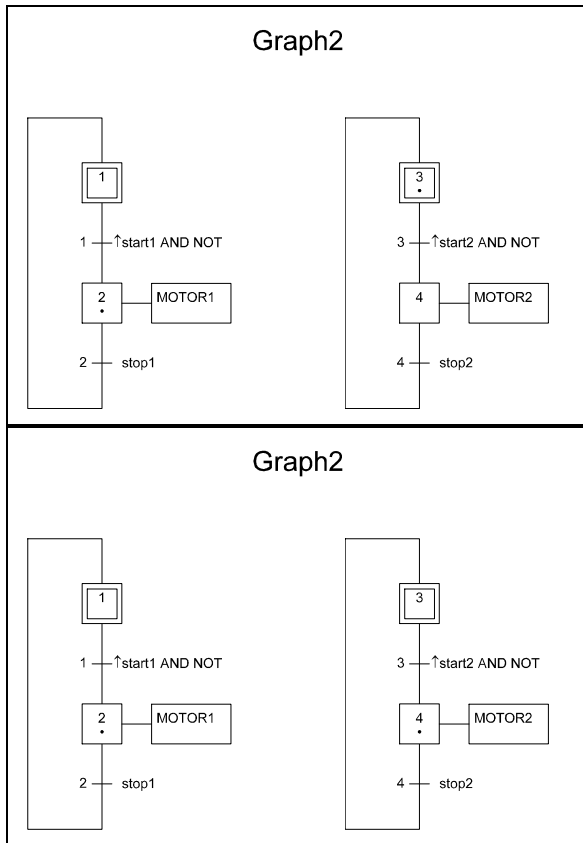


Figure 4.13

Figure 4.14

Operation of the graph

When the programmable logic controller starts to run, steps 1 and 3 are both active. Since they execute no control statements, both motors remain off. The further operation of each of the two sub-graphs works the same way as in Graph1 in Example 2. The only difference is that the two sequences evolve in parallel and each is fully independent of the other.

Example 4 - On/Off multi-graph parallel control

This fourth example presents another way to achieve parallel control, this time using multiple graphs. As we will see, this method has a big advantage--code re-usability.

Physical description

Same plant as in Example 3: two radial saws, each powered by an electric motor and controlled by start and stop buttons.

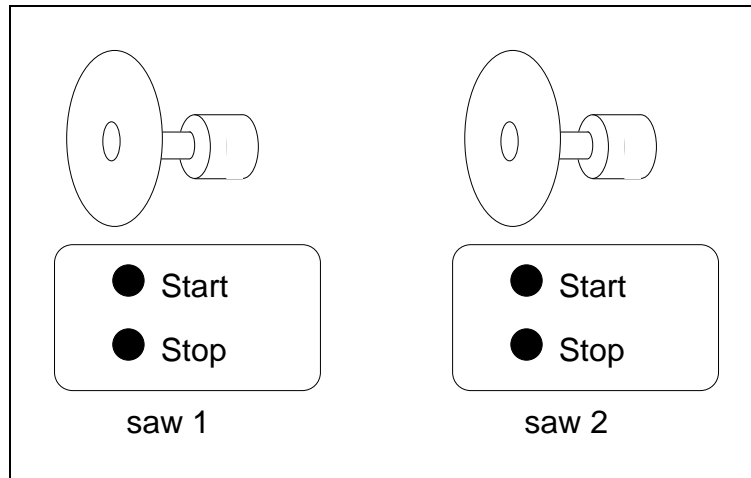


Figure 4.15

Control signal description

This time the programmable logic controller inputs and outputs are named according to an extended format. The first name refers to the Graph where they are used, the second name is used locally within the graph.

Graph3:start	input from operator push button for Motor 1
Graph4:start	input from operator push button for Motor 2
Graph3:stop	input from operator push button for Motor 1
Graph4:stop	input from operator push button for Motor 2
Graph3:MOTOR	output from PLC activates Motor 1
Graph4:MOTOR	output from PLC activates Motor 2

Automation requirement

The automation requirement is the same as in the previous example.

A Grafcet solution

Terminology and basic description

Figure 4.16 shows an *application*. An application is a set of graphs which are linked together to form a single PLC program. Application_1 is made up of two graphs, Graph3 and Graph4. Think of the graphs in an application as functional program blocks within the programmable logic controller as shown in Figure 4.17. Each block uses its own subset of the programmable logic controller inputs and outputs.

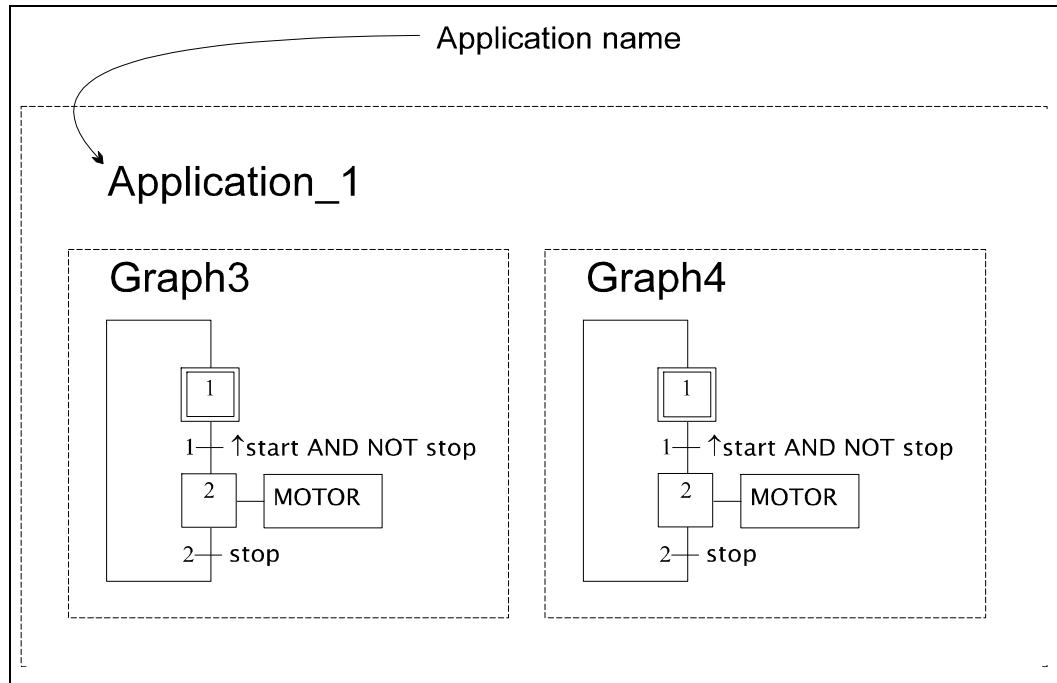


Figure 4.16

Variable names, step numbers and transition numbers are local to each graph (as opposed to global to the entire application). This means that you can re-use the same names without confusion. For instance, *stop* in Graph3 is the stop button input for the first saw, and *stop* in Graph4 is the stop button input for the second saw. From within a graph, we refer to variables simply by their names. From elsewhere in the application we call them by their full names. For instance, the variable *start* in Graph3 would be called *Graph3:start* from another graph.

This naming convention for variables is analogous to first and last names for people. At home, you use first names only. Outside the home, both first and last names are needed.

Operation of the graph

Each of two graphs in Application_1 works the same way as Graph1 in Example 2. The only difference is that now the two graphs evolve in parallel. Each is fully independent of the other.

Modularity and re-usability

The use of mutli-graph applications allows us to build modular code. Modular code is easier to maintain, and a library of standard control programs can be built up. This means that next time we want to use a code module, we don't have to start from scratch. Once you've designed a graph to control the first machine, you simply copy it for the second. The only thing you have to change is your I/O list. This lets you improve your productivity by building up a library of fully re-usable graphs. Conceptually, we can imagine a PLC programmed with modular graphs as shown in Figure 4.17.

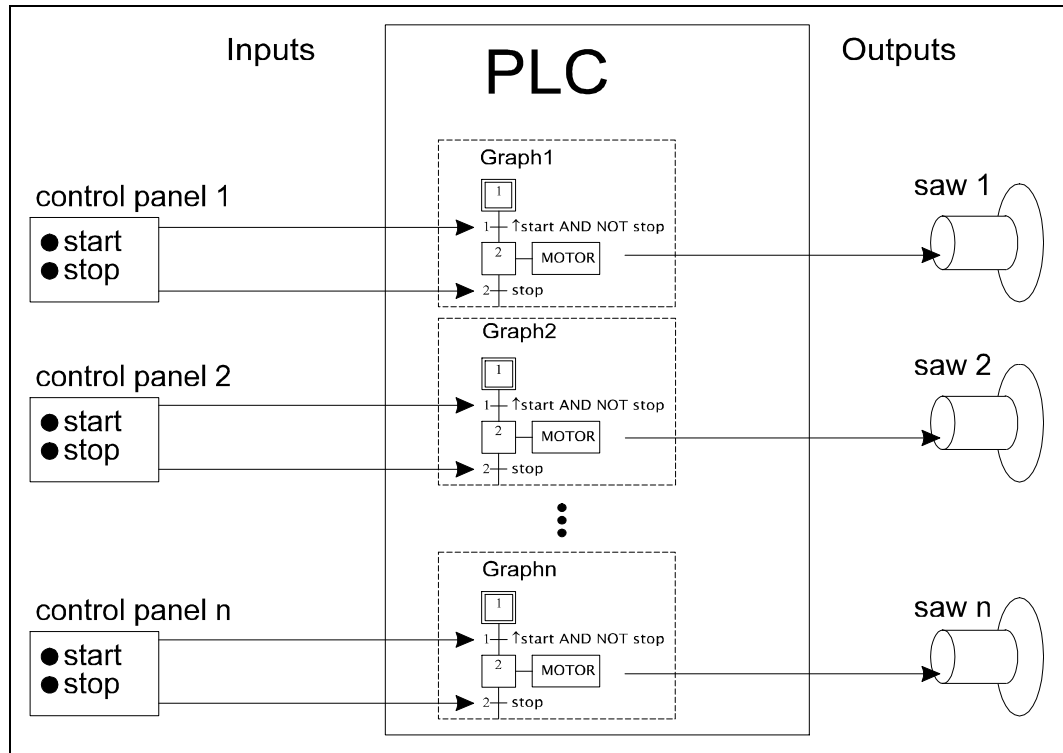


Figure 4.17

Example 5A - Sequential control

In this example, a more complex problem is presented to illustrate the use of a control sequence.

Physical description

The drill press is shown on Figure 4.18. It has two motors, one for spinning the drill bit and the other for raising and lowering the press. Limit switches detect when the drill is at the top or bottom of its course. The operator has two push buttons: Start and Stop.

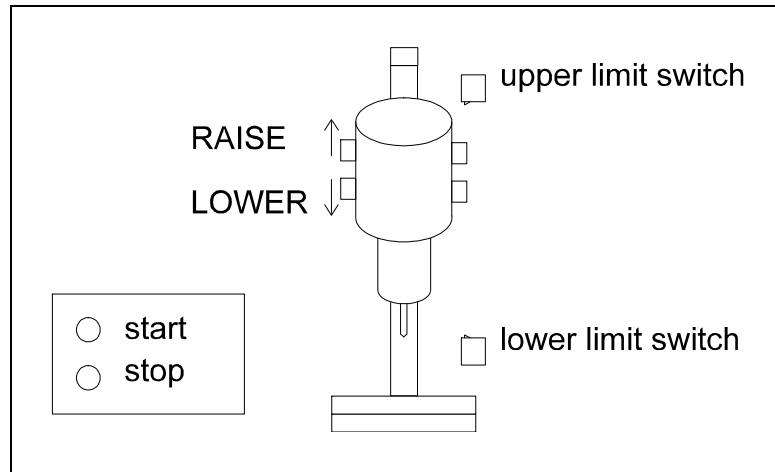


Figure 4.18

Control signal description

The control signals for the drill controller are the following:

start	input from operator push button
stop	input from operator push button
up	input from upper limit switch
down	input from lower limit switch
SPIN	output from PLC to activate drill motor
RAISE	output from PLC to raise drill
LOWER	output from PLC to lower drill

Automation requirement

The automation requirement is as follows:

1. On power-up, raise the drill (if necessary) to the upper limit switch.
2. Wait until the operator presses the start button.
3. Turn on the drill motor and give it 2 seconds to come up to speed.
4. Lower the drill until it reaches the lower limit switch.
5. Raise the drill to the upper limit switch.
6. Go to 2.

A Grafcet solution

Terminology and basic description

The graph shown Figure 4.19 introduces the ideas of the looping sequence with initialization and the OR convergence. Step 1 does the initialization; Steps 2-5 do the looping. The graph also introduces the condition action (in Step 1) and the timer (in Transition 3).

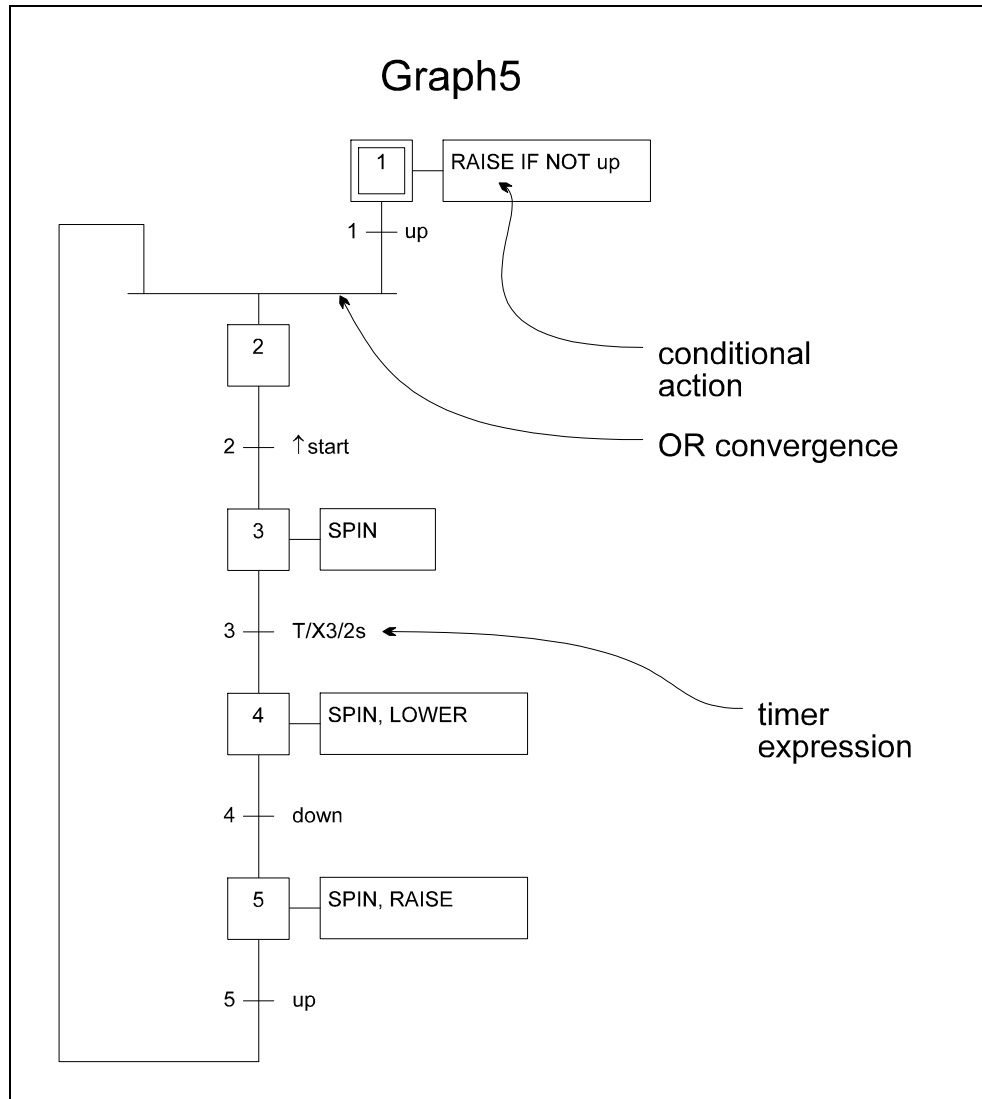


Figure 4.19

Operation of the graph

The graph is a straightforward translation of the automation requirement. On a step-by-step basis, here's how it works:

Step 1

This step is responsible for initializing the system on power-up. When the programmable logic controller is turned on, Step 1 is active. If the drill is not in contact with the upper limit switch, the statement:

RAISE IF NOT up

is executed, activating the motor which raises the drill. This continues until the drill is up, as detected by Transition 1's trigger condition:

up

When Transition 1 fires, the control token passes from Step 1 to Step 2, activating Step 2.

Step 2

This step waits for the operator to push the start button. No action is executed, i.e., none of the outputs is energized. Control passes through Transition 2 when it is triggered by the expression:

$\uparrow \text{start}$

The rising edge operator, \uparrow , ensures that the operation doesn't start until the operator has positively pushed the start button. This prevents the machine from cycling continuously if the button is held down, or from starting unexpectedly when electric service resumes after a power failure.

Step 3

This step starts the drill and lets it come up to speed for 2 seconds. Note that the statement

SPIN

is repeated in the actions of steps 3-5. This means that the drill motor output of the programmable logic controller will be energized when any of these steps is active. Control passes to Step 4 when Transition 3 is triggered by:

T/X3/2s

This is an example of a Grafset timer. It reads as:

2 seconds after the activation of Step 3

Transition 3 will therefore fire when the 2 second period is over.

Step 4

Step 4 executes two statements. The first, SPIN, maintains the output to the drill motor. The second, LOWER, lowers the drill motor. The drill is lowered until the *down* signal is received from the lower limit switch. Control passes through Transition 5 to Step 5 when the drill reaches the bottom of its course.

Step 5

Step 5 executes two statements. The first, SPIN, maintains the output to the drill motor. The second, RAISE, raises the drill motor. The drill is raised until the *up* signal is received from the upper limit switch. Transition 5 fires when the drill has reached the top of its course, passing the control token back to Step 2. This completes the control cycle.

A reduced graph

On careful examination of Graph5 in Figure 4.19, you can see that steps 1 and 2 can be merged to yield a more compact graph as shown in Figure 4.20. Remember that id numbers for steps and transitions don't have to start with 1 and don't even have to be sequential. Graph5 and Graph6 are functionally identical. This illustrates an important aspect of control programming: there is more than one correct solution to a given

problem, and some solutions are more compact and more readable than others. In general, it is a good idea to use the solution that is easiest to understand; this reduces the probability of designing an incorrect graph that doesn't do what you really wanted.

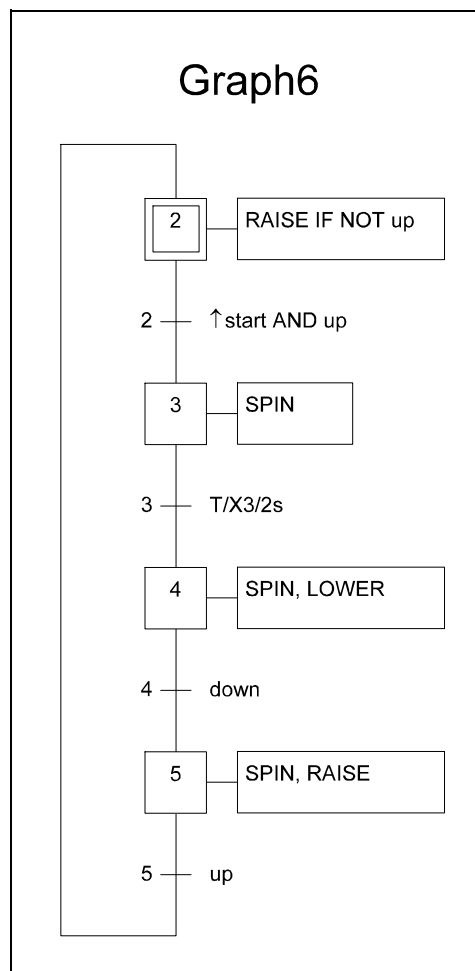


Figure 4.20

Example 5B - Sequential control with fault handling

So far we have assumed that the plant will always behave the way it is supposed to, but we all know that real life isn't like that. In this example, we re-examine Example 5A to show how to handle fault conditions.

Physical description

The same drill press as before, but now an alarm bell has been added to alert the operator to problems.

Control signal description

The same control signals as before, but now the programmable logic controller has another output, ALARM, which energizes the alarm.

Automation requirement

The basic automation requirement is still as follows:

1. On power-up, raise the drill (if necessary) to the upper limit switch.
2. Wait until the operator presses the start button.
3. Turn on the drill motor and give it 2 seconds to come up to speed.
4. Lower the drill until it reaches the lower limit switch.
5. Raise the drill to the upper limit switch.
6. Go to 2.

But now we want to add a few more conditions:

- A. If the stop button is pressed at any time, stop the drill motor and raise the drill.
- B. If the drill is in motion but doesn't reach the limit switch within 15 seconds, stop both motors, sound the alarm for 2 seconds and wait for the operator to press the start button before re-starting the cycle.

A Grafcet solution

Terminology and basic description

The graph shown Figure 4.21 builds upon Figure 4.20 and introduces the following Grafcet elements:

- Stand-alone action
- Forcing statement
- OR divergence

A stand-alone action is an action which is not connected to a step. Recall that actions are only executed when their step is active. A stand-alone action is always scanned. In Graph7, the stand-alone action:

F/Graph7:{2} IF stop

is a conditional forcing statement. If push button signal *stop* is True, then this statement forces the graph to Step 2, i.e., it activates Step 2 and deactivates all other steps.

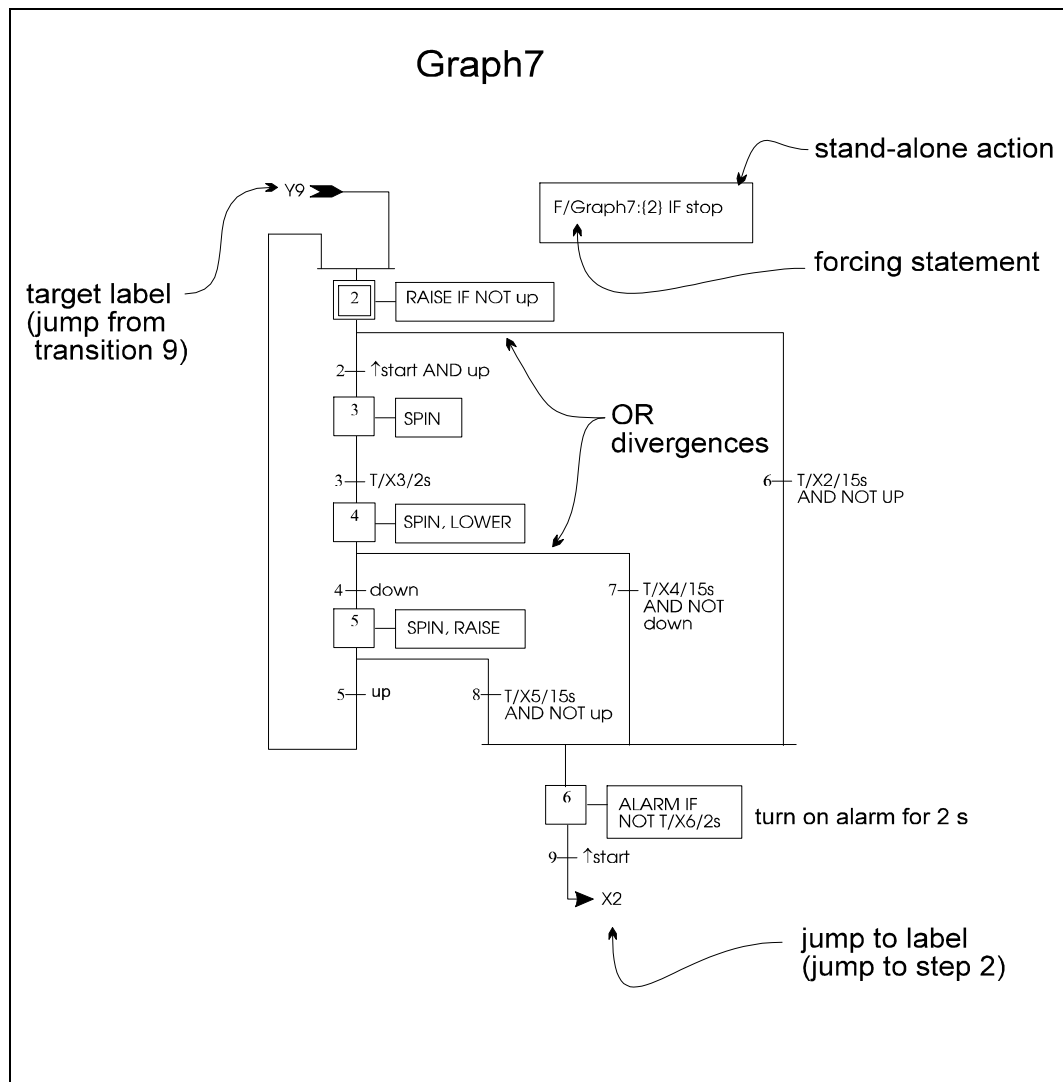


Figure 4.21

The convergences and divergences are simply branches and merges in the directed links. When a step is linked to multiple transitions by an OR divergence, it arms all the transitions and the control token can pass through any of the transitions. When a step is linked from multiple transitions by an OR convergence, it can be activated if any of the transitions fire. Graph7 also introduces the *jump to label* symbol. To avoid cluttering up the diagram, links may be replaced with a jump to label. This is purely for aesthetic reasons, as the jump to label works the same way as any other directed link.

Operation of the graph

The operation of Graph7 is as follows:

Stand-alone action

This action, which is always scanned, looks after the requirement of re-initializing the graph whenever the stop button is pressed. Since it is a stand-alone action, it forces the graph back to Step 2, no matter what step is active when the button is pressed. This looks after condition A specified under Automation Requirement.

Step 2

This initial step is responsible for initializing the system on power-up as in the previous case. Assuming all goes well, within a few seconds the *up* signal will appear, indicating that the drill is up. When the operator pushes the start button, control will pass through Transition 2 to Step 3, continuing the normal control sequence. On the other hand, if 15 seconds go by without the *up* signal being present, obviously something is wrong. Transition 6 looks after this condition. It will fire when triggered by:

T/X2/15s AND NOT up

i.e., the drill still isn't up 15 seconds after Step 2 is activated. When Transition 6 fires, the control token passes from Step 1 to Step 6. Step 6 acts as an error trap. This looks after condition B specified under Automation Requirement.

Note that both Transition 2 and Transition 6 are armed by Step 2 by means of the OR divergence. Their triggers are mutually exclusive Boolean expressions, so both can't fire at once. Beware, the OR divergence is inclusive OR. One or the other or both transitions may fire, so always make sure that the triggers are mutually exclusive.

Step 3

As in the previous example, this step starts the drill and lets it come up to speed. Transition 3 fires after 2 seconds and control passes to Step 4.

Step 4

Step 4 maintains the output to the drill motor and lowers the drill motor. The drill is lowered until the *down* signal is received from the lower limit switch or until a 15 second time-out has expired as in Step 2. Control passes through either Transition 4 to Step 5 or Transition 7 to Step 6.

Step 5

Step 5 maintains the output to the drill motor and raises the drill motor. The drill is raised until the *up* signal is received from the upper limit switch or until 15 seconds have elapsed, as in Step 2. Control passes through either Transition 5 to Step 2 or Transition 8 to Step 6.

Step 6

Step 6 acts as a fault trap. It can be activated by any of transitions 6-8 through its OR convergence. Thus it traps the control token in any case where the graph attempts a control action without receiving the appropriate response from the plant within the prescribed time. The alarm sounds for two seconds to alert the operator that there is a problem. The normal sequence resumes when the operator has corrected the problem and pressed the start button to trigger Transition 9. This completes the additional automation requirement B.

Note how time limited action is generated from a simple timer expression by the statement:

ALARM IF NOT T/X6/2s

The timer returns to False for two seconds after the activation of Step 2. Since NOT False = True, ALARM will be energized for two seconds as shown in Figure 4.22.

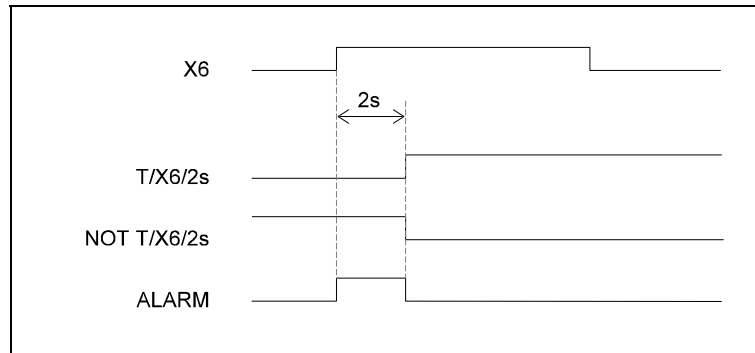


Figure 4.22

Example 6 - Synchronizing parallel sequences

In this example, a more complex problem is presented to illustrate the use of synchronized control sequences, and to introduce the remaining Grafcet elements.

Physical description

The physical plant in this example consists of a drilling station used to manufacture furniture. With two automated drill presses, as shown in Figure 4.23, it can double the production speed of certain parts by drilling two holes at once. Each drill press is like the one in the previous example. A pneumatic clamp has been added to maintain the part in position during drilling. The operator still has two push buttons, Start and Stop, and an alarm to warn of problems.

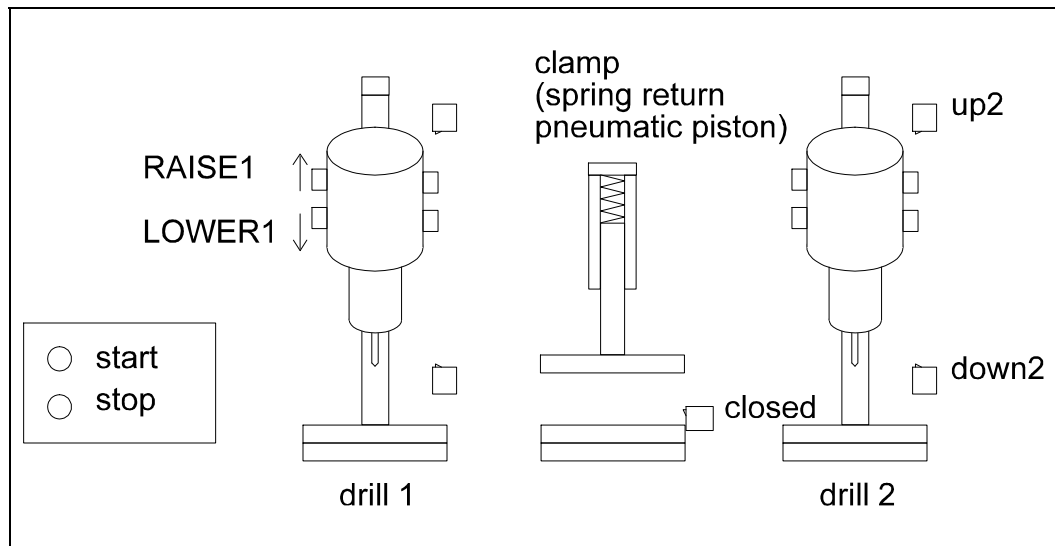


Figure 4.23

Control signal description

The control signals for the drill controller are the following:

start	input signal from operator push button
stop	input signal from operator push button
closed	input signal indicating clamp is closed
CLAMP	output from PLC to extend clamp pistons
up1	input signal from upper limit switch, drill 1
down1	input signal from lower limit switch, drill 1
SPIN1	output from PLC to activate drill motor 1
RAISE1	output from PLC to raise drill 1
LOWER1	output from PLC to lower drill 1
up2	input signal from upper limit switch, drill 2
down2	input signal from lower limit switch, drill 2
SPIN2	output from PLC to activate drill motor
RAISE2	output from PLC to raise drill 2
LOWER2	output from PLC to lower drill 2
ALARM	output from PLC to sound alarm

Automation requirement

The automation requirement is as follows:

1. On power-up, raise drills (if necessary) to the upper limit switches.
2. Wait until the operator places a part in the work area and presses the start button.
3. Extend pistons until the clamp is closed.
4. Cycle both drills through the same cycle as the previous example.
5. Open the clamp when both drills have returned to the raised position.
6. Go to 2.

While doing the above, trap any faults and sound the alarm as in the previous example.

A Grafcet solution

Terminology and basic description

The first thing you'll notice about Graph8 in Figure 4.24 is that it contains two parallel sequences, one for each drill. Grafcet's most powerful feature is its ability to show parallel control sequences like this one in a natural and obvious way.

Two new symbols are introduced to handle parallel branching and synchronization: the *AND convergence* and the *AND divergence*.

An AND divergence symbol is a double horizontal line below a transition (below Transition 2 in Graph8). The double line indicates parallelism. The AND divergence lets a transition activate more than one step when it fires. This starts executing the parallel sequences.

An AND convergence, shown by a double line immediately above a transition (above Transition 9 in Graph8), means that the transition is armed only if all the steps linked to its entrance are active. When it fires, it deactivates all of them.

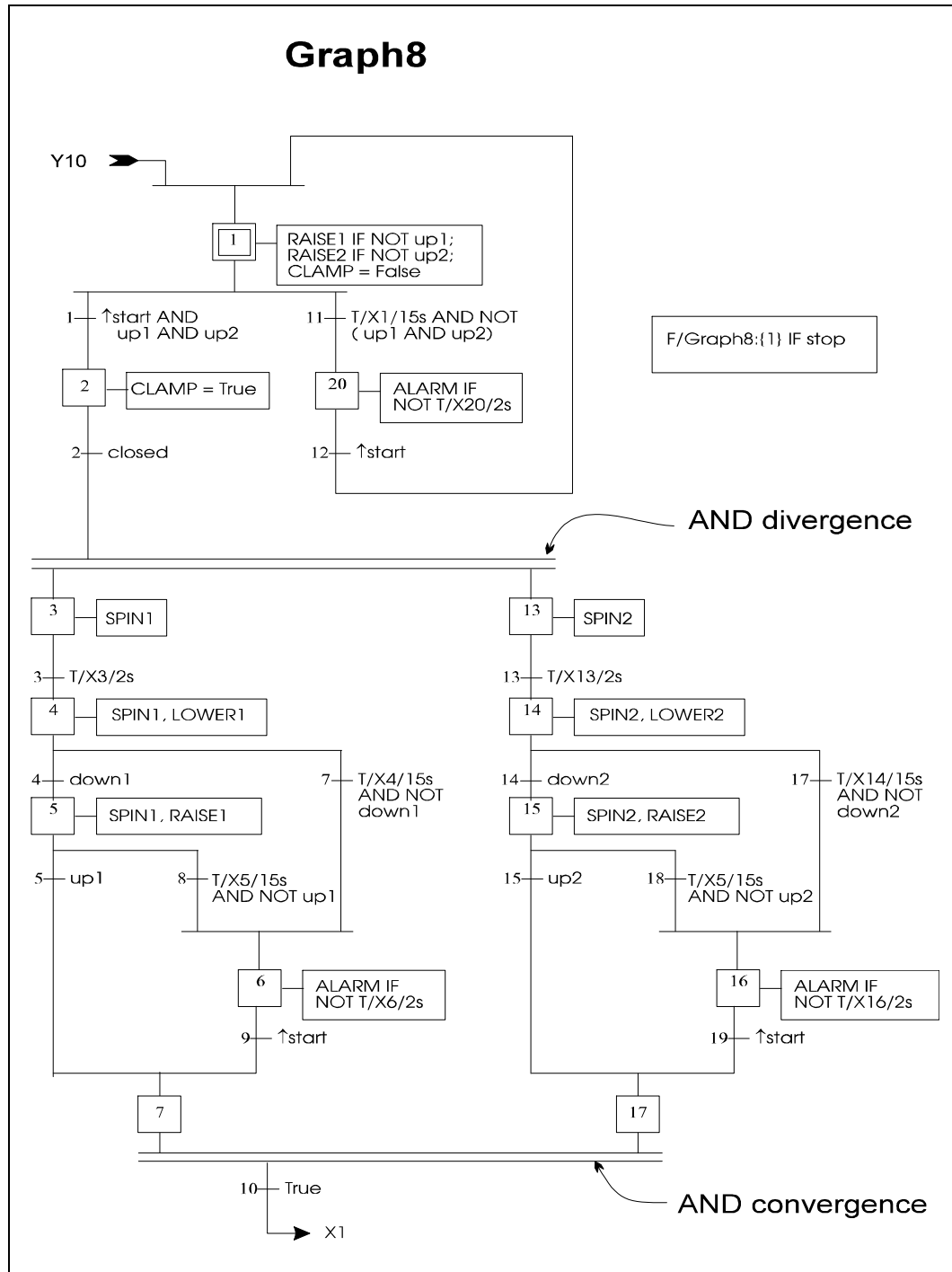


Figure 4.24

Operation of the graph

Here is how it works, step by step:

Step 1

This step is responsible for initializing the system on power-up. When the programmable logic controller is turned on, Step 1 is activated. If either drill is not in contact with its upper limit switch, the drill is raised. If 15 seconds elapse and the two drills are not in the correct position, then Transition 11 fires, passing control to the fault trap, Step 20. When the drills are positioned and the operator presses the start button, control passes through Transition1 to Step 2.

Step 2

This step closes the clamp. The clamp is powered by a spring-return pneumatic piston, so the output CLAMP must also be energized throughout both drill sequences to hold the clamp closed. That is why the stored action "CLAMP = True" is used in Step 2. Step 2 remains active until the *closed* signal is received, triggering Transition 2. Because Transition 2 is followed by an AND divergence, when it fires, it activates both Steps 3 and 13. This activates the two parallel control sequences. The evolution of these sequences then proceeds independently. Note that while the parallel sequences are active, 25 different situations are possible:

{3,13}, {3,14}, {3,15}, {3,16}, {3,17}
{4,13}, {4,14}, {4,15}, {4,16}, {4,17}
{5,13}, {5,14}, {5,15}, {5,16}, {5,17}
{6,13}, {6,14}, {6,15}, {6,16}, {6,17}
{7,13}, {7,14}, {7,15}, {7,16}, {7,17}

Since 25 situations are possible using only 10 steps, Grafcet is a more compact way of programming than methods that don't allow parallelism (for instance, flow charts and state languages).

Step 3

Step 3 starts Drill 1, letting it come up to speed for 2 seconds.

Step 4

Step 4 maintains the Drill Motor while lowering the drill. If the *down* signal is received from the upper limit switch within 15 seconds, then Transition 4 will be fired, passing control to Step 5. Otherwise, Transition 7 fires, passing control to the fault trap, Step 6.

Step 5

Step 5 maintains Drill Motor 1 while raising the drill. If the *up* signal is received from the lower limit switch within 15 seconds, then Transition 5 fires, passing control to Step 7. Otherwise, Transition 8 fires, passing control to the fault trap, Step 6.

Step 6

Step 6 acts as a fault trap. It can be activated through its OR convergence by transitions 7 and 8. Thus it traps the control token in any case where the sequence attempts a raise or lower action without receiving the appropriate response from the limit switches within the prescribed time. The alarm sounds for two seconds to alert the operator that there is a problem. The normal sequence resumes when the operator has corrected the problem and presses the start button.

Step 13 - Step 16

These steps execute the same control actions for Drill 2 as Step 3 - Step 6 did for Drill 1.

Step 7 and Step 17

Step 7 and Step 17 are wait steps. They do not execute any actions. If step sequence 3-6 finishes execution before step sequence 13-16, then Step 7 waits for the other sequence to catch up, and vice versa. Remember that the clamp shouldn't open until both drills have finished.

Transition 10

Transition 10 is not armed unless both Step 7 and Step 17 are active, i.e., until both sequences have finished. Once armed, it fires instantly because its trigger expression

True

is always True. When Transition 10 fires, it deactivates both Step 7 and Step 17 and passes control back to Step1.

Step 20

Step 20 is an error trap for the actions in Step 1. If they have not been accomplished within 15 seconds of Step 1's activation, control passes via Transition 11 to Step 20 where the alarm is sounded.

Conclusion

We have now presented all of the basics, and you are now ready to start using Grafset yourself. Your skill at defining and implementing industrial control applications will increase with practice.

Compared to the traditional relay ladder language used for programming a PLC, Grafset tends to be more compact and easier to understand. To illustrate this, a relay ladder language solution for Example 6 is presented in Appendix B (at the end of the book). The Grafset advantage is clear. You can tell at a glance what the Grafset diagram does. Most people will agree that it isn't so easy with the ladder diagram.

5 Elements of Grafcet

This chapter presents a concise definition of Grafcet. The definition considers Grafcet as a model of computation, i.e., as a method for structuring a real-time (also called a *reactive*) program for a PLC or an industrial computer.

Grafcet specifies control using diagrams called graphs according to a standard format. Just a few basic symbols specify an unlimited variety of control sequences, while simple textual statements specify the details of control procedures and algorithms.

Organization and modularity

Graphs

Grafcet is a modular control specification method where the basic module is a diagram called a graph. Think of a graph as a single page drawing showing graphical and textual control elements: steps, transitions, links and statements. A complete specification is called an application. It can include more than one page, that is, more than one logically distinct graph. The graphs are usually arranged in a hierarchical structure, with the higher level graphs controlling the lower level graphs.

Formally, a graph is defined as a set of Grafcet elements: steps, transitions, links and statements. A concept frequently confused with the graph is the connected sub-graph. Because a graph is defined as a set of elements, any subset of those elements is a sub-graph. Even the empty set is a graph!

The steps and transitions of a graph or sub-graph are not necessarily connected to each other by links.

A connected sub-graph is a subset containing all the steps and transitions such that there is a path from one element to another along a link.

The idea of connected and non-connected graphs is best illustrated by an example shown in Figure 5.1. Graph1 is a connected graph. Graph2 is non-connected graph, but it is made up of two connected sub-graphs. A sub-graph is simply a part of a graph.

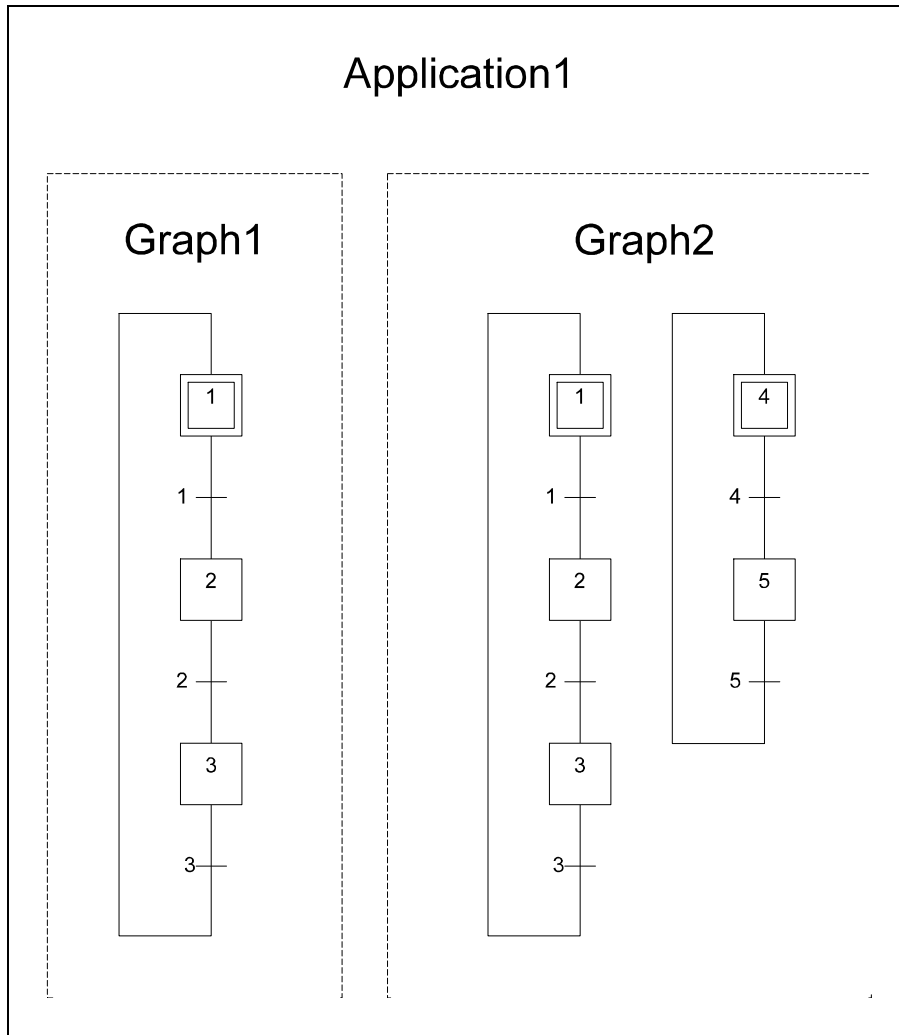


Figure 5.1

Applications

When using Grafcet for programmable logic controllers, an application consists of all the graphs running on a single controller. In a large PLC there may be hundreds of graphs running concurrently.

Variables and nomenclature

Rather than using logical or physical addresses, Grafcet lets you refer to inputs, outputs and internal storage registers by meaningful names. These are called *variable names* or *tag names*.

Types

Inputs and outputs are the physical I/O points of the controller. There are two common types of I/O: Boolean (On/Off) and numeric (0-10V, $\pm 10V$, 4-20mA, etc.). Most PLCs operate by polling, i.e., they follow a three step scan cycle:

1. Scan all physical inputs, copying the values into internal registers called the *image table* or the *I/O image*. Boolean inputs are stored as bits. Analog inputs are stored as numeric values.
2. Execute the control calculations, using internal registers.
3. Write the calculated outputs to the physical outputs.

To deal with these operations, all controllers provide at least two types of internal variables, Boolean and integer numeric. Other useful types, such as floating point, table, and set, are sometimes used. In this manual, we will limit the discussion to these two basic types.

Valid names

A name is a sequence of letters or digits, starting with a letter which represents an I/O point or an internal variable. Single letter names are OK. So are names with underscores and points. Here are some examples of valid names:

- A
- B
- B43
- temperature15
- Open_valve_36B
- open_sesame

It's up to you to choose meaningful names that improve the readability of your graph. Obviously, *Open_valve_36B* is preferable to *B12R14QLNX*.

Scope

Scope defines whether a variable can be used only in a single graph, (its scope is local), or whether it can be used throughout the application, (its scope is global). Modular programming uses local variables for greater re-usability of code. Although Grafcet's founding standard, NFC 03-190 makes no mention of scope, most software packages allow the use of local variables.

Case sensitivity

Grafcet's founding standard, NFC 03-190 makes no mention of case sensitivity. In other words, it doesn't say whether *TOTAL*, *Total* and *total* should be considered as one name or as three distinct names. However, it is a very good habit to always spell your tag names the same way, even if you use case-insensitive software, just for the sake of consistency.

Reserved words

Certain valid character strings cannot be used as the names for I/O points or internal variables because they already have a special meaning. These are called *reserved words*. The following is a list of Grafcet's reserved words and their meanings:

AND	Boolean conjunction operator
OR	Boolean disjunction operator
NOT	Boolean disjunction operator
True	Boolean True, i.e., ON or ONE
False	Boolean False, i.e., OFF or ZERO
XO, XI, X2,	Step bits, True when step is active, False when step is inactive
IF	conditional action operator

Particular implementations of Grafcet may use slightly different reserved words or may slightly modify the rules for creating new names. For instance, Famic Inc.'s CADEPA reserves the name *.INC.* as the incrementation operator and doesn't allow a period (.) to appear in names. ACS Automation's SYLGRAF reserves the words *set*, *clr* and *PID* as special operators. And Hydro-Quebec's COMPAUT reserves *CMDE*, *ET*, *OU*, and *NON*. When you start using a new Grafcet software package, check the manual to see what names are valid.

Naming conventions

A voluntary rule for choosing tag names can reduce the effort required to understand an application. In this book we use lower case for inputs and internals, and upper case for outputs, so you can tell them apart more easily.

For example:

limit_switch_1	Boolean input
total	numeric internal variable
OPEN_MAIN_VALVE	Boolean output
SPEED_CNTRL_MOTOR3	numeric output

Steps

The step is the element which performs most control actions and calculations.

Drawing and identifying steps

The symbol for a step is a square box as shown in Figure 5.2. The top of the step is its *entrance* and the bottom of the step is its *exit*.

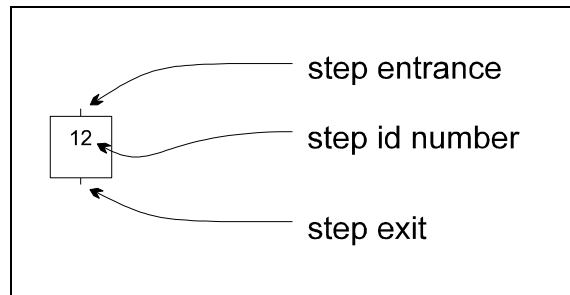


Figure 5.2

The entrance and exit are used as connections for linking the steps and transitions to form the structure of a graph. A step can only be connected to transitions, never directly to another step.

Each step is identified by a number printed inside the box. Step numbers have local scope, i.e., within an application the same step number can be reused. For instance, in Figure 5.1, Graph1 and Graph2 each contain a Step 3. To distinguish between them, we refer to *Graph1, Step 3* and *Graph2, Step 3*, respectively.

Macro steps

Macro steps are now in common use, but are not yet included in the current standards. Macros don't add any functionality to Grafcet, but they allow you to hierarchically

organize a complex graph. The idea is that a part of a complex graph can be moved to another page of paper, and a special symbol used to show that it has been moved.

The symbol is the macro step symbol shown in Figure 5.3. The macro step actually represents a connected sub-graph having one entrance and one exit.

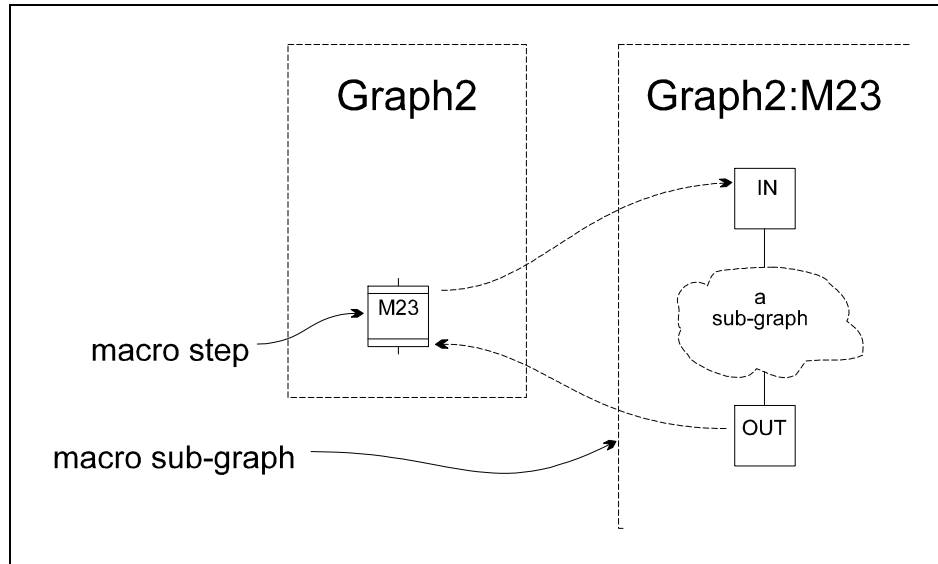


Figure 5.3

The macro step is divided into three regions. The top region represents the IN step of the sub-graph. The bottom region represents the OUT step of the sub-graph, and the center region represents the intervening steps of the sub-graph.

A detailed example is shown in Figure 5.4. It will help your understanding of macro steps to know that the graphs shown in Figures 5.4 and 5.5 function identically.

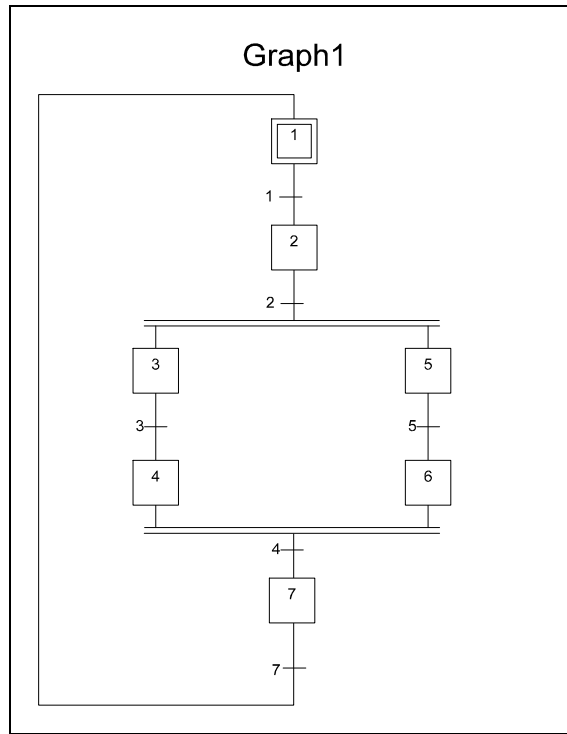


Figure 5.4

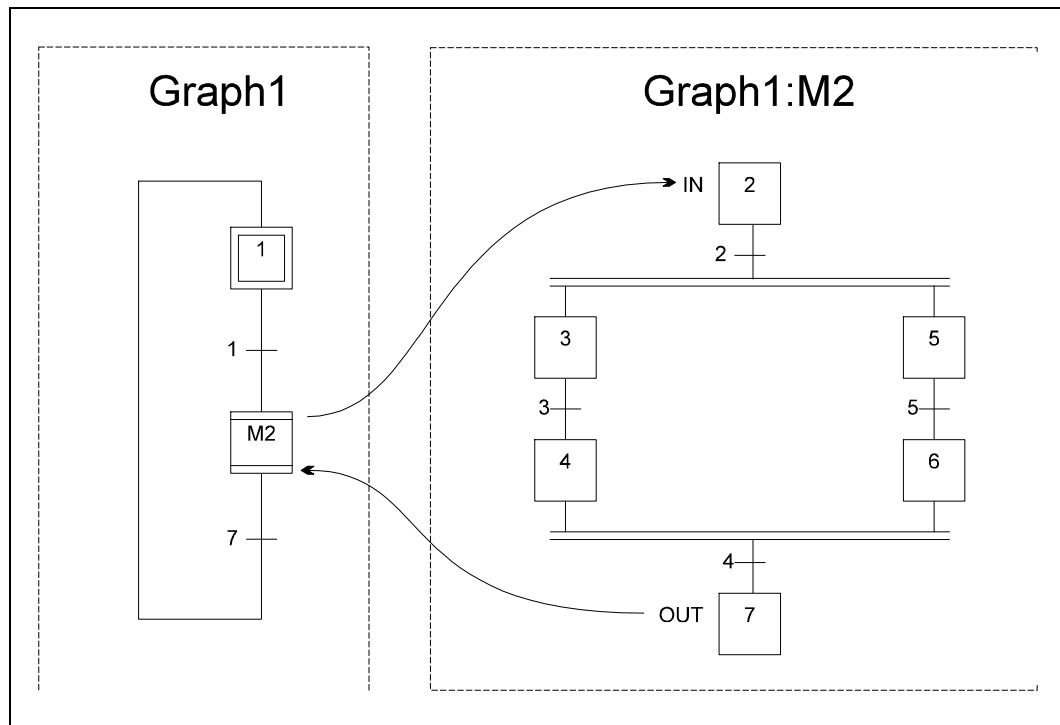


Figure 5.5

Control tokens

We say that *control enters a step* through the entrance, activating the step. While a step is active, its corresponding action is executed. An active step is marked with a dot called a control token, as shown in Figure 5.6. A step is deactivated when control leaves through the exit.

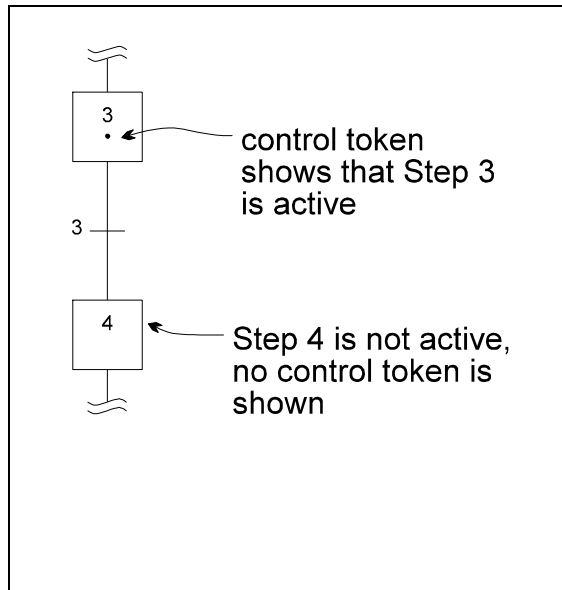


Figure 5.6

Initial steps

Grafcet lets you specify that a step should be activated when the graph is initialized. An initially activated step is called an *initial step*. It is drawn with a double box to distinguish it from a normal step, drawn with a simple box. In Graph1 shown in Figure 5.7, Steps 2 and 4 are initial steps. Note that initial steps do not have to be at the top of a connected sub-graph.

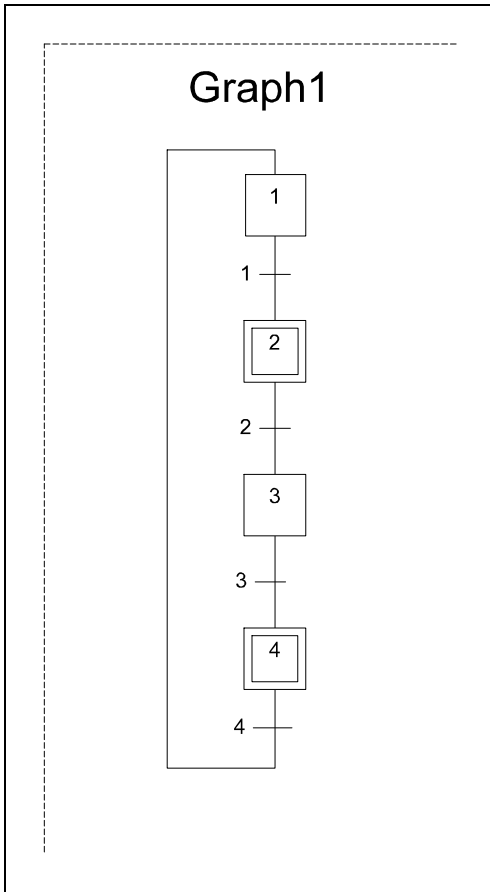


Figure 5.7

Situation of a graph

At any given time, the set of active steps in a graph is its *situation*. In Figure 5.8, Graph1 has tokens in steps 3, 5, 9, and 11. Its situation is therefore:

$$S(\text{Graph1}) = \{3, 5, 9, 11\}$$

When only the initial steps of a graph are active, the graph is in its *initial situation* and we can write $S(\text{Graph}) = \{1\}$ as an abbreviation. Graph2 in its initial situation. In this example we say:

$$S(\text{Graph2}) = \{1\} = \{1, 3\}.$$

Graph3 in Figure 5.8 has no active steps. In this case we say that Graph3 is in the *empty situation* and we can write this situation as $S(\text{Graph3}) = \{\}$, the empty set.

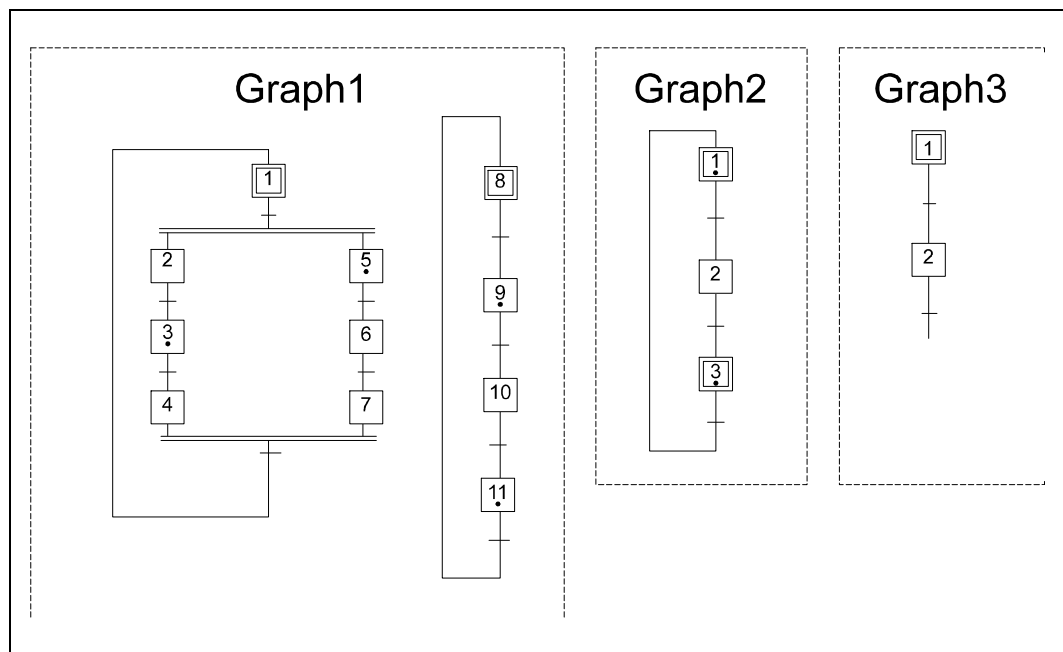


Figure 5.8

When you initialize a graph, you force it to the state where only the initial steps are active, without changing the state of internal variables or output points. Note that initializing a graph and initializing a PLC are not the same thing. Initializing the PLC initializes all the graphs in the application and also clears all internal variables and output points.

Actions

A step usually has some corresponding control action. The action consists of control code written in a textual language. A step's action is executed whenever the step is active, and

is ignored otherwise. The preferred format is to write the text within a rectangle drawn to the right of the associated step, as shown in Figure 5.9.

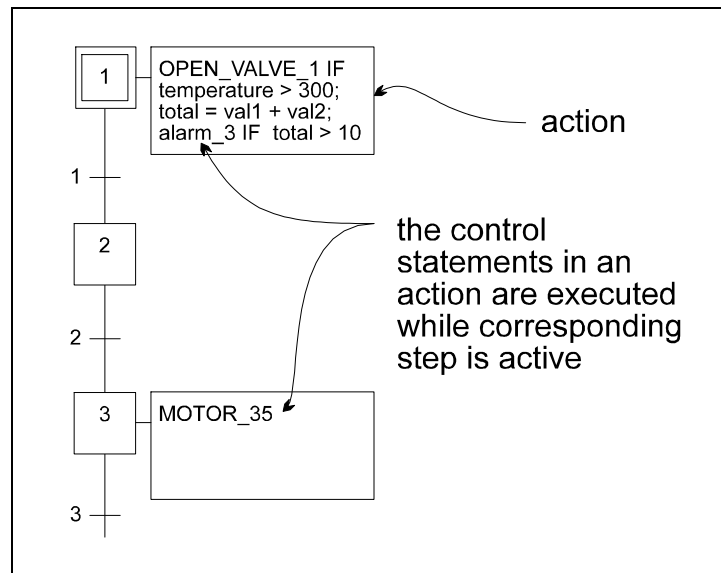


Figure 5.9

Note that it is not necessary for each step to have an action. Steps where the graph is waiting for an external event often have no explicit action. For example, Step 2 in Figure 5.9 has no action.

OR convergence and OR divergence

The entrance of a step may be connected to zero, one, two or more transitions, called the step's *entrance transitions*, as shown in Figure 5.10. If there are two or more entrance transitions, a horizontal entrance expander symbol is drawn. The more usual name for this symbol is the *OR Convergence*. For example, in Figure 5.10, the OR Convergence means that Step 6 can be activated by a control token entering from Transition 14 *or* Transition 15 *or* Transition n. In Figure 5.10:

- Step 3 has 0 entrance transitions.
- Step 4 has 1 entrance transition.
- Step 5 has 2 entrance transitions.
- Step 6 has n entrance transitions.

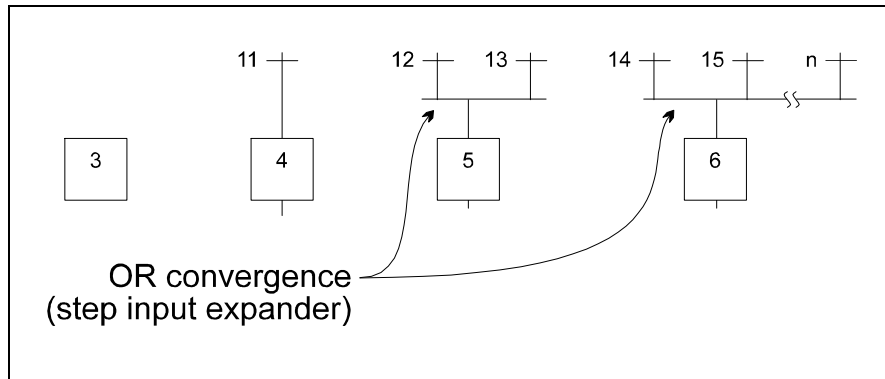


Figure 5.10

Similarly, the exit of a step may be connected to zero, one, two or more transitions, called the step's *exit transitions*, as shown in Figure 5.11. If there are two or more exit transitions then a horizontal exit expander symbol is drawn. This symbol is usually called the *OR Divergence*. In Figure 5.11, Step 16 can be deactivated by a control token exiting through Transition 24 *or* Transition 25 ... *or* Transition m. In Figure 5.11,

- Step 13 has 0 exit transitions.
- Step 14 has 1 exit transition.
- Step 15 has 2 exit transitions.
- Step 16 has m exit transitions.

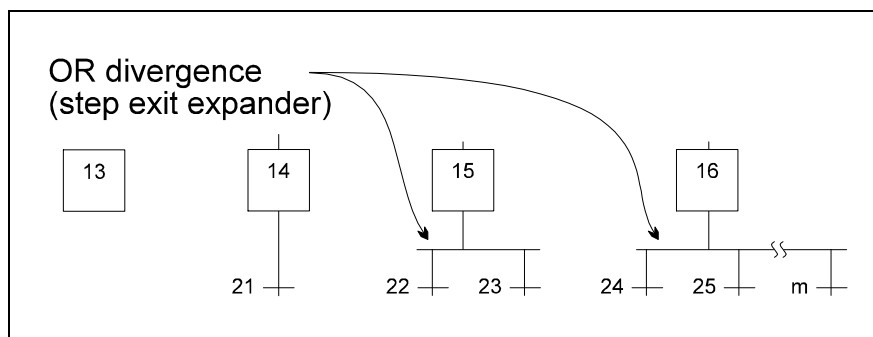


Figure 5.11

The fact that convergences and divergences are also entrance and exit expanders means that they must be connected directly to a step, and that they cannot be nested.

Note: In almost all cases, all the transitions connected to the exit of the same step by an OR divergence should have mutually exclusive trigger conditions.

The step bit

For each step there is a reserved internal variable called the *step bit*. A step bit is indicated by the step number prefixed by *X*. Thus, *X25* is the step bit for Step 25. A step bit is a Boolean variable which can be referenced elsewhere in the application. When a step is active, its step bit is True. When the step is deactivated, its step bit becomes False.

In the interest of software portability, you should consider step bits to be *read-only*. The Grafcet standard does not specify that you can activate or deactivate a step by writing to its step bit. Some Grafcet software packages let you do this and some don't, so if you want to be able to switch packages without changing your graphs, don't write to step bit.

Summary

To summarize, a step consists, at the very least, of a square with a step number written inside and a step bit. Optionally, it may also include:

- an OR convergence (entrance expander) when it has 2 or more entrance transitions
- an OR divergence (exit expander) when it has 2 or more exit transitions
- an action
- a double line indicating that it is an initial step

Transitions

The transition is the element which performs the sequential aspect of Grafcet by activating and deactivating the steps. We say that a graph *evolves* as control tokens pass through the transitions from one step to the next.

Drawing and identifying transitions

The Grafcet symbol for a transition is a short, numbered, horizontal line as shown in Figure 5.12. The top of the transition is its entrance and the bottom of the transition is its exit. The entrance and exit of a transition can only be connected to steps, never to another transition.

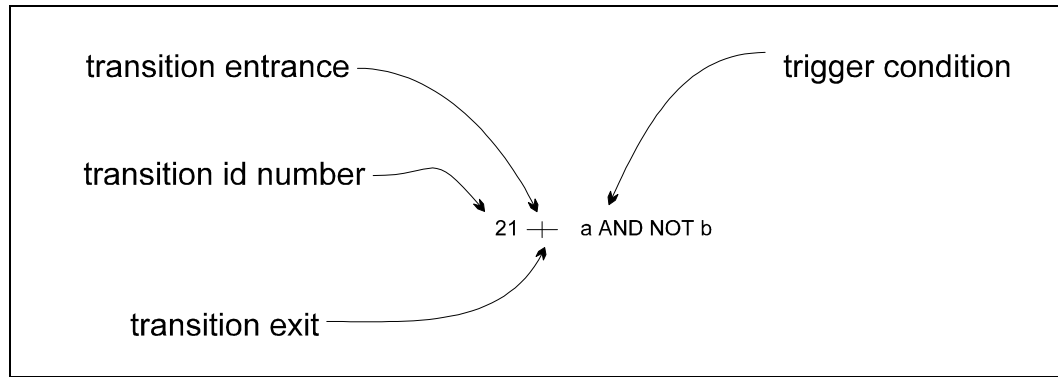


Figure 5.12

Each transition is identified by a number printed on its left hand side. Transition numbers have local scope, i.e., within an application, the same transition number can be re-used; thus Graph1 and Graph2 may each contain a Transition 3. To distinguish between them, we refer to *Graph1, Transition 3* and *Graph2, Transition 3* respectively.

Trigger conditions

To the right of each transition there is a text string of zero or more characters known as a *trigger precondition*, or just *trigger* for short. The trigger must be a valid Boolean expression. When this condition is True it allows the transition to fire, i.e., to pass a control token from its entrance to its exit.

The rules for writing triggers are the familiar rules of Boolean algebra. The following elements are allowed within an expression:

- Boolean variables and input points
- Boolean operators: AND, OR, NOT
- Rising edge operator: up arrow
- Falling edge operator: down arrow
- True/False type comparison
- Timers
- Parentheses, for establishing priority of operation

Boolean operators

The Boolean operators allowed in a trigger precondition are the same operators as introduced in Chapter 3, Basic Concepts.

Edge detection

Edge detection is an extension to normal Boolean algebra for detecting changes of state, as explained in Chapter 3. The rising edge operator,

$$\uparrow \text{expression}$$

is True during the PLC scan cycle when *expression* becomes True. The falling edge operator,

$$\downarrow \text{expression}$$

is True during the PLC scan cycle when *expression* becomes False. The truth table for these operators is shown in Figure 5.13.

a (previous scan)	a (present scan)	$\uparrow a$	$\downarrow a$
True	True	False	False
True	False	False	True
False	True	True	False
False	False	False	False

Figure 5.13

Comparison

Triggers may also compare numeric expressions. A numeric expression is one containing numeric variables or constants and arithmetic operators. Its result is always numeric, but comparing two numeric expressions gives a Boolean result, either True or False. The comparison operators are:

>	greater than
≥	greater than or equal to
<	less than
≤	less than or equal to
=	equal to
≠	not equal to

Some software packages may slightly vary these symbols because standard keyboards do not have all the necessary keys.

Here are some examples of trigger conditions containing comparisons:

- temperature > 100
- value ≤ value
- (value1 + value2 / 3) = (145 * value3)

Timers

The concept of elapsed time is so important to industrial automation that Grafset includes a special operator for this purpose. PLC programmers, used to a bewildering array of different timer types (on-delay, off-delay, set/reset, stop/start, etc.), are often surprised to see that a single time operator, acting within the context of a Boolean expression, can fulfill all control requirements.

The Grafset timer is a Boolean operator that returns to True when a given time-out period, *t*, has elapsed after a enable event. The format of a Grafset timer is:

T / enable / t

The timer's operation is defined by the timing diagram shown in Figure 5.14. The output of the timer becomes True 12 seconds after a rising edge on the enable. The timer output remains True even after the enable signal has fallen, and becomes False again only on the next rising edge of the enable signal.

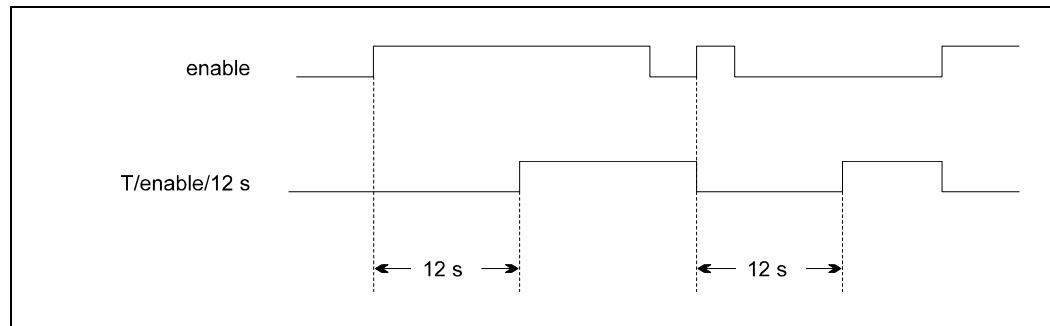


Figure 5.14

In standard Grafset, a timer enable can only be the rising edge of a step bit, while the time-out period can be a constant value or a parameter. Time is measured in seconds unless otherwise stated. Here are some examples:

- | | |
|---------------|--|
| T/X23/14s | - becomes False when Step 23 is activated, becomes True 14 seconds later. |
| T/X342/0.03 s | - becomes False when Step 342 is activated, becomes True 0.03 seconds later. |
| T/X1/delay | - becomes False when Step 1 is activated, becomes True delay seconds later, where delay is an numeric variable or input. |

A logical extension to the Grafset would allow any Boolean expression to be used as a trigger event, and any numeric expression to be used as a time-out period. (This is included in the proposed revision to the Grafset standard dated 91/3/15). Using this extension, the following expressions would be acceptable.

T/switch1 OR switch2/14s - becomes False when either switch1 or switch2 are activated, becomes True 14 seconds later.

T/temperature>300/t1 - becomes False when temperature exceeds 300, becomes True t1 seconds later.

Precedence

Within a Boolean expression, operations are carried out in order of priority. The priority of operations is shown below. Operations at the top of the list are done first.

- numeric expressions and comparisons
- timers
- edge detection, NOT operators
- AND
- OR

Parenthesis are used to define the order of evaluation, the operations within parenthesis are done first. The following examples illustrate the precedence rules of Boolean expressions:

ex. 1: a AND b OR c AND d = (a AND b) OR (c AND d) ≠
AND (b OR c) AND d

ex. 2: ↑NOT a + b = (↑(NOT (a)) + b

ex. 3: temperature ≤ 300 AND NOT over-pressure_sensor
= (temperature ≤ 300) AND NOT over-pressure_sensor

Always-true triggers

A trigger condition that is always True may be written in three ways:

- True - this is the preferred way.
- =1 - accepted by NFC 03-190 standard.
- blank - a blank trigger condition is always True, just as a transition with no entrance steps is always armed.

Syntax diagrams

Figure 5.15 shows the syntax diagram which defines the rules for forming syntactically correct trigger conditions.

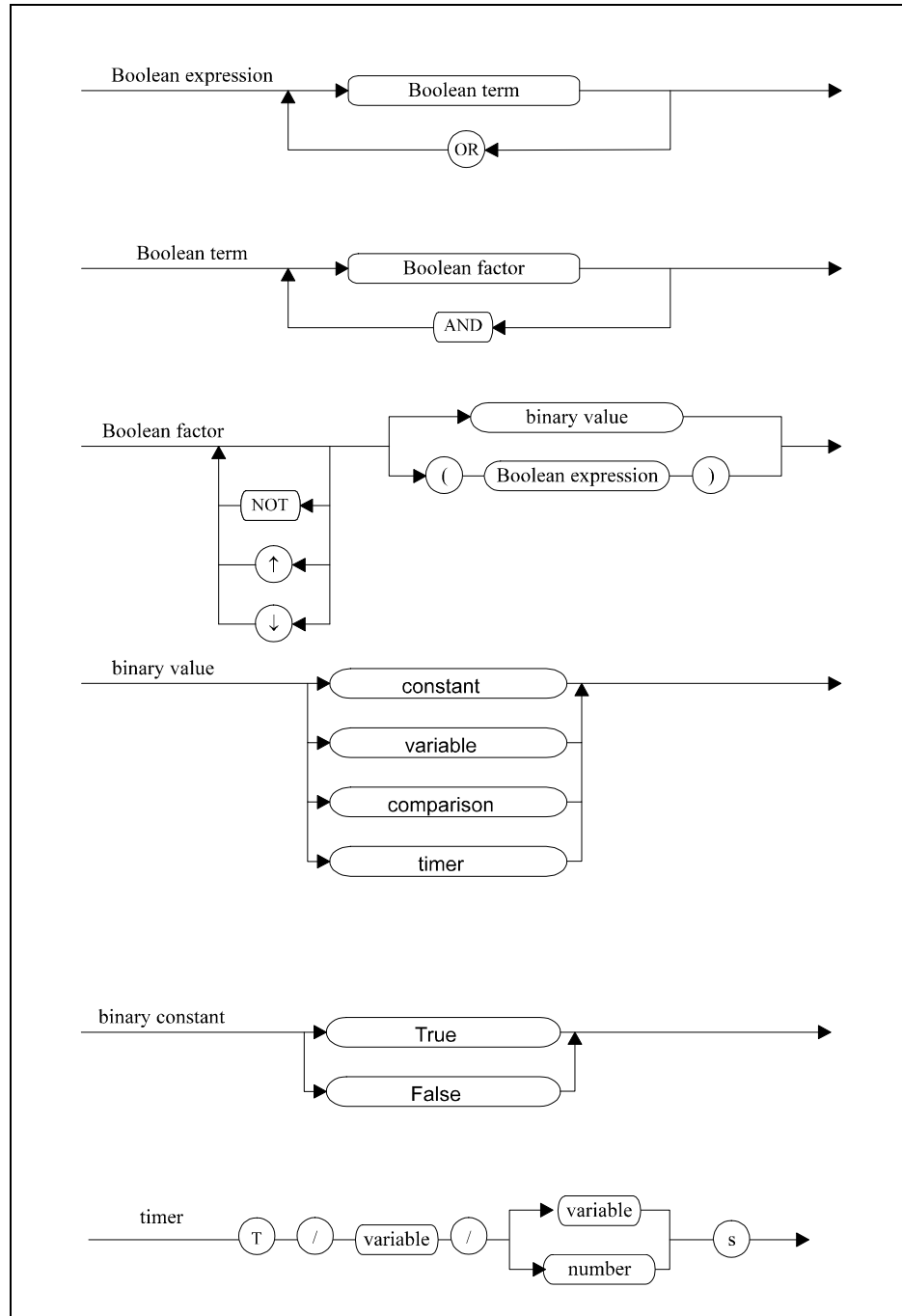


Figure 5.15

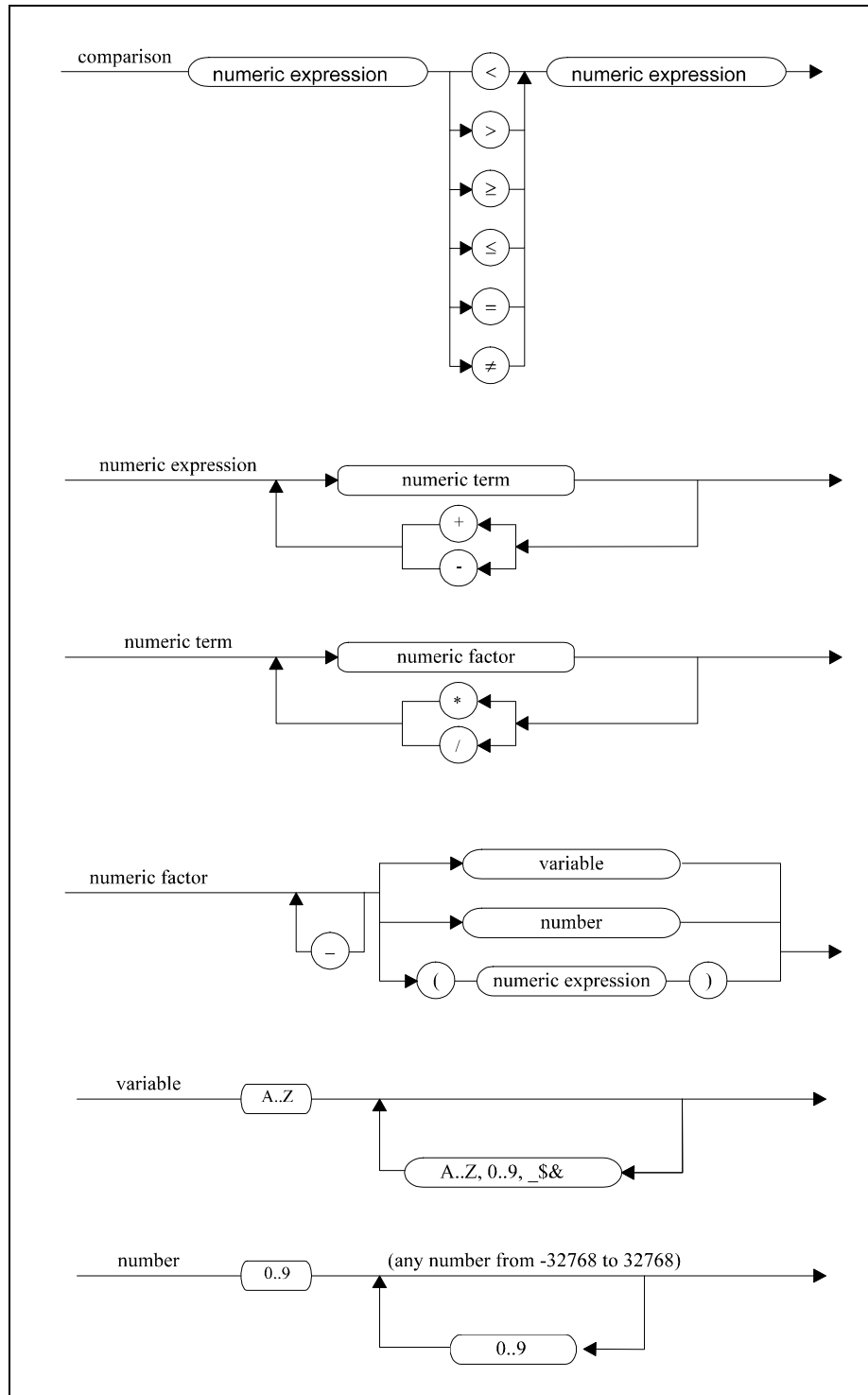


Figure 5.15 (continued)

AND convergences & divergences

The entrance of a transition may be connected to zero, one, two or more entrance steps, as shown in Figure 5.16. If there are two or more entrance steps, a double horizontal entrance expander symbol is added. The more usual name for this symbol is the AND Convergence. The choice of the name *AND Convergence* is evident from Figure 5.16, in which you will note that Transition 6 is armed only if Step 14 *and* Step 15 *and* Step n are all active.

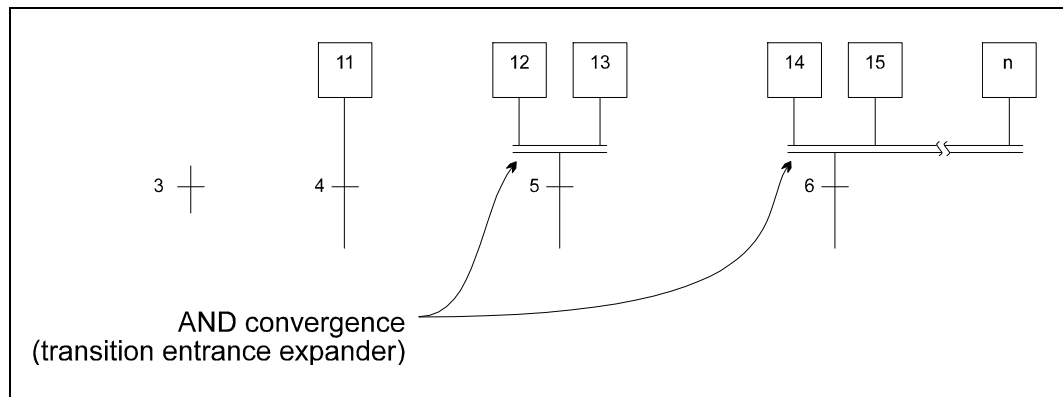


Figure 5.16

When the transition fires, it simultaneously deactivates all its input steps.

Note that a transition, such as Transition 3 in Figure 5.16 or Transitions 13 and 14 in Figure 5.17, which has no entrance steps, is called a *source transition* and is considered to be always armed.

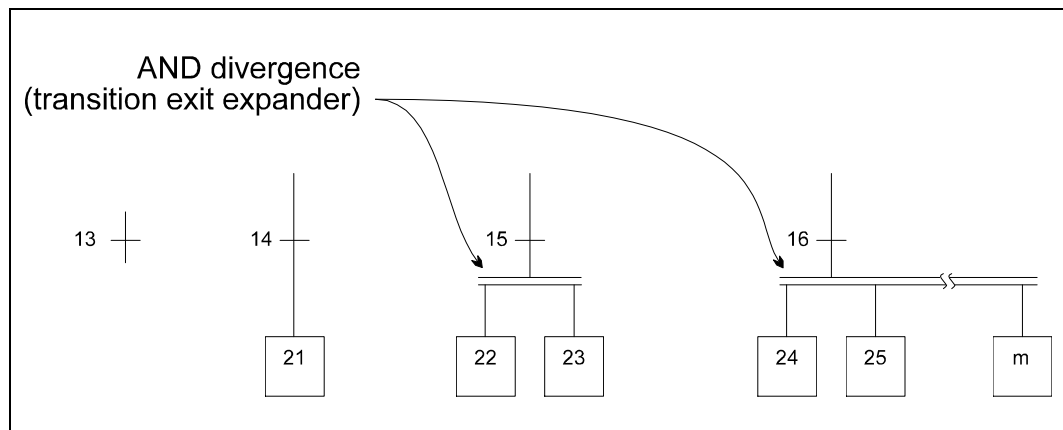


Figure 5.17

Similarly, the exit of a transition may be connected to zero, one, two or more exit steps, as shown in Figure 5.17. If there are two or more exit steps then a double horizontal exit

expander symbol, or *AND Divergence*, is used. In Figure 5.17 when Transition 16 fires, it simultaneously deactivates all (if any) of its input steps and activates all (if any) of its output steps, i.e., Step 26 and Step 27 and Step m.

The fact that convergences and divergences are also entrance and exit expanders means that they must be connected directly to a transition, and that they cannot be nested.

Grafcet evolution

A graph is said to evolve as control tokens pass from step to step, passing through the transitions. When a control token passes through a transition, the transition will *fire* (or *be cleared*) when two conditions are met:

- The transition must be armed, i.e., all of its entrance steps must be active (contain a control token). A source transition, having no entrance steps, is always armed.
- The transition's precondition must be True.

The firing time of a transition is infinitesimal.

When two or more transitions are armed and triggered simultaneously, they fire simultaneously. In the context of a PLC scan cycle, all armed transitions having True preconditions fire once during the scan. When a step is activated by the firing of an exit transition and an entrance transition simultaneously, the step remains active.

Figure 5.18 shows the evolution of a number of graphs along with their timing diagrams.

	Initial situation, expression is False	Final situation after expression becomes True	Timing diagram
A)			
B)			
C)			
D)			

Figure 5.18 (A-D)

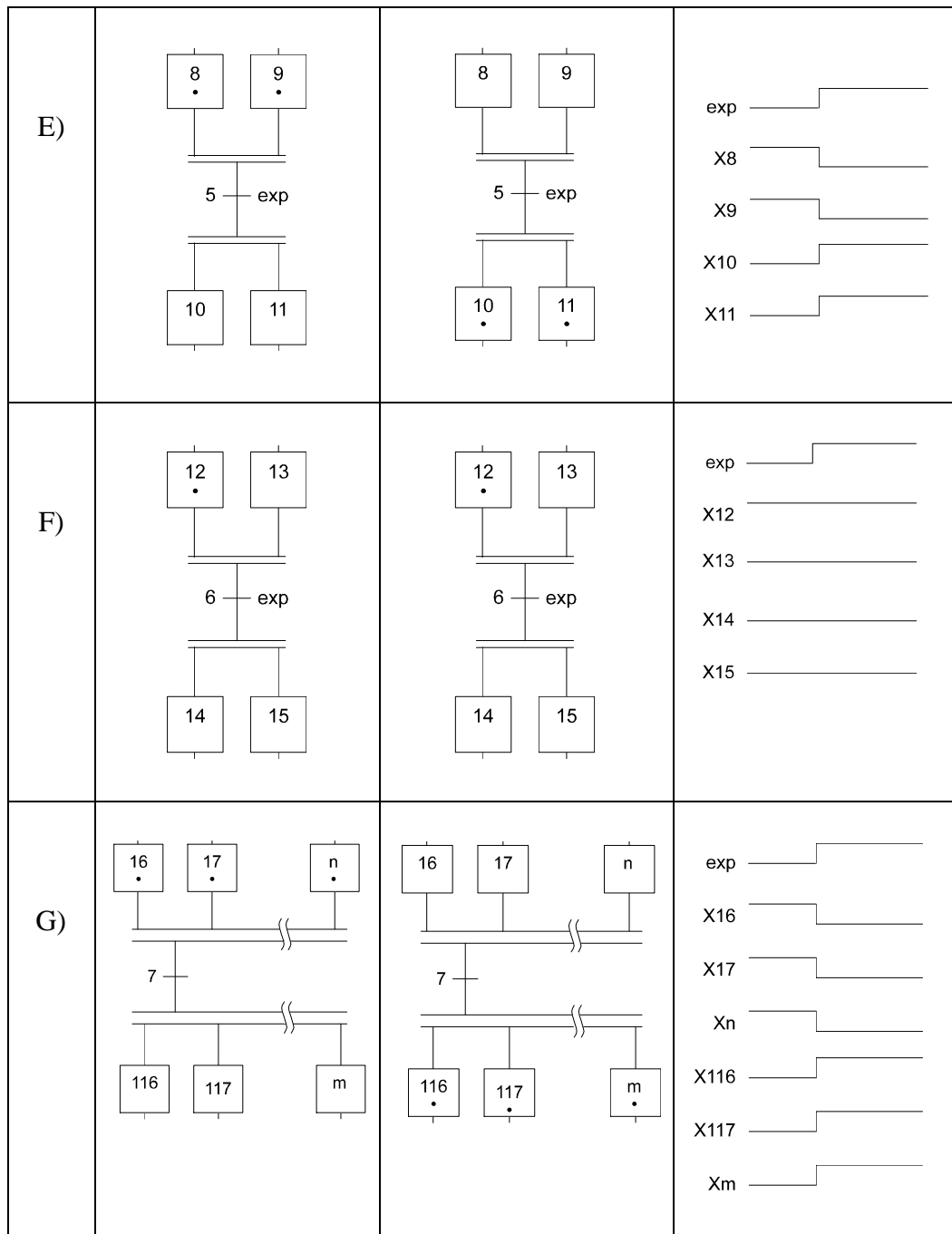


Figure 5.18 (E-G)

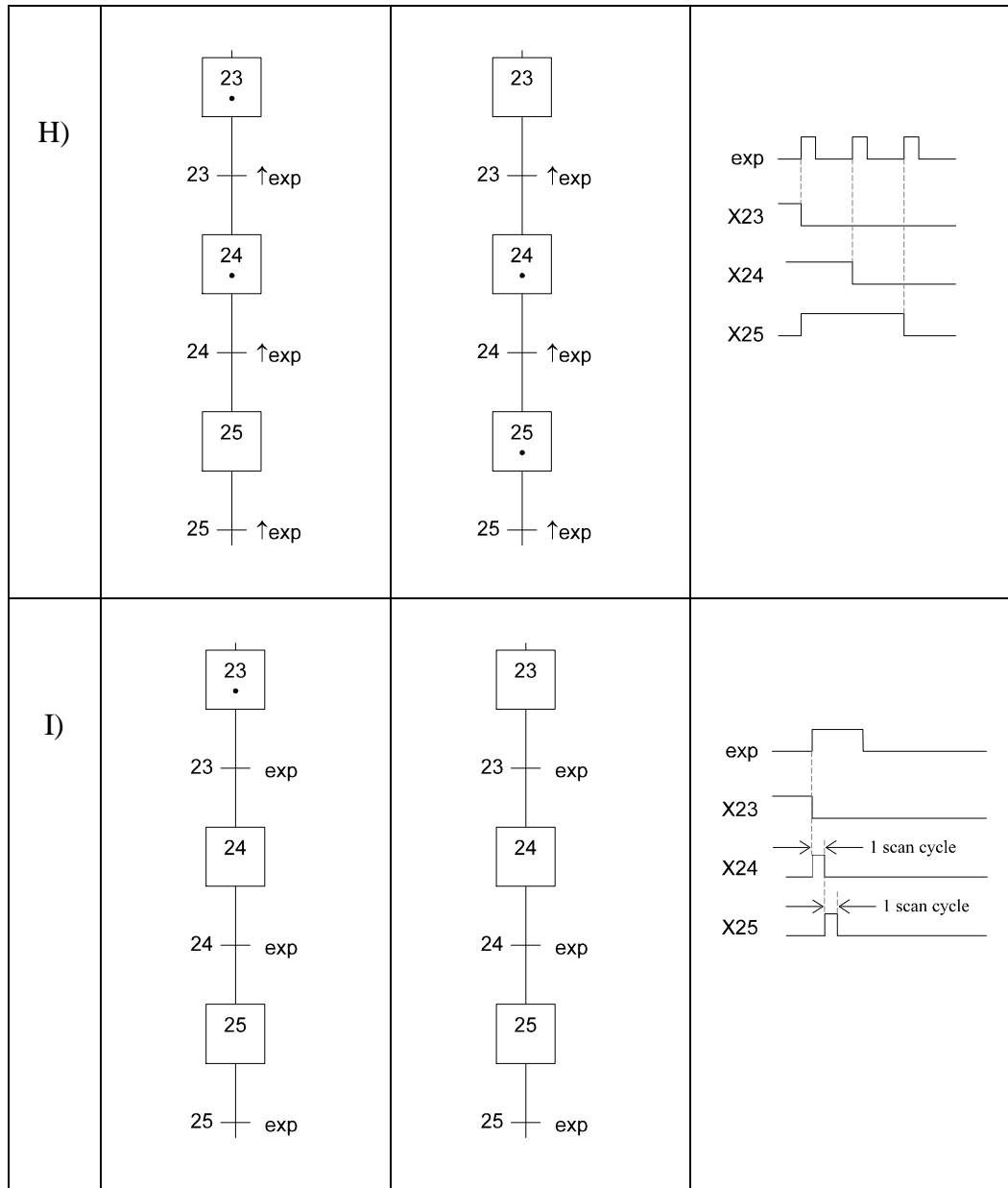


Figure 5.18 (H,I)

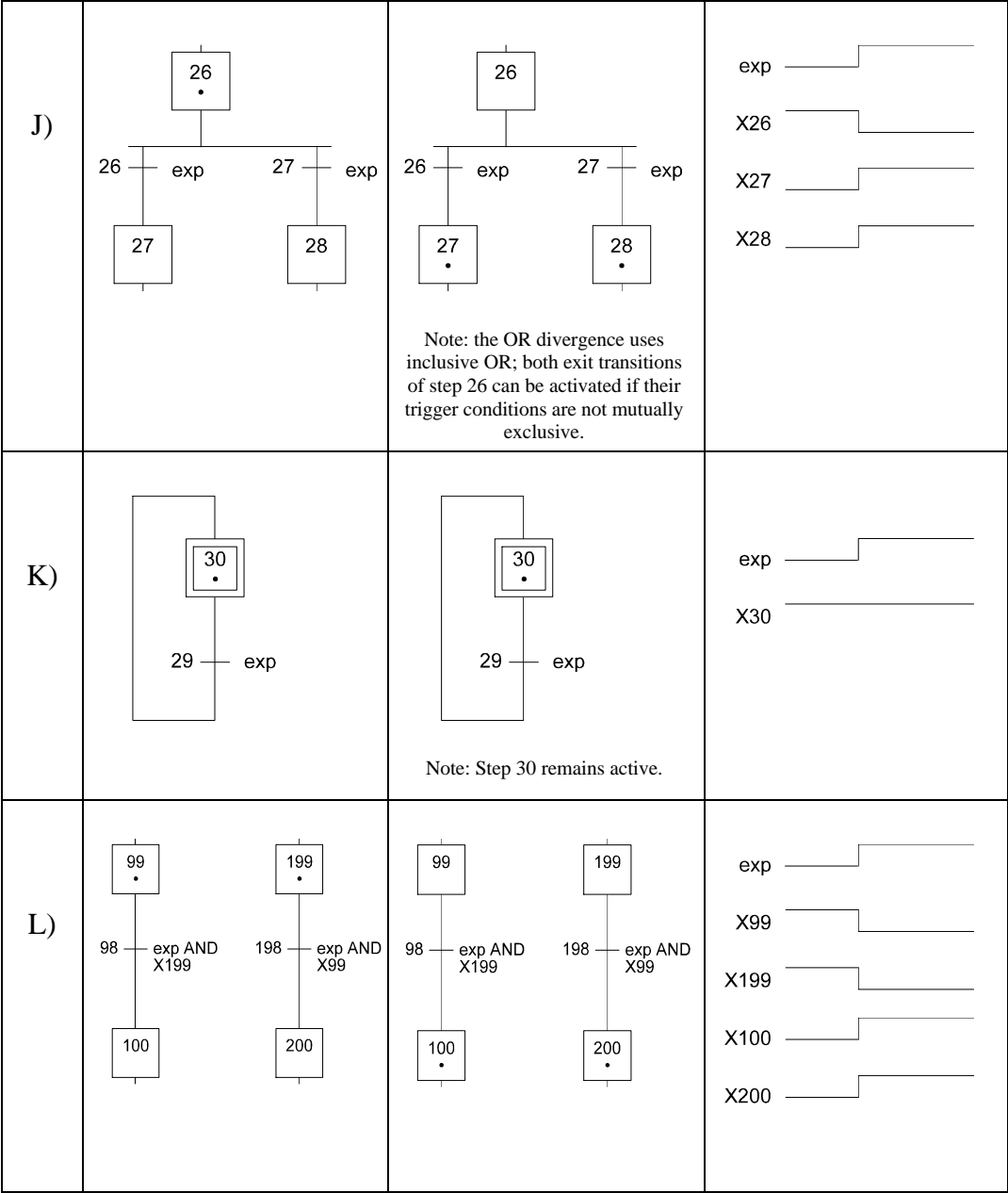


Figure 5.18 (J-L)

Summary

To summarize, a transition consists, at the very least, of a crossed vertical line with a transition number written to its left and a Boolean trigger condition written to its right. In certain cases it may also include:

- an AND convergence (entrance expander) when it has 2 or more entrance steps
- an AND divergence (exit expander) when it has 2 or more exit transitions

As Figure 5.19 shows, a transition cannot fire until it is both armed and triggered. Once a transition is armed and triggered, it fires by deactivating all its entrance steps and by activating its exit steps. Activation has priority over deactivation.

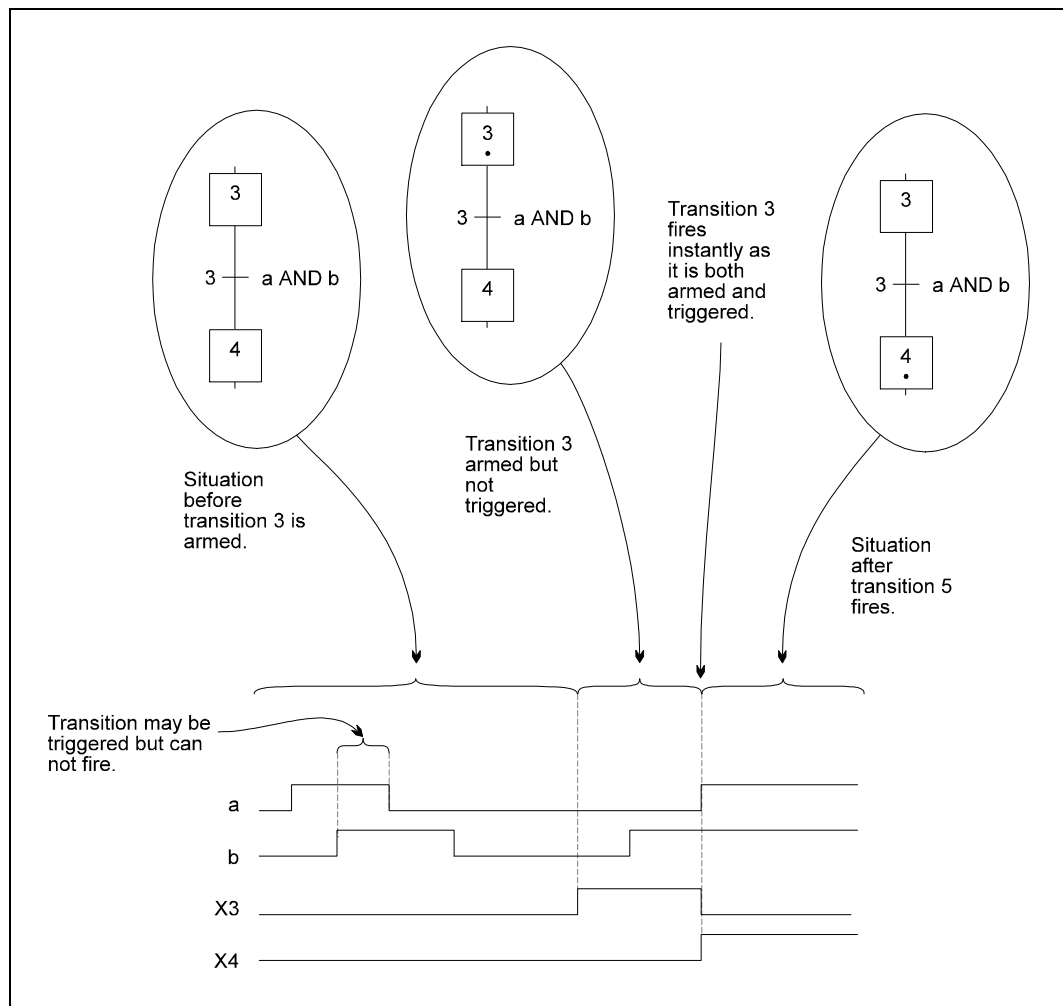


Figure 5.19

Actions

An action is a set of one or more textual statements, separated by semi-colons (;), that define a control operation or procedure. The statements in an action are executed, in the order they are written, when the action is executed.

Drawing step actions and stand-alone actions

Step actions are written in a rectangle to the right of a step and executed while the step is active. An action written elsewhere on the graph and not connected to any step is a *stand-alone action*. The formats for drawing Grafcet actions are shown in Figure 5.20.

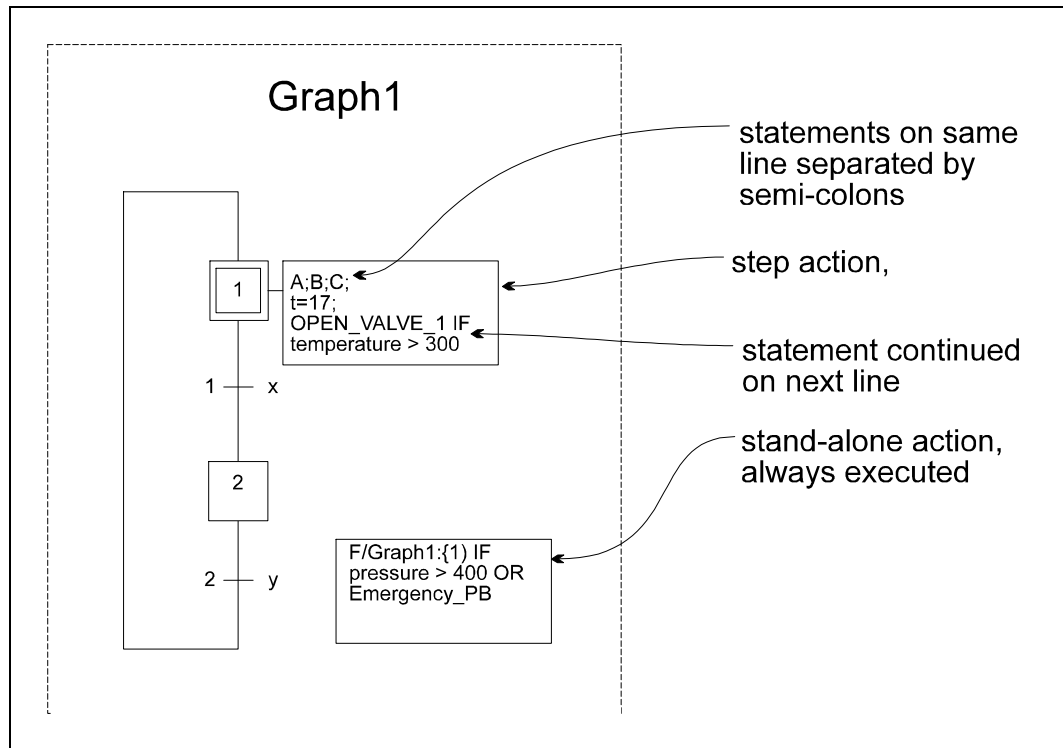


Figure 5.20

A stand-alone action is always executed, independently of the situation of the graph. In particular, a stand-alone action is even executed if $S(\text{graph}) = \{ \}$, i.e., when no steps are active.

Types of action statements

Grafcet action statements may be classified in four categories:

- Unconditional and momentary

- Unconditional and stored
- Conditional and momentary
- Conditional and stored

This classification is illustrated with a few examples in Figure 5.21. The basic types of action are described below:

	Unconditional Actions	Conditional Actions
Momentary actions - True while step is active (act only on Boolean types)	a,b,c OPEN_VALVE	a,b,c IF ↑expression OPEN_VALVE IF temperature > 300
Stored action (act on Boolean or numeric types or on the situation of a graph)	a = True b = False c = a OR b W = 29 X = w + 53 Z = 7 F/Graph1:{2,4} F/Graph1:{1}	a = True IF inp1 AND NOT inp2 b = False IF t<15 c = a OR b IF ↑expression W = 29 IF a OR b X = w+53 IF NOT c Z = 7 IF ↑a F/Graph1:{2,4} IF temperature > 150 F/Graph1:{1} IF temperature > 150

Figure 5.21

Unconditional statements

Unconditional statements are always executed when the action is executed.

Conditional statements

Conditional statements are only executed if the Boolean expression following the IF operator is True. For example, the statement

$$T = T + 1 \text{ IF } \uparrow h$$

would increment T each time h changed from False to True.

Stored statements

Stored statements have a permanent or stored effect. A typical stored action consists of writing a value to an internal variable or an output. The variable keeps the most recently written variable until overwritten by a new value. For instance, the statement

$$T = T + 1$$

would increment T continuously as long as the action were being executed.

A special case of the stored statement is the graph forcing instruction. It is different from other instructions in that operates on a graph rather than on a variable or output. For instance, the statement

$$F/\text{Graph2}:\{10,11\}$$

would force Graph2 into the situation $S(\text{Graph2}) = \{10,11\}$, i.e., it would activate steps 10 and 11 and deactivate all the other steps.

Momentary actions

Momentary actions are a special type of action which can be applied only to Boolean inputs and internal variables. Momentary actions are not stored. If a Boolean variable or input is listed as an action statement, it is activated while the action is executed. When the action is no longer executed the value returns to False. For this reason, momentary actions are sometimes called *true-in-step* actions. In the case where the same Boolean variable is activated by more than one momentary action, it remains active as long as any of the momentary actions is being executed. Think of a momentary action as a push button. Pushing the button turns the action on; the action is turned off when you release the button.

Incompatibility of stored and momentary actions

Momentary and stored activation of the same variable involves a logical contradiction. Just as a switch cannot be bi-stable and momentary at the same time, it doesn't make sense for both a momentary and stored statement to operate on the same variable. Most Grafcet compilers or translators will generate an error message or warning.

Syntax

The syntax rules for action statements are illustrated on Figure 5.22. To summarize in just a few sentences:

- Stored action statements contain an “=” (assignment operator).
- Momentary action statements contain the name of a Boolean variable, with no “=”.
- Conditional action statements may be either stored or momentary and contain an IF clause.

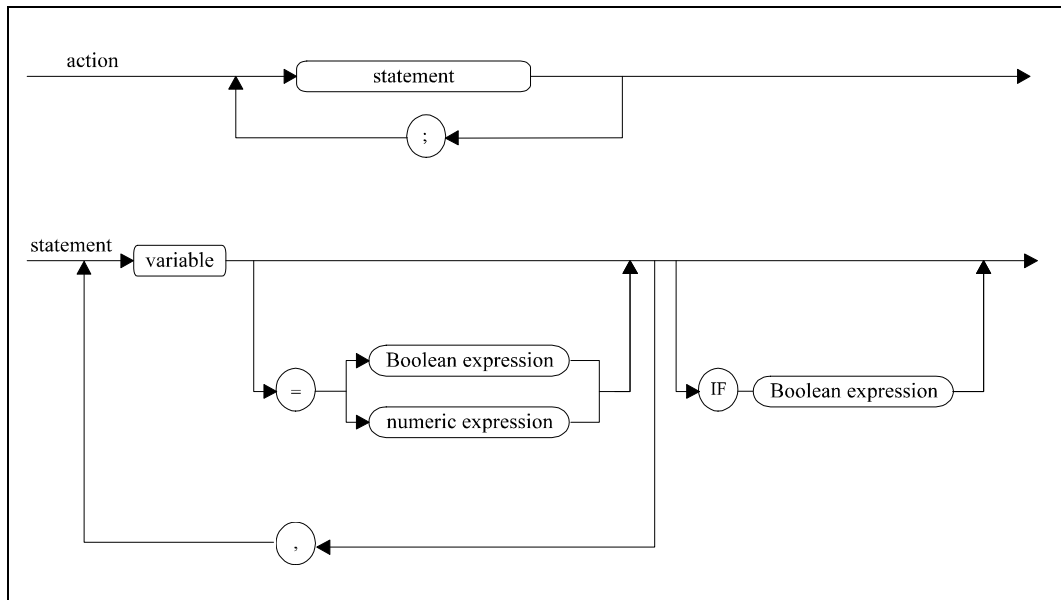


Figure 5.22

Examples

The working of Grafcet's textual language is demonstrated by the examples in Figures 5.23 through 5.26.

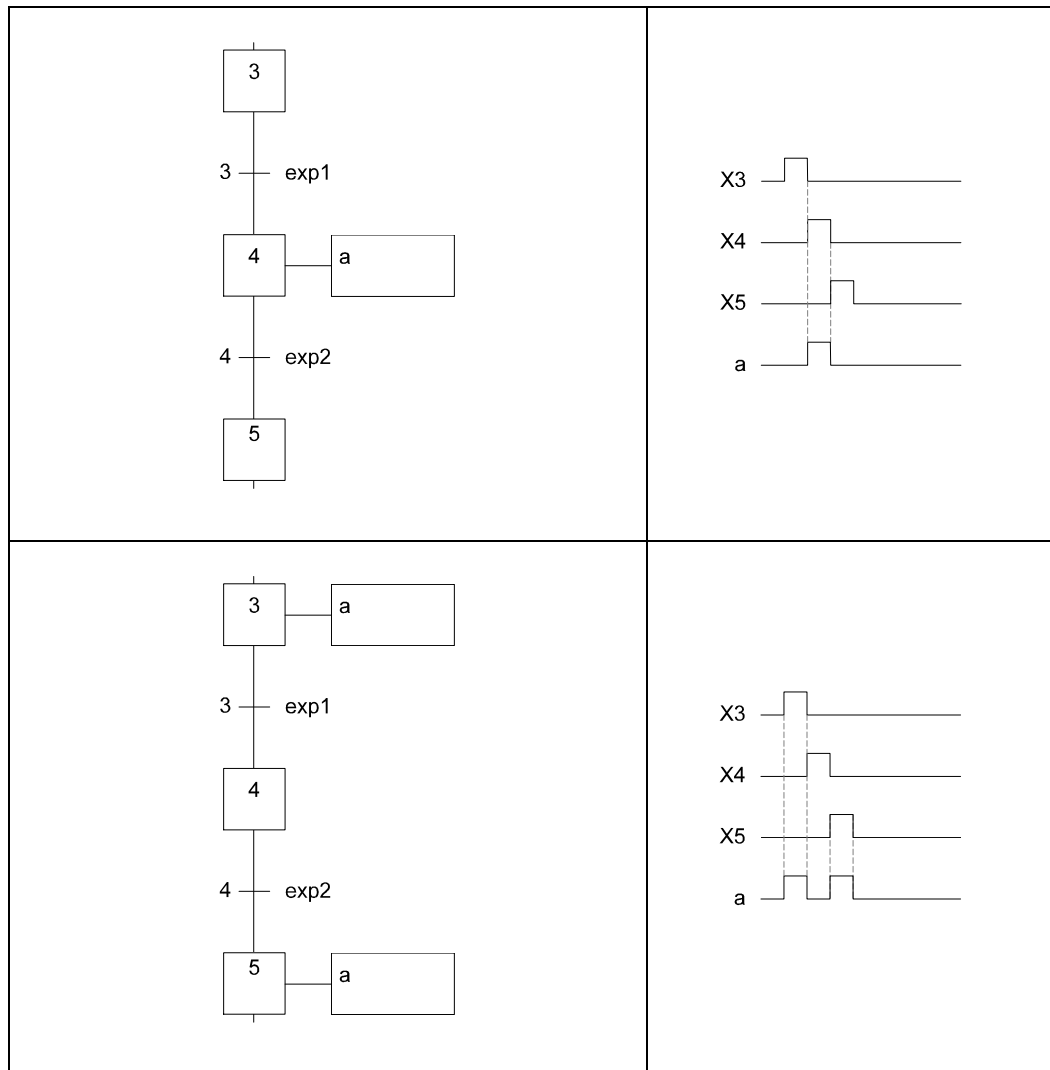


Figure 5.23 Unconditional momentary actions

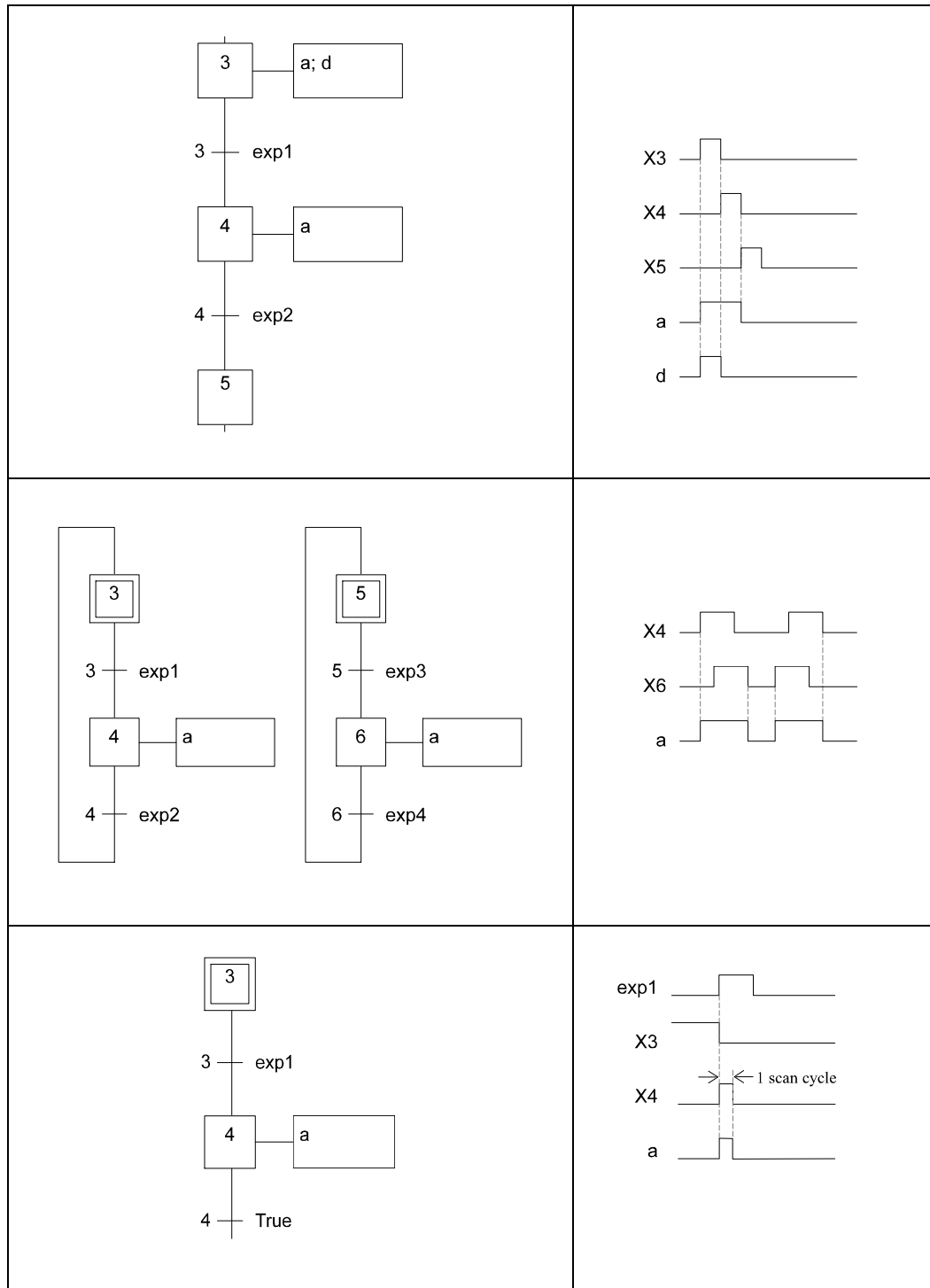


Figure 5.23 (continued)

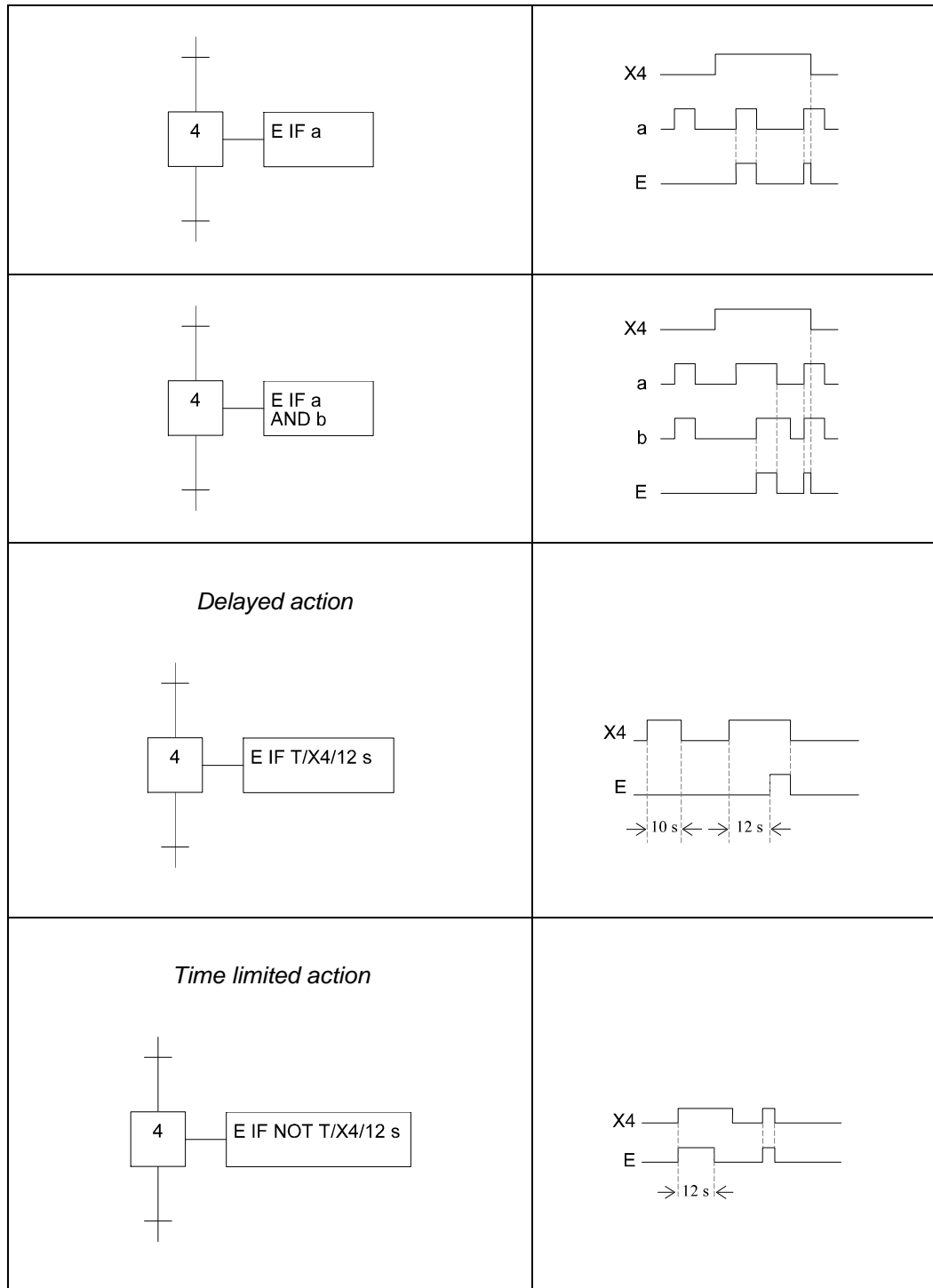


Figure 5.24 Conditional momentary actions

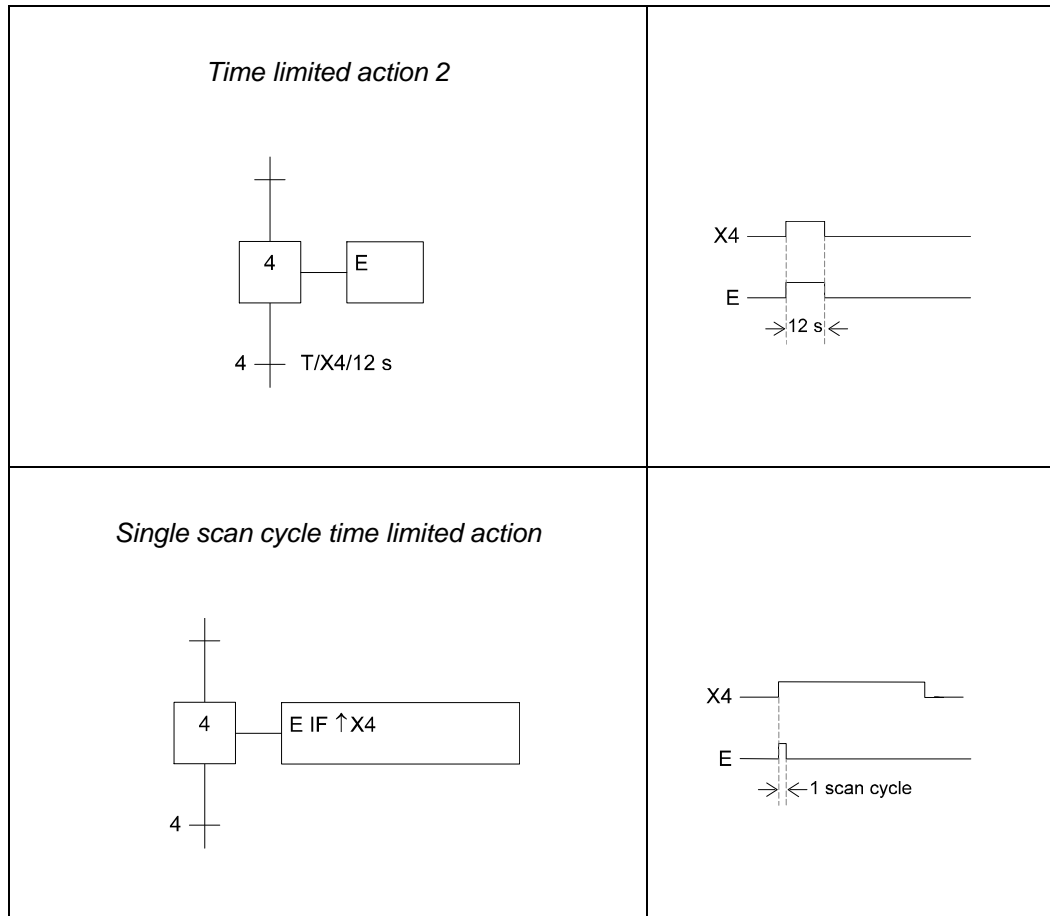


Figure 5.24 (Continued)

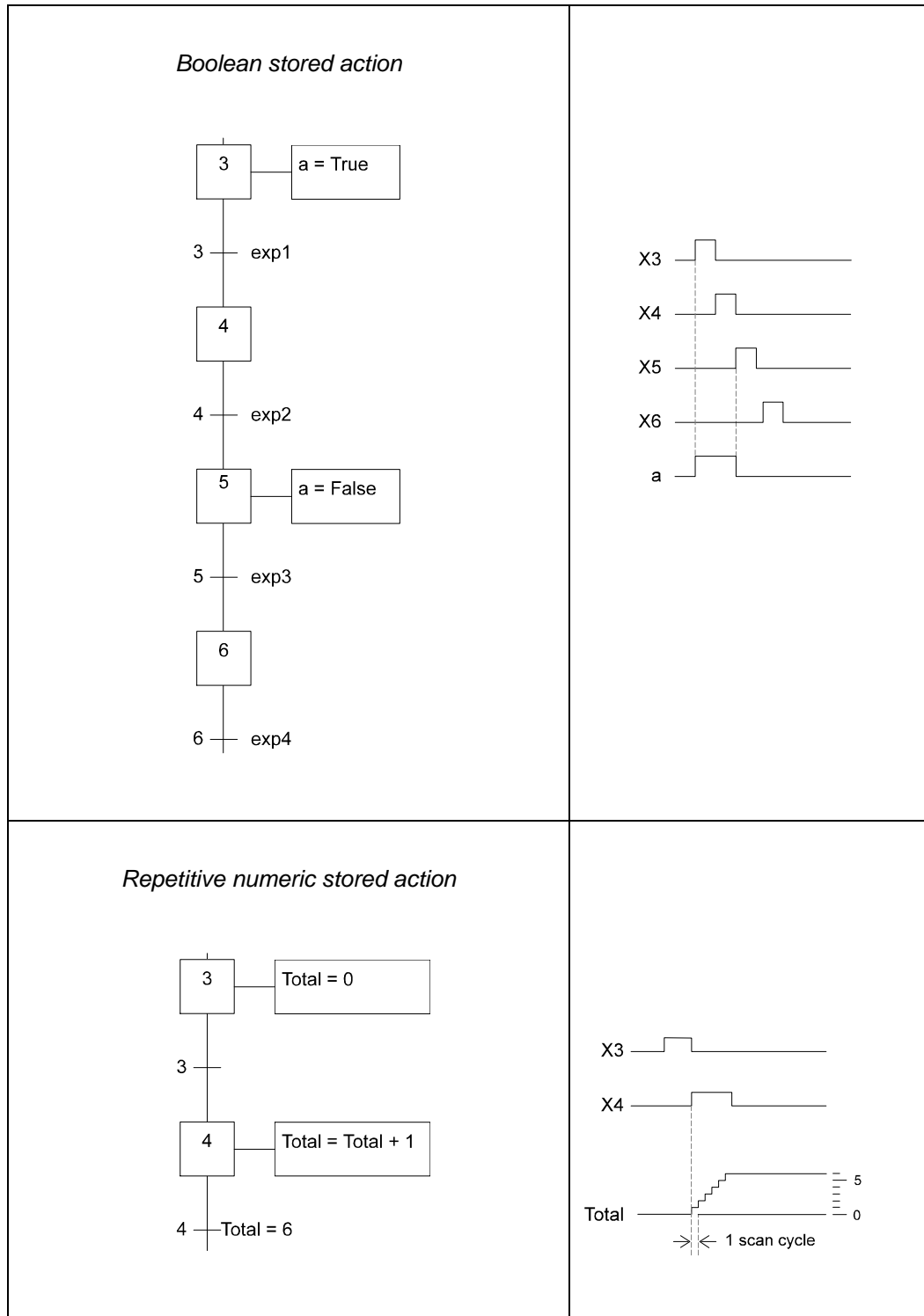


Figure 5.25 Stored actions

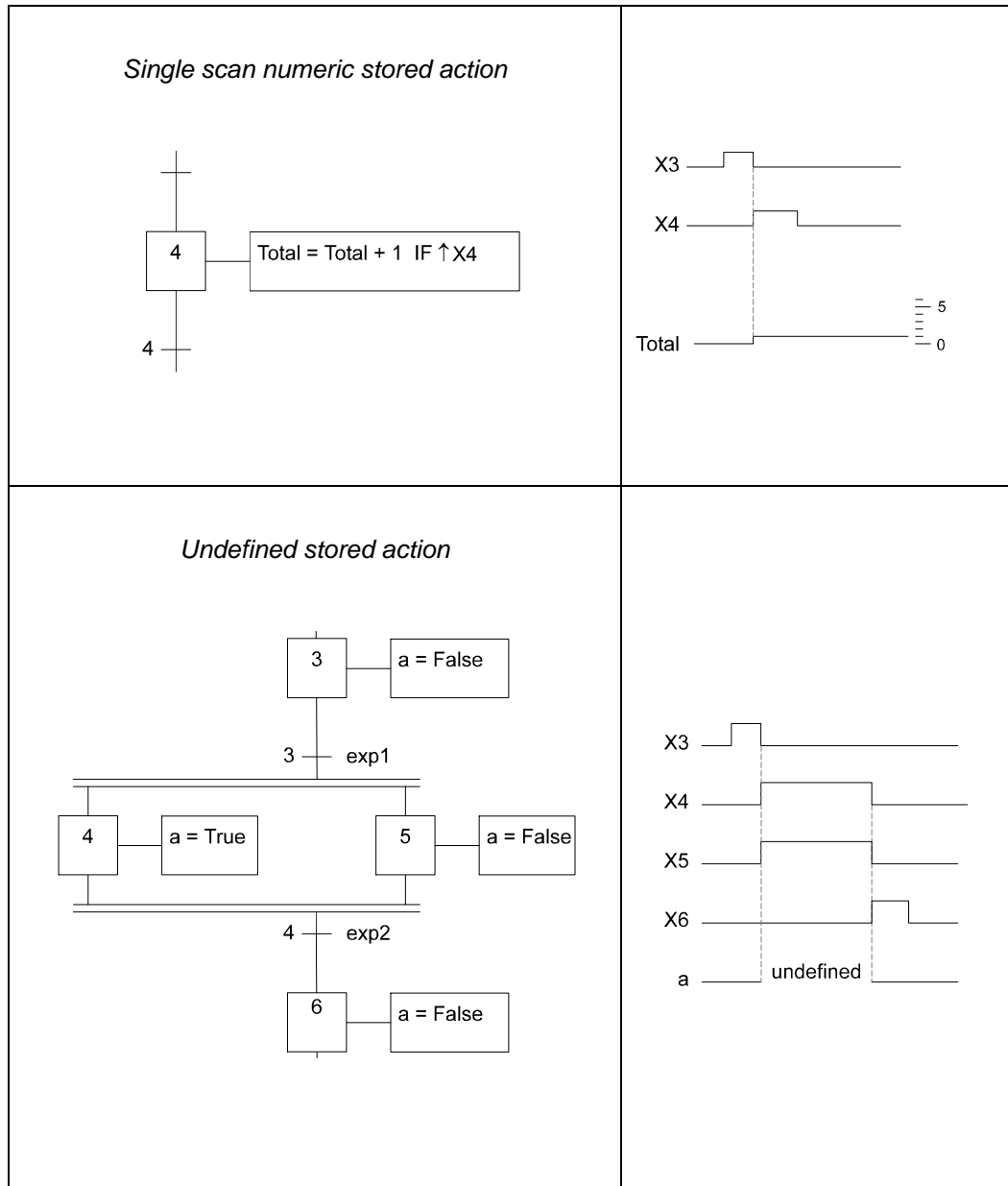


Figure 5.25 (continued)

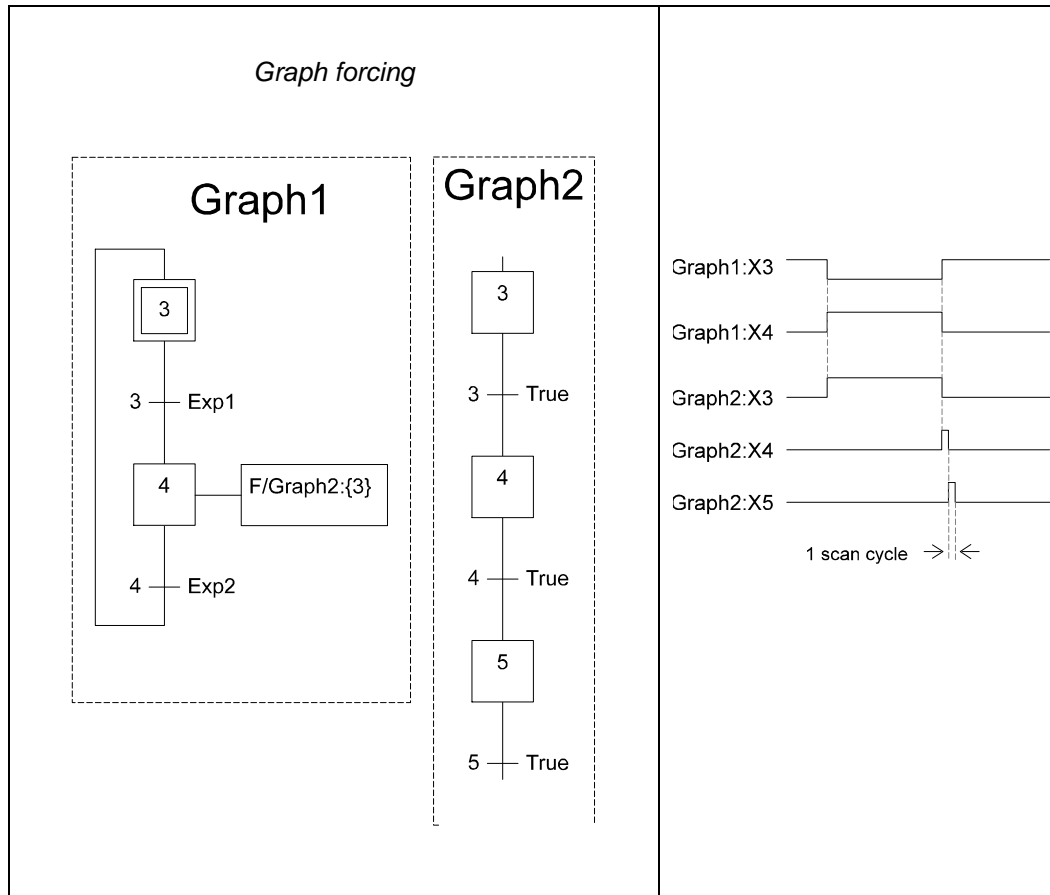


Figure 5.26

Links

Explicit links

An explicit link is simply a line drawn from a step to a transition or from a transition to a step, as shown in Figure 5.27. A control token can pass along the link when the transition fires.

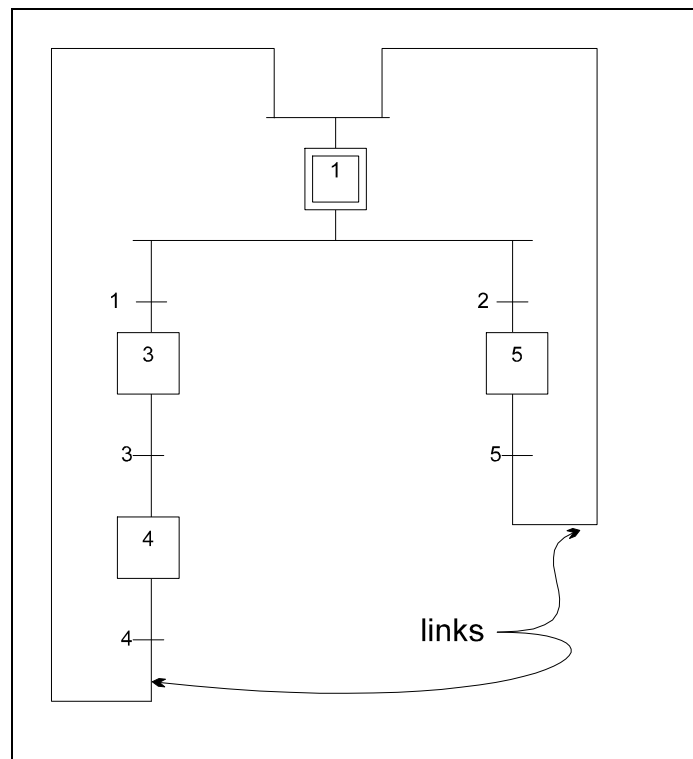


Figure 5.27

Labelled links

Sometimes when an explicit link would be too long or when crossing links would be confusing, the links are replaced by a *jump-to label* as shown in Figure 5.28. This is the format approved by NFC 03-190.

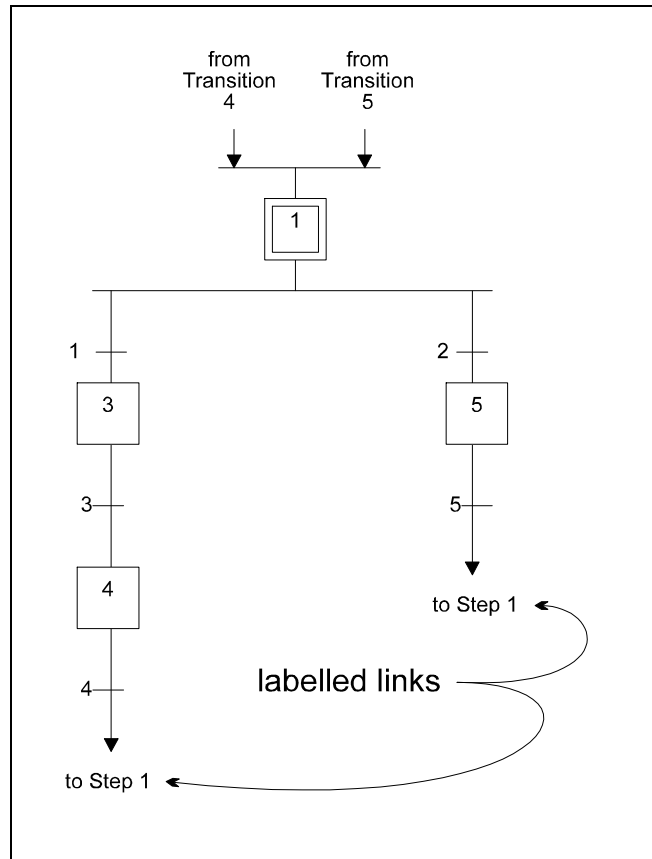


Figure 5.28

A neater format which has come into modern usage is shown in Figure 5.29.

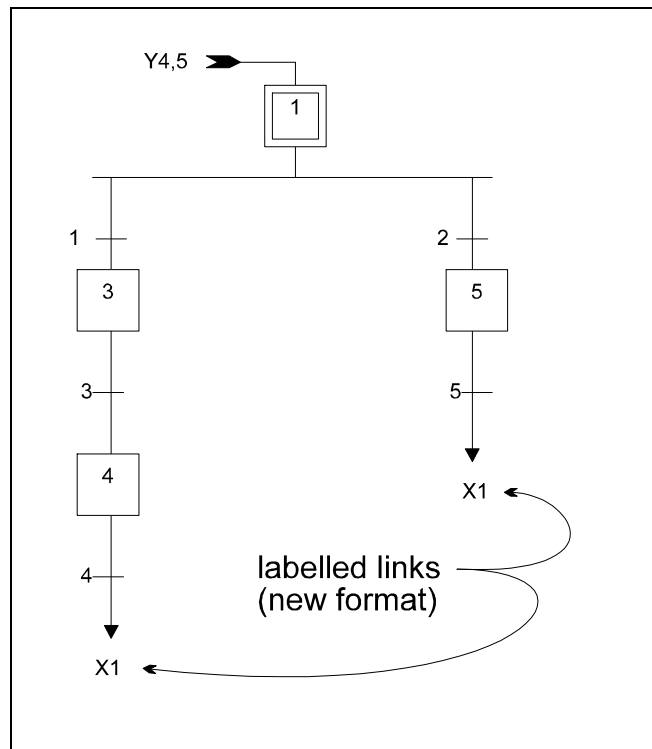


Figure 5.29

All formats as shown in Figures 5.24 to 5.29 represent the same graph.

Comments

As in other forms of programming, Grafcet programming for PLCs lets you place comments on the diagrams. It is a good idea to provide a clear, simple explanation of what each graph does to improve readability.

A graph whose actions and preconditions are described by English language comments is a *level 1* graph. Many PLC programmers find it useful to design a level 1 graph before writing out the action statements and preconditions. The final graph, including executable statements, is a *level 2* graph. It's good practice to keep the level 1 comments in the level 2 graph to enhance readability.

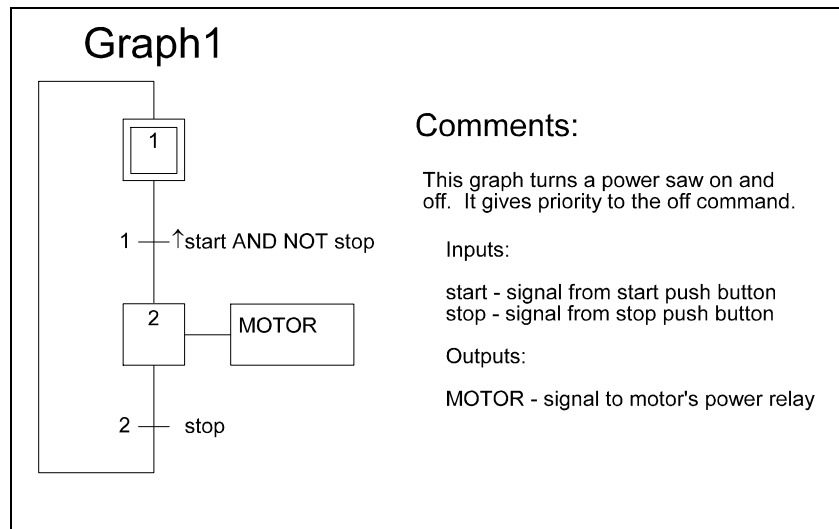


Figure 5.30

6 Grafcet Structures

This chapter presents a series of example graphs and applications which illustrate some of the structures you can build. Some of these will be useful to you for industrial control, others serve mostly to show the flexibility of Grafcet.

The stand-alone action

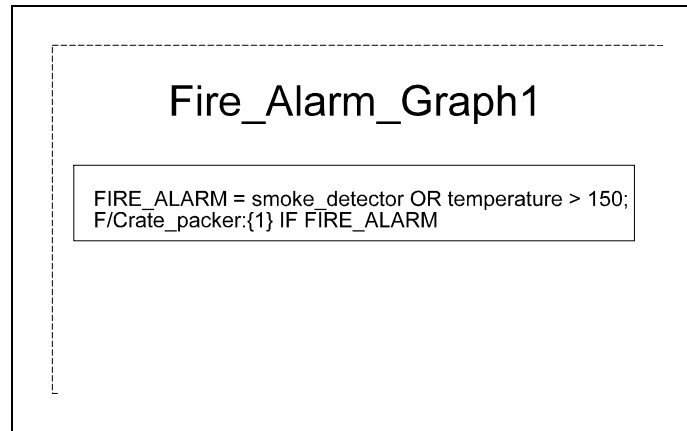


Figure 6.1

This graph has no steps or transitions, only a stand-alone action. The stand-alone action is executed continuously, and cannot be turned off by a forcing command, so it is particularly useful for safety monitoring functions.

The permanent step

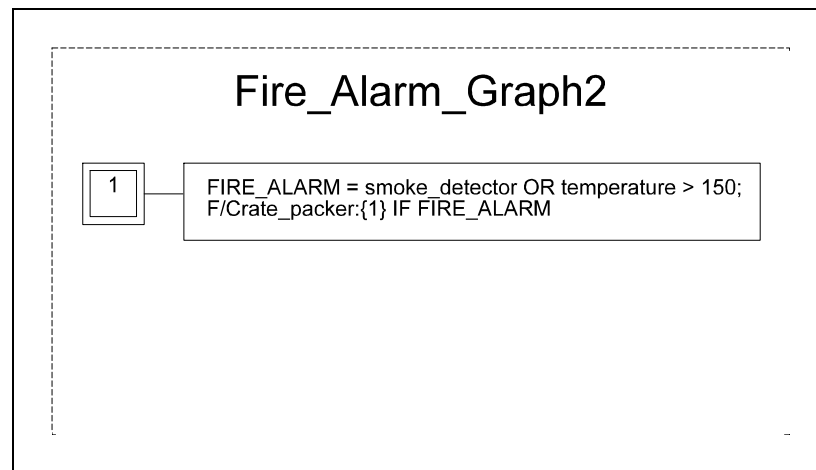


Figure 6.2

This graph has no transitions or stand-alone actions, only a single step, which is initially active. Since it is not followed by a transition, it remains active and its action is executed continuously. It differs from Fire_Alarm_Graph1 in Figure 6.1 only in that it can be forced into the empty situation, $S(\text{Fire_Alarm_Graph2}) = \{ \}$, from another graph or from a PLC programming console. This would cause execution of the action to cease. Stand-alone actions are preferred to permanent steps for safety monitoring when you do not want the maintenance operator to be able to easily turn off their execution.

Data initialization

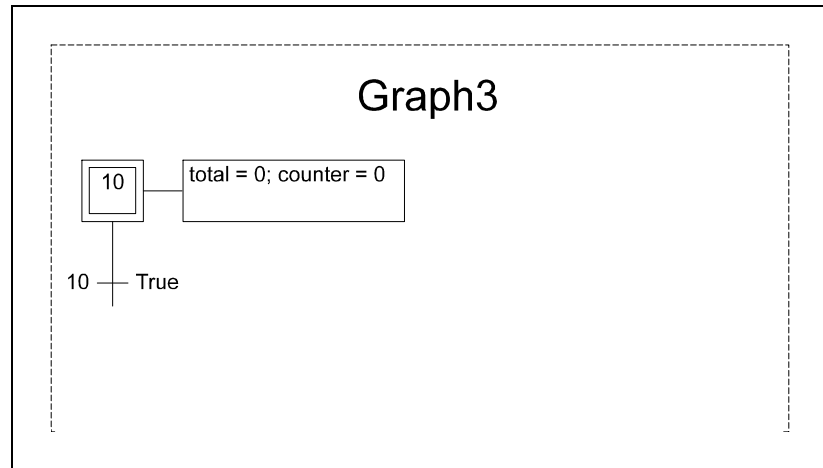


Figure 6.3

This graph illustrates a structure used for initializing data prior to a calculation. The initial Step 10 is active for only one scan cycle before Transition 10 fires to deactivate it. Transition 10 has no exit steps. Therefore, it is a *sink transition*, so named because when it fires, the control token “disappears down the drain”.

Data initialization and continuous calculation

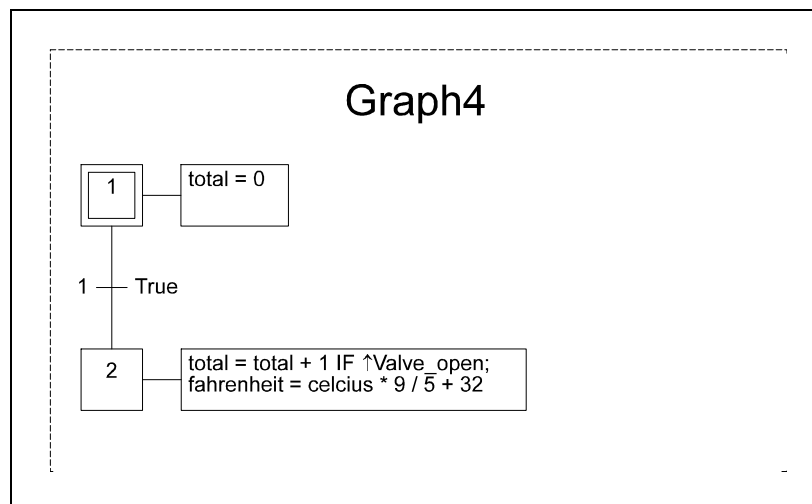


Figure 6.4

This graph shows how to carry out calculations after initializing data. The initial Step 1 is active for only one scan cycle before Transition 1 fires to deactivate it.

Step 2 is the terminal step of a (non-looping) sequence. It remains active indefinitely and its action is executed continuously.

Non-looping sequence with a permanent final step

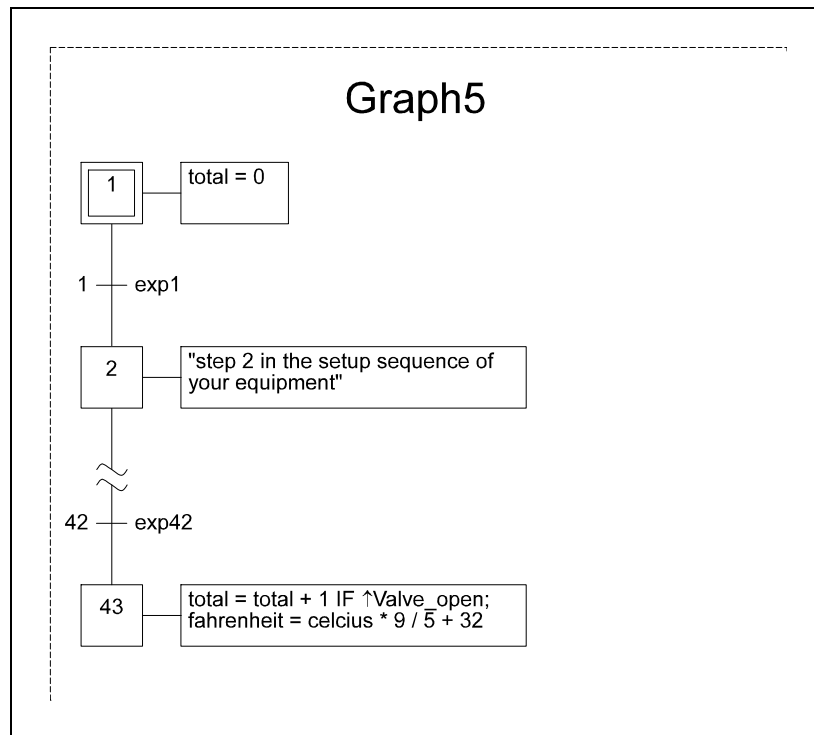


Figure 6.5

This structure is simply an extension of the previous two-step graph, and may be continued to any number of steps. It would be suitable for any sequence of operations you only want performed once, followed by an operation to be repeated continuously.

Repeating sequence (loop)

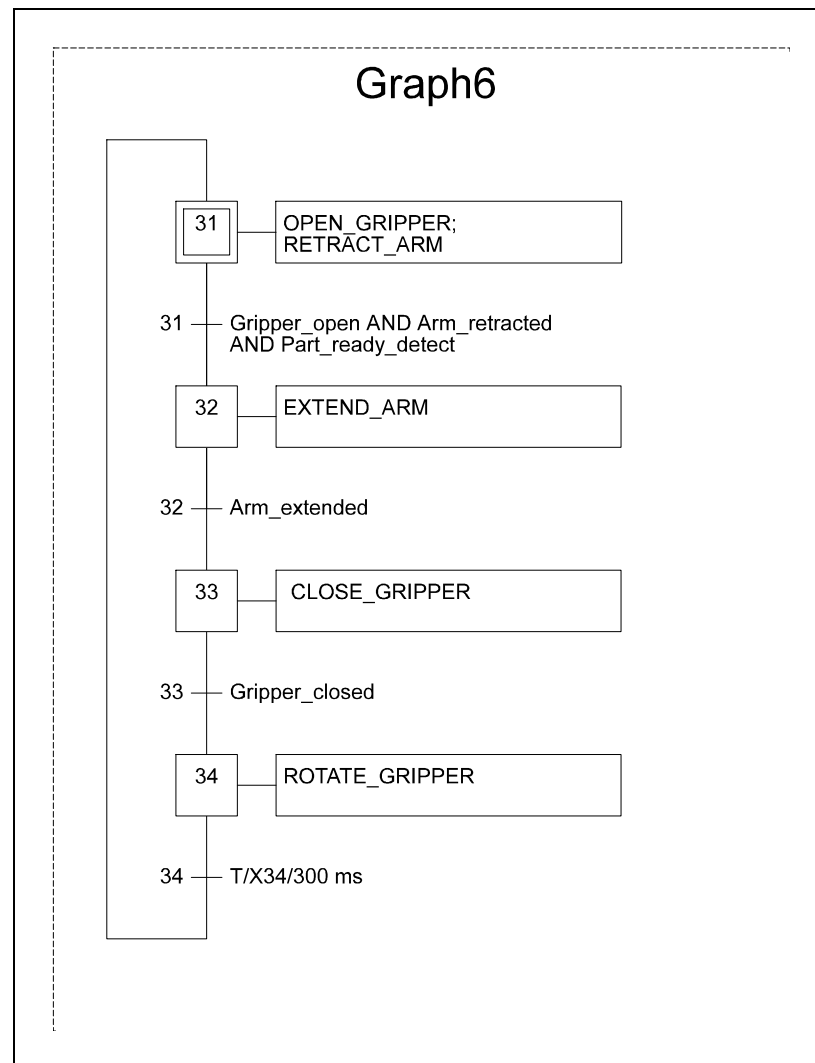


Figure 6.6

This structure is used for repetitive sequences of tasks. In this case a simple robotic gripper is being used to tighten a screw on assembly.

Loop with initialization

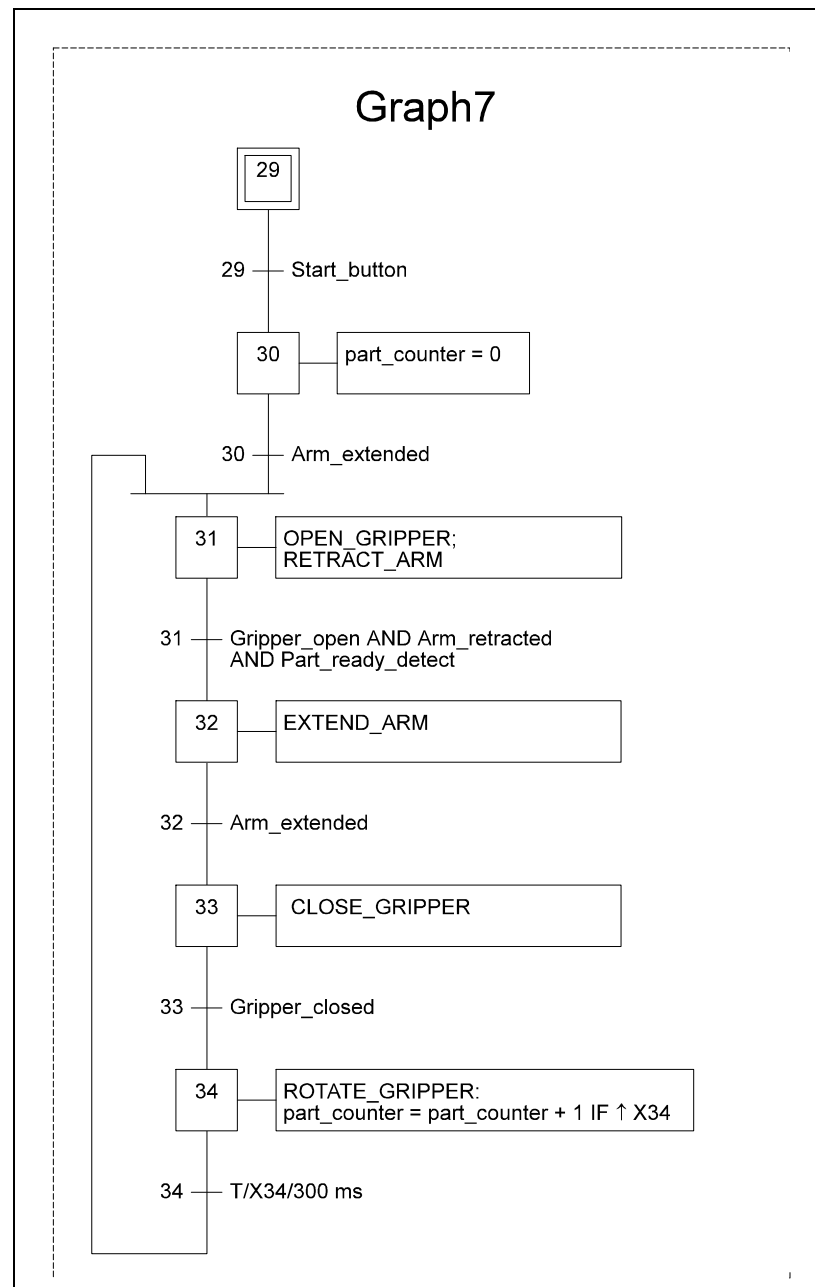


Figure 6.7

Same as Figure 6.6, but the gripper requires an initial set-up sequence before starting the repetitive cycle.

Degenerate unstable loop

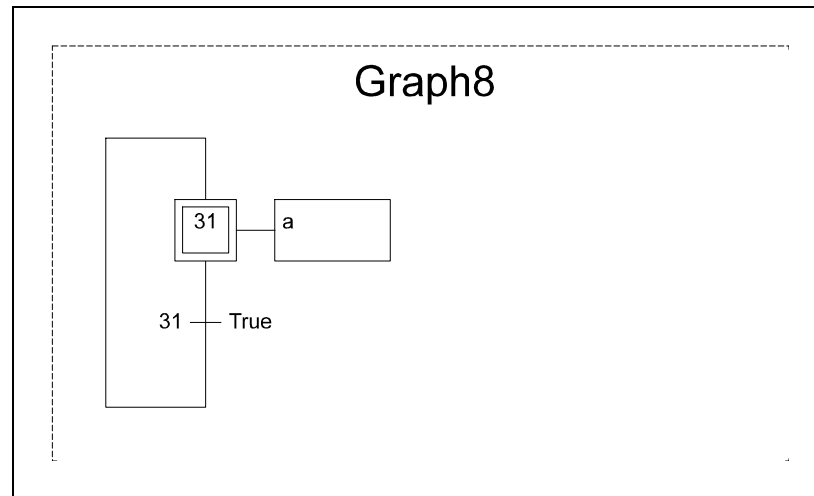


Figure 6.8

This loop is legal, but functionally it is identical to the permanent step shown in Graph 2. There is no reason for using it, except as a limiting case test of your Grafcet software. Some implementations of Grafcet cannot handle this special case.

It is the most simple example of an *unstable loop*, a loop that cycles continuously because all its trigger conditions are True. It has been proposed that future standards would prohibit unstable loops from Grafcet, but the subject is still under discussion.

Multiple initial steps

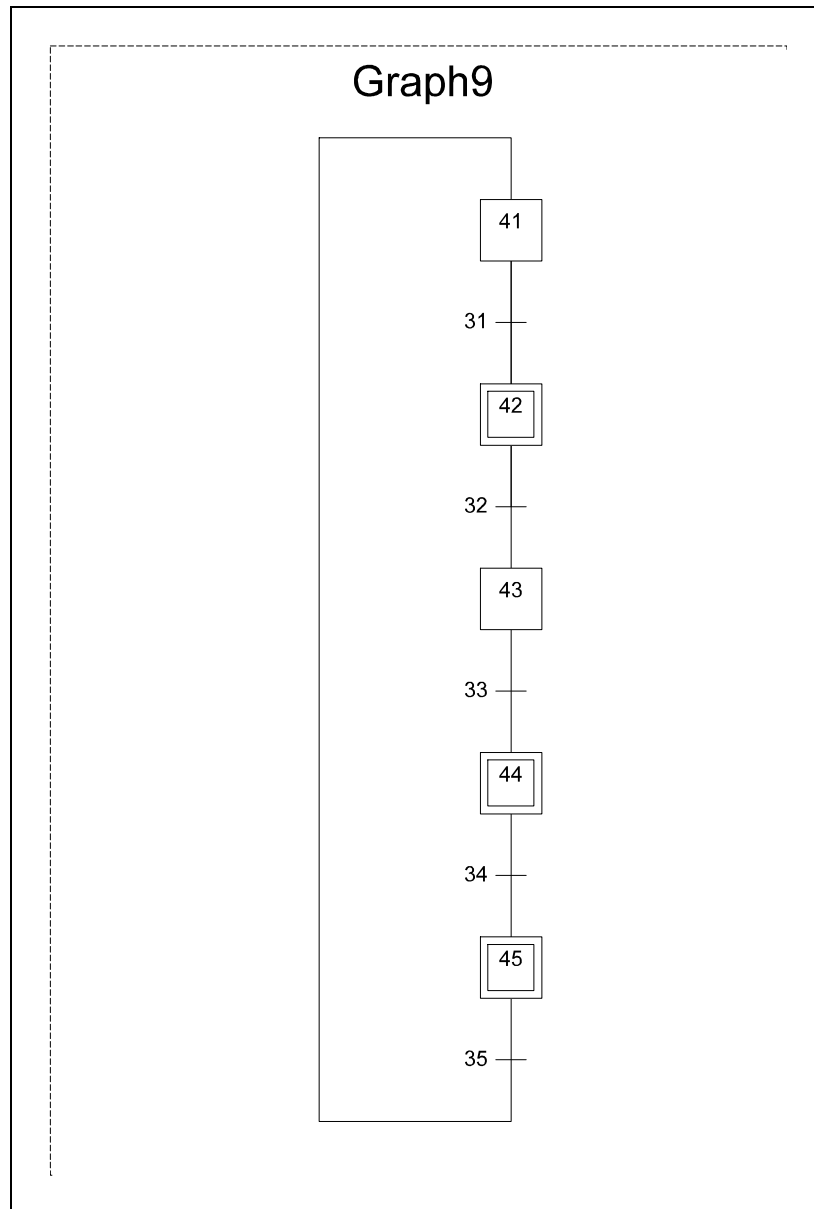


Figure 6.9

There is no rule saying the initial step must be at the top of a sequence or that a sequence can't have more than one initial step. Here is an example of a valid graph with three initial steps in the same sequence. I can't think of any application for this graph, but it is legal.

Sequence jumping

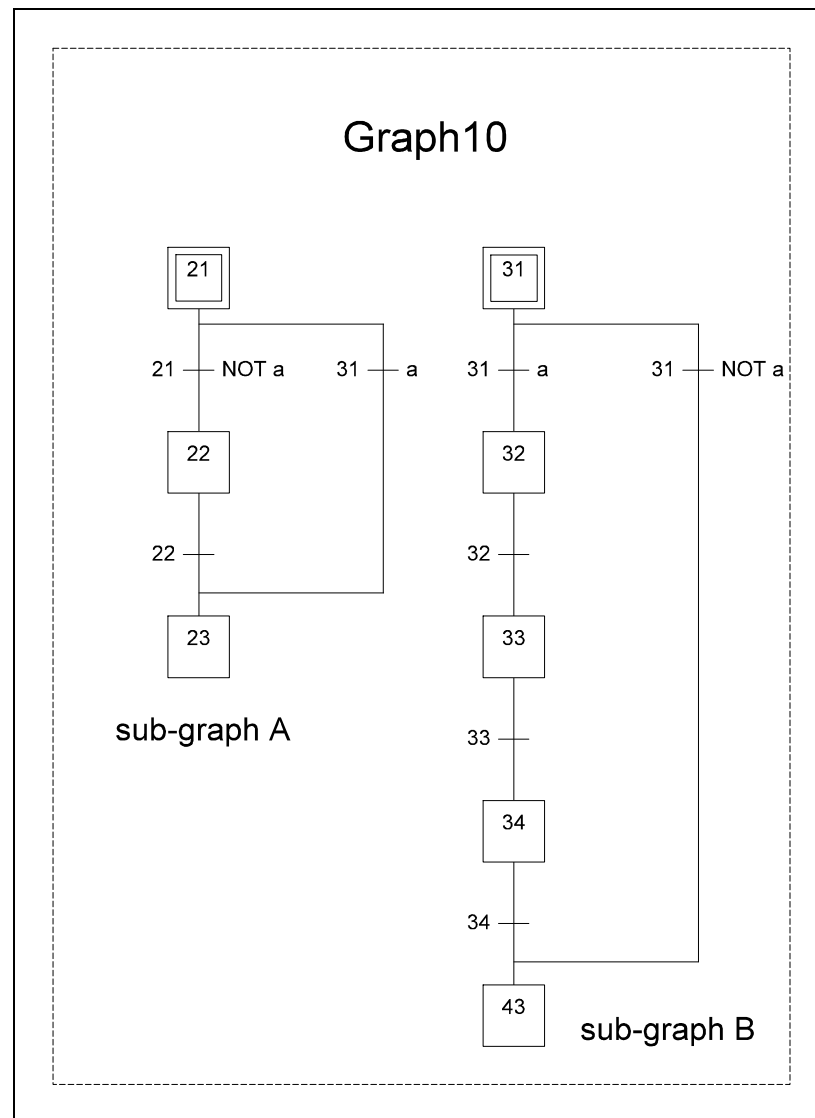


Figure 6.10

In sub-graph A, Step 22 is only executed if Variable *a* is NOT True. In sub-graph B, steps 32-34 are only executed if Variable *a* is True.

Sequence selection

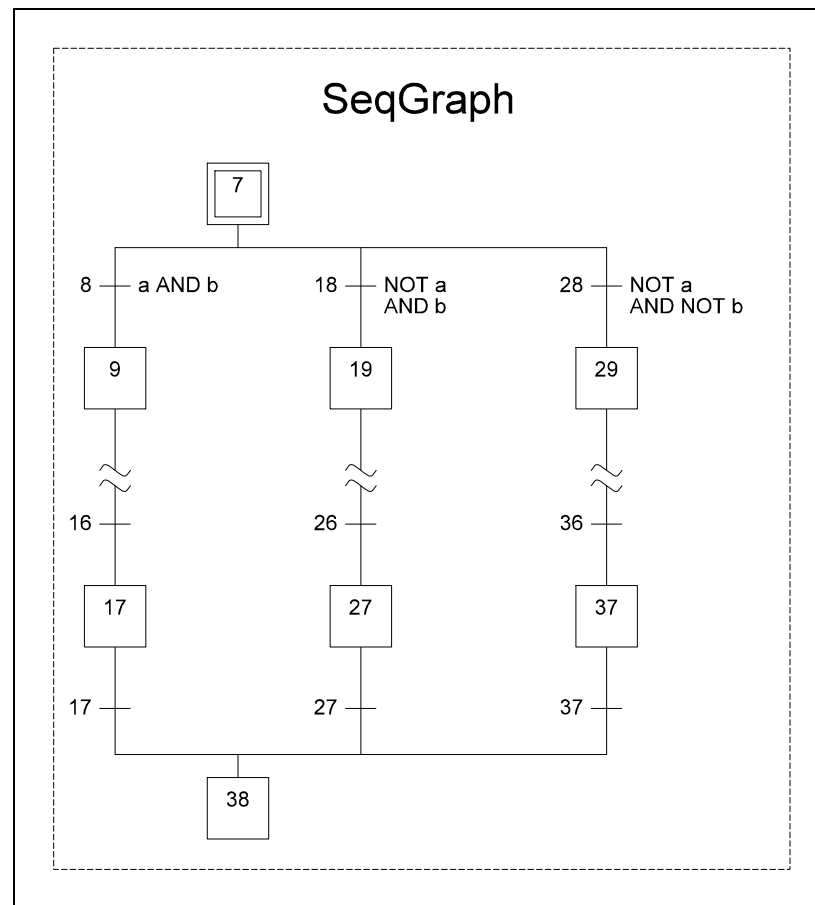


Figure 6.11

Mutually exclusive trigger conditions ensure that only one sequence is selected after Step 7. It is up to the programmer to insure that the preconditions are mutually exclusive. If they aren't, the result will likely be undesirable. (See the next page for an example.)

Interpreted parallelism

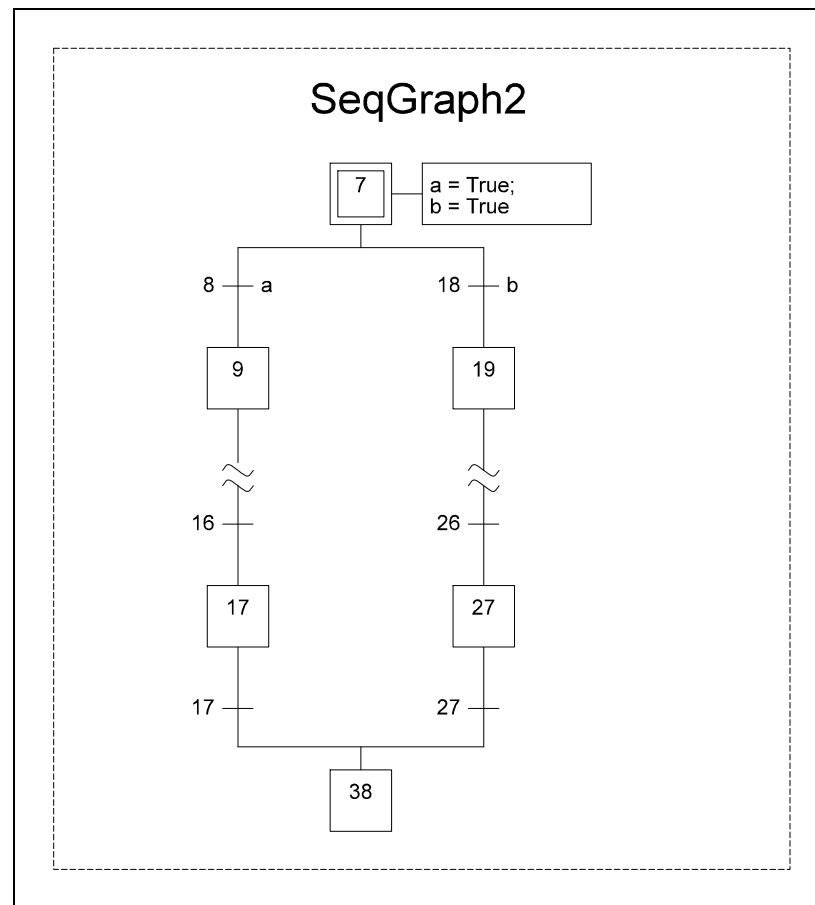


Figure 6.12

The OR divergence is inclusive OR! This means that if both transitions following the divergence have True preconditions, both will fire simultaneously. In Figure 6.12, the two sequences, steps 9-17 and 19-27, will then execute in parallel. This form of parallelism is called *interpreted parallelism* because it depends on the interpretation of the text statements. It is not recommended! Instead, use structural parallelism, i.e., the AND divergence structure.

Redundant sequence selection

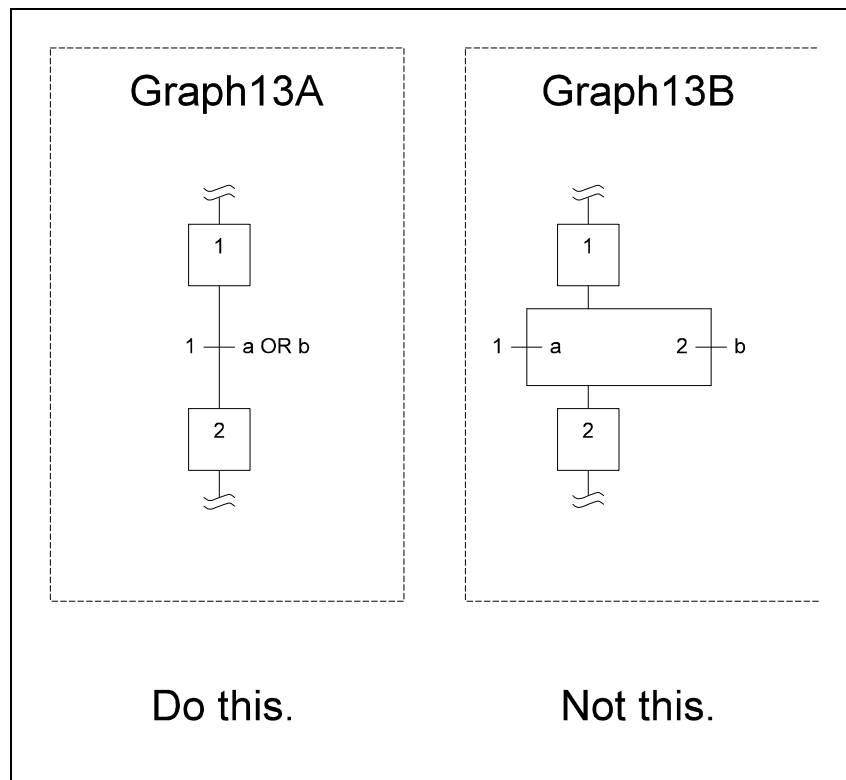


Figure 6.13

Graph13A and Graph13B are functionally identical. Both are legal Grafset structures. However, Graph13A is preferred, as it is simpler and more readable.

Another sequence selection example

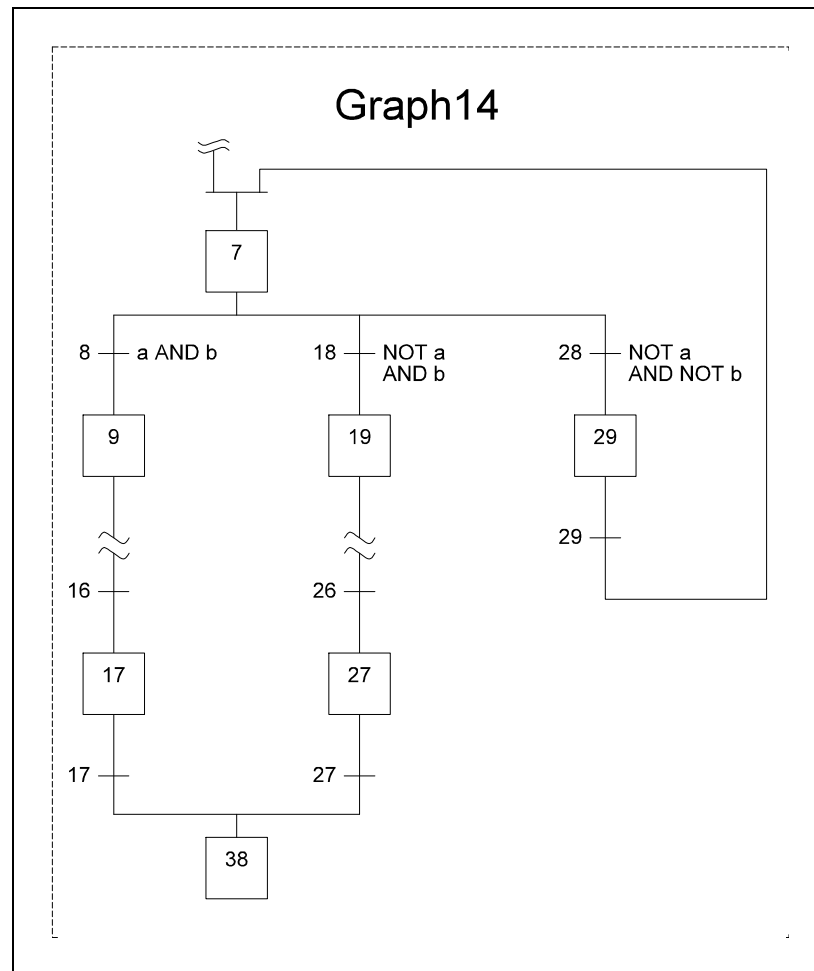


Figure 6.14

Here we see that OR divergences and OR convergences do not necessarily form matched pairs. Just because two sequences start together doesn't mean they have to end together.

Parallelism versus multi-statement actions

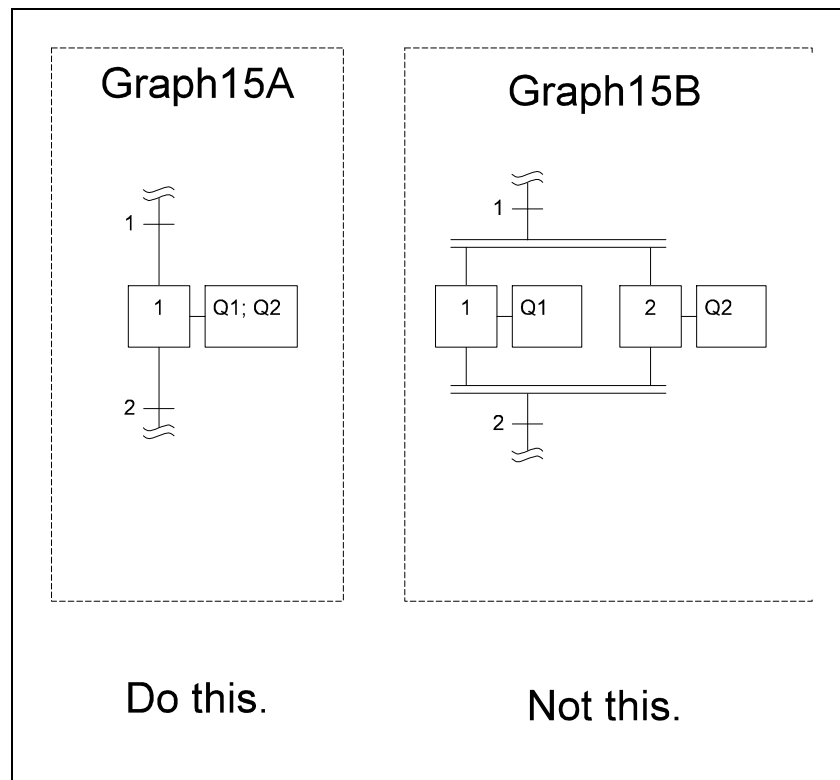


Figure 6.15

These two graphs are functionally identical.

The OR node or logic step

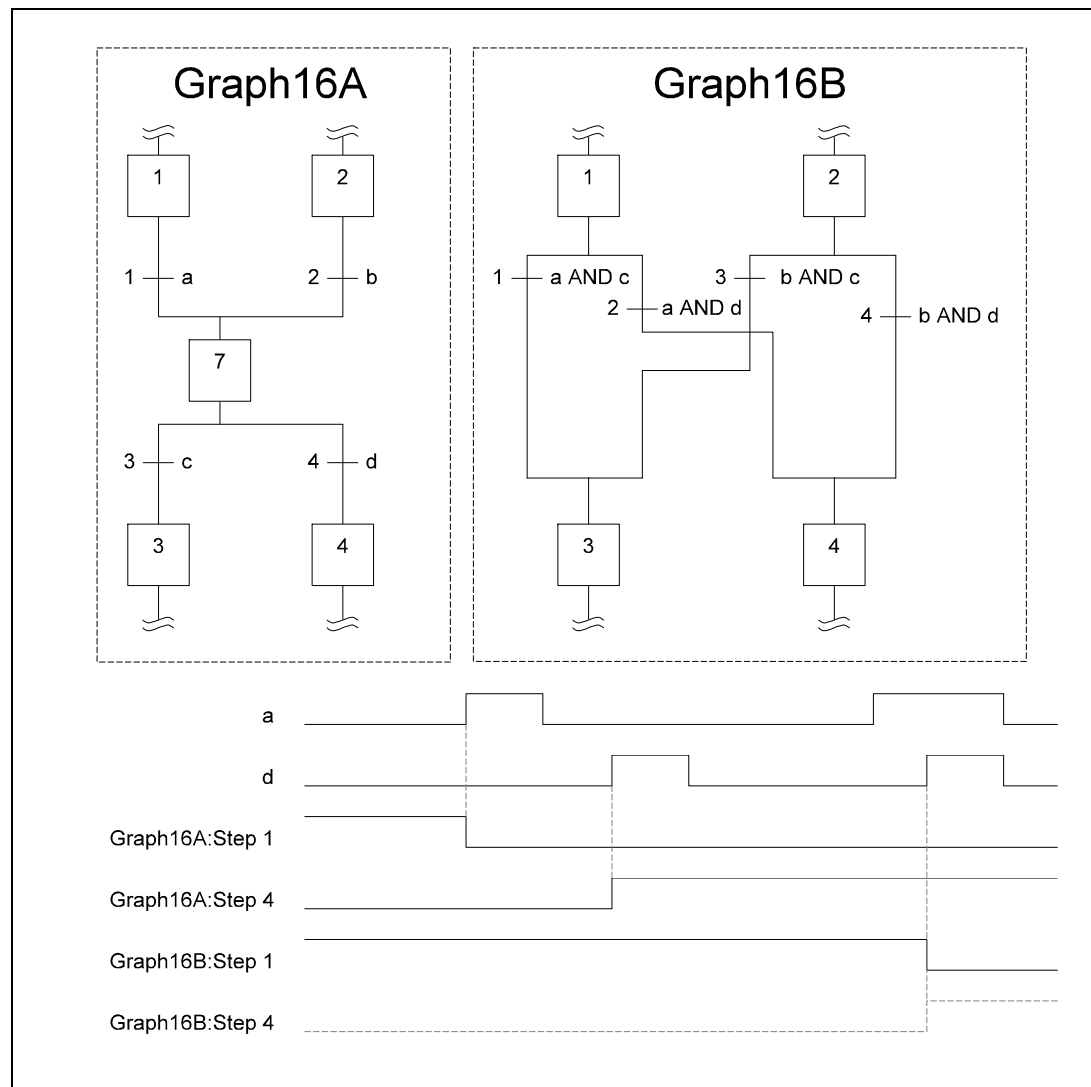


Figure 6.16

Sometimes we use a step, such as Step 7 in this example, to help simplify a graph rather than to represent control sequencing. This is called an *OR node* or a logic step. Note that the two graphs shown here do not have exactly the same behavior as is shown in the timing diagram.

Warning: When you use logic steps, make sure that the sequential evolution of the graph will always give the desired result.

Parallel sequences

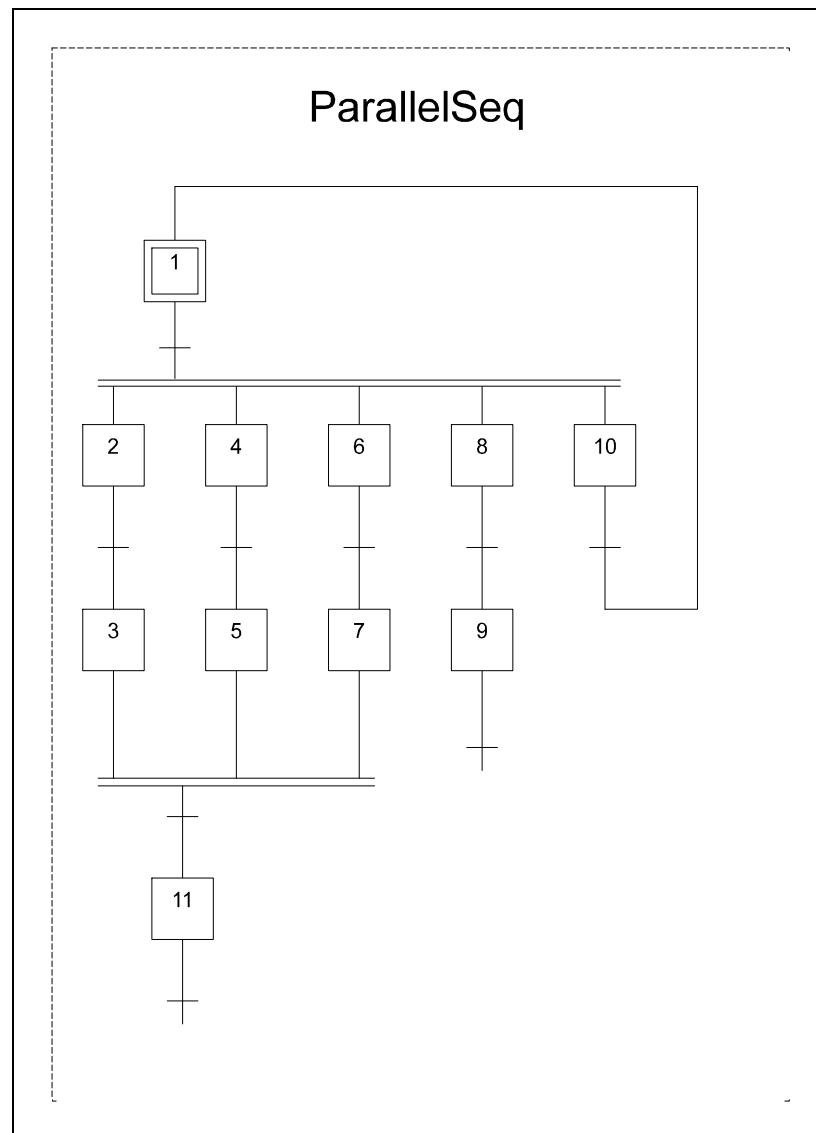


Figure 6.17

Here we see that AND divergences and AND convergences do not necessarily form matched pairs. Just because two sequences start together doesn't mean they have to end together.

Sequence synchronization (the AND node)

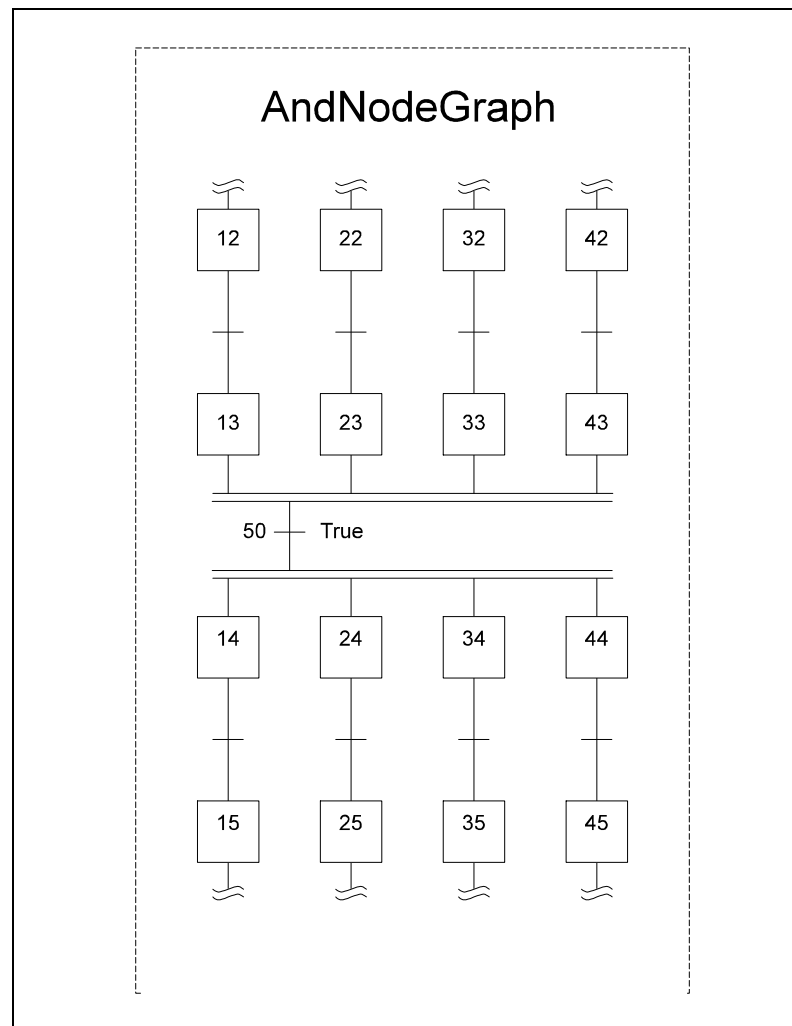


Figure 6.18

This structure, the *AND node*, is used to synchronize the execution of parallel sequences. Steps 14, 24, 34 and 44 cannot be activated until steps 13, 23, 33 and 43 are active.

Structural versus interpreted synchronization

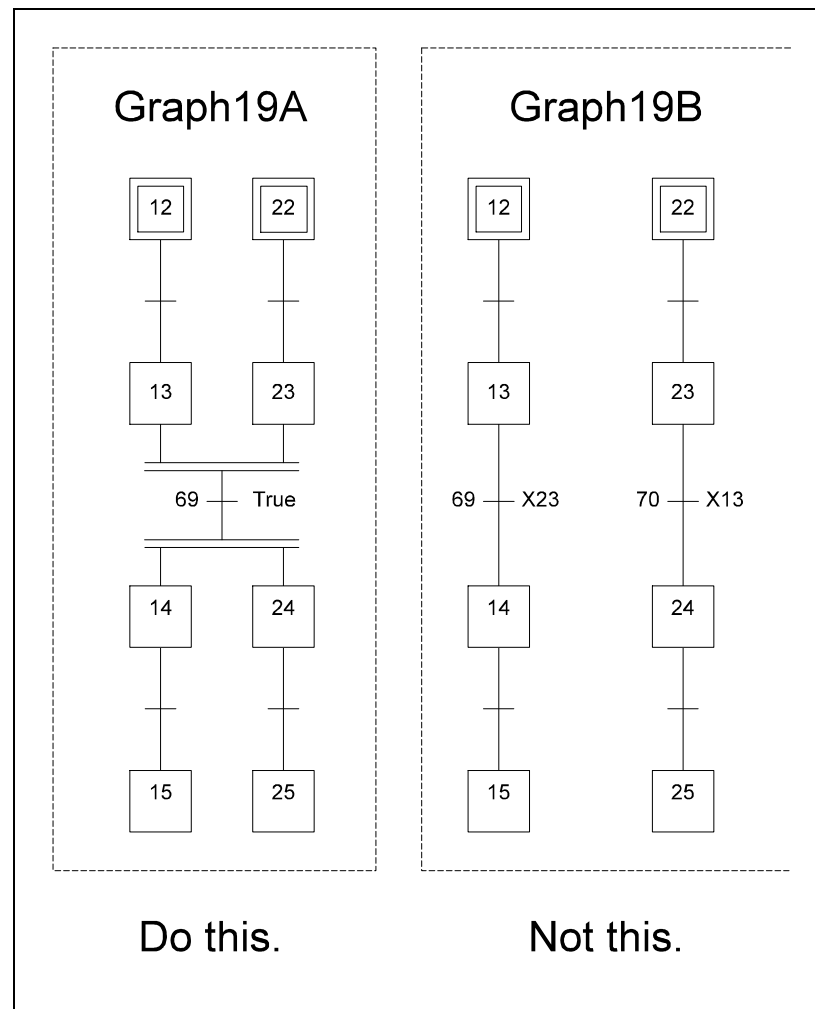


Figure 6.19

Graph19A synchronizes two sequences using an AND node. Graph19B uses the trigger condition of transitions 69 and 70 to do the same thing. The two sub-graphs are functionally identical. However, Graph19A has the advantage that you can renumber the steps without having to remember to modify the trigger conditions accordingly.

7 Cook-Book Examples

This chapter presents some Grafcet solutions to some typical control requirements. These can be modified and extended to cover many real-life control situations.

Safety interlock

For safety, the physical outputs of the controller must never be activated without first checking that all permissive conditions are satisfied and that no lock-out conditions exist.

The Grafcet Solution

Instead of operating directly on the output variables, use an output-request variable instead. Then use a stand-alone action to activate the physical output only when the necessary conditions are met.

Many companies have made this programming method standard, and insist that all physical outputs pass through an output request to ensure that interlocks and permissives are respected.

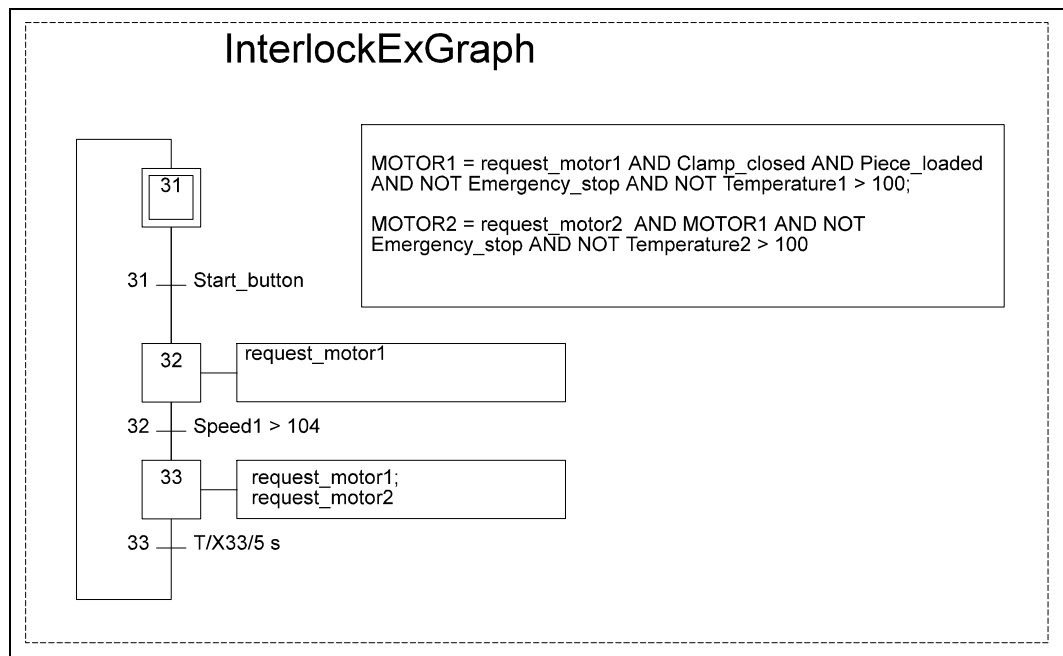


Figure 7.20

An example of safety interlock is shown in Figure 7.1. The step actions do not act directly on the outputs, MOTOR1 and MOTOR2. Instead, they issue requests for an output to be activated. These requests are handled by a stand-alone action. The statements in the stand-alone action are checked against the permissives and interlock conditions for each output. The requests to operate the motors are only carried out if the necessary conditions are True. In this case, MOTOR1 is activated only if the piece is loaded, the clamp is closed, the emergency stop button is not pressed and its winding temperature is not over 100. MOTOR2 is activated only if MOTOR1 is on, the emergency stop button is not pressed and its winding temperature is not above 100.

Permanent monitoring

Continuous monitoring of safety shut-down conditions is frequently required in control systems. For security reasons, it is important that the operator not be able to turn off this function from the PLC's programming console (at least not without shutting down the entire program).

The Grafcet Solution

The preferred way to do permanent monitoring, especially of safety conditions, is by a graph containing a stand-alone action, shown as a basic structure in the previous chapter. The stand-alone action is executed continuously, and cannot be turned off by a forcing command.

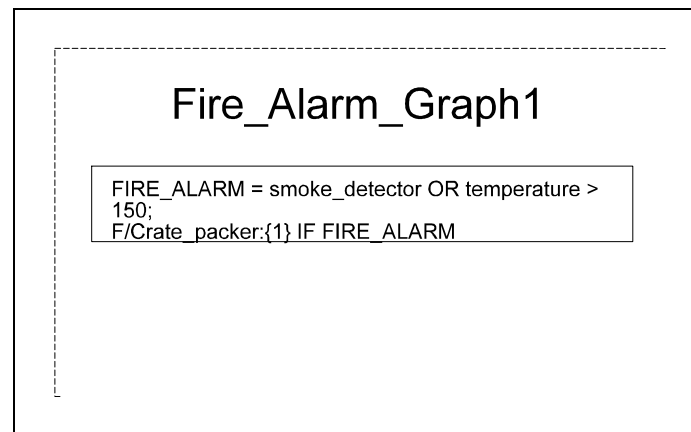


Figure 7.21

Emergency shut-down

A plant is controlled by an application containing three graphs: Seq1, Seq2 and Seq3. Each is independently responsible for controlling a section of the plant. All three control sequences are to be re-initialized if the operator pushes the emergency stop button.

The Grafcet Solution

An emergency shut-down graph uses forcing commands to return the rest of the graph in an application to its initial situation.

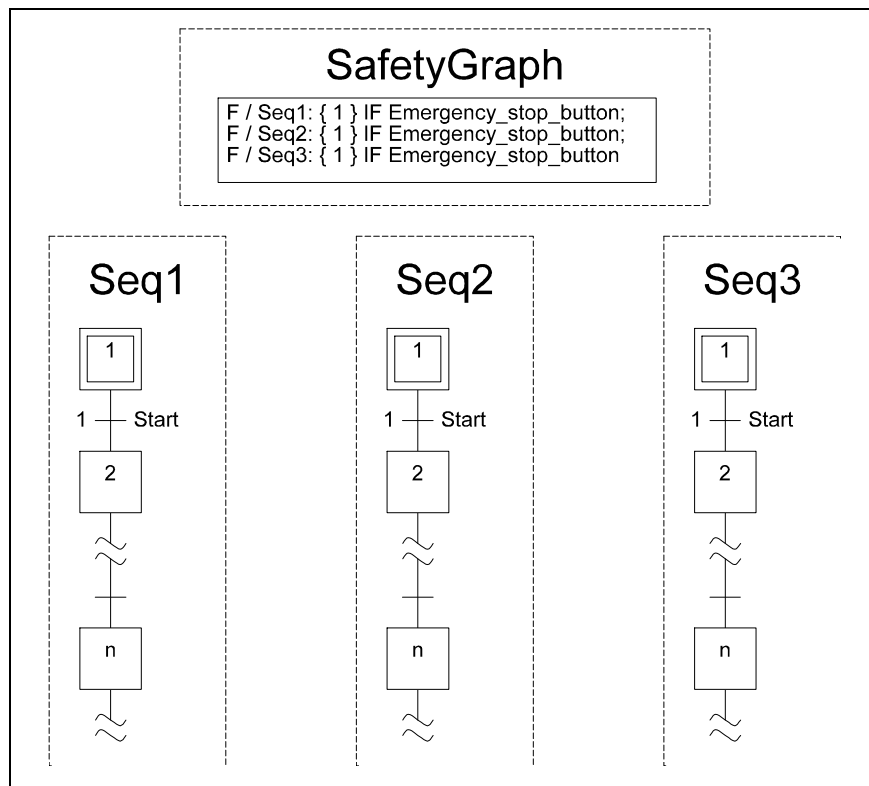


Figure 7.22

Note that the emergency shut-down graph, SafetyGraph, uses a stand-alone action rather than a permanent step. This ensures that the emergency graph cannot be bypassed by manually forcing it to the empty situation using the PLC's maintenance console.

Event counting

Within a larger program, we want to count external events.

The Grafcet Solution

Figure 7.4 presents variations on a basic method for counting events, in this case the opening of a valve. The Counter1 graph shows the most basic way of event counting. Counter2 improves on Counter1 by using a rising edge operator to detect the opening of the valve. This makes it more compact and easier to understand. The Counter3 graph does the same thing but uses source and sink transitions. It presents no advantage over Counter2 and has the drawback that not all Grafcet implementations support source and sink transitions.

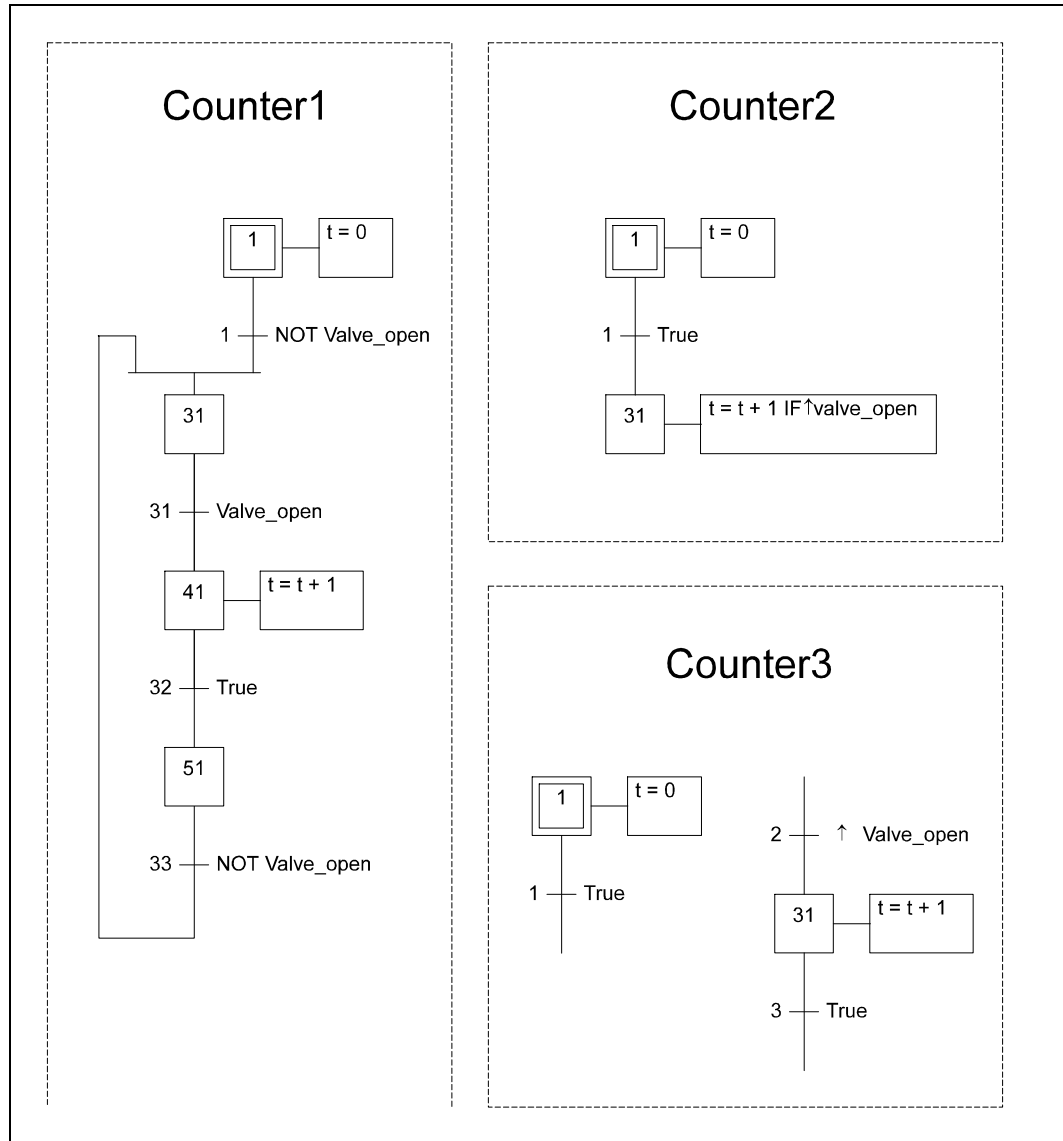


Figure 7.23

The operation of each graph is explained below:

1. Counter1 has eight elements: four steps and four transitions. Step 1 zeros the counter variable, t. The token passes to Step 31 when the valve is closed. Step 31 executes no explicit action, it simply waits for the valve to open. When this happens the token moves to Step 41, and t is incremented. Transition 32 fires immediately and the token moves to Step 51. Like Step 31, Step 51 executes no explicit action, it simply waits for the valve to close. When the valve has closed, control passes back to Step 31 and the cycle continues.
2. Counter2 has only three elements, two steps and one transition. Step 1 zeros the counter variable, t. Transition 1 fires immediately, and control passes permanently to Step 31, where t is incremented on each rising edge of the valve_open signal.

Note: You could also replace Counter2:Step 31 by a stand-alone action.

3. Counter3 has five elements: two steps and three transitions. Two of the transitions are sink transitions and one is a source transition. Step 1 initializes the counter variable. Step 31 is activated by Source Transition 2 each time the valve opens. It increments t once, and then it is deactivated on the next cycle by Sink Transition 3.

Some commercially available Grafset systems (particularly those provided by PLC manufacturers) may have problems with source and sink transitions. Some can't even handle terminal steps as in Counter2. All can handle Counter1, and a competent Grafset translator software tool will have no problems with any of these structures.

Program structuring

We want to use hierarchical structured programming to break down a large control program into modules that are easy to maintain.

The Grafcet Solution

The program is broken down into tasks that are called from a main graph. Figure 7.5 illustrates one of the most useful Grafcet structures: hierarchical tasking. In this system, one graph calls others in a top-down organization. In Figure 7.5, Main calls Task1, Task2 and Task3.

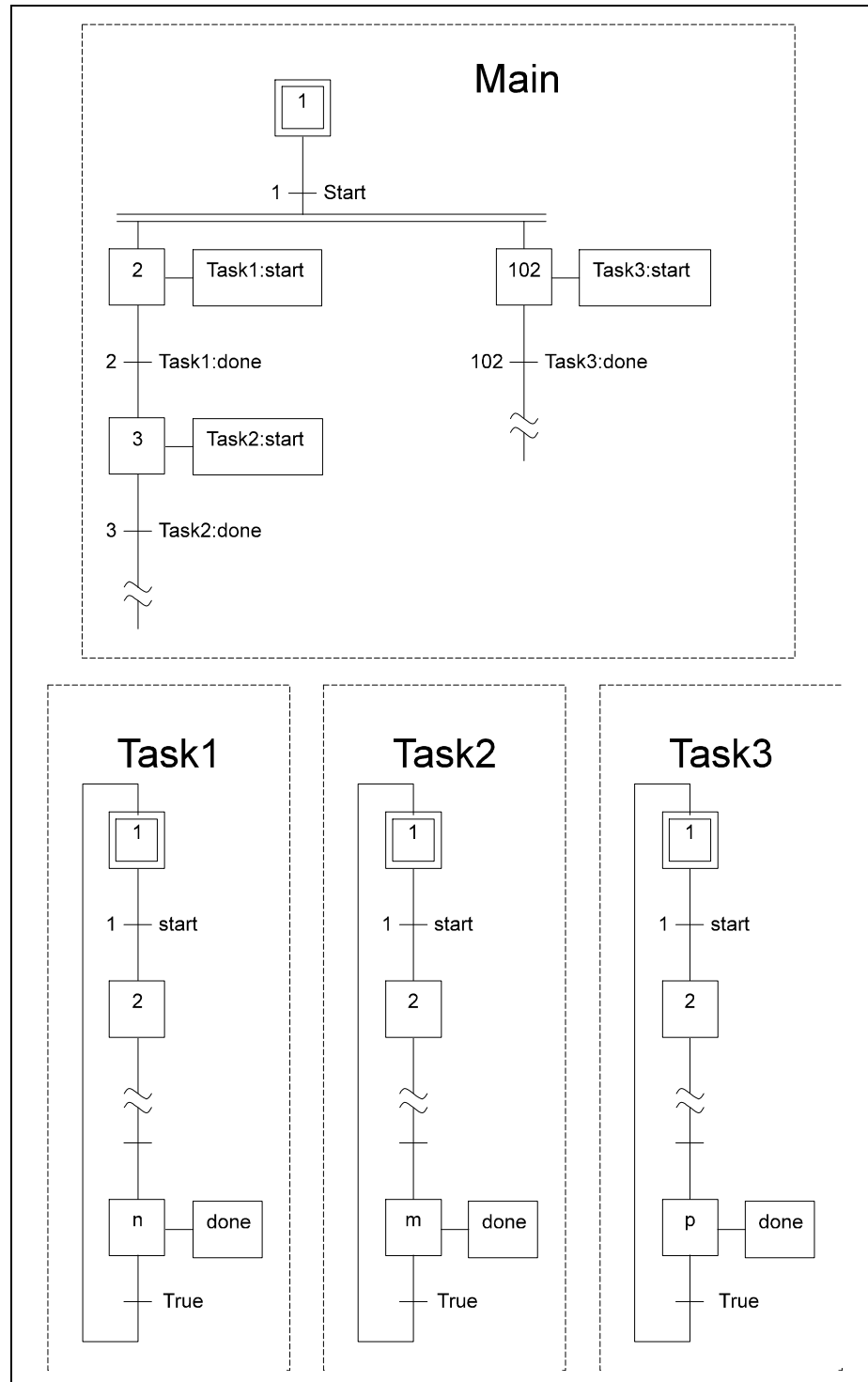


Figure 7.24

A task is called by means of its own local variables. A task needs at least two local variables:

- done - signifies to the caller graph that the task is finished its last execution and is free to be called

- start - used by the calling graph to start execution of the task

When a local variable is used in a different graph, it must be prefixed by the graph name. Thus *Task1:start* refers to local variable *start* in Task1 while *Task2:start* refers to local variable *start* in Task2.

Each task is called when the main graph activates the task's local *start* variable. For example, Task 1 is called by Step 2 in Main which momentarily activates Task1:start. Main then waits until Task1 has finished as indicated by its done variable (tested by Transition 3).

This section describes an actual automation project carried out by a major electric utility, Hydro-Québec. This company was the first North American utility to apply Grafset to control problems.

Hydro-Québec is a generating, transmitting and distribution company serving the province of Québec and exporting power to the New England states and the neighboring provinces. It has some 53 hydroelectric power plants and 26 thermal plants providing a capacity of over 30 GW, a transmission system comprising more than 10,000 km of extra-high voltage lines and 549 substations.

In recent years Hydro-Québec has started to automate its transmission and distribution substations. Functions being automated include:

- Breaker re-closing
- Transmission line switching
- Automatic substation synchronization
- Load shedding
- Service restoration
- Voltage reduction (-5%)
- Capacitor bank switching

These automatic functions are being implemented in a unique way: using intelligent remote terminal units (RTUs) programmable in the Grafset sequence control language.

Why Grafset?

In 1984 Hydro-Québec reviewed existing methods for specifying the behavior of automatic controllers. It was clear that traditional methods based on combination logic, such as relay ladder diagrams, Boolean equations or gate logic diagrams were largely inadequate. While these methods are excellent for simple On/Off type logic, none of them offer an intuitive representation of sequential control. For instance, most relay logic designers are forced to use state-transition methods to analyze the temporal behavior of their programs.

The 1984 study considered only models of control which could explicitly handle sequence control: state-transition models, Petri nets and Grafset. After considerable debate, the study group submitted its recommendations in spite of resistance from the more conservative engineers who argued that,

“We should stick with the methods we already know”,

“Grafset is new, unproven and not widely accepted

(and not even invented here!)”,

and

“Software tools such as editors, compilers, etc. are not commercially available - we’ll have to develop our own”

State-transition methods were rejected because they offered no way of specifying parallel control actions. Petri nets, while powerful, could not be recommended for general use because the analysis and understanding of Petri nets requires too much mathematical expertise. Grafcet turned out to be just right. The study recommended that Grafcet be adopted as a Hydro-Québec standard for substation and power plant automation.

The transmission line switcher

This example, an automatic substation high voltage line switching, is one of the first to have been implemented at Hydro-Québec at its Magnan substation near Montreal and has been fully commissioned since 1990. This example uses a small but powerful subset of Grafcet’s features: Boolean logic, sequencing, timing and edge detection.

The physical plant

Figure 8.1, a simplified one-line diagram, shows the electrical configuration of the Magnan substation. The substation currently has two transformers, but two more are to be installed in the future as demand for power increases. The existing two transformers, T2 and T3, may be supplied by either of two 125 kV lines, L1 or L2, through breakers B1 and B2. Normally only one of the two breakers is closed. The output of the transformers is distributed to eight 38 kV feeders (14 are planned).

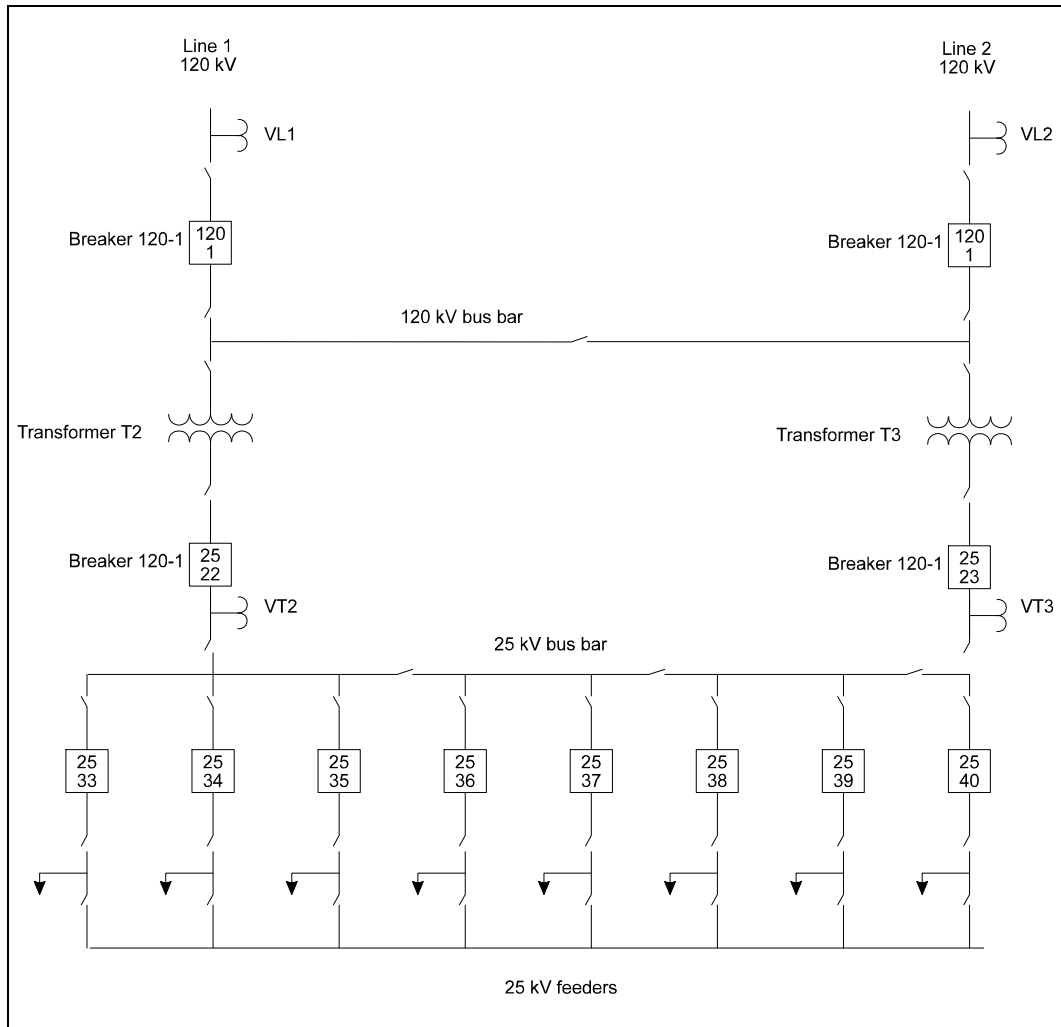


Figure 8.1

The input signals are:

Onduty	- from operator console
Unlock	- from operator console
VL1	- voltage present on line 1
VL2	- voltage present on line 2
VT2	- voltage present on transformer 2
VT3	- voltage present on transformer 3
F_B2	- fault on high voltage bus bar
Breaker_1	- breaker 120-1 closed
Breaker_2	- breaker 120-2 closed

The output signals are:

BL10	- open breaker 120-1
BL20	- open breaker 120-2
BL1C	- close breaker 120-1
BL2C	- close breaker 120-2

The internal variable used is:

Locked	Indicates that a hardware error has occurred in the substation. This variable is stored in non-volatile memory so that its value is retained even after a power failure to the controller
--------	---

The automation requirement

In normal operation, one line is in use and the other is on standby, i.e., only one of the two high voltage breakers (120-1 and 120-2) is closed.

Briefly, the line switcher is required to do the following:

- If there is a loss of voltage on the high voltage line currently in use, wait *td* seconds (parameter *td* can be set between 0.5 to 10 seconds) to see if the problem corrects itself, then switch operation to the other line (if it is working).
- If a breaker refuses to operate or if there is a fault on the high voltage bus bar then go to the “locked” state and remain there until the “unlock” signal is received.
- Memorize (in non-volatile RAM) if “locked” and return immediately to the locked state on power-up following a power failure.
- Obey on/off duty signals from operator interface.

The Grafcet solution

The Grafcet program that implements the automatic line switching is shown in Figure 8.2 (next page).

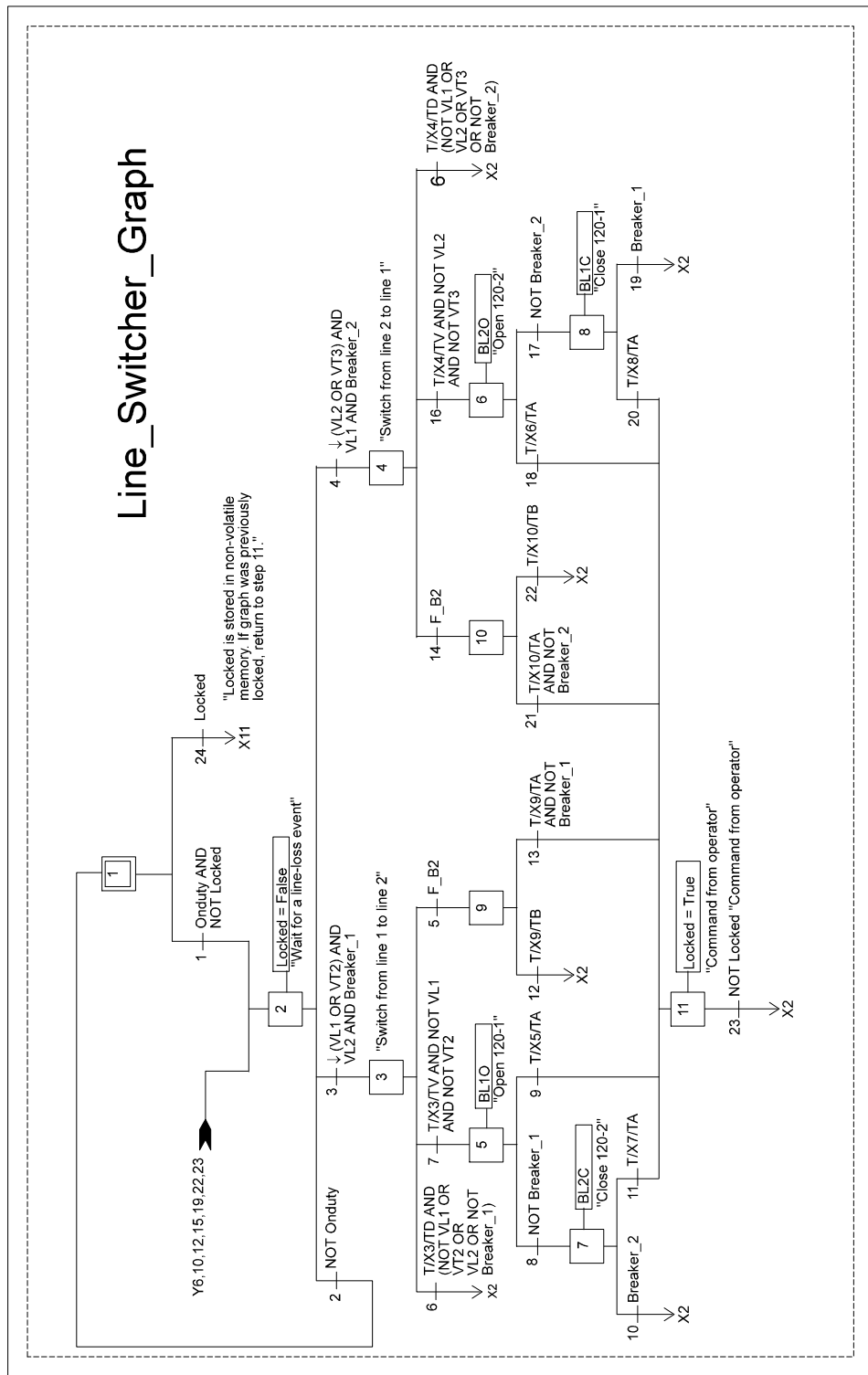


Figure 8.2

Step 1 is active on power-up. Step 2 is the normal on-duty step, where the program waits for a line loss. If line 1 is lost, the normal sequence of events is Step 3, Step 5, Step 7, and return to Step 2. Step 3 waits TD (0.5-10) seconds to see if the problem corrects itself. Step 5 opens breaker 120-1, cutting off line 1. When it has opened, Step 7 closes breaker 120-2, connecting line 2. When line 2 is connected, control returns to Step 2. If there is a high voltage bus bar fault, the program goes to Step 9 and waits for the protection to open the breaker before going back to Step 2. If a breaker refuses to operate, the program goes to Step 11, the locked state.

If line 2 is lost, a similar sequence runs through Steps 4, 6, 8 and 2.

The various steps and transitions that make up the graph are described in more detail below:

Step 1

When the system is initialized, Step 1, shown by a double box, is active. The graph executes no action while in Step 1. The graph remains in Step 1 while it is not on duty.

If the *On-duty* bit is True and the *locked* bit (in non-volatile RAM) is False then Transition 1 fires, transferring control to Step 2.

If the *Locked* bit is True, indicating that the graph was in the locked state prior to a power failure, then Transition 24 fires, passing control to Step 11.

Step 2

Normally, the graph waits in Step 2 for an event. An event is loss of voltage on one of the high voltage lines or a command from the operator. The only action executed by Step 2 is to clear the *locked* bit.

If the *On-duty* bit becomes False (due to a command from the operator), then Transition 2 fires, returning the graph to Step 1.

Transition 3 starts the sequence of switching from line 1 to line 2. It is triggered by $\neg(VL1 \text{ OR } VT2)$, a loss of voltage on line 1 or on transformer 2, and is conditional on, *VL2*, the presence of voltage on line 2 and on *Breaker_1*. Obviously there is no need to switch if line 2 is dead or if *Breaker_1* is opened by a protection relay. If *Breaker_1* has been opened by a protection then the substation should be left unpowered until the situation that provoked the protection has been repaired.

Transition 4 starts the sequence of switching from line 2 to line 1. Its operation is similar to that of transition 3.

Step 3

The first step of the line switch sequence is to wait to see if the line loss situation corrects itself. The delay time is a parameter that can be chosen by the user.

If voltage returns to line 1 during the first TD (0.5-10) seconds or if line 2 is also down, no switch is made and control return via Transition 6 to step 2.

If the condition is still present after TV (0.5-12) seconds then control passes via Transition 7 to Step 9.

If there is a fault on the high voltage bus bar, the graph goes via Transition 5 to Step 9.

Step 5

This step opens breaker 120-1. If the breaker opens normally, the sequence continues through Transition 8 where breaker 120-2 is closed. If after TA (0.5-2) seconds the breaker has not opened, the control passes through Transition 9 to the locked state, Step 11.

Step 7

Executes the action of closing breaker 120-2 to complete the line switch sequence. If the breaker closes normally then the sequence returns through Transition 10 to Step 2 where it awaits another event. If after TA (0.5-2) seconds the breaker has not closed the control passes through Transition 11 to the locked state, Step 11.

Step 9

Step 9 is activated if a fault appears on the bus bar. Its job is too make sure that Breaker_1 has been opened (by the protection circuits). If the breaker is still closed after TA (0.5-2) seconds then control passes through Transition 13 to the locked state, Step 11. Normally, the breaker does open, and control passes through Transition 12 back to Step 2 (normal operation) after tb seconds. Note that tb must be greater than TA for correct operation.

Step 11

The locked state, Step 11 sets the non-volatile variable *Locked* which is displayed on the operator console and which is used when the system is returning to operation after a power failure.

When the *Unlock* signal is received from the operator console then the graph returns to its normal operation in Step 2.

Step 4, 6, 8, 10

Similar to steps 3, 5, 7, 9.

Comparison to the traditional solution

Prior to the introduction of Grafcet to Hydro-Québec, the line switcher would have been implemented by a 25 page Relay Ladder Language program on a programmable logic controller. The Grafcet program on a single page is not only much more compact, but also more intuitive. The shorter and more intuitive a program, the less risk of programming error.

Conclusions - Hydro-Québec's experience with Grafcet

Hydro-Québec's experience with Grafcet has been very positive:

- The method has lived up to its promise of saving time and money. Its compact and rapid automation programming has shortened development and start-up compared to traditional methods.
- Grafcet has helped Hydro-Québec improve its internal documentation of automation projects.
- Grafcet has been enthusiastically accepted by Hydro's technicians and engineers who find it easier to understand than Relay Ladder Language. A typical comment is that “when you know Grafcet, you never want to use ladder programming again”.
- Real-time supervision - animated Grafcet, where the active steps are highlighted on your monitor screen, has been a great help in speeding start-up. Animation shows you not just the state of the controlled process but also what the controller is trying to do.

On the expenses side of the balance sheet, considerable extra development costs were incurred. Because Grafcet is a relatively new method, no tools were commercially available when Hydro-Québec started this project. The decision to adopt an innovative method meant that Hydro-Québec had to develop its own Grafcet compiler. This required several years of work and cost hundreds of thousands of dollars. Nonetheless, this activity fell within Hydro-Québec's mandate for R&D.

Although it is difficult to produce exact figures for how much Hydro-Québec would have spent on automation if the Grafcet method had not been used, we believe that the R&D investment has already been recovered internally in reduced engineering costs. And as an unexpected silver lining (it should come as no surprise to engineers that R&D pays off), Hydro-Québec is now able to offer this technology to other utilities through its subsidiary, Famic Automation Inc.

9 Implementing Grafcet on the PLC

Basic Assumptions

In this chapter we will consider Grafcet as a model of computation, i.e., as a method for structuring real-time programs for the PLC or industrial computer. In general, a real-time system should be responsive, effective and deterministic.

Responsive means a real-time system responds to all external events in a timely way, and no events are missed.

Effective means a real-time system can always compute a response to each event.

Deterministic means a real-time system always behaves the same way in identical circumstances.

All real-time systems have to satisfy the above conditions as closely as possible, but no real-time system ever succeeds completely because of the finite response speed of the hardware.

In implementing Grafcet, in addition to strictly following the rules of evolution, we want to pay special attention to two considerations.

- The firing time of a transition is in theory infinitesimal, so we want to make it as short as possible in our PLC too. In practice, a PLC's scan cycle time is the minimum possible interval. This means that all enabled and triggered transitions will fire once in the same scan cycle, and that OUT in Figure 9.1 will be active for exactly one PLC scan cycle.
- Our implementation should work well in the case of races. It should remain robust even if the graph has structurally or functionally unstable situations, including unstable loops. Remember that a race is an unstable situation where the graph can evolve even when the input field doesn't change. For instance, Race_Graph in Figure 9.1 is functionally unstable when inputs A and B are both True when the controller starts execution of the graph. This is because the situation $S(\text{Race_Graph}) = \{32\}$ would last for only an infinitesimal time. Robustness under race conditions is particularly important in designing interrupt driven Grafcet systems.

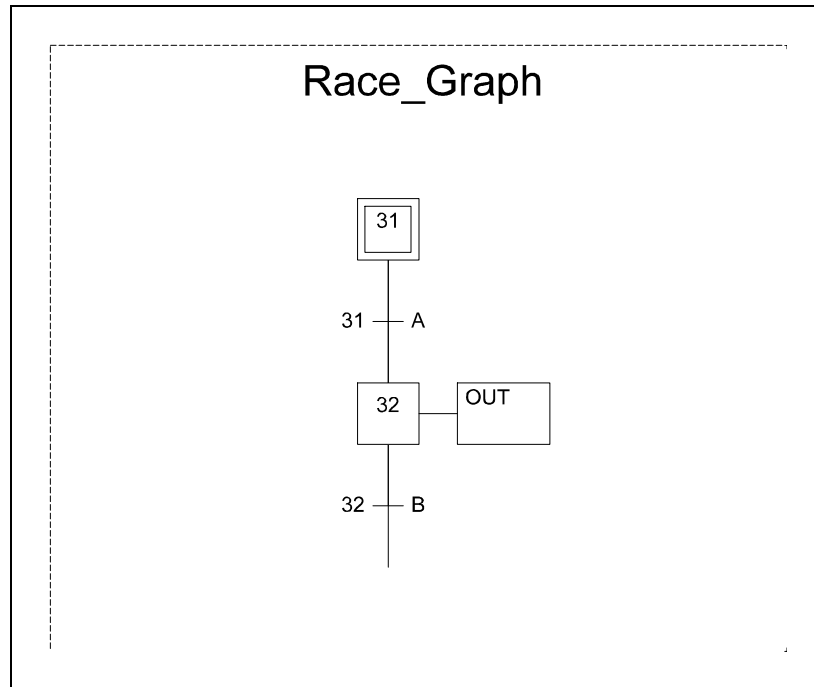


Figure 9.25

Implementation models

There are a number of basic approaches to designing a Grafset based control system (or any other control system). The most common of these are considered below:

The scan cycle model

PLCs and most industrial computers work by scan cycle. This has important consequences for the way that our Grafset implementations will work. A scan cycle is a single pass through a polling loop: First the PLC reads its input cards, copying the values into an area of memory called the *input image*. Next, the PLC performs internal calculations to determine its reaction to the inputs. Any output actions are stored in a region of memory called the *output image*. Finally, the output image is copied to the physical output cards. This cycle is illustrated in Figure 9.2.

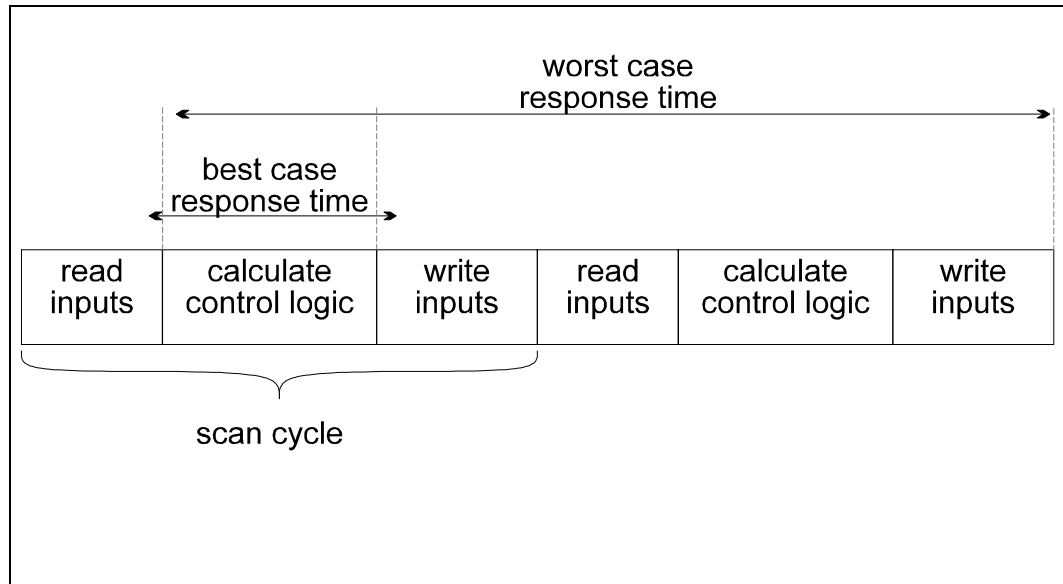


Figure 9.2

The scan time limits the responsiveness of a PLC. Referring again to Figure 9.2, we see that the best case response time is almost one scan cycle and the worst case is almost two scan cycles.

The algorithm for scan cycle based execution of Grafcet is given by the following pseudo-code:

```

Repeat
begin
    Execute various system chores
        Update real-time clock
        Communicate with supervisory systems
        Check system integrity
        Etc.
    Read inputs
    Detect rising edges
    Fire all armed and triggered transitions
    Execute stand-alone actions and actions of active steps.
    Write outputs
end.

```

To satisfy the evolution rules regarding the simultaneous firing of simultaneously armed and triggered transitions, it is important that all the steps be fired before executing any of the actions.

The interrupt model

Although PLCs work by scan cycle, industrial computers often work by interrupt. In fact, some engineers assume that *real-time* and *fully pre-emptive interrupt driven* are synonymous. An interrupt driven control computer makes use of hardware interrupts from the inputs to initiate control calculations. An interrupt driven Grafcet system is considerably more complex than a scan cycle implementation. At the time of writing only one such system has been implemented, Hydro-Quebec's Smart Station System.

State machine models

A small graph can easily be translated into a finite state machine where each state corresponds to a situation of the original graph. The big advantage of state machines is their efficiency. State machine implementations, whether in hardware or in software, tend to yield very fast response times. The problem is that there can be many times more situations than steps, and in general the state machine representation can be intractably large. For instance, it could take more than 400,000 states to replace a graph with only 100 steps.

Manual translation into PLC language

In this section, we will demonstrate how to translate Grafcet into PLC Relay Ladder Code. We use a simplified ladder language in our examples, but it will be easy for you to adapt the method to your own PLC's native ladder.

A simple example

To demonstrate the translation of graphs in ladder diagram format, let's take a simple example shown in Figure 9.3. The Relay Ladder Language implementation of this graph is shown in Figure 9.4.

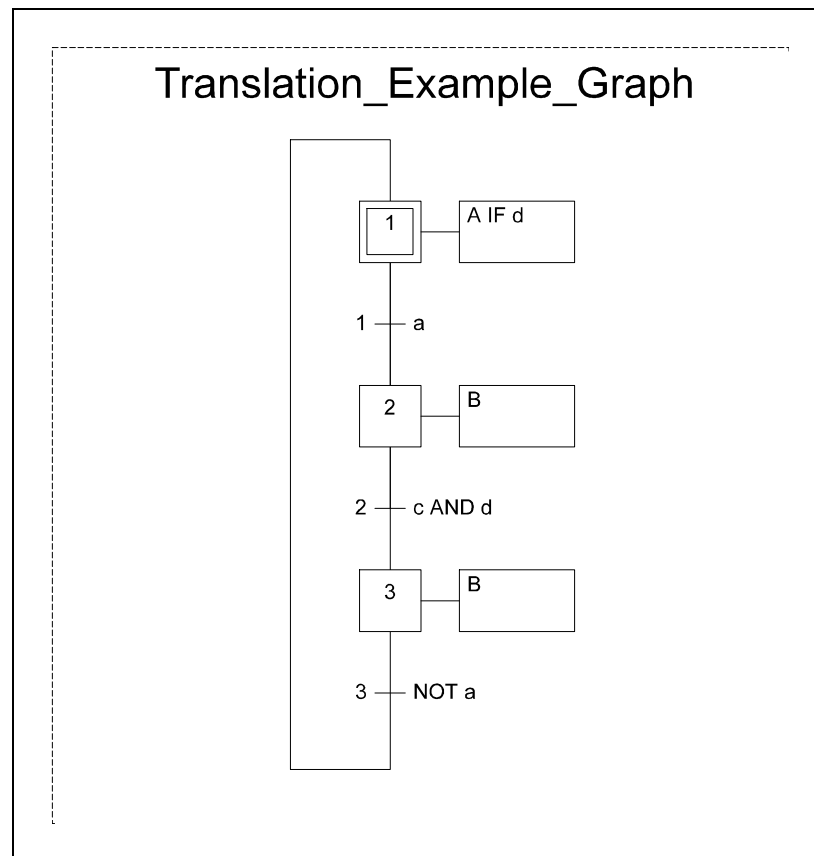


Figure 9.3

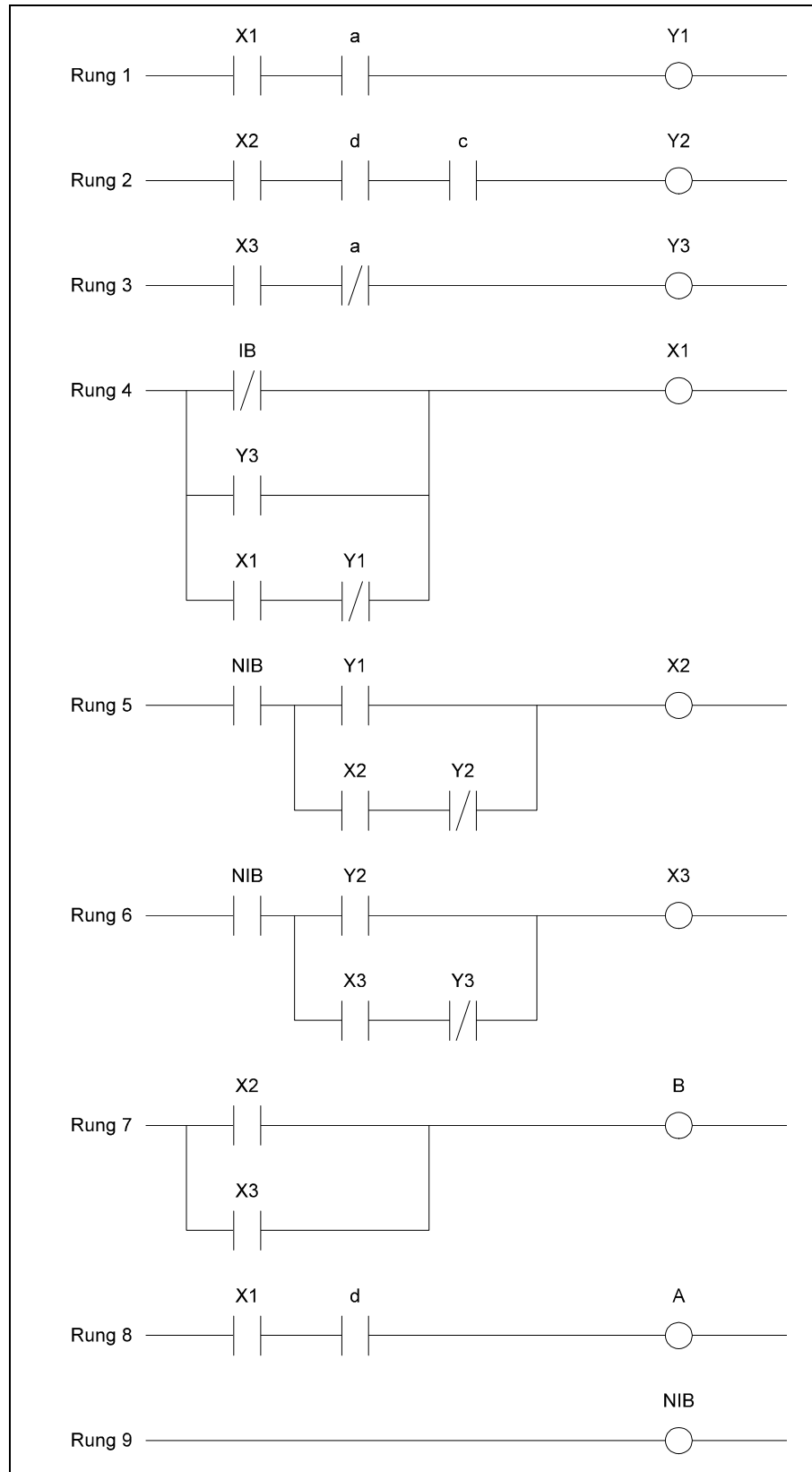


Figure 9.4

In the above example there are four different program blocks:

- rungs 1-3 - transition rungs, one for each transition in the graph
- rungs 4-6 - step rungs, one for each step in the graph
- rungs 7,8 - action rungs, one for each output variable
- rung 9 - the initialization rung.

You must place these four blocks in the order given if you want a method that always works. In some simple cases, you can save a few contacts and/or rungs by trying to amalgamate the rungs, but you run the risk of creating a program that doesn't behave properly under certain conditions, or that cannot be easily modified.

General method

The general method for translating Grafcet uses six functional program blocks:

1. Initialization rung
2. Transition rungs
3. Step rungs
4. Timer rungs
5. Edge-detection rungs. (Only required for PLCs that do not have edge detection contact instructions (—|↑|—)).
6. Action rungs

The different program blocks are described in more detail in the following sections.

Initialization rung

Although the initialization rung, Figure 9.5, is always the last rung in a program, we consider it first, because it is used in the step rungs. The initialization bit is used so that the PLC program can distinguish between the initial execution of the program and all subsequent executions. This is important for initializing the graphs. The initialization bit, NIB, is False at power-up. It remains False through the first scan cycle of the program and it is set at the end of the first cycle by the initialization rung. When NIB is True it means the program is not on its initial scan. The only way that NIB can be cleared is if a F/Graph:{ I } instruction is executed to re-initialize the graph.

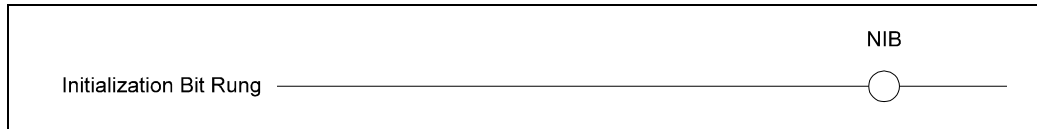


Figure 9.5

Figure 9.6 shows an initialization rung for a graph including a forcing instruction.

F/Graph:{ I } IF Emergency_PB.

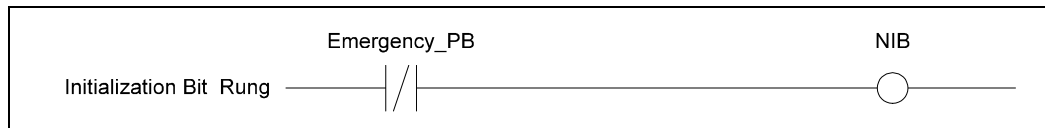


Figure 9.6

Transition rungs

Each transition n is represented by a transition bit, Y_n , which takes the value True during each scan in which the transition should fire. As you may recall, a transition fires if it is both armed and triggered. Use a rung for each transition such that each transition coil is energized if these conditions are met. An example is shown in Figure 9.7, where transition bit Y_2 is set when:

- Transition 2 is armed (X_1 and X_2 are True, indicating that all the entrance steps are active,
- The Trigger precondition, c AND NOT a , is True

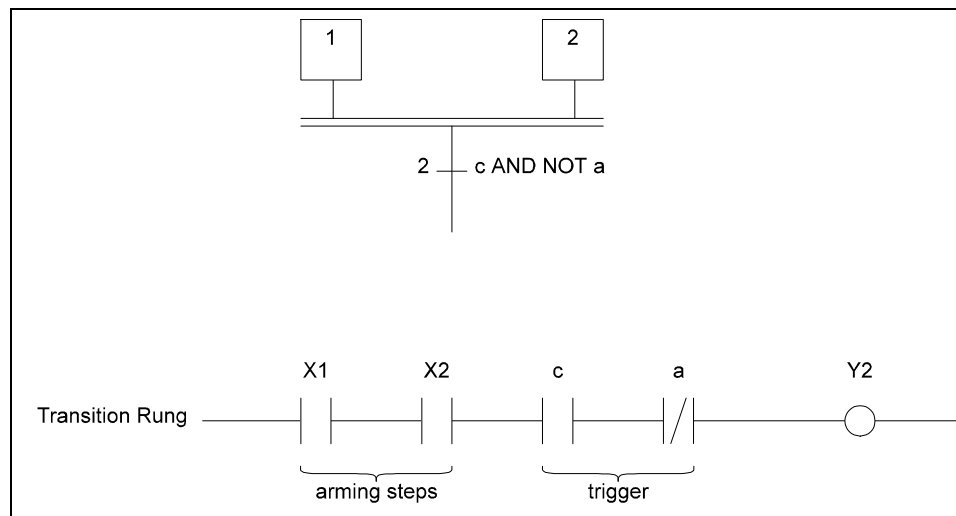


Figure 9.7

Step rungs

Each Step m is represented by a step bit, X_m , such that X_m is True when Step m is active. Recall that a step is activated if any of its entrance transitions fire. It is deactivated if any of its exit transitions fire. If both an exit transition and an entrance transition fire at the same time, the step remains active. On initialization, all initial steps are active and all other steps are inactive.

For each normal (non-initial) step, the above behavior is achieved by a rung as shown in Figure 9.8. Initialization bit, NIB, is False on the first scan, so it looks after initial deactivation. NIB will be set at the end of the first scan thus allowing the steps to be activated. Once NIB is set, then X_1 can be set by any of the entrance transition bits, Y_1 or Y_2 . The latch bit, which provides the feed back loop from X_1 , memorizes the active state. Any of the exit transition bits, Y_{11} or Y_{12} , can deactivate the step by clearing X_1 .

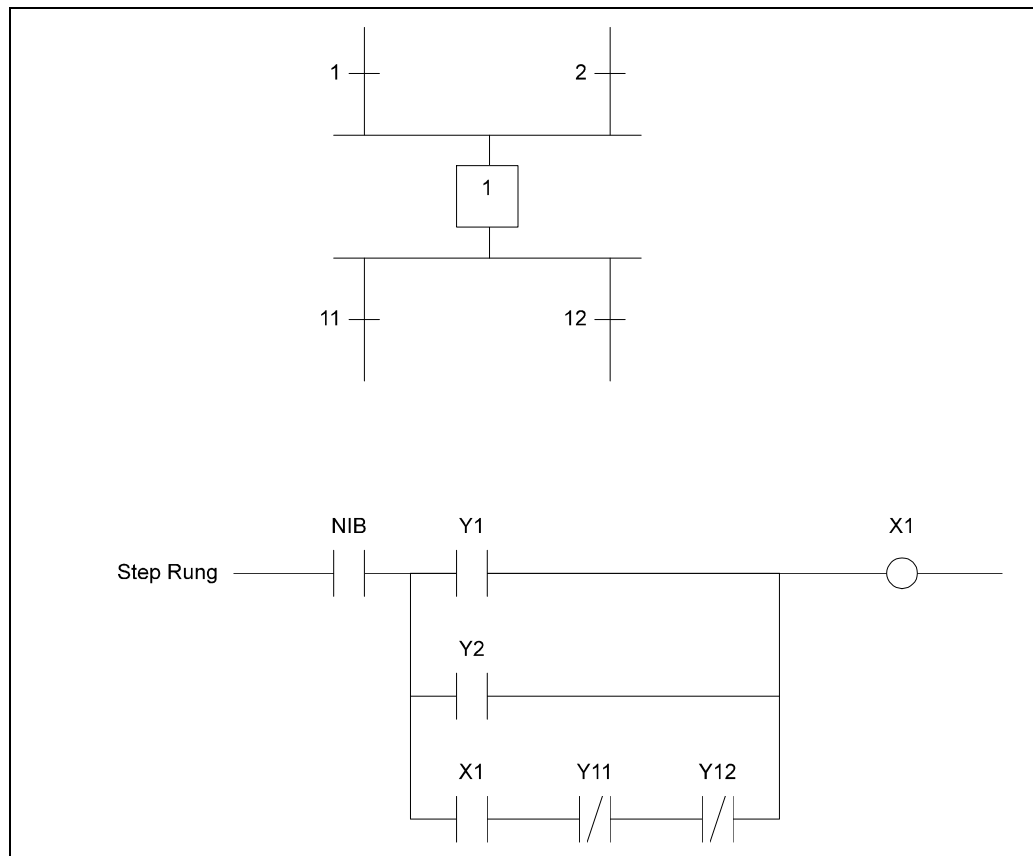


Figure 9.8

Initial step rungs

An initial step requires a slightly different rung, as shown in Figure 9.9. At power-up, NIB is False and will be set at the end of the first scan. Therefore, on the first scan, X0 will be energized by the normally closed contact. This activates the initial step on the first scan. After that, the initial step rung works in basically the same way as normal step rungs.

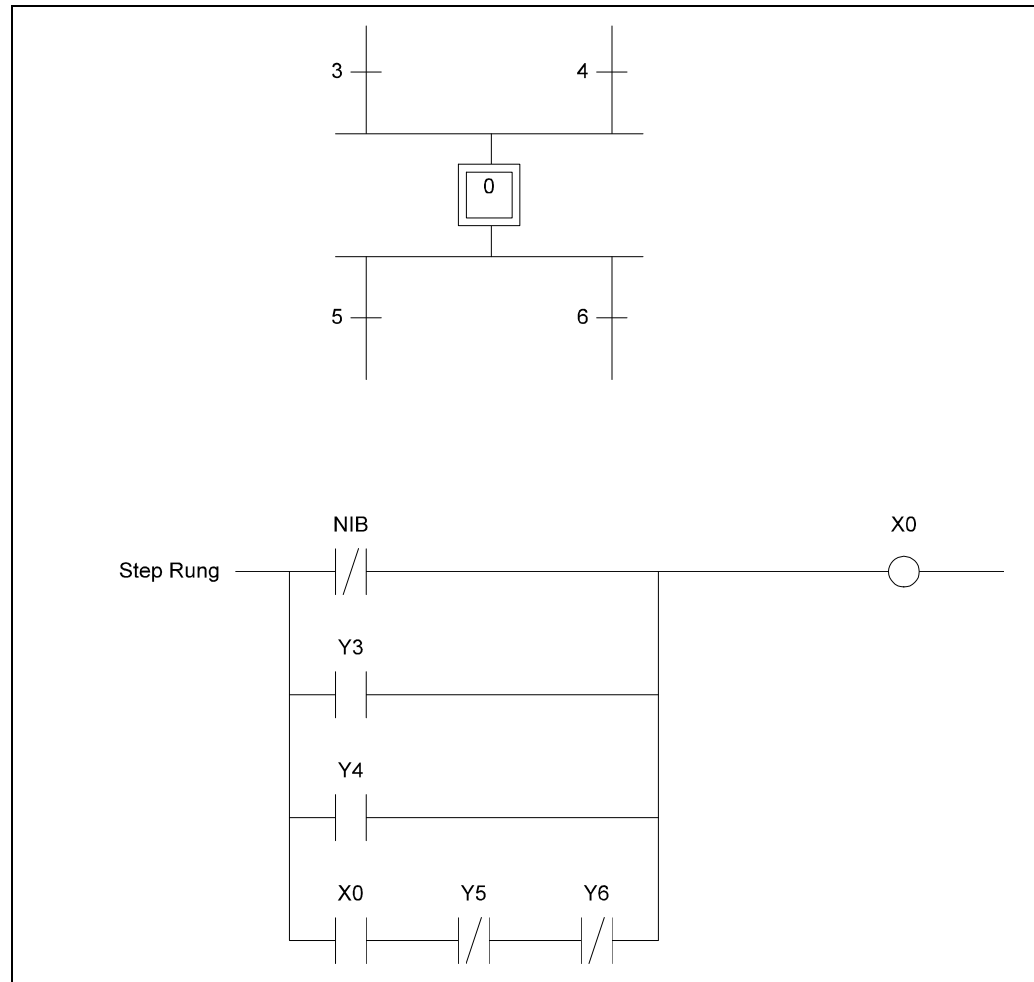


Figure 9.9

Forcing

Two more internal bits are added to the graph for each forcing instruction. One is used to activate the steps that are contained in the forced situation, the other is used to deactivate steps not in the forced situation. An example of this is shown in Figure 9.10.

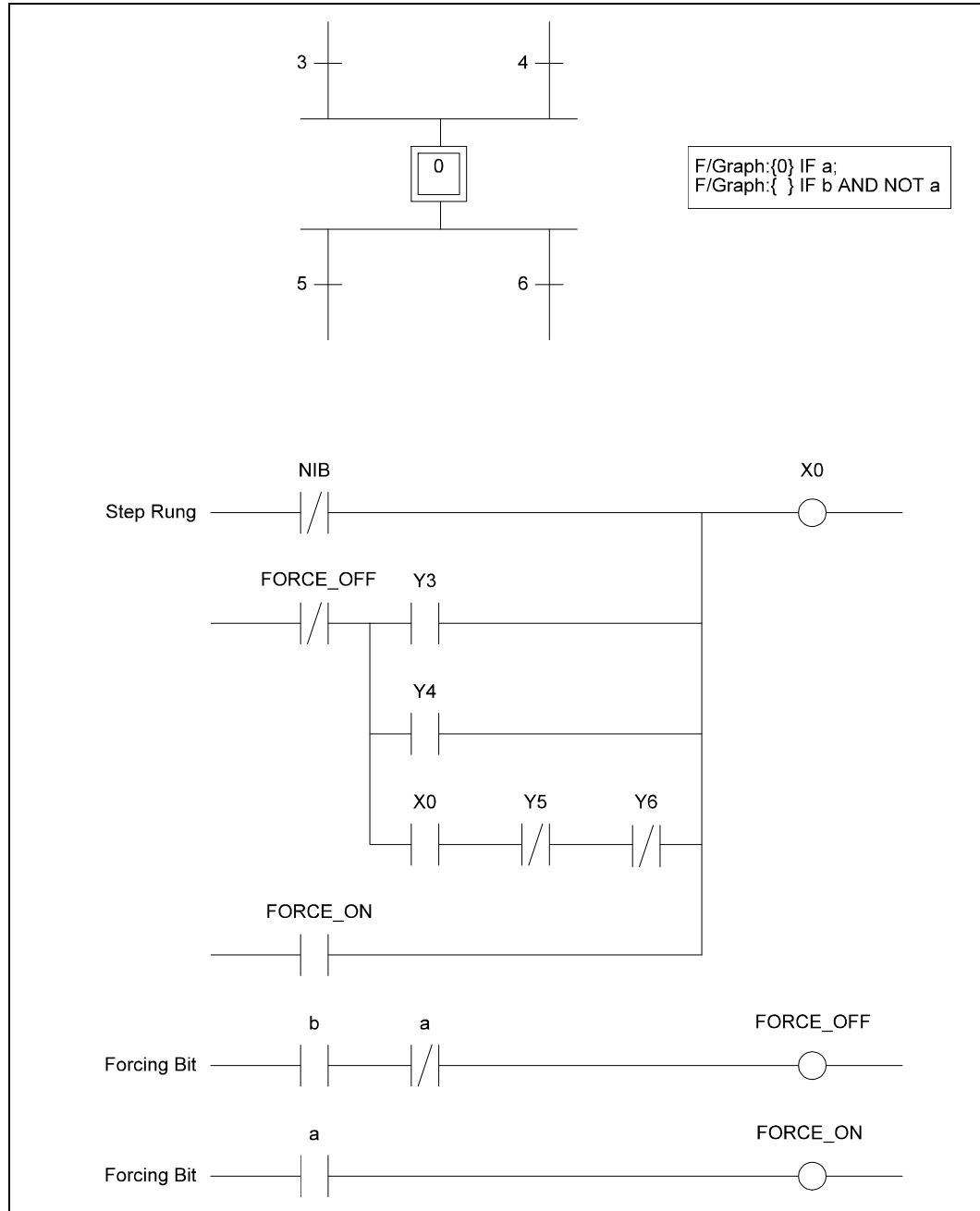


Figure 9.10

Action rungs

If a variable is set by momentary actions in more than one step, the variable must be True whenever any of the actions are executed. Use a rung as shown in Figure 9.11 for each action.

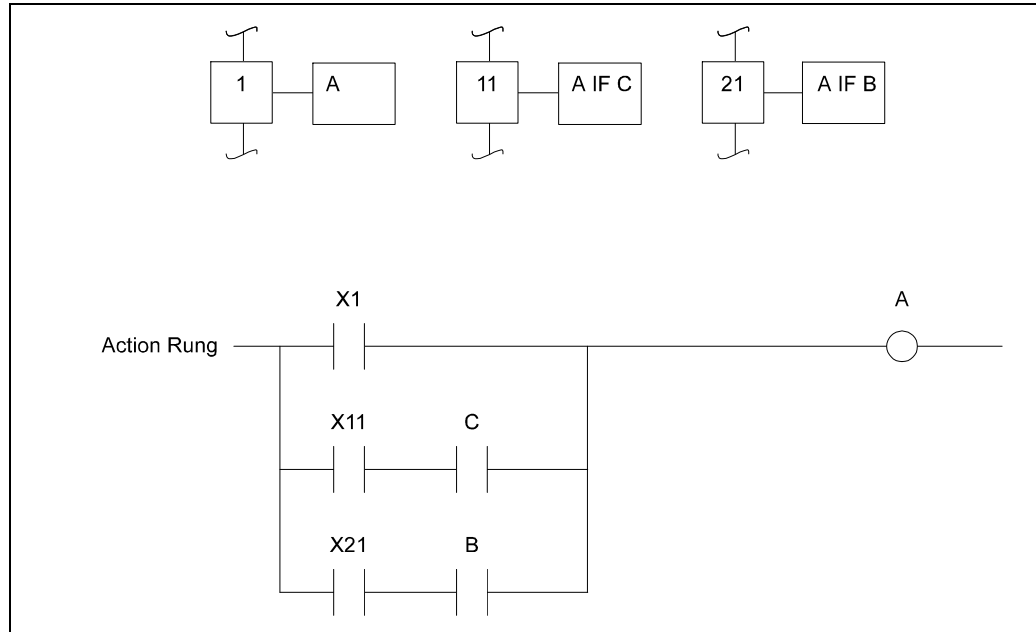


Figure 9.11

Timer rungs

Each Grafcet timer expression of the form

T/ enable / preset

requires the generation of one rung. The timer rung is constructed using the timer block provided by all PLCs. The actual usage of timer blocks may vary from one PLC to the next, but it will always be similar to the generic timer shown in Figure 9.12.

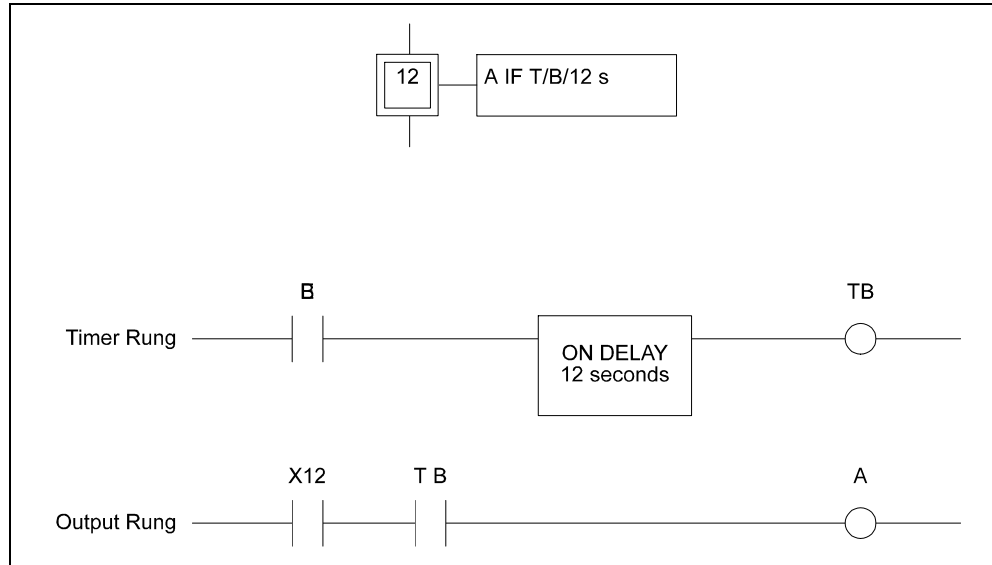


Figure 9.12

Note that the timer returns a True value whenever 12 seconds have elapsed since the rising edge of its enable signal. Thus, if Step 12 is activated 13 seconds after B becomes True, then A will become True instantly on Step 12's activation.

Edge-detection rungs

A rising edge occurs whenever a variable passes from False to True. Because the minimum measurable time interval under the scan cycle model is one scan cycle, a rising edge is detected if a variable changes state from one scan to the next. To detect a rising edge, we need to have previously memorized the state of the variable during the last scan, and we compare it with the state during the present scan. In Figure 9.13, we see that the comparison is made in the action rung which is followed by the memorization rung. The memorization rung must follow the action rung for this to work correctly.

The rising edge of an input cannot be detected on the first scan because the previous state of the input is unknown. Therefore, the initialization bit, NIB, is included in the action rung to block edge detection during the initial scan. (Remember that NIB is not True during the first scan, but is True for all subsequent scans.)

Falling edge detection works the same way, except that we look for a variable which passes from True to False.

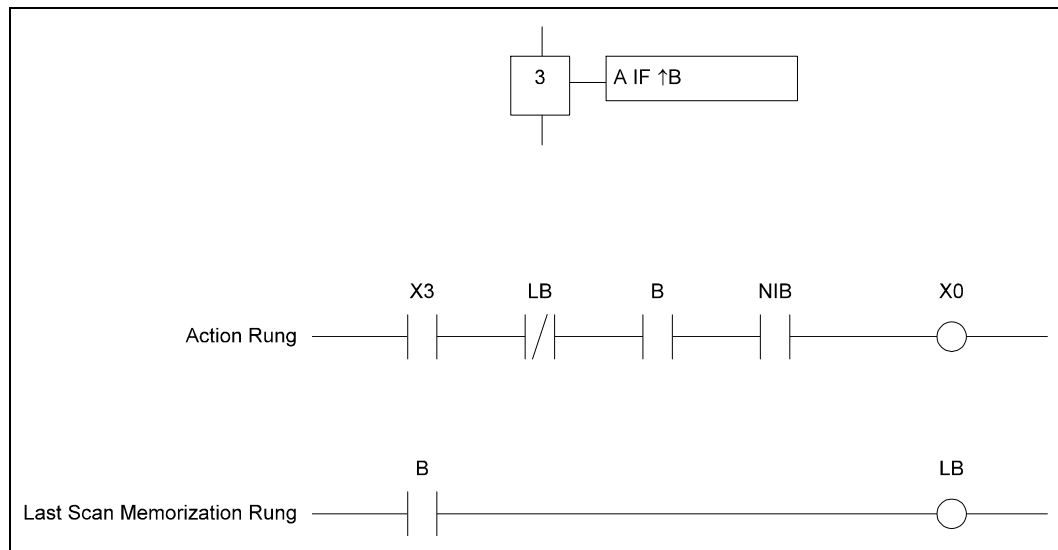


Figure 9.13

10 Glossary

Actions

momentary action - analogous to a momentary push button, an action which sets a bit only as long as the action is executed. When the action is no longer executed the bit is cleared. When more than one momentary action acts on the same bit, the bit remains True as long as any of the actions are executed. Note that momentary actions are incompatible with stored actions.

stored action - an action which assigns a new value to a logical or numerical variable. The variable keeps the new value after the action has been executed.

conditional action - an action which is subject to a Boolean expression, if the expression is True the action is executed, otherwise the action is not executed. Note that a conditional action using a timer can be used to implement delayed or time limited actions.

delayed or time limited action - an action whose behavior changes with time.

Application - The group of graphs running simultaneously on a single PLC.

Arm - A transition is armed when all its input steps are active. A transition with no input steps, called a source transition is always armed (by convention).

Bit - a logical variable that can take the values True or False.

Clear ...

a bit - Assign the value False to a logical variable.

a register - To zero a register.

a transition - This is the term used by the IEC's English translation to mean fire a transition. The preferred word, already used for years in Petri net theory, is *fire*.

Condition - a Boolean expression, evaluates to either True or False.

Connected sub-graph - a part of a Grafcet diagram such that all the elements are connected by links.

Controller - an information processor, which executes a control algorithm whose input, consists of data signals and whose output constitutes control signals. In this book a controller is usually a PLC or an industrial computer.

Directed link - see Link.

Enable - to start a timer.

Entrance - the top of a step or a transition, the control token enters through the entrance.

Exit - the bottom of a step or transition, the control token leaves through the exit.

False - the logical condition False (opposite of True)

Fire - a transition fires when it is enabled and its precondition is True. When it fires it deactivates all (if any) of its entrance steps and activates all (if any) of its exit steps.

Force

a graph - cause a graph to assume a given situation, ex.: the instruction F/Graf2:{3,7} activates steps 3 and 7 of the graph Graf2 and deactivates all other steps in Graf2.

a step - cause a step to change state, from active to inactive or vice-versa. (You can do this in the on-line mode of CASE tools.)

a transition - cause an enabled transition to fire even if its precondition is False. (You can do this in the on-line mode of CASE tools.)

a variable - cause a bit to change value. (You can do this in the on-line mode of CASE tools.)

Functional instability - when the trigger condition of a step's exit transition is True at the moment the step is activated. A functional instability depends on the state of the inputs, a race occurs only when certain input conditions are met. (see also Structural instability)

Graph - a Grafcet diagram.

Initial step - A step drawn as a double box to indicate that it is to be activated on initialization of the graph.

Integer - A 16 bit numeric variable (between -32767 and 32768).

Interrupt driven - an operating system system where various tasks are called according to interrupts coming from external sources.

LADDER - see Relay Ladder Logic

Link - A line that joins the exit of a step to the entrance of a transition, or that joins the exit of a transition to the entrance of a step. Also known as *directed links* because they can only join an exit to an entrance, they never go in the other direction.

Macro expansion - The sequence of steps and transitions represented by a macro-step. A macro expansion must have exactly one entrance and one exit.

Macro step - A single step (drawn as a triple box), that stands for a sequence of steps and transitions.

Plant - The physical system to be automated, also called the process or the physical process.

PLC - Programmable logic controller, a computer designed specifically for plant control

Process - The physical system to be automated, also called the plant.

Relay Ladder Logic - a Boolean logic notation using electrical symbols such as relay contacts and coils

Race - The common name for an unstable situation.

Reactive system - a real-time system.

Real-time system - a control system which responds to external events within a bounded time.

Receptivity - The preferred term is *trigger condition*.

Reset - The preferred term is clear (a bit).

Scan cycle - the time required for a programmable controller to scan its inputs and generate appropriate control responses at its outputs.

Set - Assign the value True to a bit.

Situation - The set of active steps in a graph.

Stable situation - a situation of a graph such that the graph cannot evolve until an input changes or a timer expires.

Step - Basic Grafcet element which executes zero or more non-sequential control actions.

Structural instability - when the trigger condition of a step's exit transition is always True. (see also Functional instability)

Sub-graph - Part of a Grafcet diagram, may or may not be connected by links.

Tag name - see Variable name.

Timer - Boolean expression whose value depends on an **enable** and a **preset** period. A timer is written as:

T/enable/preset

It becomes True when preset seconds have elapsed since the latest rising edge of enable. The timer is False on initialization and becomes False on the rising edge of enable.

Transition - Basic Grafcet element which sequences the activity of the steps within a graph.

Translate - Compile a Grafcet application from graphical form into executable code for a target PLC.

Trigger condition - Boolean expression which determines if an enabled transition should fire. The armed transition will fire if its trigger condition is True.

True - The logical condition True (opposite of False).

Unstable situation - a race.

Variable - Name (or tag name) which represents a region in the memory map of a PLC.

Word - A 16 bit numeric variable that can take a value between 0 and 65536.

Index

- action
 - Boolean stored, 82
 - conditional, 75
 - conditional momentary, 80
 - delayed, 80
 - incompatibility of stored and momentary actions, 77
 - momentary, 74, 76
 - repetitive numeric stored, 82
 - single scan cycle time limited, 81
 - single scan numeric stored, 83
 - stand-alone, 74, 89
 - step, 74
 - stored, 75
 - time limited, 80
 - undefined, 83
- action, 21
- action rungs, 132, 137
- actions, 56, 74, 140
- actuators, 5
- AFCET, 2
- always-true triggers, 63
- Analog inputs, 49
- AND, 6
- AND convergence, 42
- AND convergences, 67
- AND divergence*, 42
- AND divergences, 67
- AND node, 105
- application**, 140
- Applications, 48
- arm, 140
- assembler, 3
- automatic substation synchronization, 117
- automation productivity, 2
- automation specification, 4
- batch process automation, 3
- bit**, 140
- bits, 49
- Boolean equations, 117
- Boolean expressions, 6
- Boolean inputs, 49
- Boolean logic, 2, 6
- Boolean notation (advantages of), 9
- Boolean operators, 60
- Boolean signals, 6
- breaker re-closing, 117
- C.A. Petri, 12
- C.E. Shannon, 10
- capacitor bank switching, 117
- case study, 117
- clear**, 140
- code re-usability, 27
- combinational control, 6
- combinational function, 6
- comments, 88
- comparison, 61
- condition**, 140
- conditional action, 140
- conditional momentary actions, 80
- conditional statements, 76
- connected graph, 47
- connected sub-graph, 140
- control theory, 5
- control token*, 21
- Control token, 54
- controller**, 141
- data initialization, 91

data initialization and continuous calculation, 91
 degenerate loop, 95
 delayed action, 80
 derivative control, 6
directed links, 20
 DIS 1131-3, 2
 documentation, 125
 E.F. Moore, 10
 edge detection, 61
 edge-detection rungs, 139
 elapsed time, 62
 elements of Grafcet, 16
 Elements of Grafcet, 47
 emergency shut-down, 111
enable, 141, 143
entrance, 141
 equal, 61
 event counting, 111
 evolution, 68
exit, 141
 explicit links, 85
 falling edge, 139
 falling edge operator, 7
false, 141
 Famic Automation Inc, 125
fire, 141
 firing time, 68
 flow chart, 10
 flowcharts, 2
 force, 141
 forcing, 136
 forcing statement, 37
 functional instability, 141
 G.H. Mealy, 10
 George Boole, 6
 glossary, 140
 grafcet structures, 89
 grafcet tutorial, 15
 graph, 47, 141
 evolution, 68
 forcing, 84
 level 1, 88
 level 2, 88
 situation, 56
 structure, 51
 greater than, 61
 greater than or equal to, 61
 hierarchical tasking, 114
 Hydro-Québec, 117
 Hydro-Quebec's experience with Grafcet, 124
 I/O image, 49
 I/O list, 29
 IEC 848, 2
 image table, 49
 implementing Grafcet on the PLC, 126
 IN step, 52
 inclusive OR, 99
 incompatibility of stored and momentary actions, 77
 initial situation, 25
 initial state, 11
 initial step, 54, 141
 initial step rungs, 135
 initialization, 32
 initialization rung, 132
 inputs
 $\pm 10\text{V}$, 49
 4-20mA, 49
 numeric, 49
 On/Off, 49
integer, 141
 integral control, 6
 internal registers, 49
 International Electrotechnical Commission, 2

- interpreted parallelism, 99
- interrupt driven, 141
- labeled links, 86
- LADDER, 142
- less than, 61
- less than or equal to, 61
- level 1 graph, 88
- level 2 graph*, 88
- link**, 142
- links, 85
 - explicit, 85
 - labeled, 86
- list, 3
- load shedding, 117
- local variables, 50
- loop, 93
- loop with initialization, 94
- looping sequence with initialization, 32
- low level machine languages, 3
- macro, 142
- manual translation into PLC language, 130
- model of computation, 47
- modular graphs, 29
- modular programming, 50
- momentary action*, 21, 76, 140
- multiple initial steps, 96
- mutually exclusive trigger conditions, 98
- naming conventions, 51
- NFC 03-190, 50, 63, 86
- NFC-03-190, 2
- non-connected graph, 47
- non-looping sequence with a permanent final step, 92
- NOT, 6
- not equal, 61
- not invented here, 118
- Numeric signals, 6
- On/Off control, 19
- OR, 6
- OR convergence, 32, 57
- OR divergence, 37, 57
- OR node, 103
- organization and modularity of Grafcet applications, 47
- OUT step, 52
- pant**, 142
- parallel sequences, 42, 104
- parallelism
 - interpreted, 99
 - structural, 99
- parallelism versus multi-statement actions, 102
- Pascal, 2
- permanent monitoring, 110
- permanent step, 90
- Petri nets, 12, 117
- physical plant, 5
- physical process, 142
- PID, 3, 6
- places (Petri net)*, 12
- PLC, 142
- pneumatic piston, 44
- polling, 49
- precedence, 63
- preset, 143
- priority of operations, 63
- process, 5
- process, 142
- program structuring, 114
- Prolog, 2
- proportional control, 6
- pseudo-code, 128
- queuing systems, 12
- race, 142
- radial saw example, 20
- reactive program, 47
- reactive system, 142

- readability, 49
- read-only, 59
- real-time system, 126, 142
- receptivity**, 142
- reduced graph, 34
- redundant sequence selection, 100
- relay ladder logic, 9
- Relay Ladder Logic, 142
- repeating sequence, 93
- reserved words*, 50
- reset**, 142
- response time, 128
- re-usable graphs, 29
- rising edge, 139
- rising edge operator, 7
- RTU, 117
- safety interlock, 108
- safety monitoring, 90
- saving time and money, 125
- scan, 142
- scan cycle, 49, 127
- scope, 29, 50
- sensors, 5
- sequence
 - jumping, 97
 - selection, 98
 - synchronization, 105
- sequences
 - parallel, 104
- sequential control, 31
- sequential control with fault handling, 36
- sequential function, 6
- Sequential Function Charts*, 2
- service restoration, 117
- set**, 142
- SFC, 2
- situation, 56, 142
 - empty*, 56
- spring-return pneumatic piston, 44
- stand-alone action, 37, 38, 74, 89
- state machine, 129
- state machine theory, 10
- state transition diagram, 10
- state-transition methods, 117
- step
 - bit, 59
 - entrance, 51
 - exit*, 51
 - IN, 52
 - initial, 54
 - macro, 51
 - number, 51
 - OUT, 52
 - permanent, 90
- step rungs, 132, 134
- steps
 - drawing and identifying, 51
- stored action, 140
- stored statements, 76
- structural instability, 143
- structural parallelism, 99
- structural versus interpreted
- synchronization, 106
- structured Grafset programming, 114
- sub-graph, 47, 143
- substation automation, 117
- synchronizing parallel sequences, 40
- syntax diagrams, 64, 77
- tag name*, 49, 143
- tag names, 17
- tasks, 114
- textual language, 56
- time limited action, 80
- timer, 34
- timer, 143
- timer rungs, 138

- timers, 62
- timing charts, 9
- timing diagram, 69
- token, 54
- transition, 143
 - entrance, 59
 - exit, 59
 - number, 60
 - trigger, 60
- transition rungs, 132, 133
- transitions, 59
 - drawing and identifying, 59
- translate**, 143
- transmission line switcher, 118
- transmission line switching, 117
- trigger
 - mutually exclusive conditions, 98
- trigger condition*, 21, 143
- trigger conditions, 60
- true**, 143
- true-in-step, 76
- types of control signal and control function, 6
- unconditional momentary actions, 78
- unconditional statements, 75
- undefined stored action, 83
- unstable situation, 143
- valid names, 49
- variable**, 143
- variable name*, 49
- variable names, 17
- word**, 143

Appendix A – Bibliography

In English:

1. ADEPA, *GRAFCET: A function chart for sequential processes*, Agence nationale pour le Développement de la Production Automatisée, 17 rue Perier, B.P. 54, 92123 Montrouge, France, 1979.
2. Baracos, P.C., *Binary Decision Machines: Theory, Design and Applications*, Ph.D. Thesis, McGill University, Montreal, Canada, 1987, pp. 3.17-3.20, 5.1-6.18.
3. Chocron, D. and Cerny, E., *A Petri-net based industrial sequencer*, IECI '80 Spring Conference and Exhibit on Industrial Control and Instrumentation Applications of Mini and Microcomputers, Philadelphia, March 17-19, 1980.
4. David, René and Alla, Hassane, *Petri Nets & Grafcet: Tools for modelling discrete event systems*, ISBN 0-13-327537-X, Prentice Hall, London, 1992.
5. IEC, *Standard 848, Preparation of function charts for control systems*, International Electrotechnic Commission, 3 rue de Varembé, Geneva, Switzerland, 1988.
6. IEC, *Draft Standard DIS 1131-3, Programming Languages for Programmable Controllers*, International Electrotechnic Commission, 3 rue de Varembé, Geneva, Switzerland, 1989.
7. Lloyd, Mike, *Graphical Function Chart Programing for Programmable Controllers*, in Control Engineering, Vol.32, No. 10, October 1985, pp. 73-76.
8. Peterson, J.L., *Petri Net Theory and Modelling of Systems*, Prentice-Hall, 1981.

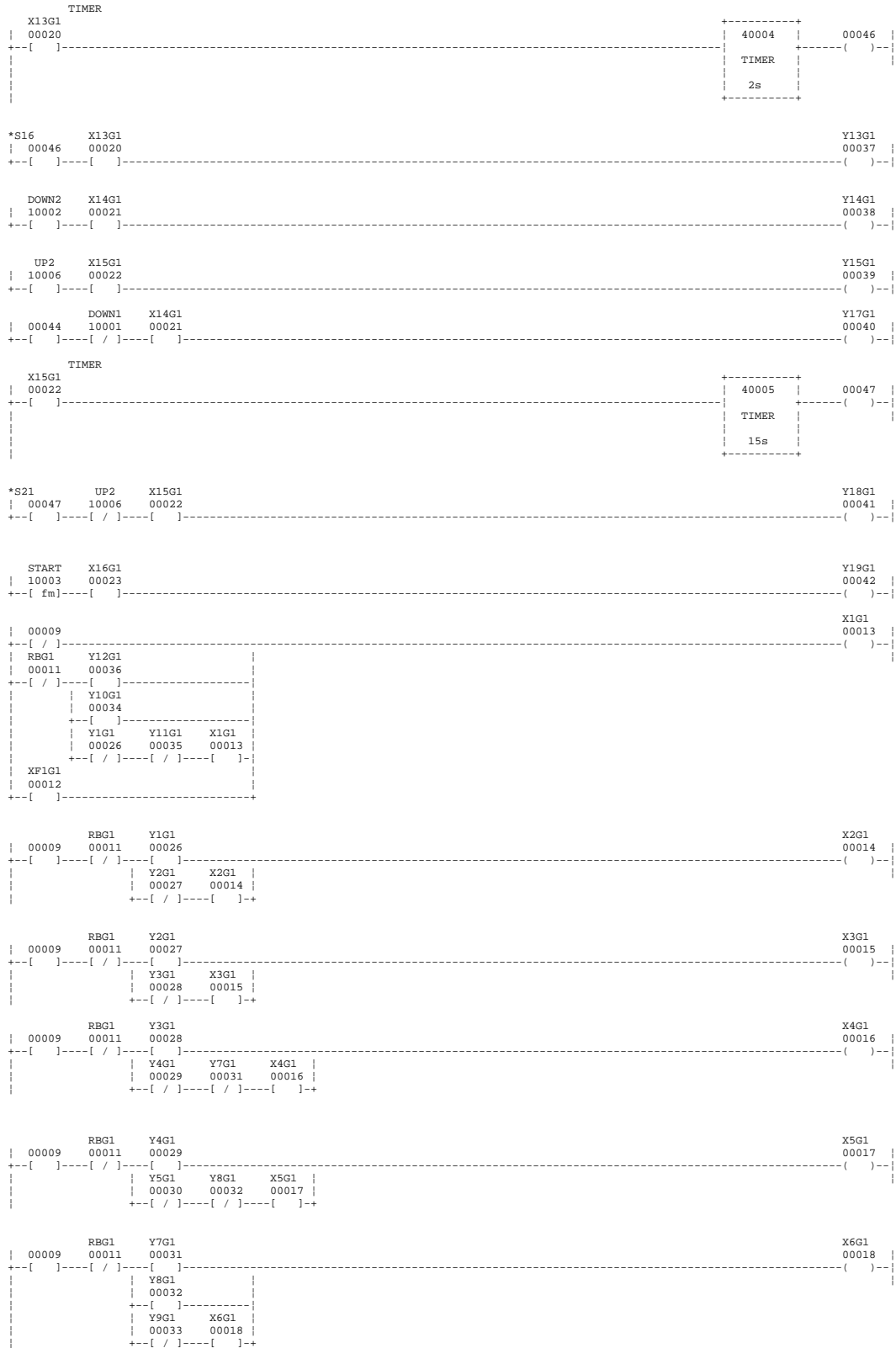
In French:

1. ADEPA (Agence nationale pour le Développement de la Production Automatisée), LE GRAFCET, Cépaduès-Éditions, 111 rue Nicolas-Vauquelin, 31100 Toulouse, France, 1992.
2. ADEPA, *GRAFCET: Diagramme fonctionnel des automatismes séquentiels*, Agence nationale pour le Développement de la Production Automatisée, 17 rue Perier, B.P. 54, 92123 Montrouge, France, 1979.
3. Blanchard M., *Comprendre, Maîtriser et appliquer le GRAFCET*, Cépaduès-Éditions, 111 rue Nicolas-Vauquelin, 31100 Toulouse, France, 1979.
4. Bossy, J.C., *Le GRAFCET: Sa pratique et ses Applications*, 2e Ed., UDULIVRE, Paris, 1984.
5. CEI, *Norme 848: Établissement des diagrammes fonctionnels pour systèmes de commande*, Commission Électrotechnique Internationale, 3 rue de Varembe, Genève, Suisse, 1988.
6. Chevalier G., *Le GRAFCET: Les Automatismes par le Diagramme Fonctionnel et la Technologie Modulaire*, DUNOD, Paris, 1980.
7. Chocron, D., *Un système de Programmation par Réseau de Petri de Contrôleurs Industriels*, Thèse de M.Sc. , Université de Montréal, 1980.
8. David, René et Alla Hassane, *Du Grafcet aux réseaux de Petri*, Éditions Hermes, Paris, 1991.
9. GREPA (GRoupe Equipement de Production Automatisée), *LE GRAFCET de nouveaux concepts*, Cépaduès-Éditions, 111 rue Nicolas-Vauquelin, 31100 Toulouse, France, 1985.
10. Thelliez S. et Toulotte J.M., *Applications Industrielles du GRAFCET*, Éditions Eyrolles, Paris, 1983.
11. Thelliez S. et Toulotte J.M., *GRAFCET et Logique Industrielle Programmée*, Editions Eyrolles, Paris, 1980.
12. Thomas R., *Programmation du GRAFCET sur les Automates Programmables*, Centre Technique des Industries Mécaniques, Étude CETIM: No. 10080, 1986.

Appendix B– RLL Listing

This appendix lists the Relay ladder language program described in Example 6 in Chapter 4. The program is for an Allen-Bradley PLC5-12 programmable controller. It was generated using Taylor Industrial Software Inc's ladder programming software.

GRAPH F4-21									
START	UP1	UP2	X1G1	Y1G1					
10003	10005	10006	00013	00026					
+--[fm]---[]---[]---[]-----				()--					
X2G1				Y2G1					
00014				00027					
+--[]-----				()--					
TIMER									
X3G1									
00015				40001				00043	
+--[]-----				()--					
TIMER									
2s									
*S4	X3G1			Y3G1					
00043	00015			00028					
+--[]---[]-----				()--					
DOWN1	X4G1			Y4G1					
10001	00016			00029					
+--[]---[]-----				()--					
UP1	X5G1			Y5G1					
10005	00017			00030					
+--[]---[]-----				()--					
TIMER									
X4G1									
00016				40002				00044	
+--[]-----				()--					
TIMER									
15s									
*S8	DOWN1	X4G1		Y7G1					
00044	10001	00016		00031					
+--[]---[/]---[]-----				()--					
TIMER									
X5G1									
00017				40003				00045	
+--[]-----				()--					
TIMER									
15s									
*S10	UP1	X5G1		Y8G1					
00045	10005	00017		00032					
+--[]---[/]---[]-----				()--					
START	X6G1			Y9G1					
10003	00018			00033					
+--[fm]---[]-----				()--					
X7G1	X17G1			Y10G1					
00019	00024			00034					
+--[]---[]-----				()--					
START	UP1	X1G1		Y11G1					
10003	10005	00013		00035					
+--[fm]---[/]---[]-----				()--					
UP2									
10006									
+--[/]--									
START	X20G1			Y12G1					
10003	00025			00036					
+--[fm]---[]-----				()--					



```

|      RBG1      Y5G1      X7G1
| 00009 00011 00030      00019
|--[ ]---[ / ]---[ ]---( )--|
|      |      |      |
|      | Y9G1  |      |
|      | 00033 |      |
|      |---[ ]---|
|      | Y10G1  X7G1 |
|      | 00034 00019 |
|      |---[ / ]---[ ]-+
|
|      RBG1      Y2G1      X13G1
| 00009 00011 00027      00020
|--[ ]---[ / ]---[ ]---( )--|
|      |      |      |
|      | Y13G1  X13G1 |
|      | 00037 00020 |
|      |---[ / ]---[ ]-+
|
|      RBG1      Y13G1      X14G1
| 00009 00011 00037      00021
|--[ ]---[ / ]---[ ]---( )--|
|      |      |      |
|      | Y14G1  Y17G1  X14G1 |
|      | 00038 00040 00021 |
|      |---[ / ]---[ / ]---[ ]-+
|
|      RBG1      Y14G1      X15G1
| 00009 00011 00038      00022
|--[ ]---[ / ]---[ ]---( )--|
|      |      |      |
|      | Y15G1  Y18G1  X15G1 |
|      | 00039 00041 00022 |
|      |---[ / ]---[ / ]---[ ]-+
|
|      RBG1      Y17G1      X16G1
| 00009 00011 00040      00023
|--[ ]---[ / ]---[ ]---( )--|
|      |      |      |
|      | Y18G1  |      |
|      |---[ ]---|
|      | Y19G1  X16G1 |
|      | 00042 00023 |
|      |---[ / ]---[ ]-+
|
|      RBG1      Y15G1      X17G1
| 00009 00011 00039      00024
|--[ ]---[ / ]---[ ]---( )--|
|      |      |      |
|      | Y19G1  |      |
|      | 00042  |      |
|      |---[ ]---|
|      | Y10G1  X17G1 |
|      | 00034 00024 |
|      |---[ / ]---[ ]-+
|
|      RBG1      Y11G1      X20G1
| 00009 00011 00035      00025
|--[ ]---[ / ]---[ ]---( )--|
|      |      |      |
|      | Y12G1  X20G1 |
|      | 00036 00025 |
|      |---[ / ]---[ ]-+
|
|      STOP      XF1G1
| 10004      00012
|--[ ]---( )--|
|
|      STOP      RBG1
| 10004      00011
|--[ ]---( )--|
|
|      UP1      X1G1      RAISE1
| 10005 00013 00004
|--[ / ]---[ ]---( )--|
|
|      UP2      X1G1      RAISE2
| 10006 00013 00005
|--[ / ]---[ ]---( )--|
|      |      |
|      | X15G1 |
|      | 00022 |
|      |---[ ]---+
|
|      CLAMP      00051
| 00009 00002      00051
|--[ ]---[ / ]---( )--|
|      |      |
|      | X1G1  |
|      | 00013 |
|      |---[ ]---+
|
|      00009 00014      00002
|--[ ]---[ ]---( )--|
|      |      |
|      | CLAMP |
|      | 00002 00051 |
|      |---[ ]---[ / ]-+

```

