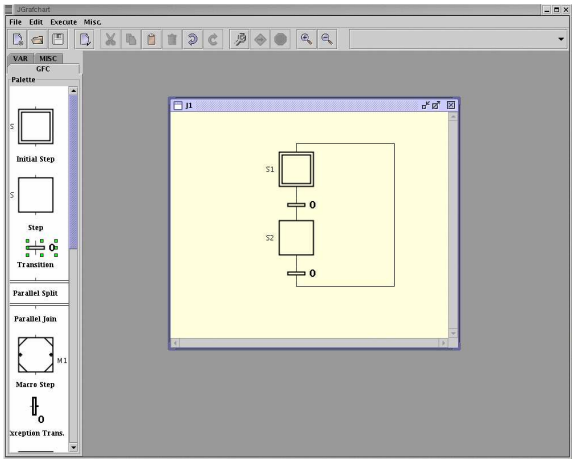
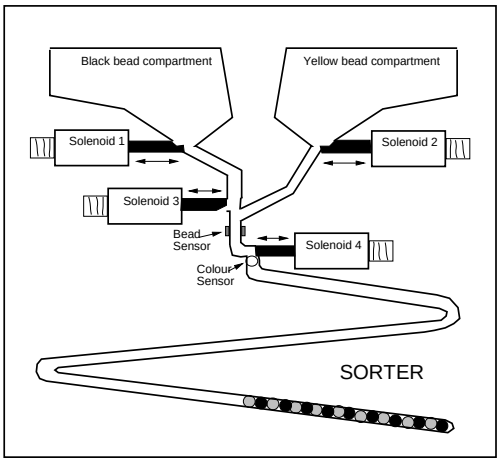
Real-Time Systems

Laboratory 2: Sequence Control





Karl-Erik Årzén, Rasmus Olsson,
Mattias Grundelius, Vanessa Romero Segovia

Department of Automatic Control
Lund Institute of Technology
October 2012

Introduction

Sequence Control

The topic of the laboratory is sequence control of a bead sorter process using Grafcet/SFC. Although a toy, the bead sorter process contains many of the control problems found in discrete manufacturing control applications. A Grafcet/SFC editor and execution environment called JGrafchart is used to implement the sequence control.

SFC (Sequential Function Chart) defined in the IEC 848 standard is a graphical programming language used for PLCs (Programmable Logic Controllers). It is one of the five languages defined in IEC 61131-3 standard. The SFC inherits its characteristics from the French standard Grafcet which itself is based on Petri nets.

The basic concepts of this discrete system model are steps, actions, transitions, and conditions associated with transitions. A step represents a partial state of the system, in which an action can be performed. The step can be active or inactive; an action associated with a step is only performed when the step is active. A transition represents a logic condition and defines the direct link between steps.

Grafcet/SFC allows programming of sequential logic and parallel control execution (multiple control flows can be active at once). This makes Grafcet/SFC very suitable for solving problems related to manufacturing control applications.

Before the laboratory you must have read the laboratory manual and answered the preparation exercises.

1. JGrafchart

JGrafchart is a Grafcet editor and execution environment developed at the Department of Automatic Control, Lund Institute of Technology. JGrafchart is written in Java using Swing graphics. It also uses JGo, a class package for graphical object editors from Northwoods Corporation, and JavaCC, a Java parser generator from Sun. In the laboratory you will run JGrafchart on top of Linux.

Note: JGrafchart has a built-in Undo/Redo function. However, in certain situations it does not work as intended. Therefore it is currently disabled. Hence, be careful before deleting any items and make sure that you save your Grafcet application to file regularly.

In the laboratory you will also use a stripped down version of JGrafchart that only contains the language elements that you will actually use.

1.1 Workspaces

Grafcet sequence diagrams are created interactively using drag-and-drop from a palette containing the different Grafcet language elements. The sequence diagrams are stored on JGrafchart workspaces, implemented as Java internal frames, see Fig.1.1.

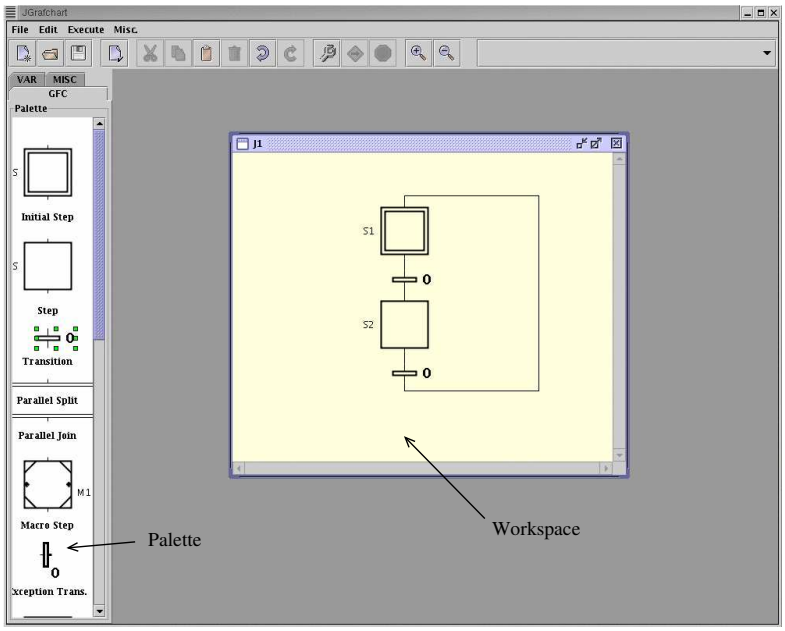


Figure 1.1 JGrafchart main interface

Workspaces can be stored to a file and loaded from a file. The XML format is used for storing. Workspaces support scroll, pan, and zoom. It is possible to change their size, to iconize them, etc. The default name of a workspace (J1,

J2, ...) can be changed from the *Properties* menu choice in the *File* menu. If multiple workspaces are used, only one of them is the current focus for menu choices. This is indicated through a blue workspace border, rather than the ordinary gray border. The focus is changed by clicking on a workspace. This also automatically moves the workspace to the front.

On a workspace it is possible to select an object or an area containing multiple objects in the standard fashion. A selected object can be moved, cut to the clipboard, or copied to the clipboard. The contents of the clipboard can be pasted to a workspace.

Grafcet objects are connected together graphically by clicking on the connection stubs. A connection can be moved by selecting it and moving a corner point. This is specially needed when a Grafcet object is connected to another object that lies above the first object.

A selected Grafcet object or connection is deleted using the `Delete` key.

1.2 Grafcet elements

JGrafchart supports the following Grafchart elements: steps, initial steps, transitions, parallel splits, parallel joins, macro steps, digital inputs, digital outputs, and internal variables (boolean and integer).

Steps

Grafcet steps have action blocks that may be made visible or hidden through menu choices on the step menu that is obtained by double-clicking on the step, see Fig. 1.2. Step actions are entered as text strings, either directly by

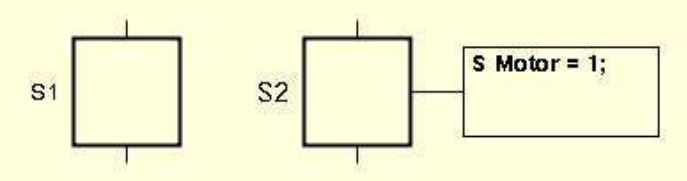


Figure 1.2 Step with action block hidden and visible.

clicking on the text string in the action block or through the *Edit* step menu choice. The latter is an advantage, since the editor type-in-field in the action block does not adjust its size with the text during typing. Multiple step actions are separated by semi-colons.

Four different action types are supported. Stored actions (impulse actions) are executed once when the step is activated. The syntax for step actions is:
`S "variable-or-output" = "expression";`

Periodic actions (always actions) are executed periodically, once every scan cycle, while the step is active. The syntax for periodic actions is:
`P "variable-or-output" = "expression";` The default scan cycle is 40 ms.

Exit actions (finally actions) are executed once, immediately before the step is deactivated. The syntax for exit actions is:

```
X "variable-or-output" = "expression";
```

Normal actions (level actions) associate the truth value of a digital output with the activation status of the step. The syntax for a normal action is:

```
N "output";
```

The expression syntax follows the ordinary Java syntax, with some minor exceptions. One important exception is that the literal 0 (1) is used both to represent the boolean literal false (true) and the integer literal 0 (1). The context decides the interpretation.

The operators supported are: + (plus), - (minus), * (multiplication), / (integer division), ! (negation), & (and), | (or), == (equal), != (not equal), < (less than), > (greater than), <= (less or equal), >= (greater or equal).

Expressions may contain name references to inputs, outputs, and variables. JGrafchart uses lexical scoping based on workspaces. For example, a variable named X on workspace W1 is different from a variable named X on workspace W2. References between workspaces are expressed using dot-notation. For example, a step action in a step on workspace W1 can refer to the variable Y on workspace W2 using `W2.Y`.

By default steps do not have any names. In order to give a step a name use the *Set Name* menu choice of the step. During compilation unnamed steps are automatically given temporary names on the form *#0*, in order to make it possible to identify a step in case of compilation errors.

Initial steps

Initial steps are ordinary steps that are active initially when the execution of the sequence diagram starts. Initial steps may have actions in the same way as ordinary steps.

Transitions

Transitions represent conditions or events that should be true in order for the Grafcet to change state. The transition expression is represented by a text string associated with the transition, see Fig. 1.3. A transition expression is edited either directly by clicking on the expression or through the *Edit* transition menu choice.

The transition expression should return a boolean value. The expression syntax is the same as for step actions with a few additions. The expression "step".x returns true if the step is active and false otherwise. The expression "step".t returns the number of scan cycles since the step was last activated. The expression "step".s returns the number of seconds since the step was last activated. The expression /"boolean-input" represents a positive trigger event. It is true if the value of an input was false in the previous scan cycle and is true in the current cycle. Similarly, the expression \"boolean-input" represents a negative trigger event. For example, the expression `(/y | \y)` is true whenever the boolean input `y` changes its value.

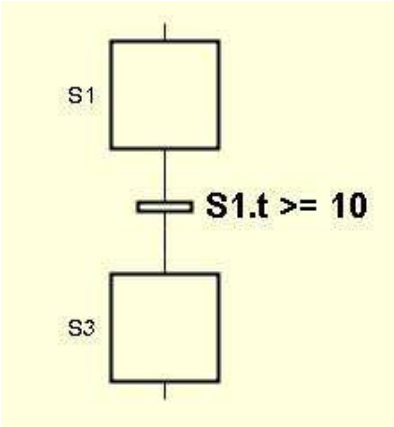


Figure 1.3 Transition

Parallel Splits and Joins

Parallel branches are created and terminated with parallel splits and parallel joins. The parallel objects only allow two parallel branches. If more branches are needed, the parallel elements can be connected in series, see Fig. 1.4.

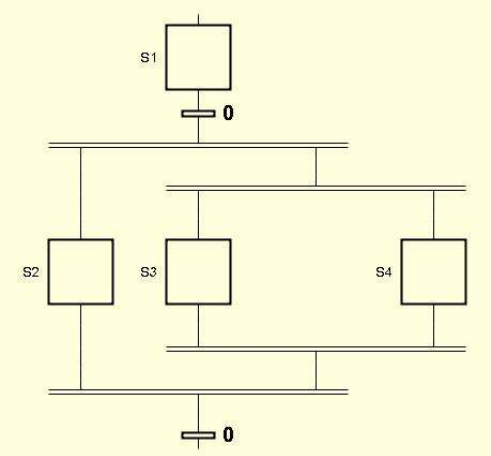


Figure 1.4 Parallel branching with three branches.

Macro Steps

A macro step represents a hierarchical abstraction. The macro step contains an internal structure of steps, transitions, and macro steps represented on a separate (sub)workspace. The sub-workspace is made visible and hidden by double-clicking on the macro step. The first step in the macro step is represented by a special enter step. Similarly the final step of the macro step is represented by a special exit step. Both the enter step and exit step are ordinary steps and may have actions. The macro step itself may also have actions. The situation is shown in Fig. 1.5.

The sub-workspace of a macro step has a local namespace lexically contained within the namespace of the macro step itself. For example, the sub-

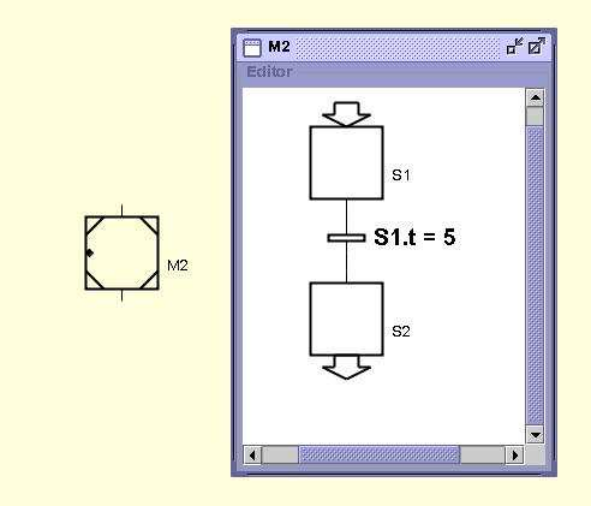


Figure 1.5 Macro step M2 with internal structure. The internal structure contains the enter step S1 and the exit step S2.

workspace of the macro step M1 may itself contain a macro step named M1, without causing any ambiguities.

Digital Inputs and Outputs

Digital inputs represent boolean variables that can be read by the steps and transitions in a sequence diagram. Similarly, digital outputs represent boolean variables that can be written to by the step actions in the sequence diagram. Each input and output has an associated value (0 or 1), a name, and a channel number, see Fig. 1.6. The name and channel numbers can be changed through click-and-edit. For digital inputs, the value can be toggled by double-clicking on the input. Digital inputs have the initial value 0. Two types of digital output exists. One with ordinary logic (initial value 0) and one with inverted logic (initial value 1). Digital inputs and outputs may only be located on top-level workspaces.

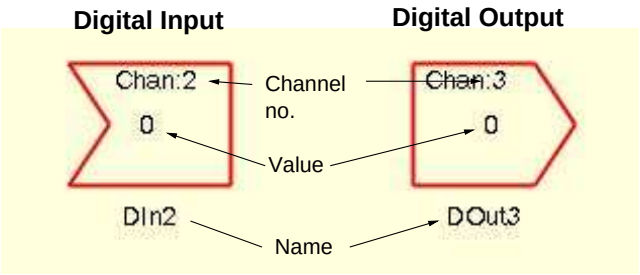


Figure 1.6 Digital input and output.

Internal Variables

Internal variables are variables that can be both read from and written to. Four types of variables are available: real variables, boolean variables, integer variables, and string variables. Associated with each variable are its value and its name, see Fig. 1.7. Both can be changed by click-and-edit.

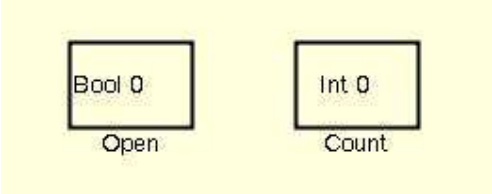


Figure 1.7 Boolean variable (left) and integer variable (right).

By default, a variable retains its value when the application is stopped and later restarted. Optionally, the user may give a variable an initial value. If the variable has an initial value its value will be set to the initial value every time the application is started. If you do not use any initial value you must make sure that the variable is initialized in the appropriate step.

1.3 Execution

Grafcet sequence diagrams are executed by a periodic thread associated with each top-level workspace. The thread cyclically performs three operations:

1. Read Inputs. The values of the digital inputs are read.
2. Execute Diagram. All the transitions in the diagram are checked. Steps are activated and deactivated.
3. Write Outputs. The values of the digital outputs are written.

A Grafcet sequence diagram can be executed in two different modes. In simulated mode the inputs and outputs are only connected to the graphics on the screen. In on-line mode (non-simulated mode), additionally, inputs are read from the digital I/O and outputs are written to the digital I/O. This mode only works when executing on the machines in the lab rooms of our department. The execution mode and the thread sleep interval determining the scan rate may be changed using the *Properties* menu choice. Note that changing the thread sleep interval also affects all wait intervals, since "step".t returns the number of scan cycles.

Before a sequence diagram can be executed it must be compiled. This is done through the *Execute* menu or by pressing the wrench button. During compilation two things are performed. First, for every transition two lists are built up. One list containing references to all the steps preceding the transition, and one list containing references to all the steps succeeding the transition. Second, the transition expressions and step actions are compiled. Compilation errors are shown in the message menu just below the tool bar.

The execution is started through the *Execute* menu or by pressing the arrow button. The execution is stopped through the *Execute* menu or by pressing the stop sign button.

Two types of problems may arise during compilation: parsing errors and symbol table lookup errors. Parsing errors are actually detected already when the step actions and transition expressions are entered. For example, the transition expression (y OR z) would generate a parsing error. (The syntactically correct expression should be `(y | z)`). Symbol table lookup errors occur if a name reference does not exist, e.g., if there does not exist any variables named `y` or `z` in the previous example. Both parsing errors and symbol table lookup errors are indicated by a change in the text colour of the transition expression or step action from black to red. NOTE: In the current version of JGrafchart symbol table lookup errors involving name references that point to the wrong type of objects are not caught! For example, assume that `y` is the name of a variable, and the following transition expression is compiled: `(y.t > 10)`. For this to work `y` should be the name of a step rather than a variable. Errors of this type may generate run-time faults that may crash JGrafchart. You are therefore recommended to regularly save your work on file.

The syntax for transition expressions and step actions is expressed by formal grammars. The parser generator tool JavaCC is used to generate Java parsers for these text expressions. During the parsing, a syntax tree is built up. During compilation the syntax tree is traversed, and all nodes representing name references are replaced by Java references to the corresponding Grafcet object. The expressions are evaluated on-line, again by traversing the syntax tree. For example, assume that a transition contains the transition expression `y = 5` and that `y` is the name of an integer variable. During parsing the syntax tree in Fig. 1.8 is generated and during compilation the symbol reference is created.

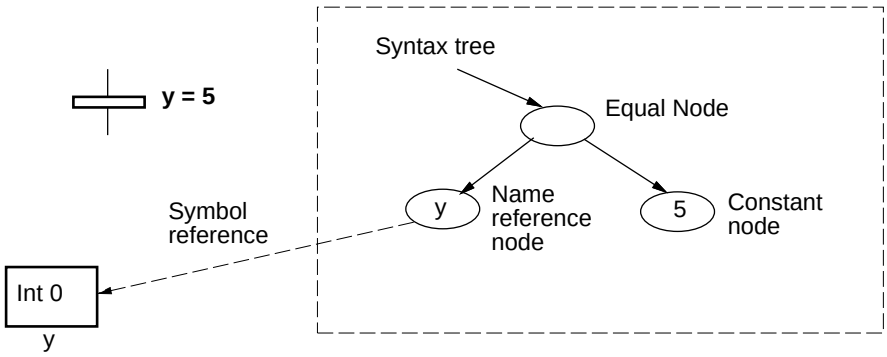


Figure 1.8 Syntax tree for the expression y = 5.

In the `Execute Diagram` part of the execution cycle the following operations are performed. For each transition in the diagram, the transition expression is evaluated. If it is false, then the transition icon is changed to red. If it is true, the transition icon is changed to green. If, additionally, all steps preceding the transition are active, then the steps preceding the transition are marked to become deactivated in the next cycle, and all the steps succeeding the transition are marked to become activated in the next cycle. When all transitions have been checked, the change of step state is effectuated. In addition to the things above, step actions are executed and the step timing information is updated.

2. Bead sorting

The bead sorting process is a laboratory version of a process that is similar to processes often found in the manufacturing industry. The main objective is to select sequences of two different components and subsequently re-sort them. Beads of two different colours (black and yellow) are used to represent the components. The beads of each colour are provided from separate compartments and when it is required to select a particular bead colour, the relevant solenoid is activated. An opto-electronic sensor is situated in the bead path and signals from this are used to determine whether or not a bead has actually been released from the compartment or not. If one of the bead sources is empty, or if a bead is not dispensed when requested, a LED lamp can be activated.

The sequence of coloured beads (e.g. 1 black, 1 yellow, etc. or 2 black, 1 yellow, 2 black etc) that has been selected will finally be shown in the track at the bottom of the unit. At this point the unit is turned over and the beads run back to a colour sensor. According to the colour detected, the program should activate the sorting solenoid. This is repeated until all the beads are back in the correct compartment. At this point the bead sensing opto-electronic device will indicate that no further beads are passing and the LED lamp can again be activated. The process is shown in Figure 2.1. Here, the sequencing mode of the process is shown. The sorting mode is obtained by rotating it 180 degrees.

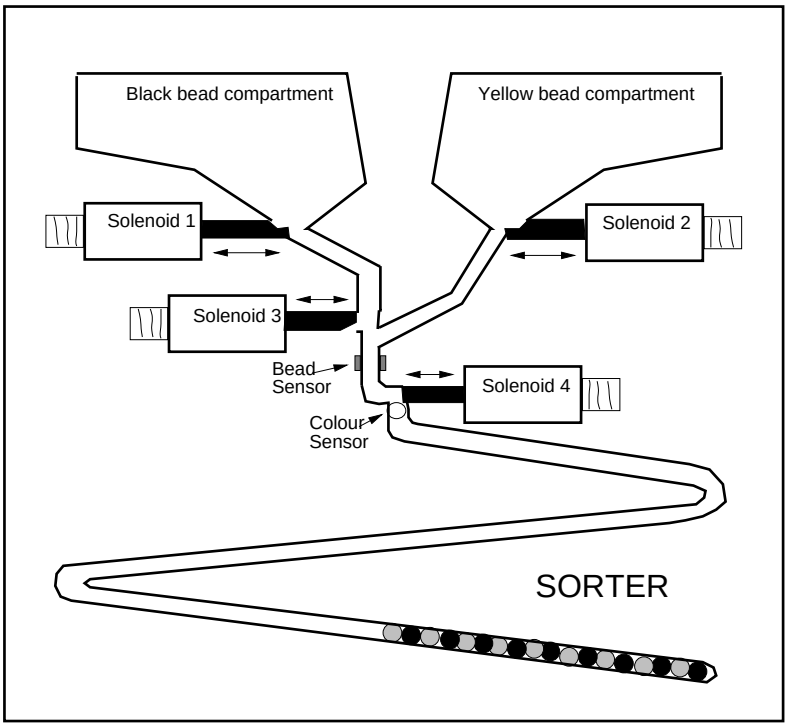


Figure 2.1 Bead sorter unit. The solenoids are in their initial (open) positions.

Template Grafchart

A Grafchart is available with inputs, outputs and parameters predefined, see Figure 2.2. A template of the sequences including the logic for switching between sorting and sequencing mode is also in the Grafchart. The scan cycle of the Grafchart is 10 ms.

The following output variables are available:

Name	Channel	Type	Description
`Sol1-Sol4`	30-33	digital	outputs with inverse logic, open (true), close (false) the solenoids
`LED`	37	digital	output used to indicate the completion of modes
`ResetBead`	35	digital	output used to reset (initialized to 0) the value in the analog bead sensor
`ResetColour`	36	digital	output used to reset (initialized to 0) the value in the analog colour sensor

The following input variables are available:

Name	Channel	Type	Description
`Tilt`	30	digital	input that is true when the process is turned 180 degrees (uses an accelerometer)
`AnalogBead`	32	analog	input for the presence of a bead
`Col`	33	analog	input for the bead colour

Simple logic contained within the ColourLogic and BeadLogic macro steps convert the analog inputs to corresponding boolean variables `Colour` and `Bead`. When you write your programs you should use these boolean variables (virtual sensors) in the same way as if they had been real sensors.

The bead and colour sensors must be reset each time they have detected a bead. This is done by sending a short pulse to the reset signal. It is important that the reset signal is TRUE for at least one cycle, and then FALSE for a sufficiently large time before the sensor values are read. Suitable initial values for this are available in the integer variables `SortReleaseTime` and `SeqReleaseTime`. In your program you should use these variables rather than numerical values directly.

Additionally the following variables are also defined and should be used as parameters: `SortWaitTime`, `SeqWaitTime`, `NbrBlack` and `NbrYellow`.

The sorting and sequencing algorithms should be entered in the corresponding macro steps, see Figure 2.2.

Name	Type	Description
`SortReleaseTime`	integer	number of scan cycles during which the sorting solenoid should be open (Sol3, Sol4)
`SeqReleaseTime`	integer	number of scan cycles during which a sequencing solenoid should be open (Sol1, Sol2)
`SortWaitTime`	integer	number of scan cycles to wait for the bead to pass solenoid 3
`SeqWaitTime`	integer	number of scan cycles to wait for the bead sensor to detect a bead
`NbrBlack`	integer	number of black beads in the sequence pattern
`NbrYellow`	integer	number of yellow beads in the sequence pattern
`Sort`	boolean	variable that shows activation of the sort mode
`Seq`	boolean	variable that shows activation of the sequence mode

Preparatory exercise 1: *Draw the Grafcet sequence for the sequencing mode of the bead sorting unit. Let the sequence pattern be 1 black, 1 yellow, 1 black, etc.. To release a bead from a compartment you activate solenoid 1 or 2 for a short time. The sequence should describe the following sequencing algorithm:*

1. *Reset the sensors as described above.*
2. *Release a bead from compartment 1 and wait long enough for the bead to pass the bead sensor.*
3. *If a bead is detected, go to step 4, otherwise repeat steps 1 and 2. If no bead has been detected after 5 attempts, go to step 7.*
4. *Reset the sensors.*
5. *Release a bead from compartment 2 and wait long enough for the bead to pass the bead sensor.*
6. *If a bead is detected, go to step 1, otherwise repeat steps 4 and 5. If no bead has been detected after 5 attempts, go to step 7.*
7. *At least one of the compartments is now empty, and the sequencing algorithm may be finished.*

Preparatory exercise 2: *Draw the Grafcet sequence for the sorting mode of the bead sorting unit. Use the following sorting algorithm:*

1. *Reset the sensors.*

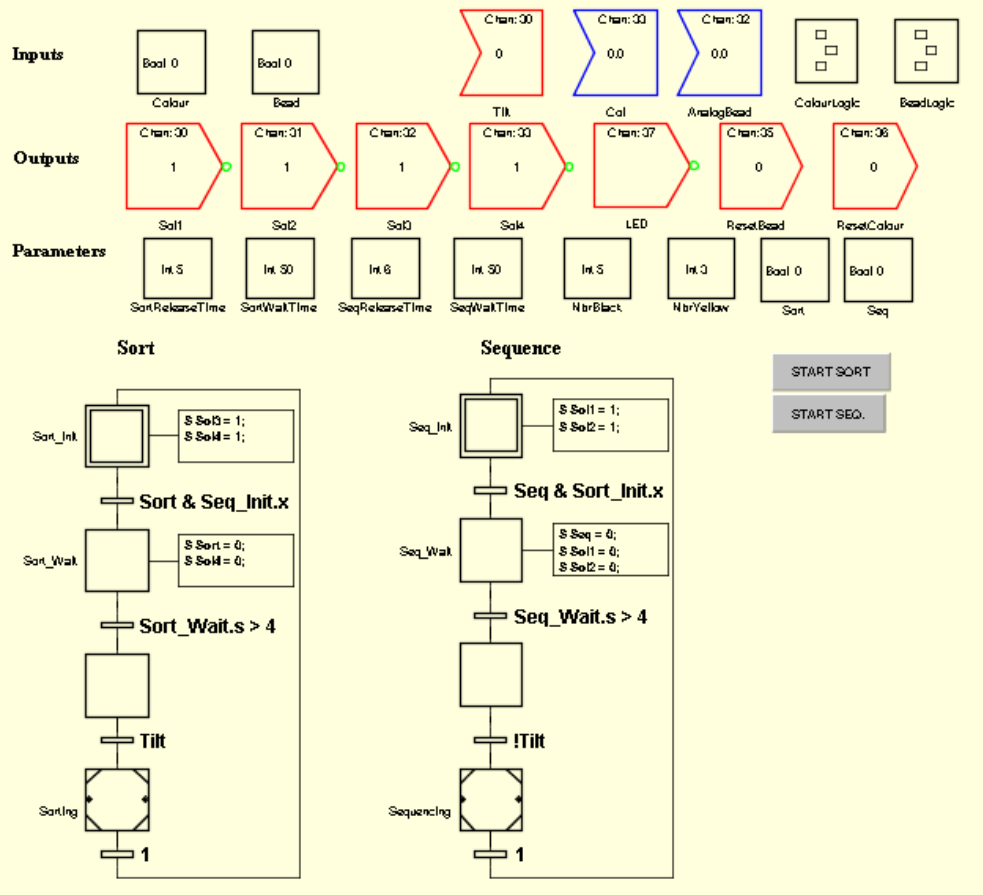


Figure 2.2 The template Grafchart.

2. *Determine the colour of the bead above solenoid 4, and let this decide the position of solenoid 3.*
3. *Release the bead by opening solenoid 4 for a short time.*
4. *Wait long enough for the bead to pass solenoid 3.*
5. *As long as there are still beads above solenoid 4, repeat steps 1-4. If no bead has been released during the 3 last cycles, go to step 6.*
6. *The sorting algorithm is finished.*

Exercise 1: Enter the two main sequences inside the macro steps Sorting and Sequencing respectively, and ensure that they work properly. The JGrafchart program should be started after you have turned on the bead sorter process.

Exercise 2: Modify step 7 in the sequencing algorithm above so that all remaining beads are released from compartment 1 or 2 after the sequencing has completed.

Lamp alarm

The sorter unit is equipped with a LED lamp that can be used to signal that a mode has run to completion.

Exercise 3: Add the lamp alarm function to your sequences.

Exercise 4: Modify the sequence in order to allow other parametrized sequences (e.g. n black beads, m yellow beads etc).