Project Acronym: MEDIS

Project Title: A Methodology for the Formation of Highly Qualified Engineers at Masters Level in the Design and Development of Advanced Industrial Informatics Systems

Contract Number: 544490-TEMPUS-1-2013-1-ES-TEMPUS-JPCR

Starting date: 01/12/2013 **Ending date:** 30/11/2016

Deliverable Number: 2.4.3

Title of the Deliverable: Development of the AIISM teaching resources - Industrial Networks and Fieldbuses – Introduction - Discrete Event Control

Task/WP related to the Deliverable: Development of the AIISM teaching resources

Type (Internal or Restricted or Public): Internal

Author(s): Armando Sousa

Partner(s) Contributing: FEUP

Contractual Date of Delivery to the CEC: 30/09/2014

Actual Date of Delivery to the CEC: 30/09/2014

<u>Project Co-ordinator</u>

Company name :	Universitat Politècnica de València (UPV)
Name of representative :	Houcine Hassan
Address :	Camino de Vera, s/n. 46022 - Valencia (Spain)
Phone number :	+34 96 387 7578
Fax number :	+34 96 387 7579
E-mail :	husein@disca.upv.es
Project WEB site address :	http://medis.upv.es/

Context

WP 2	Design of the AIISM-PBL methodology
WPLeader	Universitat Politècnica deValència (UPV)
Task 2.4	Development of the AIISM teaching resources – Industrial Networks and Fieldbuses - Introduction
Task Leader	UP
Dependencies	UPV, MDU, TUSofia, USTUTT, UP

Author(s)	Armando Sousa
Contributor(s)	Mário de Sousa
Reviewers	

History

Version	**Date**	**Author**	**Comments**
0.1	01/04/2014	Armando Sousa	Creation, first chapter
0.2	12/04/2014	Armando Sousa	Lecture chapter
0.3	14/14/2014	Armando Sousa	Add Lab
0.4	15/05/2014	Armando Sousa	Add Seminars and Mini-Project
0.5	19/09/2014	Mário de Sousa	Add Seminars and Mini-Project

Table of Contents

1. Executive summary

WP 2.4 details the learning materials of the Advanced Industrial Informatics Specialization Modules (AIISM) related to the Industrial Networks and Fieldbuses.

The contents of this package follows the guidelines presented in the Partner's documentation of the WP 1 (Industrial Networks and Fieldbuses)

- The PBL methodology was presented in WP 1.1
- The list of the module's chapters and the temporal scheduling in WP 1.2
- The required human and material resources in WP 1.3
- The evaluation in WP 1.4

During the development of this WP a separate document has been created for each of the chapters of the Industrial Networks and Fieldbuses Module (list of chapters in WP1.1). This document is for the third chapter – Discrete Event Systems.

In this document, section 2 defines the lecture, sections 3 and 4 describe the laboratory work for weeks 3 and 4, sections 5 and 6 explain the seminar topics for weeks 3 and 4, and sections 7 and 8 define the requirements to fulfill for the mini-project during weeks 3 and 4. Section 9 lists the bibliography and the references.

2. Introduction

This chapter will deal with computer control of real world systems.

Frequently, such systems can be modeled as discrete systems without loss of important information. Control of this class of systems is introduced in this chapter.

An important tool called Grafcet, in accordance to IEC 60848 is to be introduced and used throughout his chapter.

1.10 Problem Statement

This chapter aims to enable the student to learn about discrete event systems and control of this class of problems by using healthy approaches such as the IEC 60848 Grafcet graphical programming language, adequate for parallel streams of control.

At the end of this lecture, the student should:

- Recognise and transform a suitable system adequate into a discrete event system
- Recognise the control architecture of an embedded system
- Use IEC 60848 Grafect as a programming language to design a control strategy
- Implement Grafcet in general purpose computer controller

3. Lecture

This Lecture will deal with the specifics of control of systems modelled as discrete event systems. A very fast approach to the Grafcet graphical programming language is made, with its formal definitions. Latter in this chapter, its implementation in general purpose platforms is addressed.

1.11 Context

Computer control of real world systems is very often and such systems are called “embedded systems”. They are hidden inside a larger setup may range from a shoe that flicks a LED light at each step, to a full blown Unmanned Airplane control system. Other examples include washing machines and semaphores (traffic lights). Other level of complexity arise from communications to outside world but this latter issue will be dealt with latter.

1.12 Embedded Systems

In the mentioned examples the limits of the word are well defined and the controller uses inputs to read sensors and uses outputs to drive actuators (as depicted in Figure 1). The so called control loop is said to read the state of the System to be controlled, compute needed changes and enforce changes by commanding actuators – if the overall system is well designed the system to be controlled will be led to the overall state wanted the designer of the system, implemented in the software control logic of the controller. Thus, an embedded controller also includes a software part.

Figure 1 – Embedded control of a system by using a computer controller

1.13 Discrete Event Systems

Flying an airplane seems adequate for a model using equations in continuous time but the same type of model seems awkward for the traffic lights.
Even if human time is inherently continuous in nature, a very useful model will come handy for control of certain systems that can be converted to discrete event without significant loss of information.

1.14 Conversion to discrete event

Throughout this chapter, let us consider an example of an oven temperature control, with a temperature sensor and a resistor heater, controlled by a computer system.

Firstly, let as consider that we need a high degree of quality and that we have high quality sensors and actuators. It would be conceivable to make a control system that would use “analog” readings and produce an “analog” output so that the heater resistance would produce just the perfect amount of heat to take the oven to the chosen temperature. This arrangement would make continuous time system. Naturally, the analog sensor and the analog actuator would have to be converted into digital signals and the control strategy to produce just the perfect amount of heat would not be trivial and would always depend on what was inside the oven.

Another version of the same system would use a single on-off sensor and a single on-off heater resistor. Let as assume that the sensor turns on at the adequate chosen temperature. The control strategy would be trivial: turn on the heater when the sensor is off.

Is the complexity of the continuous time system necessary?

Can the system be transformed into a discrete event system without loss of functionality?

Both approaches are valid and the approach taken depends upon the requirements of the project.

A different but similar question is if a given system can be considered to be of discrete event nature. Computer controlled systems are inherently discrete time systems and so taking time samples of the input is always necessary and when control software is done, the outputs are also changed to request action from the actuators. If the control logic is closer to an on-off system, then most likely, a discrete event control system is adequate. If the control logic is closer to scalar math and based mostly on the same equations, the system is adequate for continuous control.

Frequently, real world complex systems rapidly become parallel in nature and detailed control math for the global system becomes cumbersome and inaccurate due to real world non-linearities such as sensor and actuator saturation limits.

Discrete event systems are useful for systems that:

1. Are complex
2. Have many variables
3. Have frequent interrelations
4. Need to recognize complex events
5. Need to track time sequences
6. Are highly parallel
7. Are non-linear
8. The control strategies change largely over time

It is possible to change a given system to make it or consider it to be of discrete event nature but the designer of the project must be aware that this simplification involves some tradeoffs and may, for example, reduce accuracy of an output.

1.15 *Events*

A model of a system is an equation that describes its evolution over time. Similarly, instead of an equation, the evolution of a system may be given by a chart.

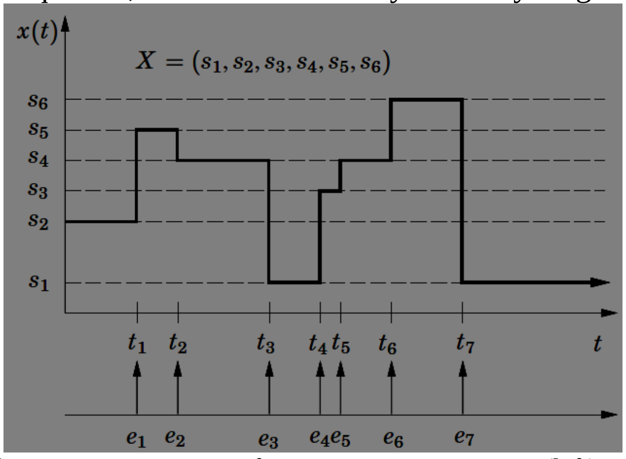

Figure 2 – Comparison of systems: continuous time (left) and discrete time (right)

Figure 2 compares two hypothetical systems, one of continuous nature (model) and the other of discrete nature (model). While the continuous time uses curved continuous line, the discrete one uses "floors", horizontal lines that change by means of vertical state changes that are said to have been caused by events. The discrete event system must be in one of its 6 possible states (s1, s2, …, s6) and changes among them by means of transitions that are (ideally) vertical. This means that thus just before t1 ($t=t_1^-$), the system is in state s_2 and in $t=t_1$ the system is in state s_5 and the change took very little time (virtually no time at all).

1.16 Introduction to Grafcet

Retaking the previous example of the on-off temperature sensor and on-off heater actuator, it is possible to consider as a control law the infinite repetition of `heater = not(sensor)`. This trivial example is understandable but large control code and many variables with intricate time relations would produce a single large block of unstructured code that would be difficult to document, debug and maintain.

The IEC 60848 standardized previous industry practice and defined a graphical programming language named Grafcet. It is also very close the IEC 61131 language named Sequential Function Block (SFC). This lecture will use the ideas of the textual language called "Structured Text" (ST), defined by IEC 61131[1] but will stick with the IEC 60848 Grafcet for the graphical part.

Basic Elements

The Grafcet graphical programming language inherits some ideas from the Finite State Machines (FSM) diagram. As in state diagrams of FSMs, the Grafcet way of thinking transforms large text programs into very small amounts of textual code interconnected in a visual diagram easy to read, document, debug and maintain.

The Grafcet equivalent of the states of the FSM are called steps, take a square shape and are to be numbered. Steps represent some kind of stability in the system (horizontal line in the right side of figure 2).

Steps are interconnected by transitions that are draw as perpendicular small horizontal lines or small rectangles perpendicular to the expected vertical downward "motion" of flow of a Grafcet diagram.

[1] *IEC 60848, GRAFCET specification language for sequential function charts, 2nd ed. International Electrotechnical Commission, 2002*

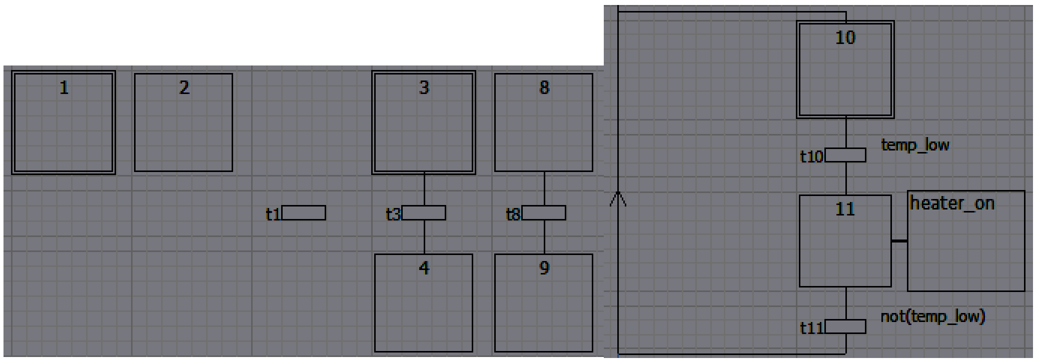

Figure 3 – Examples of Grafcet elements (steps, transitions and an action) and a small Grafcet

This lecture uses images from the "FEUPAutom[2]" tool and Figure 3 shows (from left to right):

- Step number 1 – marked with double frame, an initial step, active after system start
- Step 2 – a normal step with single frame
- Transition t1 – shown alone, no condition shown; t10 and t11 show transitions with conditions (Boolean expressions, in this case written in ST language)
- Transition t3 will deactivate (above) the initial step 3 and activate step 4 (below)
- Transition t8 will do similarly the same for steps 8 and 9 respectively

The right hand side of Figure 3 also shows a small Grafcet diagram:

- Step 10 is initial
- When the Boolean variable `temp_low` is true, transition t10 is fired and deactivates step 10 and activates step 11
- Step 11 has an Action and that action make the `heater_on` Boolean variable become true when step 11 is active (and False when not)
- When `temp_low` is no longer true, transition t11 deactivates step 11 and activates 10
- In spite of the drawing, step 10 is said to be below t11 (in the connection sense of the Grafcet, below means succeeds, that is step 10 succeeds t11)
- Upward flow in Grafcet must always be stated explicitly by an upward arrow

One of the strengths of the Grafcet is that it allows for different Grafcet to stand alone interconnected or not and their evolution may be interconnected or not.

Time Evolution

Discrete event systems are complex in nature because they deal with several signals over time and thus it is important to analyze in detail the workings of a simple Grafcet.

[2] *http://paginas.fe.up.pt/~asousa/wiki/doku.php?id=proj:feupautom*

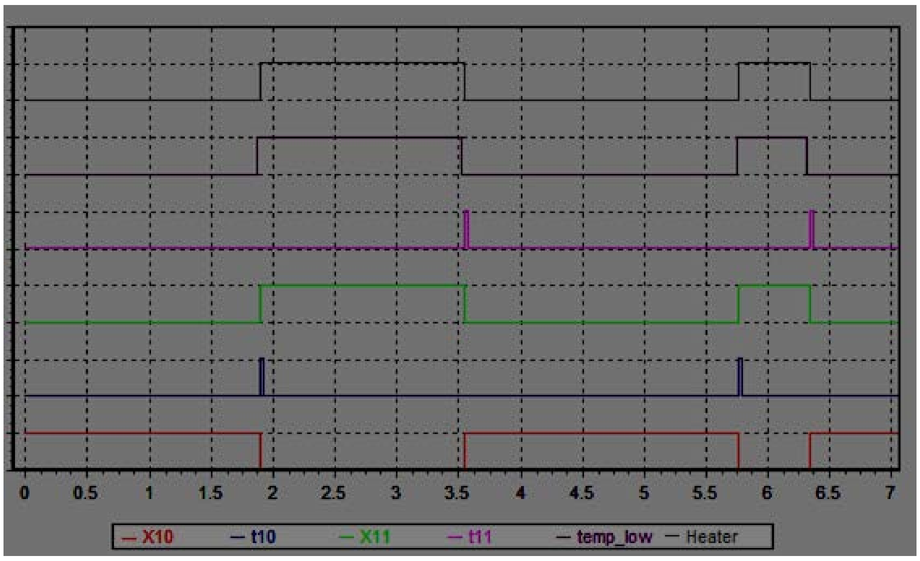

Figure 4 – Time evolution of Grafcet diagram of figure 3

Figure 4 depicts the temporal evolution of figure 3. The initial Step 10 (associated with variable X10) starts active at t=0. Just before t=2s, `temp_low` becomes True and t10 is fired (is active over a small time impulse). After that, X11 and associated Heater outputs become active. When `temp_low` becomes inactive, t11 is fired and X10 is activated again. Naturally, X11 and Heater become inactive.

Choice and Parallelism

The Grafcet language is adequate for parallel systems and as many steps as wanted may be simultaneously active and such it is possible to represent alternative paths and start of simultaneous "threads". Such configurations are shown in Figure 4. The mentioned (vertical) "threads" are the sequence of diagrams that start from step 31 up to step 41; similarly, another "thread" is 32 to 42 and so on until the last shown "thread", diagram starting in step 53 up to step 63.

Still regarding Figure 4, in the top left part of the diagram, t30 will activate all steps bellow its double line, steps 31, 32 and 33. When steps 41, 42 and 43 (all of them) are active, t40 may be fired and that event will deactivate all steps above double line (41 to 43) and activate steps below if any (in the example, step 40). The double line emphasizes that more than one step will be (de-)activated simultaneously.

Another different situation in Figure 4 (right) is that if step 50 is active, one of its single line connected transitions may be fired and a single one of the steps 51, 52, 53 will become active. If step 61 is active, t61 may be fired and 61 will be deactivated and 60 activated. The same for steps 62 and 63 with transitions t62 and t63 respectively.

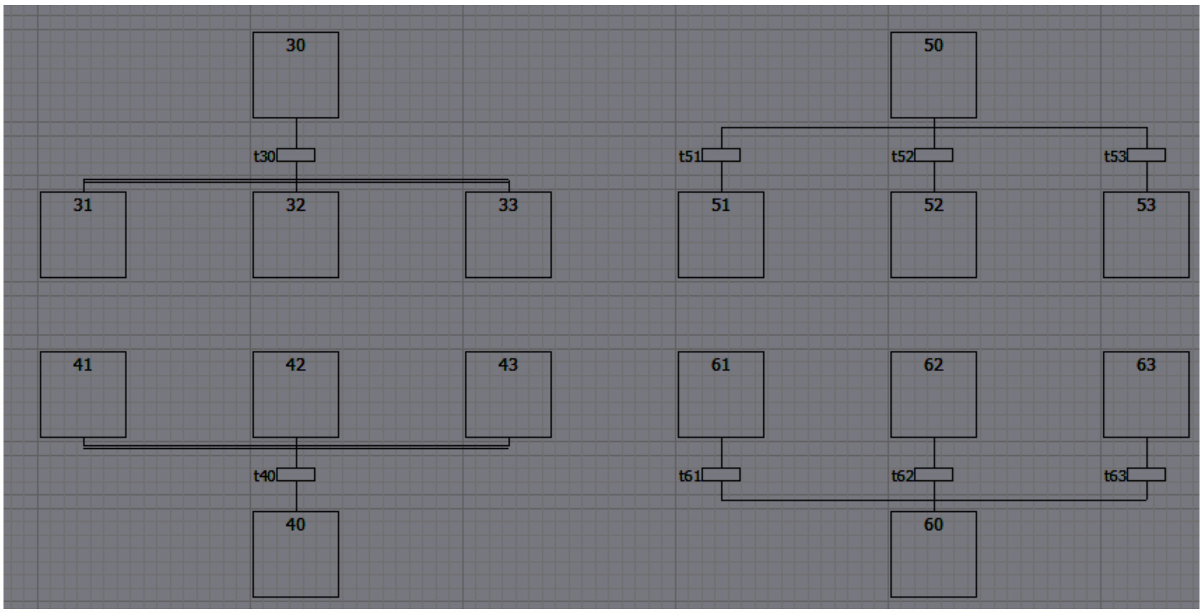

Figure 5 – Possible control structures – left: start of parallelism and end of parallel threads (synchronization); right: choosing and joining threads

In summary, the left part of Figure 5 shows 3 independent simultaneous threads, started by the firing of t30 (the threads start with steps 31 to 33). When such threads activate steps 41, 42, 43 and only then, t40 may be fired leading to steps 41, 42 and 43 being deactivated and step 40 activated.

The right part of Figure 5 shows a choice of thread, either t51, t52 or t53 can be taken; t61, t62 and t63 are also independent and each of them will activate step 60.

It is important to note that the code (or the problem) should enforce that only one of the choice transition (in this case t51, t52, t53) may be at most simultaneously active.

The situation of a Grafcet is the listing of its active steps, for example if, in Figure 5 situation {30}, then t30 may be fired; after its condition is evaluated as true, the transition is in fact fired and the system evolves into situation {31,32,33}; the overall evolution can be written as {30} → t30 → {31,32,33}.

Formal evolution rules

The Grafcet formal rules are:

1. Initial situation – upon start, all initial steps are activated
2. Firing a transition – if all steps above are active, the transition is said to be enabled; if also the Boolean condition associated with the transition is evaluated as true, then, the transition fires
3. Evolution – for a fired transition, deactivate steps above and activate steps below
4. Simultaneity – all fired transitions evolve at the same time
5. Conflict – if an active step is simultaneously deactivated and activated, it remains active; if activated more than once, it also remains active

Special Variables and Timer

As briefly mentioned earlier, there is a special Boolean variable associated with each step: X*stepnumber* (example: for step 10, variable X10) that is true if step is active and false if not.

Such “internal” variables can be used for example in the conditions of transitions (as well as input variables, etc.).

Time is a common problem in real world control systems.

This lecture will use de simplified version of the timers proposed in IEC 80484, and will only mention the most common, T*On* mode of timers.

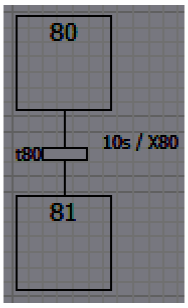

Figure 6 – Using a T*On* timer as the condition of a transition – t80 will fire after step 80 is active for 10 seconds

The example shown in Figure 6 illustrates the general Timer of Grafcet that follow the general notation

time / *variable*

The mentioned example is a timer of 10 seconds associated with variable X80 (temporal activity of step 80). This kind of timers can be used in the condition of a transition, example t80. The expression "10s / X80" will become true 10 seconds after step 80 is active. Overall, the Grafcet diagram of Figure 6 makes t80 fire 10 seconds after step 80 is active and consequently step 80 will be active for 10 seconds.

Shared Resources

Grafcet is adequate to solve problems involving management of shared resources.

Consider the problem in Figure 7 where two semi-automated cars C1 and C2, commanded centrally, that C1 must drive from P1 to Px and C2 from P2 to Px, sharing a piece of track that can only be used by one of them at a time (a shared resource).

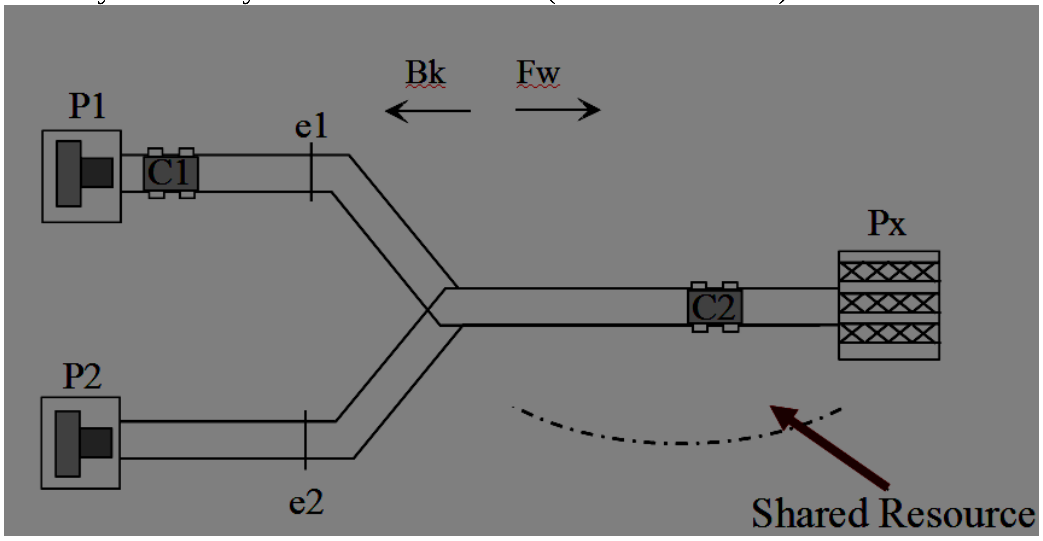

Figure 7 – Shared Resource Problem

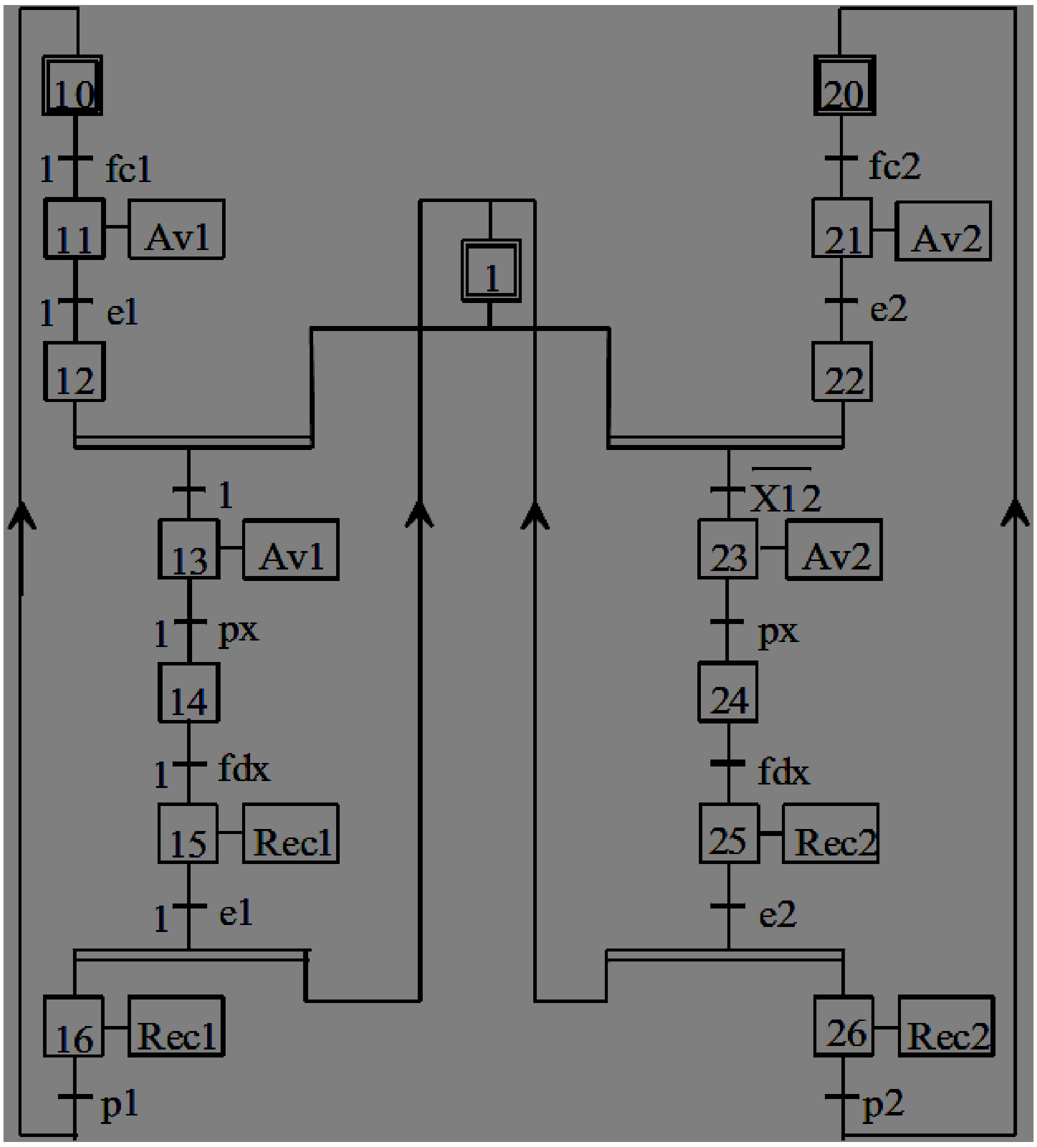

Figure 8 – Solution for problem in Figure 7

Figure 8 shows a possible solution for the problem presented in Figure 7, using notions form shared resource management in Grafcet.

The initial situation is {1, 10, 20}. If steps 11 and or 21 become active there is no problem. When any of them reaches e1 and e2 (entry to shared part of track), they both stop (steps 12 and or 22). After that, one of three situation may arise:

1. {1,12} → {13}
2. {1,22} → {23}
3. {1,12,22} → {13}

If step 12 or 22 becomes active and step 1 is not active, that step will remain active and "wait" for step 1.

This working ensures the shared resourse (the shared track) is only assigned to one of the requestants (cars), that is, only one car is allowed enter in the shared track.

Still regarding Figure 8, the activity of step 1 simbolizes that the shared resource is free/unused, that is, if X1 is active, then the resource (shared track) is free (of cars). After using the resource, it is necessary to free it (make step 1 active again) and this is done when either car leaves the shared portion of the track. When C1 leaves the shared resource the situation changes {15} → {1, 16}; when car C2 leaves shared track, the situation will evolve {25} → {1, 26}.

Generally, shared resource treatment is made by ensuring that the entry to the shared resource section is allowed by deactivating a step and activating it at the end of the shared resource use.

Many other solutions using Grafcet would be possible, for example, substituting the structure by variables or treating the resource as a separate Grafcet.

Tree of situations

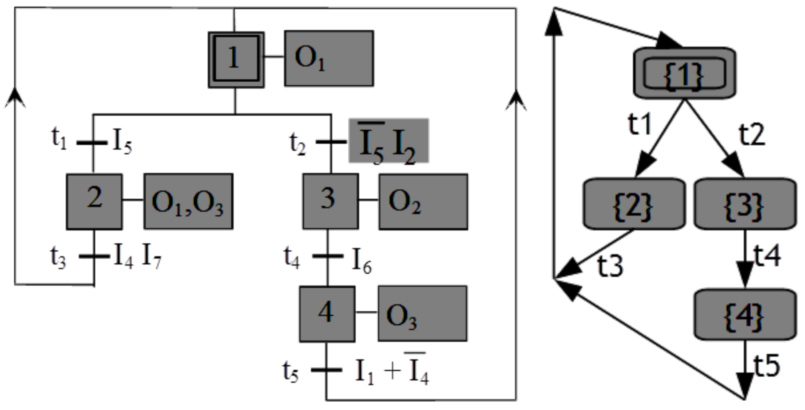

Figure 9 – Example Grafcet diagram and associated tree of situations

Figure 9 shows the tree of accessible situations that the grafcet may reach. For example this analisys demonstrates that in no circunstance will steps 2 and 3 be active at the same time nor will O1 and O2 ever be simultaneously active.

1.17 Implementation

This section details how to control a system using general purpose equipments (example PC hardware).

Control Cycle

Healthy controllers start by reading all inputs to get a single global coherent image of the process. All physical inputs are copied into image variables. Only then the controller goes on to execute the control code, using images of inputs and outputs. If, during the cycle there are conflicting orders, the last one wins and the physical outputs are not affected. Only at the end of the control cycle the images of the outputs are written to the physical outputs. This is the healthy controller control cycle, interesting for implementing control of problems that have parallelism such as Grafcet diagram.

Care should be taken not to assume that all control cycles take the same time, thus counting cycles is not an option to count time.

Code Generation

One of the strengths of IEC 60848 Grafcet is that many manufactures of control equipments support it and make available compilers but it is interesting to know how to produce code to comply with Grafcet rules.

In order to compile, it is interesting to list the data structures.

In the case of Figure 10, the data structures are:

- Step_List - E[] = [1,10,11,20,21,30,40,50,60]
- Transition_List - T[] = [1,10,20,29,39,49,59,40,50,60]
- List of Steps above each transition
  Steps_Above_Transition[][] =
  [t1 => (1) ; t10 => (10) ; … ; t29 => (11,21) ; … t60 => (60) ; …]
- List of Steps below each transition
  Steps_Below_Transition [][]=
  [t1 => (10,20) ; t10 => (11) ; … ; t29 => (30) ; …; t60 => (1) ; …]

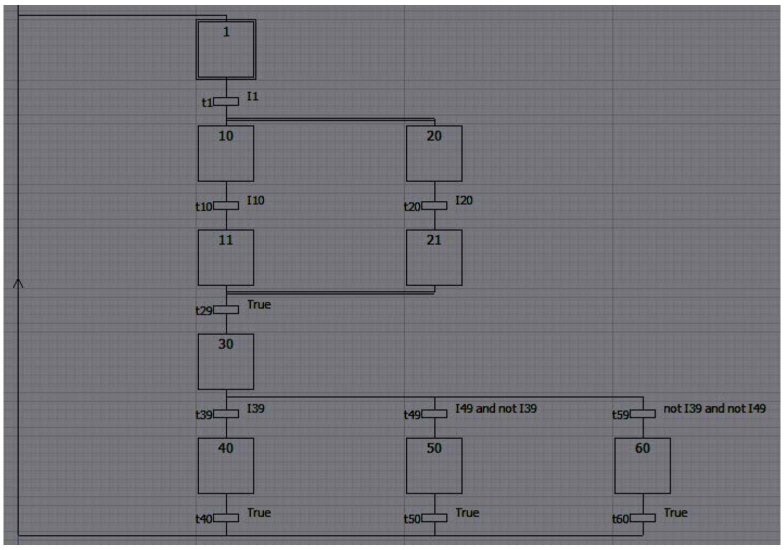

Figure 10 – Example Grafcet for implementation purposes

Additionally, it is interesting to have auxiliary Boolean variables for each step (X_{number}) and for each transition (t_{number}). To allow easiness of use, it is also interesting to have an integer that measures the time each step is active (X_{number}_T) – this is an implementation detail, not the letter of the standard.

Algorithm for code generation:

1. If boot, activate initial steps and directly goto Step 6
2. Calculate fired transitions (calculate t*number*=…)
3. For fired transitions, deactivate steps above (X*number*=false)
4. For fired transitions, activate steps below and reset timers associated with each step (X*number*=true; X*number*_T=0;)
5. At a given time rate, for all active steps: increment time Xnumber_T++
6. First, make all outputs inactive and then activate the outputs listed in the active steps

Such code generation can be done “by hand” using the Grafcet as a healthy model or can be generated by several tools such as the FEUPAutom or the Beremiz (MatPLC / MATIEC).

Taking the diagram shown in Figure 10, the code generated by the FEUPAutom automated C code generator is shown below:

```c
////////////////////////////////////////////////////////////
// FEUPAutom _ C _ v4.03 -
////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////
///////////// If boot => Set Initial Steps ////////////////
////////////////////////////////////////////////////////////

  // ObjIdx=0 => INI_Step "X1"
  if( (_SysWords[0]==0) ) {     _MemBits[0] = 1;  };

  if( (_SysWords[0]>0) ) {  // No evolution in initial cycle

////////////////////////////////////////////////////////////
///////////////// Calc Fired Transitions //////////////////
////////////////////////////////////////////////////////////

  // ObjIdx=1 => Transition "t1"
  // Steps Above: id=0 => X1 ;
  _MemBits[1] =  _MemBits[0] &&  (_InBits[1]()) ;
  // ObjIdx=4 => Transition "t10"
  // Steps Above: id=2 => X10 ;
  _MemBits[4] =  _MemBits[2] &&  (_InBits[10]()) ;
  // ObjIdx=5 => Transition "t20"
  // Steps Above: id=3 => X20 ;
  _MemBits[5] =  _MemBits[3] &&  (_InBits[20]()) ;
  // ObjIdx=8 => Transition "t29"
  // Steps Above: id=6 => X11 ;id=7 => X21 ;
  _MemBits[8] =  _MemBits[6] &&  _MemBits[7] &&  (1) ;
  // ObjIdx=10 => Transition "t39"
  // Steps Above: id=9 => X30 ;
  _MemBits[10] =  _MemBits[9] &&  (_InBits[39]()) ;
  // ObjIdx=11 => Transition "t49"
  // Steps Above: id=9 => X30 ;
  _MemBits[11] =  _MemBits[9] &&  (I49 && !_InBits[39]() ) ;
  // ObjIdx=12 => Transition "t59"
  // Steps Above: id=9 => X30 ;
  _MemBits[12] =  _MemBits[9] &&  (!_InBits[39]() && !I49) ;
  // ObjIdx=16 => Transition "t40"
  // Steps Above: id=13 => X40 ;
  _MemBits[16] =  _MemBits[13] &&  (1) ;
  // ObjIdx=17 => Transition "t50"
  // Steps Above: id=14 => X50 ;
  _MemBits[17] =  _MemBits[14] &&  (1) ;
  // ObjIdx=18 => Transition "t60"
  // Steps Above: id=15 => X60 ;
  _MemBits[18] =  _MemBits[15] &&  (1) ;
}; // No evolution in initial cycle

////////////////////////////////////////////////////////////
///////////////// ReSet Steps Above fired Tr //////////////
////////////////////////////////////////////////////////////

// ObjIdx=1 => Transition "t1"
  // Steps Above: id=0 => X1 ;
  if( (_MemBits[1]) ) {
     _MemBits[0]=0;
  };
// ObjIdx=4 => Transition "t10"
  // Steps Above: id=2 => X10 ;
  if( (_MemBits[4]) ) {
     _MemBits[2]=0;
  };
// ObjIdx=5 => Transition "t20"
  // Steps Above: id=3 => X20 ;
  if( (_MemBits[5]) ) {
     _MemBits[3]=0;
  };
// ObjIdx=8 => Transition "t29"
  // Steps Above: id=6 => X11 ;id=7 => X21 ;
  if( (_MemBits[8]) ) {
     _MemBits[6]=0;  _MemBits[7]=0;
  };
// ObjIdx=10 => Transition "t39"
  // Steps Above: id=9 => X30 ;
  if( (_MemBits[10]) ) {
     _MemBits[9]=0;
  };
// ObjIdx=11 => Transition "t49"
  // Steps Above: id=9 => X30 ;
  if( (_MemBits[11]) ) {
     _MemBits[9]=0;
  };
// ObjIdx=12 => Transition "t59"
  // Steps Above: id=9 => X30 ;
  if( (_MemBits[12]) ) {
     _MemBits[9]=0;
  };
// ObjIdx=16 => Transition "t40"
  // Steps Above: id=13 => X40 ;
  if( (_MemBits[16]) ) {
     _MemBits[13]=0;
  };
// ObjIdx=17 => Transition "t50"
  // Steps Above: id=14 => X50 ;
  if( (_MemBits[17]) ) {
     _MemBits[14]=0;
  };
// ObjIdx=18 => Transition "t60"
  // Steps Above: id=15 => X60 ;
  if( (_MemBits[18]) ) {
     _MemBits[15]=0;
  };

////////////////////////////////////////////////////////////
///////////////// Set Steps below fired Tr ////////////////
////////////////////////////////////////////////////////////

// ObjIdx=1 => Transition "t1"
  // Steps Below: id=2 => X10 ;id=3 => X20 ;
  if( (_MemBits[1]) ) {
    _MemBits[2] = 1;  _MemBits[3] = 1;
    _MemWords[2] = 0; _MemWords[3] = 0;
  };
// ObjIdx=4 => Transition "t10"
  // Steps Below: id=6 => X11 ;
  if( (_MemBits[4]) ) {
    _MemBits[6] = 1;
    _MemWords[6] = 0;
  };
// ObjIdx=5 => Transition "t20"
  // Steps Below: id=7 => X21 ;
  if( (_MemBits[5]) ) {
    _MemBits[7] = 1;
    _MemWords[7] = 0;
  };
// ObjIdx=8 => Transition "t29"
  // Steps Below: id=9 => X30 ;
  if( (_MemBits[8]) ) {
    _MemBits[9] = 1;
    _MemWords[9] = 0;
  };
// ObjIdx=10 => Transition "t39"
  // Steps Below: id=13 => X40 ;
  if( (_MemBits[10]) ) {
    _MemBits[13] = 1;
    _MemWords[13] = 0;
  };
// ObjIdx=11 => Transition "t49"
  // Steps Below: id=14 => X50 ;
  if( (_MemBits[11]) ) {
    _MemBits[14] = 1;
    _MemWords[14] = 0;
  };
// ObjIdx=12 => Transition "t59"
  // Steps Below: id=15 => X60 ;
  if( (_MemBits[12]) ) {
    _MemBits[15] = 1;
    _MemWords[15] = 0;
  };
// ObjIdx=16 => Transition "t40"
  // Steps Below: id=0 => X1 ;
  if( (_MemBits[16]) ) {
    _MemBits[0] = 1;
    _MemWords[0] = 0;
  };
// ObjIdx=17 => Transition "t50"
  // Steps Below: id=0 => X1 ;
  if( (_MemBits[17]) ) {
    _MemBits[0] = 1;
    _MemWords[0] = 0;
  };
// ObjIdx=18 => Transition "t60"
  // Steps Below: id=0 => X1 ;
  if( (_MemBits[18]) ) {
    _MemBits[0] = 1;
    _MemWords[0] = 0;
  };

////////////////////////////////////////////////////////////
///// If step active increment MW timer of step @ %s16 /////
////////////////////////////////////////////////////////////

  if (_SysBits[16]) {  // a tenth of second has gone by
    // ObjIdx=0 => Step "X1"
    if( (_MemBits[0]) ) { _MemWords[0] = _MemWords[0]+1; };
    // ObjIdx=2 => Step "X10"
    if( (_MemBits[2]) ) { _MemWords[2] = _MemWords[2]+1; };
    // ObjIdx=3 => Step "X20"
    if(  (_MemBits[3]) ) { _MemWords[3] = _MemWords[3]+1; };
    // ObjIdx=6 => Step "X11"
    if( (_MemBits[6]) ) { _MemWords[6] = _MemWords[6]+1; };
    // ObjIdx=7 => Step "X21"
    if( (_MemBits[7]) ) { _MemWords[7] = _MemWords[7]+1; };
    // ObjIdx=9 => Step "X30"
    if( (_MemBits[9]) ) { _MemWords[9] = _MemWords[9]+1; };
    // ObjIdx=13 => Step "X40"
    if( (_MemBits[13]) ) { _MemWords[13] = _MemWords[13]+1; };
    // ObjIdx=14 => Step "X50"
    if( (_MemBits[14]) ) { _MemWords[14] = _MemWords[14]+1; };
    // ObjIdx=15 => Step "X60"
    if( (_MemBits[15]) ) { _MemWords[15] = _MemWords[15]+1; };
  }

////////////////////////////////////////////////////////////
////////// Unset all Outputs
////////////////////////////////////////////////////////////

  _OutBits[0]=0;   _OutBits[1]=0;   _OutBits[2]=0;
  _OutBits[3]=0;   _OutBits[4]=0;   _OutBits[5]=0;
  _OutBits[6]=0;   _OutBits[7]=0;   _OutBits[8]=0;
  _OutBits[9]=0;   _OutBits[10]=0;  _OutBits[11]=0;
  _OutBits[12]=0;  _OutBits[13]=0;  _OutBits[14]=0;
  _OutBits[15]=0;  _OutBits[16]=0;  _OutBits[17]=0;
  _OutBits[18]=0;  _OutBits[19]=0;  _OutBits[20]=0;
  _OutBits[21]=0;  _OutBits[22]=0;  _OutBits[23]=0;
  _OutBits[24]=0;  _OutBits[25]=0;  _OutBits[26]=0;
  _OutBits[27]=0;  _OutBits[28]=0;  _OutBits[29]=0;
  _OutBits[30]=0;  _OutBits[31]=0;  _OutBits[32]=0;
  _OutBits[33]=0;  _OutBits[34]=0;  _OutBits[35]=0;
  _OutBits[36]=0;  _OutBits[37]=0;  _OutBits[38]=0;
  _OutBits[39]=0;  _OutBits[40]=0;  _OutBits[41]=0;
  _OutBits[42]=0;  _OutBits[43]=0;  _OutBits[44]=0;
  _OutBits[45]=0;  _OutBits[46]=0;  _OutBits[47]=0;


////////////////////////////////////////////////////////////
//////// If step active, execute its action code //////////
////////////////////////////////////////////////////////////

  // ObjIdx=0 => Step "X1" (code...)
  // ObjIdx=2 => Step "X10" (code...)
  // ObjIdx=3 => Step "X20" (code...)
  // ObjIdx=6 => Step "X11" (code...)
  // ObjIdx=7 => Step "X21" (code...)
  // ObjIdx=9 => Step "X30" (code...)
  // ObjIdx=13 => Step "X40" (code...)
  // ObjIdx=14 => Step "X50" (code...)
  // ObjIdx=15 => Step "X60" (code...)

/********** End of C Code **********/
```

1.18 *Further Reading*

- Rene David ; Hassane Alla; “Petri Nets and GRAFCET: Tools for Modelling Discrete Event Systems” - New York ; PRENTICE HALL Editions, 1992; ISBN-10: 013327537X; ISBN-13: 978-0133275377
- Rene David ; Hassane Alla; “Du Grafcet aux réseaux de Petri” - Hermes Sciences Publications; - Hermes Sciences Publications, 1992 (Language: French); ISBN-10: 2866013255; ISBN-13: 978-2866013257

4. Lab – Week 5

In the previous week you controlled a very simple plant floor using an *ad hoc* approach. For this lab session you are requested to re-implement that exact same problem, but now using the state machine based approach proposed in the lecture. Assume that each conveyor can only have one single work-piece at any time. When transporting a work-piece from one conveyor to the next, these conveyors only become free to handle other work-pieces once the transfer is complete.

1. Produce a Grafcet to control the system;
2. Compile it “manually” and implement by making use of the routines developed earlier in the course;
3. Test the system by making sure global throughput is as fast as possible;

You should re-use the software you developed that implements the Modbus communication protocol, so you can communicate with the plant floor simulator.

5. Lab – Week 6

The lab session for this week consists on extending your previous lab program to be able to handle multiple work-pieces simultaneously. You will need to consider each conveyor as a shared resource, as each conveyor is only allowed to have at most one work-piece present at all times.
Use one the approach to handle shared resources that you think most appropriate. All of them will work, although they will differ in the final resulting code..

6. Seminar – Week 5

The lecture this week proposed that the discrete event system be modeled using the Grafcet modeling language. There are however other similar modeling languages that could also be used. We suggest that each group summarize the properties (syntax and semantics) of a specific discrete event system modeling language, highlighting the similarities and differences to Grafcet.

Other suggested modeling languages to consider: state machines, Petri Nets, SFC (defined in IEC 61131-3), UPPAAL modeling language, etc.

7. Seminar – Week 6

Find in the internet the most prominent brands of industrial controllers (PLC Programmable Logic Controllers) - for example the top 5 brands in value of equipment sold. For each of these browse through commercial documents to find if or up to what point each brand supplies compilers for any flavor of Grafcet and or discrete event controller language. Find which standards they do comply with.
Also find some prominent free solutions, and repeat the previous characterization.
Build a comparison table ordered by sales value that shows:

- Grafcet support / type of language for discrete event control
- Range of hardware that is supported and its price range
- Limitations of the implementation with regard to international standards
- Cost and type of license of the software solution
- Include portability and O.S. Issues

8. Mini-project – Week 5

In the previous two weeks you have built a program that is able to communicate with the plant floor simulator. In these weeks you focused on the communication aspect of the program – i.e. you implemented the Modbus/TCP client. The control application itself was simplified by having the plant floor reduced to only 3 linear conveyors.

In this week and the next, you will extend this very basic program to start taking into account all the requirements of the full cell described during week 1.

In this first week, we ask you to implement a control program able to control all the machines and conveyors, considering that only one piece is processed at any time in the cell. You should consider that both P1 type pieces (red) and P2 type pieces (blue) may be introduced in the cell, in any order.
We suggest you start off by considering that only one P1 Type piece is processed at a time. Once this is working, you can then do the same for P2 type pieces.
It is highly advisable that you first model your control application as a state machine, and only then you translate the state machine into code.

9. Mini-project – Week 6

In this week you will extend the previous control application so it can handle both P1 and P2 type pieces simultaneously. Multiple pieces of P1 and P2 should be able to be processed at the same time. Take care with the shared resources.
If you wish to make your control program more interesting, you can add a requirement that each piece will be processed with a different type of tool at the bottom machine. This means you may have to change tool if the previous piece that was processed was of a different type.
For even more challenging requirements, can try to handle the situation in which both P1 and P2 type pieces are waiting to be processed in T1 and T2 respectively (because T3 is occupied) using one of several algorithms. For example, you may opt to give alternating priorities (P1, P2, P1, P2, …). Or you could always give priority to the first workpiece to be placed in the respective conveyor.

10. Bibliography

- IEC 60848, GRAFCET specification language for sequential function charts, 2nd ed. International Electrotechnical Commission, 2002
- Paulo Portugal and Adriano Carvalho; “The industrial information technology handbook 2005” - "Chapter 64 - The GRAFCET Specification Language" – CRC press – Ed. Richard Zurawski, 2005. ISBN: 978-0849319853; http://www.crcnetbase.com/isbn/9780849319853