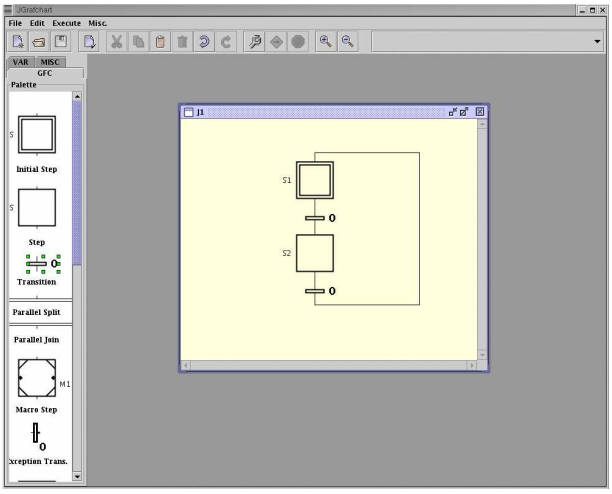
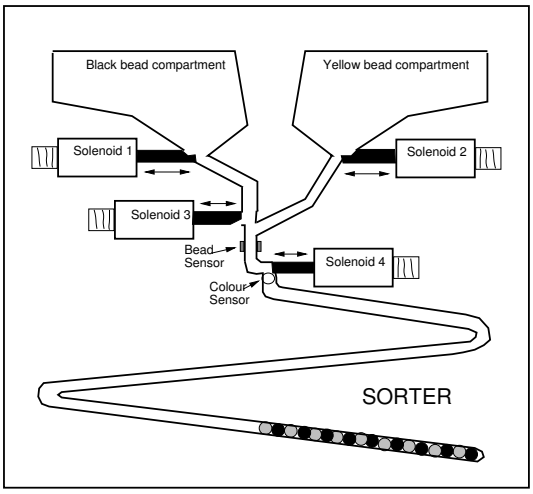
Real-Time Systems

Laboratory 2: Sequence Control





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

Department of Automatic Control
Lund University
July 2014

Introduction

Preparatory exercise 0:

Read through the laboratory manual carefully.

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/Grafchart 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.

Note:

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

Chapter 1

JGrafchart

JGrafchart is a Grafchart integrated development environment developed at the Department of Automatic Control, Lund University. It is written in Java using Swing graphics and JGo, a class package for graphical object editors from Northwoods Corporation.

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 application regularly.

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

Note: The JGrafchart version in the lab is not always the latest. If you download and do the preparations in JGrafchart somewhere else and then cannot open it in the lab, simply download and use the latest version in the lab too.

1.1 Workspaces

Grafchart applications are created interactively using drag-and-drop from a palette (containing the different Grafchart language elements) to workspaces, see Fig.1.1.

Workspaces support scroll, pan, and zoom. It is possible to change their size, to iconize them, etc. The application name can be changed from *Properties* in the *Edit* menu.

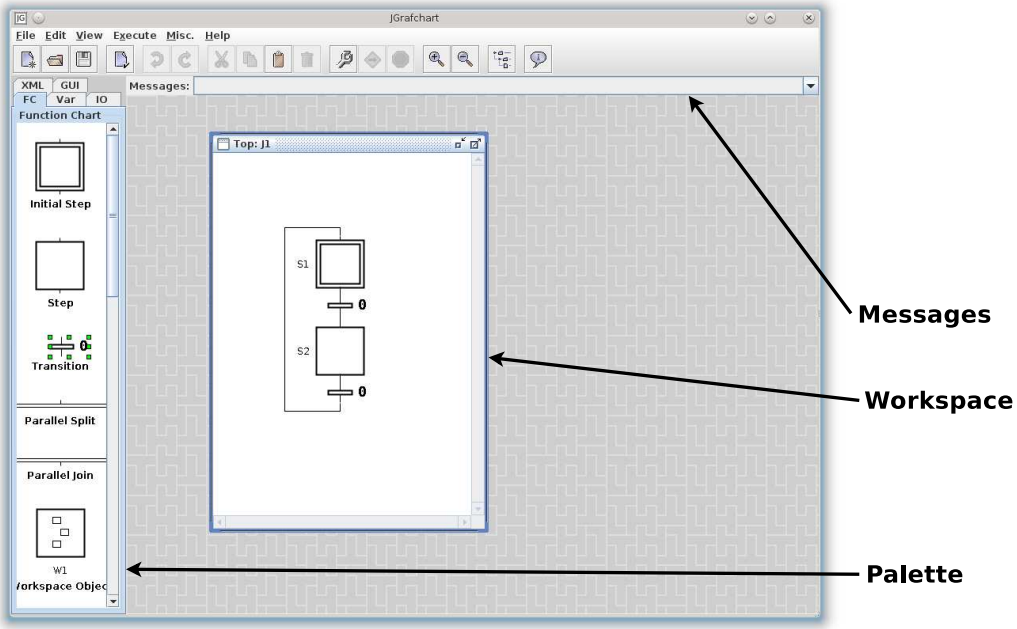


Figure 1.1: JGrafchart main interface

Grafchart objects are connected together graphically by click-dragging the connection stubs.

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

1.2 Grafchart 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).

1.2.1 Steps

Grafchart steps have action blocks that may be shown or hidden through the step's context menu (right-click menu), see Fig. 1.2.

Step actions are entered as text strings, either directly by clicking on the text string in the action block or through *Edit* in the step's context menu. Multiple step actions are separated by semi-colons.

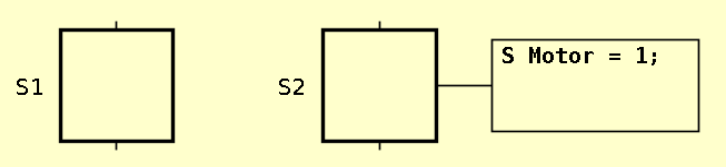


Figure 1.2: Step with action block hidden (S1) and visible (S2).

Four different action types are supported. Stored actions (impulse actions / enter actions) are executed once, immediately when the step is activated. The syntax for stored actions is:
`S <variable> = <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> = <expression>;`
Note: The default scan cycle time is 40 ms.

Exit actions (finally actions) are executed once, immediately before the step is deactivated. The syntax for exit actions is:
`X <variable> = <expression>;`

Normal actions (level actions) associate the truth value of a Boolean variable with the activation status of the step. The syntax for normal actions is:
`N <BooleanVariable>;`

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 value false (true) and the integer value 0 (1). The context decides the interpretation.

The supported operators 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 internal variables. JGrafchart uses lexical scoping based on workspaces. For example, a variable named `Y` on workspace `W1` is different from a variable named `Y` on workspace `W2`. References between workspaces are expressed using dot-notation. For example, an 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. To give a step a name use *Set Name* in step's context menu. 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.

1.2.2 Initial steps

Initial steps are steps that are activated when the execution of the application starts.

1.2.3 Transitions

Transitions represent conditions or events that should be true in order for the application to change state. The condition is an expression that is evaluated as a Boolean value, see Fig. 1.3. A step can be connected to multiple transitions, thus creating alternative paths. However, in order for this to work correctly the conditions must be mutually exclusive. Multiple transitions can also be connected to the same step.

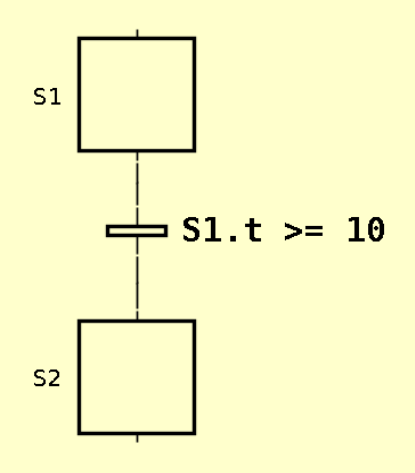


Figure 1.3: Transition

The condition is edited either by clicking on it or with *Edit* in the transition's context menu. The condition expression syntax is the same as for step actions.

The expression <step>.x is 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.

1.2.4 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.

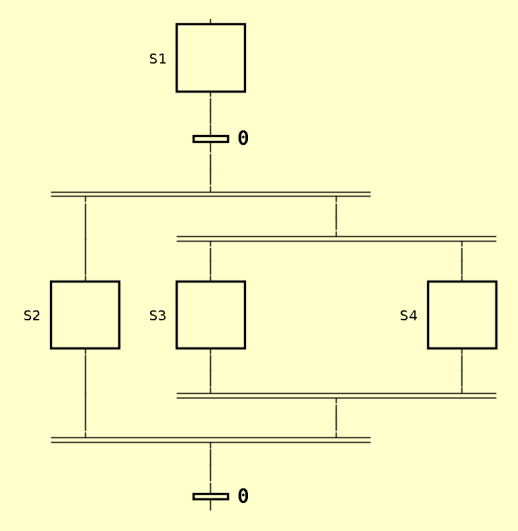


Figure 1.4: Parallel branching with three branches.

1.2.5 Macro Steps

A macro step represents a hierarchical abstraction. The macro step contains an internal (sub-)workspace which can be shown and hidden in the macro step's context menu. 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. 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-workspace of the macro step M1 may itself contain a macro step named M1, without causing any ambiguities.

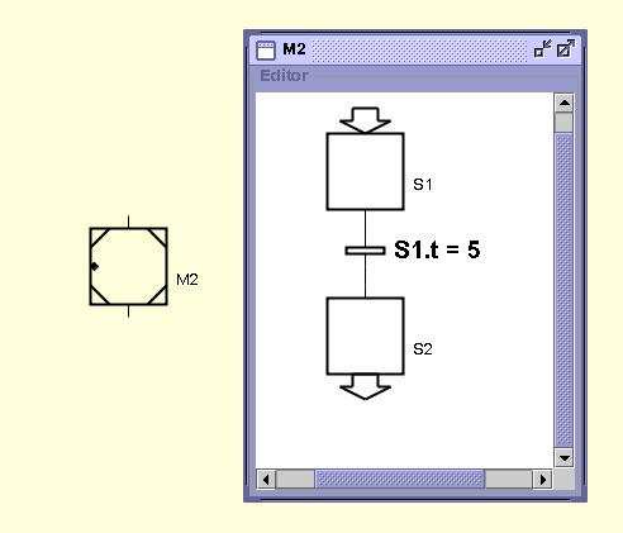


Figure 1.5: Macro step M1 and its workspace which contains the enter step S1 and the exit step S2.

1.2.6 Digital Inputs and Outputs

Digital inputs are Boolean variables that can be read by the application's steps and transitions. Similarly, digital outputs represent Boolean variables that can be written to by the application's step actions. Each input and output has an associated value (0 or 1), a name, and a channel name, 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 inputs and outputs exist, one with ordinary logic (initial value 0) and one with inverted logic (initial value 1).

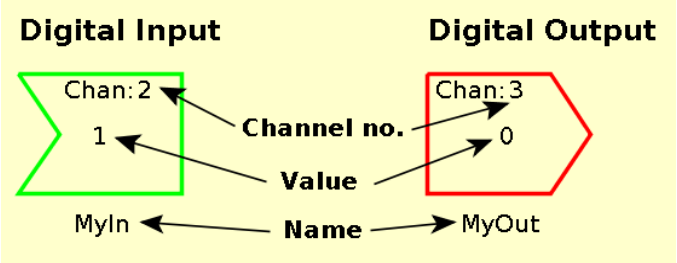


Figure 1.6: Digital input and output.

1.2.7 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.

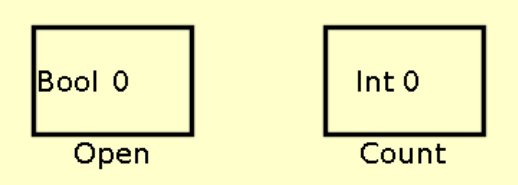


Figure 1.7: Boolean variable (left) and integer variable (right).

By default, a variable retains its value when the application is stopped and later restarted. Optionally, it may have an initial value. Then 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 properly.

1.3 Execution

Grafchart applications are executed by one periodic thread each. The threads cyclically perform the following:

1. Read digital and analog inputs.
2. Evaluate and fire transitions. Steps are activated and deactivated (executing S and X actions).
3. Execute P actions.
4. Update variables subject to N actions.

An application 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 I/O and outputs are written to the I/O. This mode only works when executing on the machines in the lab rooms of our department. Execution

mode and scan cycle time are changed with *Properties* in the *Edit* menu. Note that changing the scan cycle time also affects wait intervals using <step>.t, since it counts the number of scan cycles.

Before an application 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 list just below the toolbar.

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

Two types of problems may arise during compilation: syntax errors and semantic errors. For example, the transition condition `(y OR z)` has a syntax error (OR is a variable and is not allowed there). The syntactically correct expression is `(y | z)` (`|` is the OR operator). An example of a semantic error is a name lookup failure, for example if there does not exist any variables named `y` or `z` in the previous example. Transition conditions and step action blocks which contain errors are highlighted red.

The syntax for transition conditions 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 Grafchart object. The expressions are evaluated on-line, again by traversing the syntax tree. For example, assume that a transition contains the transition condition `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.

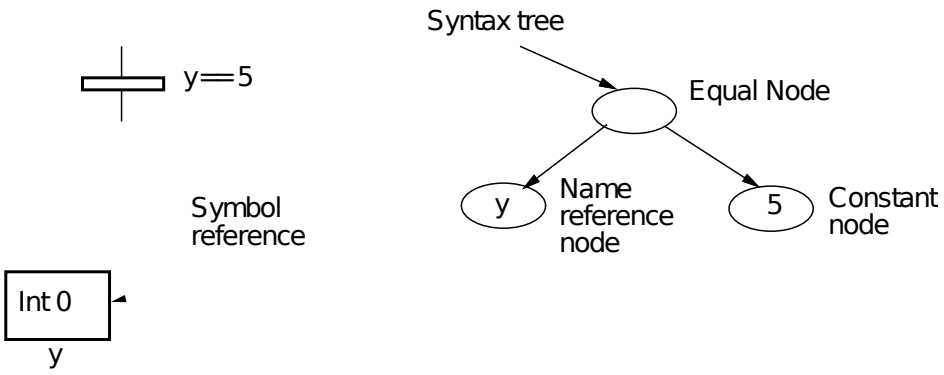


Figure 1.8: Syntax tree for the expression y == 5.

Chapter 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 colors (black and yellow) are used to represent the components. The beads of each color are provided from separate compartments. When the process should select a particular bead color 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. When the compartments are empty, a LED can be activated.

The sequence pattern of colored beads (e.g. 1 black, 1 yellow, 1 black, 1 yellow, ... or 2 black, 1 yellow, 2 black, 1 yellow, ...) 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 color sensor. According to the color detected, the application 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.

2.1 Template Application

A template is available with inputs, outputs, and parameters predefined, see Figure 2.2. Logic for switching between sorting and sequencing mode is also included. The scan cycle time is 10 ms.

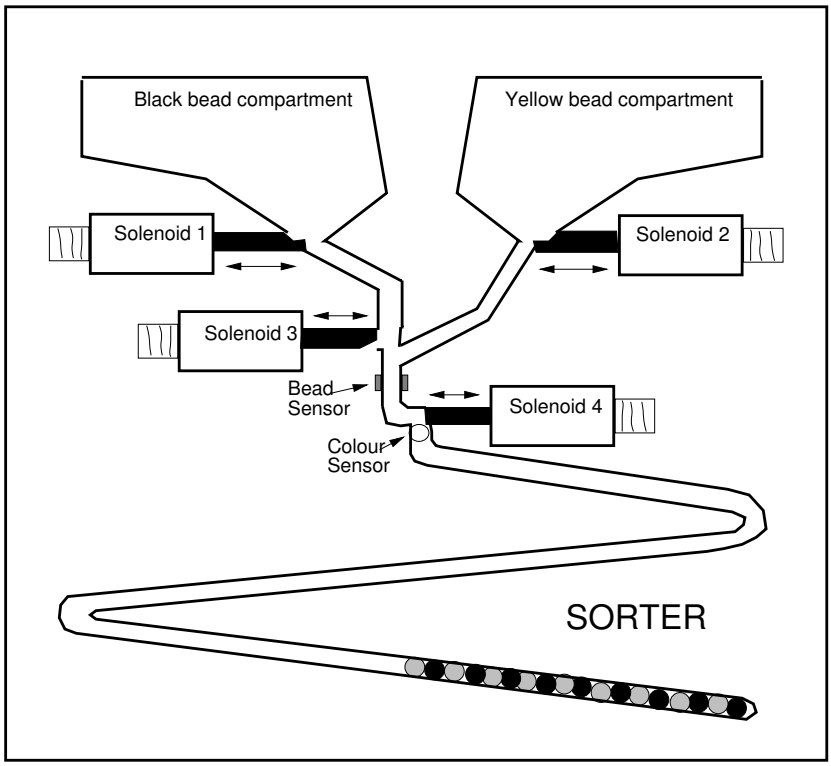


Figure 2.1: Bead sorter unit. The solenoids are in their initial (open) positions.

The following output variables are available:

Name	**Channel**	**Type**	**Description**
`Sol1-Sol4`	30-33	digital	the solenoids, open (true), close (false)
`LED`	37	digital	output used to indicate completion
`ResetBead`	35	digital	output used to reset (set to 0) the value in the analog bead sensor
`ResetColor`	36	digital	output used to reset (set to 0) the value in the analog color sensor

The following input variables are available:

Name	Channel	Type	Description
`Tilt`	30	digital	input that is true when the process is upside down, that is, in sorting mode (uses an accelerometer)
`AnalogBead`	32	analog	input for the presence of a bead
`Col`	33	analog	input for the bead color

Simple logic contained within the ColorLogic and BeadLogic macro steps convert the analog inputs to corresponding Boolean variables `Color` (true = yellow, false = black) and `Bead`. When you write your applications you should use these Boolean variables (virtual sensors) in the same way as if they had been real sensors.

The bead and color sensors must be reset after each 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 virtual sensor values are read.

There are also some predefined integer that you should use in your application to avoid hard coding the timing, see Table 2.1.

Name	Description
`SortReleaseTime`	number of scan cycles during which the sorting solenoid should be open (Sol4)
`SeqReleaseTime`	number of scan cycles during which a sequencing solenoid should be open (Sol1, Sol2)
`SortWaitTime`	number of scan cycles to wait for the bead to pass solenoid 3
`SeqWaitTime`	number of scan cycles to wait for the bead sensor to detect a bead
`NbrBlack`	number of black beads in the sequence pattern
`NbrYellow`	number of yellow beads in the sequence pattern

Table 2.1: Predefined integer variables.

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

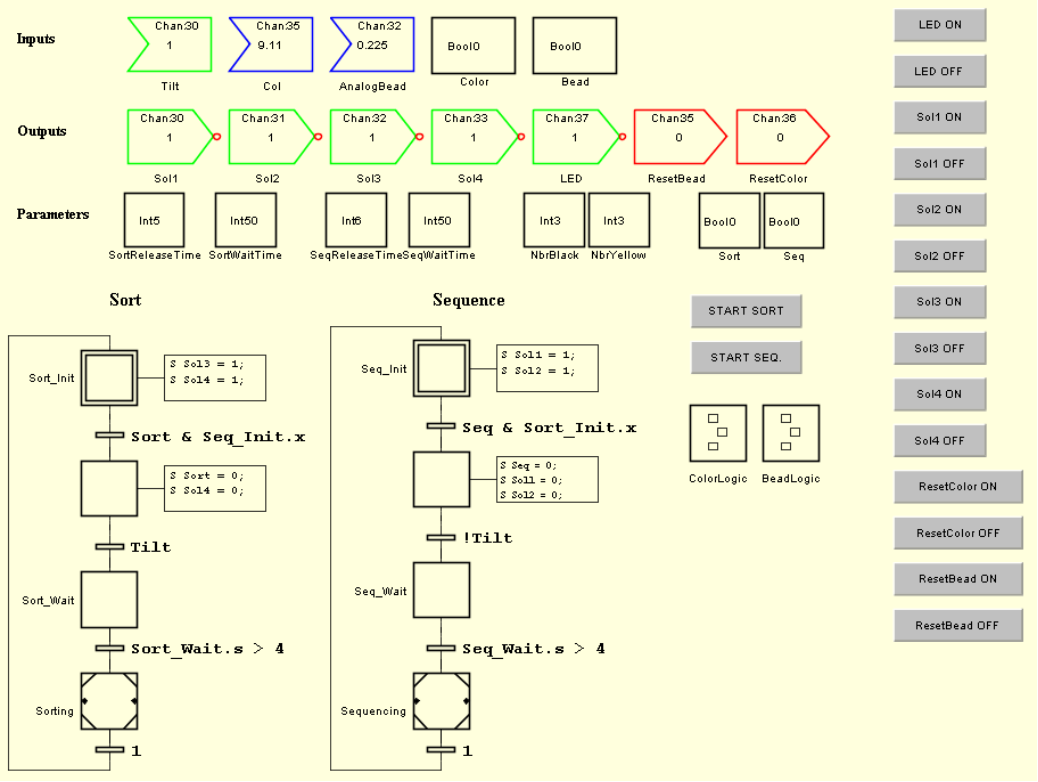


Figure 2.2: The template application.

2.2 Basic Functionality

Preparatory exercise 1:

Implement the Grafcet sequence for the sequencing mode of the bead sorting unit using JGrafchart. 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.

Implement your sequence inside the block named ***Sequencing***

The sequence should describe the following sequencing algorithm:

1. *Reset the bead sensor.* `Resetbead =1`
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 bead sensor.*
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:

Implement the Grafcet sequence for the sorting mode of the bead sorting unit.

Implement your sequence inside the block named ***Sorting***

Use the following sorting algorithm:

1. *Reset the sensors.*
2. *Determine the color 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.*

Preparatory exercise 3 (recommended):

During the lab you will share the real process with some other students, and to make it go as smooth as possible we recommend you verify your controller from preparatory exercise 1 and 2 against the simulated process in `BeadSorterSim.xml`.

The simulator is run by loading the provided function chart from `BeadSorterSim.xml`, *building it and running it. You also need to set your template application to simulation mode as well as bulding and running it.*

The simulator has an arrow displaying the current direction of gravity, so beads will travel in the direction of the arrow.

To make the sorting and sequencing go smooth, make sure to start the application before flipping the gravity (done by double changing the value of the `Tilt`*-input). If beads start behaving strange you might want to reset the simulation.*

Exercise 1:

Make sure that JGrafchart is started after you have turned on the bead sorter process. Enter the two main sequences inside the macro steps Sorting and Sequencing respectively, and ensure that they work properly.

Note: You might need to tweak the release and wait times. Due to problems with the process, the solenoids may end up in the wrong place if you try to move them too fast.

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.

2.3 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.

2.4 Custom Patterns

Exercise 4:

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