CODEN:LUTEDX/(TEIE-5342)/1-46/(2014)

Industrial Electrical Engineering and Automation

Web Browser Based Ladder-Editor

Fredrik Bengtsson och Ivan Flink

Division of Industrial Electrical Engineering and Automation
Faculty of Engineering, Lund University

Web Browser Based Ladder-Editor

MSC Project in Industrial Electrical Engineering and Automation

Performed at B&R-Automation

Fredrik Bengtsson
Ivan Flink

Supervisor LTH:
Gunnar Lindstedt

Supervisor B&R-Automation:
Patric Thysell

Examinator:
Ulf Jeppsson

Abstract

The goal for this master thesis is the creation of a graphical ladder-editor in JavaScript and HTML5. The editor is made in such a manner that the user inserts blocks by selecting them from a menu. Then the block is inserted simply by them clicking where the block should be inserted. The editor itself should handle the required lines. The editor also contains a menu with custom made buttons which allows the user to easily browse through the different possibilities and options. Most common ladder blocks are included in the editor, and it is easy to add more if desired. Furthermore, the editor described here contains the possibility for the user to delete the blocks added and it also contains undo and redo functionality. Menus have been designed and implemented that allows the user to change the attributes of the contacts, coils and function blocks that he or she has inserted. There is also the possibility to copy a block from one part of the system and insert it somewhere else. The system that the user edits graphically using the editor is described in code using the Json format. This in turn allows the program the user is working on to be saved and loaded. This is also implemented, in this case by establishing communication with B&Rs program Automation Studios which then handles the saving and loading from and to file.

What was done by whom

- General structuring, planning and discussions with B&R were done in unison.
- Experimenting and self study of the software (JavaScript and HTML5) was done by both, but individually.
- Research and testing to determine a good way to portray a ladder program in Json was done together.
- Drawing of the ladder structure on a canvas was done by both but individually.
- Button design and features was done by Ivan Flink
- Button implementation was done by Ivan Flink.
- The applications of the buttons (such as undo, redo, delete, rename etc) was done by Fredrik Bengtsson along with implementation of inserting components (both in "and" and in "or").
- Communication with Automation Studios was done by Fredrik Bengtsson.

Preface

The goal of this master thesis is to build a Ladder-Editor in HTML5 and JavaScript for BR-Automation in Malmo. The authors, Ivan Flink and Fredrik Bengtsson worked as a team with continuous contact with the BR-Automation manager Patric Thysell and LTH professors Gunnar Lindstedt and Ulf Jeppsson. The thesis topic was a suggestion from the company and they wanted this product for their own use.

Many of the tools that were used were completely new to the authors, and much work was the study of these tools from various resources. As much of the work could be done anywhere, the authors did much of the work at LTH since it was convenient for both of them. The original time plan was set from 1st of January to the 31st of May, with the option to continue in June or July.

Our expectation for this thesis were rather big. Our goal was to present the absolute best solution that we were capable of. There are a lot of different areas that we also explored during these months. HTML5 and JavaScript are two big programming languages and we hope that the skills developed here, will serve us in our future careers.

We would like to thank all the people who have helped us with our project, especially our examiner Ulf Jeppsson and our two supervisors, Gunnar Lindstedt and Patric Thysel for making this thesis possible.

Table of contents

1. Introduction

In automation various control systems is used for operating equipment such as machinery, processes in factories, boilers and heat treating ovens, switching in telephone networks, steering and stabilization of ships, aircraft and other applications with minimal or reduced human interaction. Some processes have been completely automated. The biggest benefit of automation is that it saves labor, however it is also used to save energy and materials and to improve quality, accuracy and precision.

One problem with most automated processes is that it is difficult to edit the software once your process is installed and running, for instance if a company would like to add a sensor to their factory. This would require hire a few skilled consultants - very expensive for the company.

The goal for this thesis is to make a software structure which allows people, with no automation skills, to manipulate and edit the program. The demand for simple editing has drastically increased over the last couple of years. For the ladder program, such changes would be to add, delete or change the various elements of the program. This was initially planned to be done through an online ladder-editor using drag and drop functions. However, during the course of the project it was determined that adding components by clicking was preferable. Basically the user should be able to select an element from a panel in the editor, and insert it anywhere in the ladder-editor by clicking where it should be placed and the code will re-write itself. In other words, changes can be made to the system without the user changing one line of code!

The main task is to program an editor in HTML5 [1]/JavaScript, with a very smooth and easy interface and appearance, and the user should not have to worry about format, changes in the code, errors and so on. The ladder-program will be processed by an interpreter that will work via Json communication with the ladder-editor. Our main focus is on the ladder-editor so we assume that we have got a working interpreter. Hence, the thesis will only focus on the Json-structure.

2. Method

2.1 Programming languages

For the thesis project many different programming languages were utilized or worked with.

2.1 .1 Ladder programming

The software behind automation systems is often ladder programming. This is what the editor was built to program with. Ladder programming is rather easy to understand once the most important parts are clear. A very basic ladder-program can be seen in figure 1.

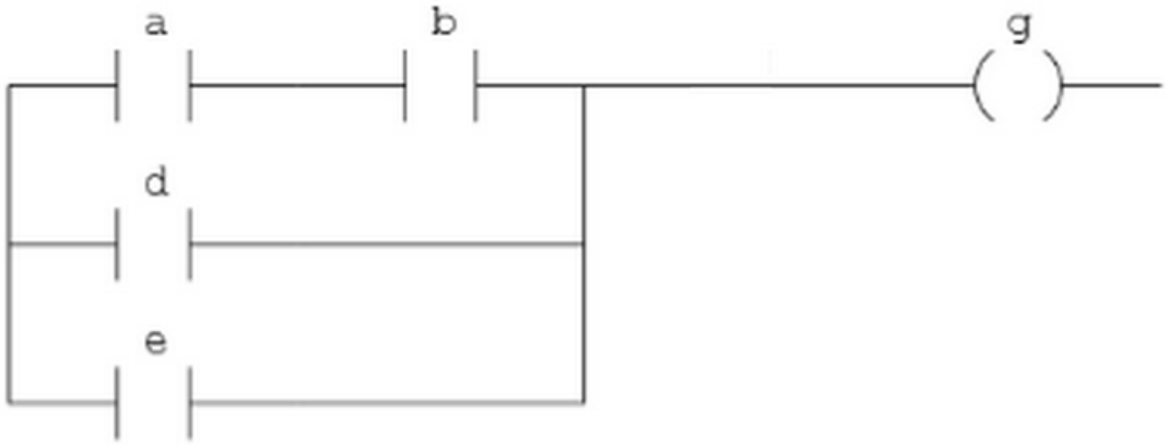

Figure 1 Simpel ladder-program

The schematic contains contacts and coils. They are all Boolean values, meaning they can either be true or false. Each contact might represent a sensor in a system and they are true if something is blocking the sensor, otherwise a contact will have the value false. The coil, might represent that whole system, and when it is true the system is working, and when it is false the system might be paused. With this in mind, one can easy see that here are 3 combinations that will result in a working machine. The first combination is seen in figure 2, where contacts “a” and “b” are both true.

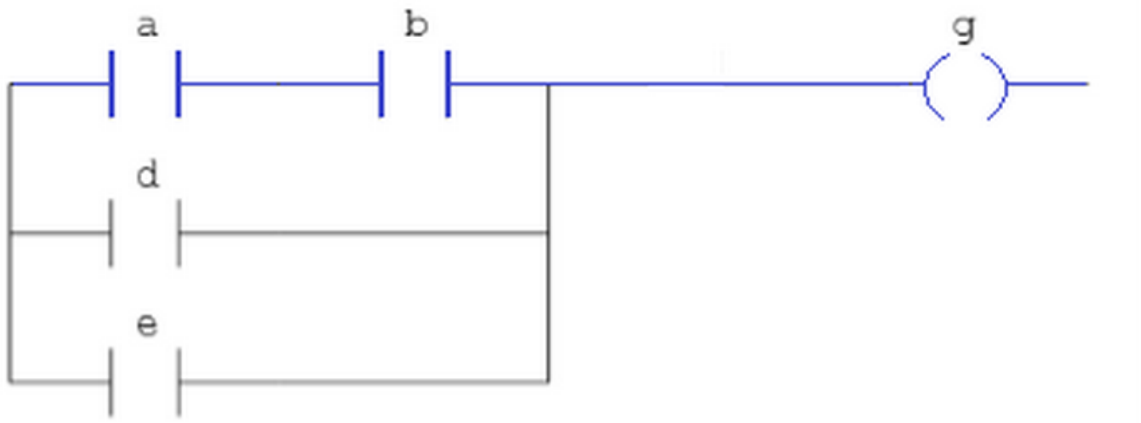

Figure 2 Example 1

The second and third way, for the coil (system) to be true is if sensor d or sensor e are blocked , as seen in figure 3 and figure 4.

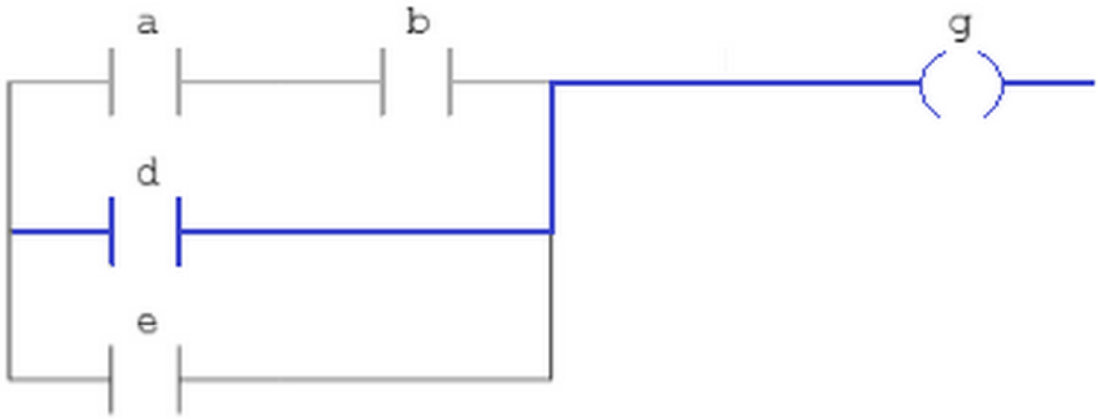

Figure 3 Example 2

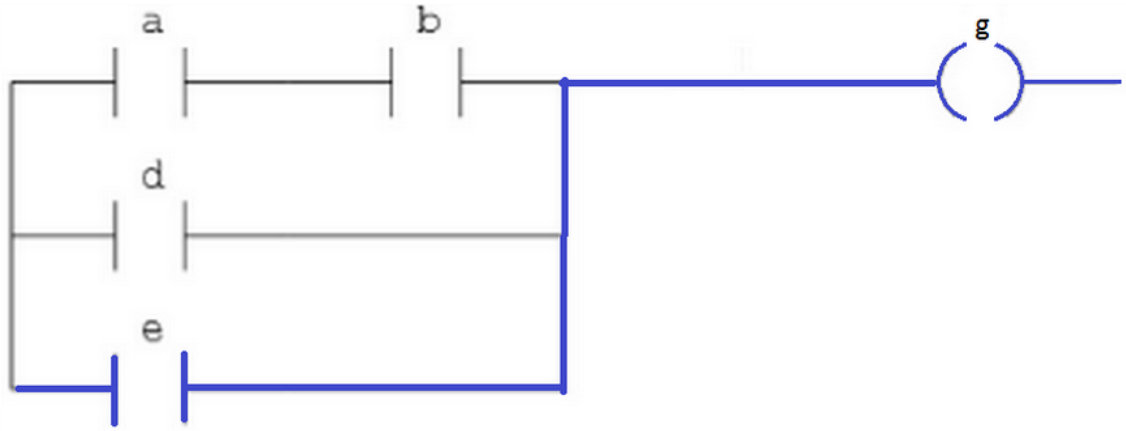

Figure 4 Example 3

In ladder logic, one can also choose elements such as an inverted contact, as seen in figure 5 and an inverted coil. As one can hear from the name, they work quite the opposite of a normal contact and coil. If a contact needs to be true in order to set the coil to true, the inverted coil needs to be false.

Figure 5 Inverted contact

There are other frequently used components that are worth mentioning, such as the addition-block seen in figure 6. The addition-block is in the operator-category and here we also find subtraction, multiplication and division -blocks. These blocks can have more than one input. One might for example add the value of four variables and return it as an output. The block described will in that case have four inputs and one output.

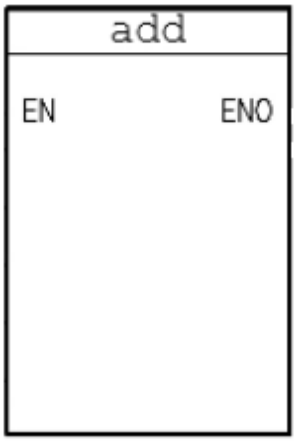

Figure 6 A add-block

One might want to compare variables, and therefore blocks such as greater than (GT) and less than (LT) are also included in the ladder-logic. These types of elements have two inputs and one output.

2.1.2 JavaScript

Most of the programming done for the editor was done using JavaScript programming language. This language is the most common language used for programming web pages.

As a programming language JavaScript is quite similar to Java. It contains much of the same structures and formatting. Moreover, like Java it automatically handles memory allocation and references.

One way JavaScript differs from most other programming languages is, however, its lack of type definitions. In other programming languages the programmer must clearly define the variables type (integer, strings vectors etc). However, in JavaScript this is not necessary. Moreover, a variable that was first assigned as an integer can then be linked to a string or some other data type. This simplifies programming and removes the need for wrapper classes.

As a programming language used mainly for websites, JavaScript is closely linked to HTML5. JavaScript code can modify HTML5 objects such as buttons or canvases. Furthermore, HTML5 buttons can be linked to JavaScript code, so the code is executed when the button is clicked.

2.1.3 HTML5

HTML5 is a core technology language of the Internet used for structuring and presenting content for the World Wide Web. It is the fifth revision of the HTML standard, as one might notice from the name. Since HTML is used to program websites, the language has always had a lot of design features. For instance there are thousands of colors to choose from and hundreds of ways to design and present your text-content. In recent years, the demand for displaying videos and high quality audio on your website has drastically increased. HTML5 was created to support this demand. HTML5 also supports the canvas-object which will be used frequently during this thesis project. More about that further on.

2.1.4 Json structure

For this section [7] [8] [9] have been frequently used.

The Json-structure is rather easy to understand. In JavaScript an object is simply given some attributes like name, type and in some cases content. A coil with the name demo would simply be represented with this object:

obj = { name:”demo” , type:”coil” };

Now it is important to clearly notice the difference in obj and demo which are both names. In the future code we will refer to obj if we want to use this coil. However when we draw the ladder program we want to use the name of the object, and we can simply call this attribute by writing “obj.name” , and we will get “demo”. In the same way obj.type will give “coil”. There are a lot of other elements that are frequently being used, such as the contact and add-element (these can be seen in figure 7):

obj1 = { name:”demo2” , type:”contact” };

obj2 = { name:”demo3” , type:”add” };

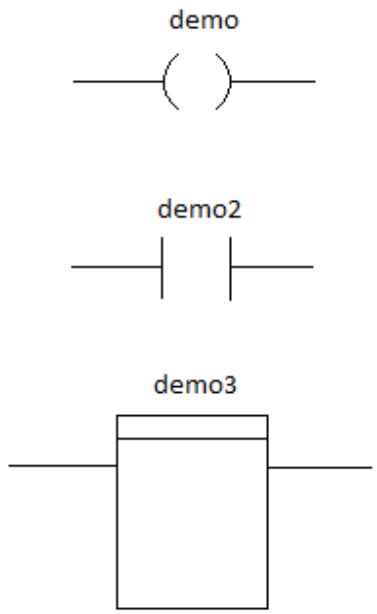

Figure 7 Three common elements

There are two very important types of objects that are not elements themselves, like the example above, but instead they describe the structure of the ladder-program and this is objects of type “or”/“and” (these can be seen in figure 8). An object of this kind has no attribute name, instead these objects have a vector called content, which stores other objects. The code for producing two easy circuits would be:

figure 8 The upper circuit

```
obj2= {type:”contact” , name:”demo3”};

obj3= {type:”contact” , name:”demo4”};

obj1 = {type:”or” , content:[obj2,obj3]} ;
```

figure 8 The lower circuit

```
obj2= {type:”contact” , name:”demo5”};

obj3= {type:”contact” , name:”demo6”};

obj1 = {type:”and” , content:[obj2,obj3]};
```

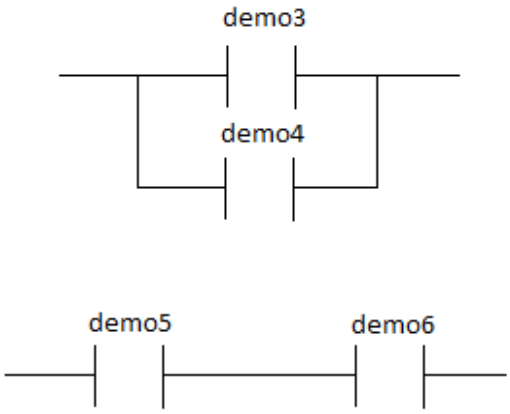

Figure 8 The ”And” / ”Or” structure

The content-vector can of course be of any size and does not only have to include two elements as the example above. The whole ladder-program is often described as one big or-object and in that case the content-vector could include 20,30,40 or more objects, depending on the total size of the ladder-program.

It is important to define objects in the right order to avoid errors. An object must be defined before used in another object. For example the code below would generate an error, since we use obj2 and obj3 in the first row, when they are not yet defined.

```
obj1 = {type:”and” , content:[obj2,obj3]};

obj2= {type:”contact” , name:”demo5”};

obj3= {type:”contact” , name:”demo6”};
```

Build…

ERROR!

This might look obvious in this short piece of code, but when you are dealing with 50 objects, it is rather easy to make these kinds of mistakes.

The reason the Json object structure was used, is that a Json object can easily be converted to a string, and from this string back to a Json object. This makes it easy for one to save the Json object as a string, so it can be sent to another program. This is something that B&R desired, and something that was used later in the project.

2.2 Format of a ladder-program

2.2.1 Introduction and the first solution

Of central importance to the implementation of the ladder editor was how a ladder program can be represented and saved in code. This is vital as the ladder program needs to be encoded so that it can be sent to and from the PLC. The structure therefore must be something that can be transferred using Json. It was decided that the ladder program should be represented using JavaScript objects, which makes them easy to transfer using Json. They also have the advantage that they are the preferable way of representing the ladder program for the editor, hence the same system can be both used to produce and edit the ladder program in the editor, as well as to transfer it to the PLC.

How to best represent the ladder program using JavaScript was an important question for this thesis project. The solution decided upon needed to be easily transferable using Json. It also needed to be easy to change the structure as to enable users to change their ladder program (as this structure is the representation of their ladder program).

To demonstrate the alternative solution, a sample ladder program is represented below in figure 9.

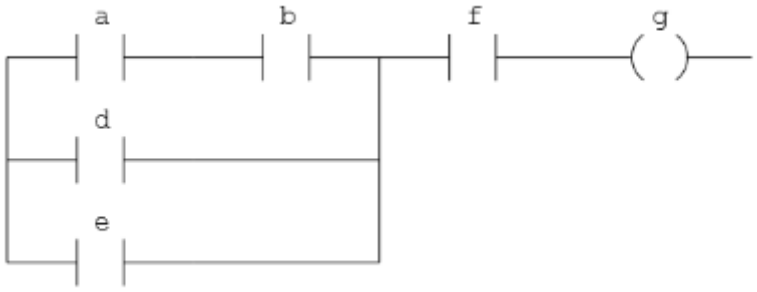

Figure 9 Example structure

Our initial approach was to use a method to represent it based on its graphical appearance, representing each row as an object, containing information about connections to the other rows, so our representation for the top row would be:

```
obj row1=[
"line":1 //this row is graphically placed on the uppermost row
so it has a number 1

// the components of the row are stored in a vector.
"components": [
{"type":"contact"
```

```
"value":"a"},
{"type":"contact"
"value":"b"},
{"type":"contact"
"value":"f"}
]
"start":"init" //the row is directly connected to the left
side.
"end": {"type" coil,   // the row ends with coil f.
"value":g,}
"inConnection":[{"name": "row2" // row2 is connected to row1
"outpos":1, // the connection is from after the first
component of row2.
"inpos":2,} // the connection is from after the second
component of row1.
{"name":"row3" // We also have a connection with the third row
in a similar structure.
"outpos":1,
"inpos":1,
}]
```

This structure, while being able to graphically represent the ladder program, had a major disadvantage. Namely that if a component were added, one had to go through all the different row objects, and update their connection vectors to represent this.

After consultation with B&R Automation, it was made clear that it was not necessary that the program was graphically consistent. That is to say that it was acceptable if the program changed appearance, as long as the ladder logic was maintained.

2.2.2 A better approach

This allowed for another approach, instead using nested objects and logical operators (“and” and “or”) to properly portray the structure of the program. With this method the aforementioned ladder system would be described in code in the following way:

```
system={type:”and”, content:[obj1,obj2,obj3];
obj1={type:”or”, content:[obj4,obj5,obj6]};
obj2={type:”contact”,name:”f”};
obj3={type:”coil”,name:”g”};
obj4={type:”and”,content:[obj7,obj8]};
obj5={type:”contact”,name:”d”};
obj6={type:”contact”,name:”e”};
```

```
obj7={type:”contact”,name:”a”};
obj8={type:”contact”,name:”b”};
```

As previously stated, defining the system object before the objects in its content vector would result in errors. However, it is done here for the sake of clarity for the reader (in implementation the program will add new objects straight into the content vector [as opposed to the references to objects used here] resulting in a single very large object, so this will not be a issue).

This new method has several advantages. Not only is the code shorter and easier to get an overview of, but more importantly it is relatively easy to edit the code to adapt to changes in the ladder program.

So if one wants to remove a contact, the program simply searches the system until it finds the array (an "and" or "or" structure) the contact is located in and deletes the contact from the array. The structure might then need to be modified, for instance if this causes there to be an “and” object containing only one component. This is fixed by removing the "and" object and replacing it with the component it contained. So for example in the described structure, if contact b was to be removed from the diagram (as can be seen in figure 10), the code would be changed into:

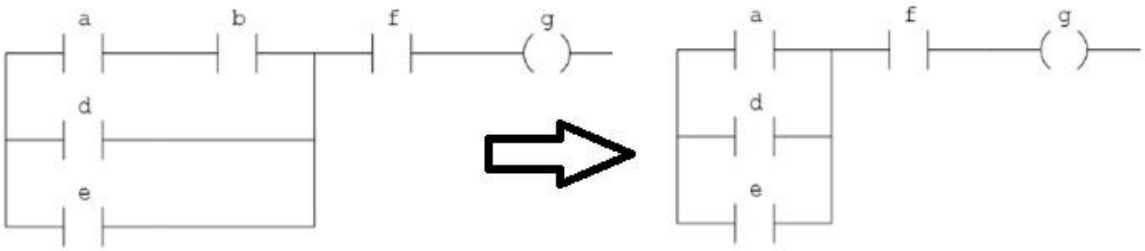

Figure 10 Contact ”b” has been removed

```
system={type:”and”, content:[obj1,obj2,obj3];
obj1={type:”or”, content:[~~obj4~~ obj7,obj5,obj6]};
obj2={type:”contact”,name:”f”};
obj3={type:”coil”,name:”g”};
~~obj4={type:”and”,content:[obj7,obj8]};~~
obj5={type:”contact”,name:”d”};
obj6={type:”contact”,name:”e”};
obj7={type:”contact”,name:”a”};
~~obj8={type:”contact”,name:”b”};~~
```

Similarly if one wants to add a contact in “and” with another one then one simply creates an “and” object, then places the two objects in it, and put this new object in the array were the previous object was. So if one was to add a new contact c after contact d (see figure 11), the code would become:

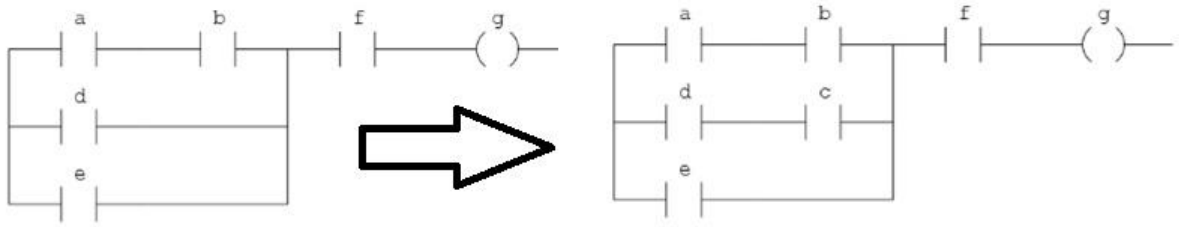

Figure 11 Contact ”c” has been added

```
system={type:”and”, content:[obj1,obj2,obj3];
obj1={type:”or”, content:[obj4,~~obj5~~ obj10,obj6]};
obj2={type:”contact”,name:”f”};
obj3={type:”coil”,name:”g”};
obj4={type:”and”,content:[obj7,obj8]};
obj5={type:”contact”,name:”d”};
obj6={type:”contact”,name:”e”};
obj7={type:”contact”,name:”a”};
obj8={type:”contact”,name:”b”};
obj9={type:”contact”,name:”c”};
obj10={type:”and”,content:[obj5,obj9]};
```

It is not always necessary to create a new “and” object, if the object that a new contact is to be attached to is already in an “and” object, the new object will directly be added to this “and” object.

To add a component in “or” with the component the method is as follows. One creates a new “or” object. This “or” object contains the component one wishes to add in its content vector, as well as the components one wishes to place the component over (if this is more than one component, they are placed in a newly created “and” object). So with the system used previously, if one would wish to place the contact c in “or” with the contact a (see figure 12), the code would be rewritten as:

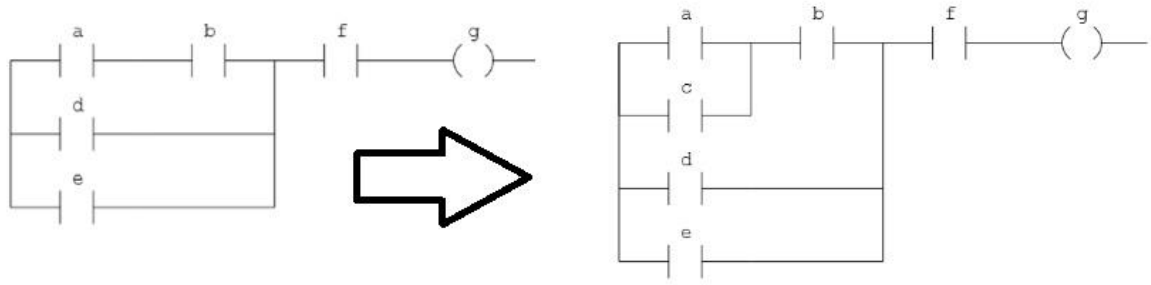

Figure 12 Contact ”c” has been added

```
system={type:”and”, content:[obj1,obj2,obj3];
obj1={type:”or”, content:[obj4,obj5,obj6]};
obj2={type:”contact”,name:”f”};
obj3={type:”coil”,name:”g”};
obj4={type:”and”,content:[~~obj7~~obj10,obj8]};
obj5={type:”contact”,name:”d”};
obj6={type:”contact”,name:”e”};
obj7={type:”contact”,name:”a”};
obj8={type:”contact”,name:”b”};
obj9={type:”contact”,name:”c”};
obj10={type:”or”,content:[obj7,obj9]};
```

2.3 Drawing the ladder-structure

An important step in implementing the ladder editor was to show the ladder diagram graphically. This needed to be done so the user could see the current state of the ladder diagram he or she was working on, so he or she would be able to edit it as appropriate. The drawing was done using the HTML5 canvas class [7].

The whole ladder structure is drawn with one “pencil”, which will be referred to as pointer from now on. The pointer is given start-coordinates that places the pointer somewhere on the screen, and those are the coordinates of the ladders top left corner. Every time something is being drawn on the screen, the pointer coordinates are being changed, and this is a big challenge because one does not want any gaps when drawing the ladder-program, but one does not want to draw the same line twice either, since this will result in a thicker line which will look unpleasant, compared to other lines.

As mentioned before, the key is to think recursively when drawing a ladder-program. The editor has a function called drawer, which is called repeatedly with different input-arguments. The argument is always an object, and depending on what type of

object it is, the drawer will respond differently. The first time the function is being called, the argument is an object called system, which is a global variable, and represents the whole system (an “and” / “or” object depending on the structure of the ladder-program). The drawer function then has 5 important “if-statements”, each checking what type of object was sent to the function, and what to draw. The structure for drawing a ladder-program is not trivial but the example below is a simple explanation without any details.

If the **type** of the input object is **contact / coil / add** the system will simply draw a contact and move the pointer in the x-plane with exactly the same amount of pixels that the element required.

If the **type** of the input object is **“and“** the drawer will simply make recursive calls on the drawing function, starting with the first object in the content vector and then step by step looping through all the element until every object in the content vector is drawn.

If the **type** of the input object is **“or”** the process of drawing the object is more complex. First functions to determine the height and width of an object are needed. The “findlength()” function will go through the object. If it is a coil or contact object it will simply return the height of 1. If it is a function block it will return a 3 instead. In case of an and-object it will return the sum of the length of each of the components in the object’s content vector (calling the function recursively). In the case of an "or" object the function will return the length of the largest object in the content vector (once again calling the function recursively). The “findheight” function works similarly. The only difference is that in case of an “or" object it will return the sum of the heights of each of the objects in the content vector, while in case of an “and" object it will return the largest height of the objects in the content vector.

With these two functions, drawing the object in "or" can now be implemented. That is to say drawing components over or under each other can now be implemented. First one goes through the content vector of the “or" object one is to draw and finds the greatest length. This length is used to determine the horizontal length required to draw the “or" object. Then the program saves the initial x and y coordinates. Having done this the program then calls the drawer function for the first object in the content vector. It will draw this object, increasing the x-coordinate. The program compares the length of the object drawn to the greatest length that was previously determined. If this is less, then it will be extended with a line horizontally to compensate for this discrepancy. Having done this the program checks the height of the object drawn, increasing the y-coordinate as appropriate depending on this. Then the program resets the x-coordinate to the initial value, and then repeats the process for the next object in the content vector. Once this is done for each of the objects in the content vector, the program then draws lines from the initial y value to the final y value, both

with the initial and final x values. Then the program resets the y value to the initial value and sets the x value to the final x value.

2.4 Adding new elements

2.4.1 Drag and drop

In implementing drag and drop, [15] was used.

The first solution for the editor was that objects were to be added by dragging them from a menu to where they were to be placed in the ladder diagram. To do this, a function, which noted the graphical space each ladder component took in the canvas on which it was drawn, was noted. This means that by checking the mouse location when the mouse is clicked one could see if any of the existing components was clicked upon.

The graphical data was also used when implementing drag and drop. By clicking on a component in the menu it was noted that this component was selected. While moving the mouse over the canvas with the mouse button down, the system would draw the object at the mouse location, and then redraw the rest of the system. As this is done quickly it would simulate dragging the object across the screen. However, while this worked well for the Chrome and Internet Explorer browsers, it proved problematic for Firefox. The constant redrawing of the system caused the screen to flicker in Firefox. This was highly distracting and the code was changed so that instead of redrawing the system every time the mouse was detected moving across the screen, it was only redrawn every fifth time. This still simulated drag and drop and solved the flickering problem.

To add a component to the diagram, simply releasing the mouse where one desires the component to be would add it there. This would be done by using the graphical data of each component, as to detect next to which component the new component should be added in the code. After this is done the diagram is redrawn, and hence the new component appears in its position.

However, having done this and shown it to the supervisor at B&R Automation, it was concluded that the drag and drop function initially requested might be problematic if the program is run on an iPad. This was because no good way could be found to distinguish if the user wished to place the component in series with another component or in parallel when dragging. Hence it was determined that the editor should be implemented by clicking instead.

2.4.2 Insert by clicking

Throughout this part [14] was used.

To implement insert by clicking many of the same functions as used in drag and drop could be used. However, first a function was created that would draw two circles, one on each side, to each of the components, and the coordinates of these circles were saved. These circles would be what the user would click on to add new components. Circles in objects that were in the same row were moved on top of each other as multiple circles next to each other were unsightly and unnecessary. Having done this, it was easy to implement adding objects in “and” with other object. Simply select the type of component one wishes to add and click on the circle where the component should be inserted. The program will then rewrite the code describing the system to reflect this, as previously explained in section 2.3 using the commands for the JavaScript array class [10] [11].

When adding coils in “and” with other objects the algorithm needed to be adapted slightly. This is because there are only specific places where a coil should be allowed to be added according to ladder logic. Hence the program had to check that the coil added would be connected to the right end of the program, before allowing the user to add the coil (so only circles there were to appear for the user).

To add a component in “or” in the system is somewhat more complex. This as it is added between two points. The user first selects the first point, then possible points to add a component to will light up, and the user will then select the second point, and the component will be added between the points. This is once again done by rewriting the code to reflect this change, then redrawing the system, ensuring that the change is done both graphically, as well as in the code.

The above functionality was, however, much more complex to implement. This because there are clear limits in ladder logic for how components are allowed to be placed. Hence when selecting the first point the user would be able to choose between the full array of possible points, but when selecting the second point the user would only be permitted to select points that are possible to add a component in or to. As one should be able to place objects over an “or” object as well, these would also have points that appear in the first step. So for instance, the user first selects the first point of which the object should be between, from all possible options (as can be seen in figure 13).

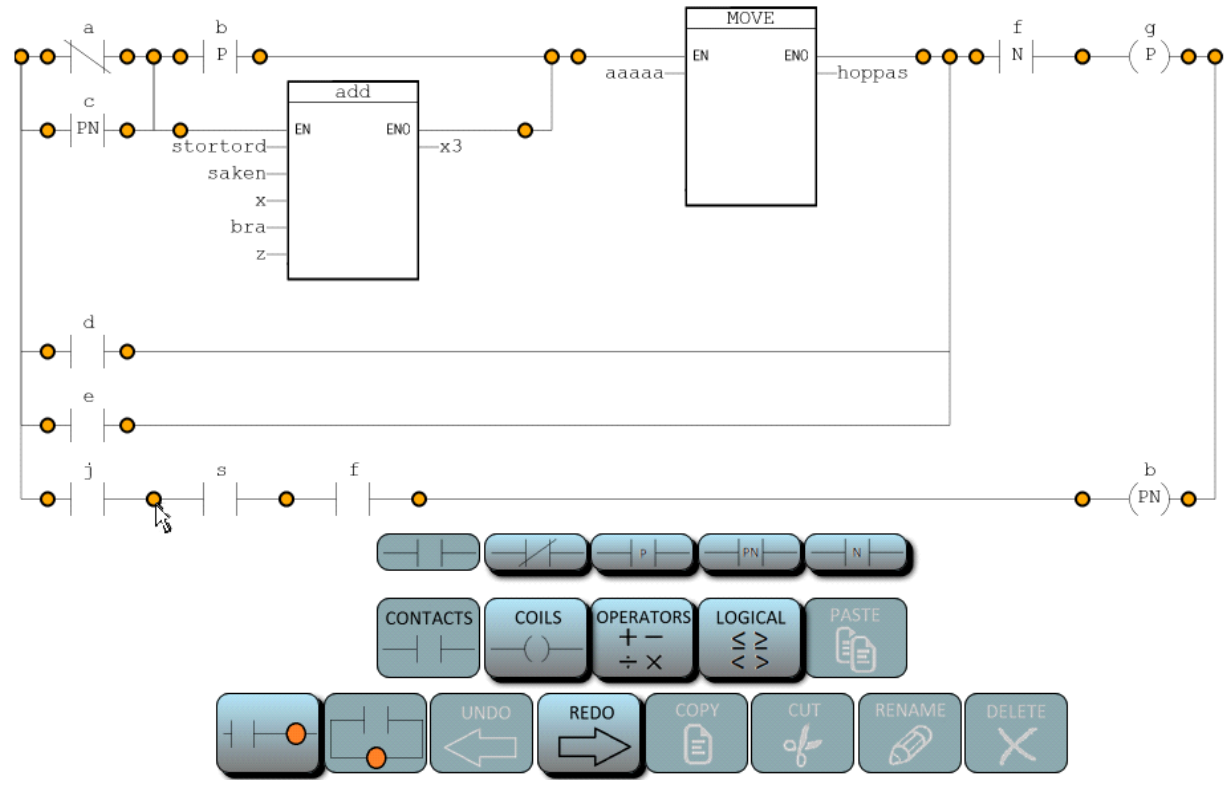

Figure 13 Different options for placing the contact in ”or”

The user is then presented with the possible points the component can be placed between(see figure 14).

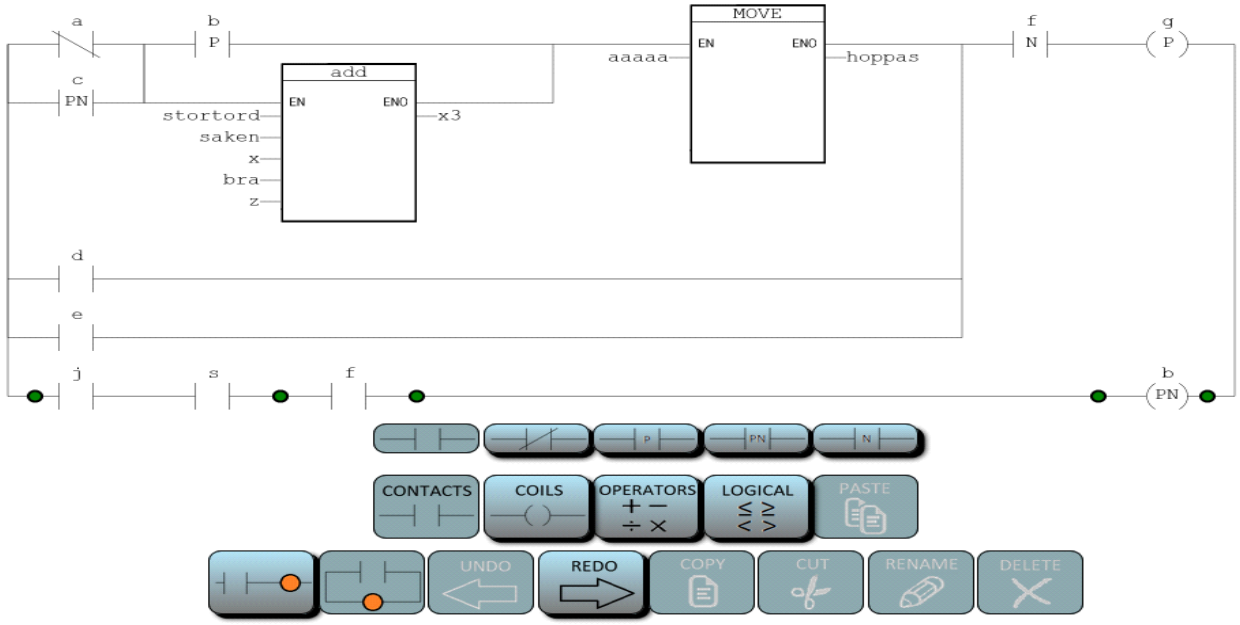

Figure 14 After clicking once, a new set of options appears

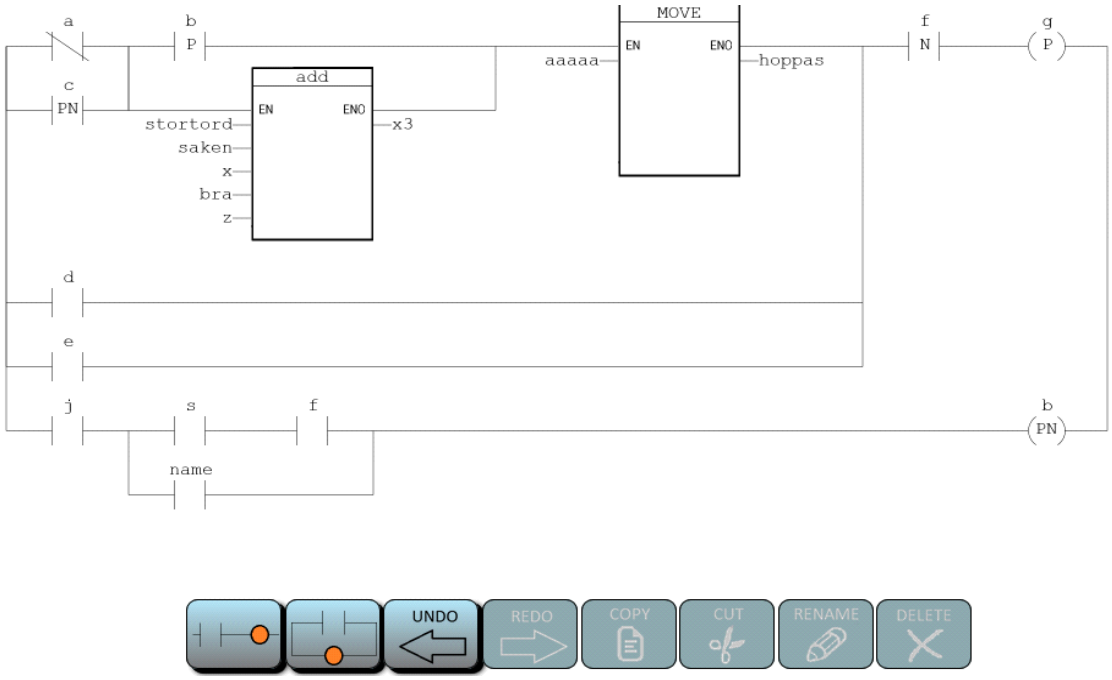

Figure 15 A new contact was placed at the desired position

After the second point is chosen, the component will be inserted as can be seen in figure 15.

So the program needs to be able to check at which points one can insert a component in “or”. This is done by the program checking all the components in the same “and" object as the component tied to the first point clicked. These all have their points lit up as they are all possible targets. Also the program checks all components which have the same point as the selected component and add points to match all components in the same “and" object as these. This is because multiple “or" objects may have points at the same place.

This works well for contact and function blocks. However, the rules for the placement of coils are more restrictive, this as coils must be placed at the right-end side of the diagram. To ensure this when inserting a coil the user can only insert it in “and" objects that are located in the “system object” (or are the “system object”). Moreover, one of the points selected must be at the end of the diagram. This ensures that coils are inserted properly.

There are some limitations on "or" structures that can be implemented in the code. It is only possible to add components between two points in the same “and” structure. While other possibilities can be implemented graphically, they cannot be implemented in code. These are also unintuitive from a ladder logic standpoint, so the inability to implement them was not considered a major concern.

2.5 More features

2.5.1 Undo and redo buttons

The next feature to be implemented was undo - and redo buttons. These were relatively simple to implement. First code was added to create a deep copy of the code describing the system. Each time the system was changed, a copy of the systems code was saved in a vector. Pressing the undo button would now replace the system code with the code last saved. The circuit would then be redrawn to reflect this change graphically. Each time the undo button was pressed the code describing the system would be saved in another vector before it was replaced. Pressing the redo button would replace the system code with the code from this vector, effectively reversing the effect of the undo button. Whenever the system was redrawn, it would check if there were objects in the undo and redo vectors, if any of the vectors were empty, the matching button would be deactivated. Also the redo-buttons vector would be emptied whenever the system was changed (other than by the undo or redo button).

2.5.2 Selecting objects on the canvas

To implement some of the other functions such as deleting components, the user needs to be able to select components on the ladder diagram. To do this a shape class was defined. This class described a rectangle-shaped location on the canvas, containing its x- and y-coordinates as well as its width and height. The shape object also contained an object that would be linked to one of the components (contact, coil or function block) in the system. Once this was done, code was implemented to go through the system object, similar to when the system was drawn, however, instead of drawing the system, each component was added to a shape object, which described a box around it. When this was done, event listeners were added, checking the mouse's location. If the mouse was on top of one of the component's rectangle, the rectangle would shine light blue, indicating that the component could be selected (see figure 16). Clicking would then select the component. It would be redrawn with thicker lines to show this and the program would note that this component was selected, as can be seen in figure 17.

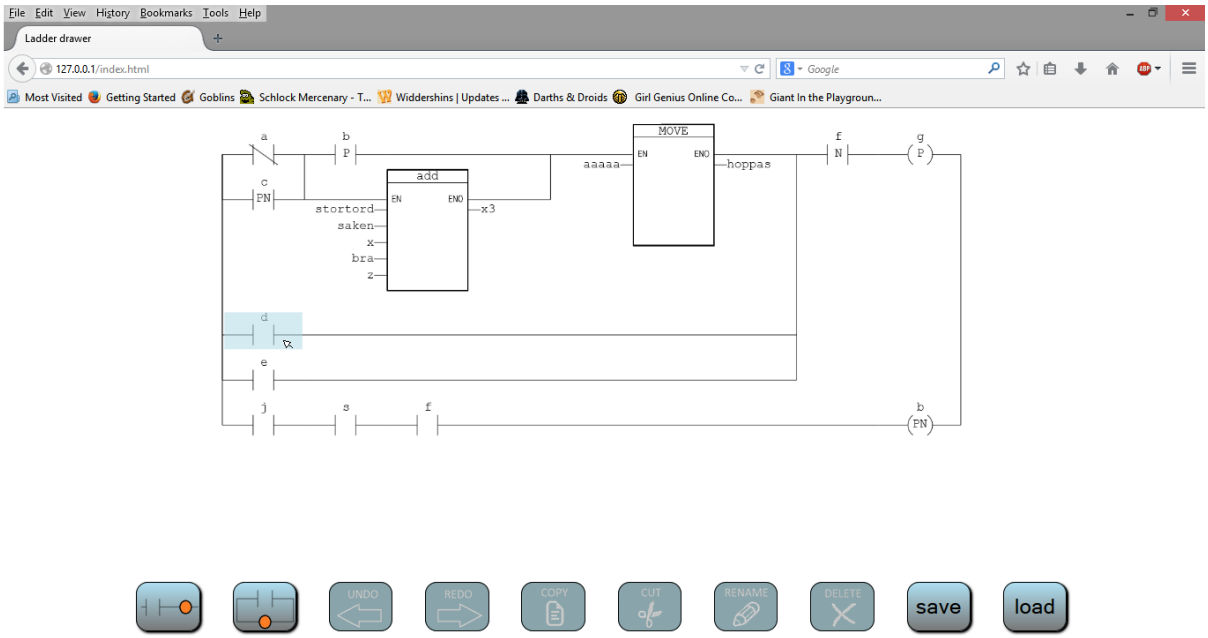

Figure 16 Moving the mouse over an object will result in a background-effect

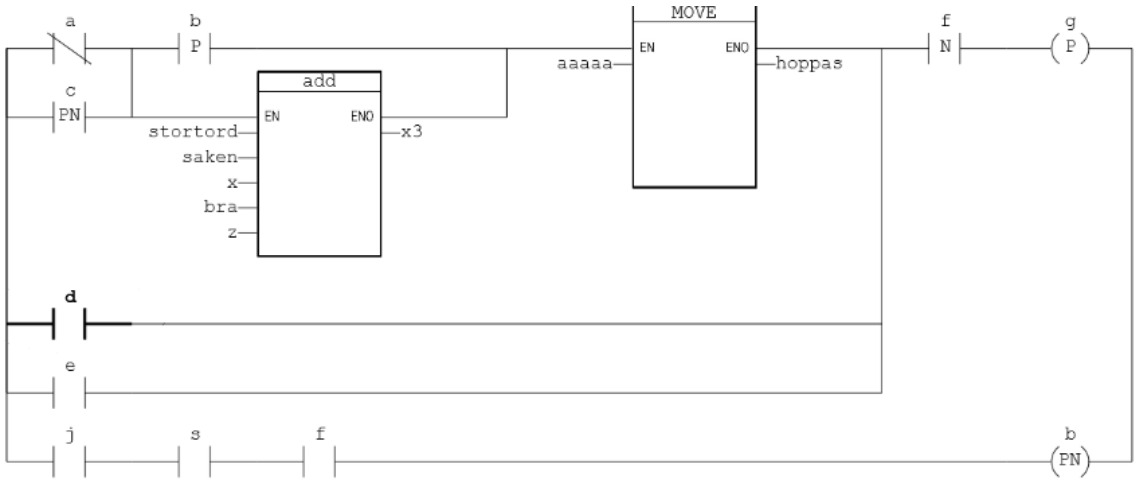

Figure 17 Clicking the component will select it, and result in thicker lines

2.5.3 Delete-function

The functionality to remove elements was also added. This was done by first selecting the component. Clicking the delete button would then delete the component, using the algorithm for removing objects previously described. It was decided that one should also be able to delete the selected object by pressing the delete key on a keyboard, so this functionality was also added.

2.5.4 More components

The first implementation had only three components, a contact, coil and add-function. The next step was to add the other components in ladder logic. These included additional function blocks such as multiplication and division, other types of coils and contacts such as inverted or pulse. To do this as efficiently as possible, the code describing each component was changed somewhat, now including a subtype as well. So for instance a normal contact would be described as:

obj2={type:”contact”,subtype:”contact”, name:”f”};

while an inverted contact would be described by:

obj2={type:”contact”,subtype:”inverted”, name:”f”};

Three main types were used, contact, coil and function. Each had numerous subtypes. This was done as each of the component types would have much of the same properties, regardless of the subtype. For instance both the coil and inverted coil would take the same space on the canvas to draw, and would abide by the same rules of placement in the ladder diagram. Indeed the only difference was that when drawing the inverted coil, a line should be made through the coil. Hence with this system very little code needed to be added to implement all the different components, which was beneficial.

Once this was done, it was easy to implement the code for buttons for each of the components. When clicking on a component button, first circles would be added showing where the component could be added. Then an object for the component would be created and saved in the *selected* global variable. Once the user had clicked the appropriate circle to show where the component was to be inserted, the *selected* component would be added to the code describing the system using one of the methods previously shown.

2.5.5 Copy, cut and paste

Using a similar method, cut and copy functions were also added. These functions were implemented so that the selected block would be saved. Pressing the paste button would then create it so that the *selected* variable was linked to the saved block. Otherwise it would function just like the component buttons.

2.6 Change name and number of inputs

An important feature that needed to be added was the functionality to change the properties of the component (such as the values tied to the contact and coils as well as the inputs and outputs of the function blocks). This was done by letting a menu appear when double clicking on a component. Therefore a menu was placed in the middle of the screen. This menu would normally be hidden, using tags [2] however, when double clicking on a component the menu would become visible. The menu would contain sliders (HTML5 select objects [16]) for changing component properties from a list. For contacts and coils this menu was simple, containing a single slider used to choose the name.

However, the function-blocks required a more complex menu (as can be seen in figure 18). Hence, another menu was placed in the same position as the previous one. The new menu contained sliders for both inputs and outputs. It also contained a slider for choosing the number of inputs, as some blocks, such as an add block, could have a varying number of inputs. This was implemented so that the menu contained five input sliders. However, when double clicking on an object, the system would check the number of inputs the function had, and hide the additional sliders. If the user would want to decrease or increase the number of inputs he or she would change the slider governing the number of inputs, and the additional sliders would appear or disappear to reflect this change (and allow him or her to choose a value for these inputs). Some function blocks only allow a fixed set of inputs (blocks such as the comparison blocks like "greater than" etc.). This was implemented by hiding the slider that decided the number of inputs, so that the user could not change the number of inputs.

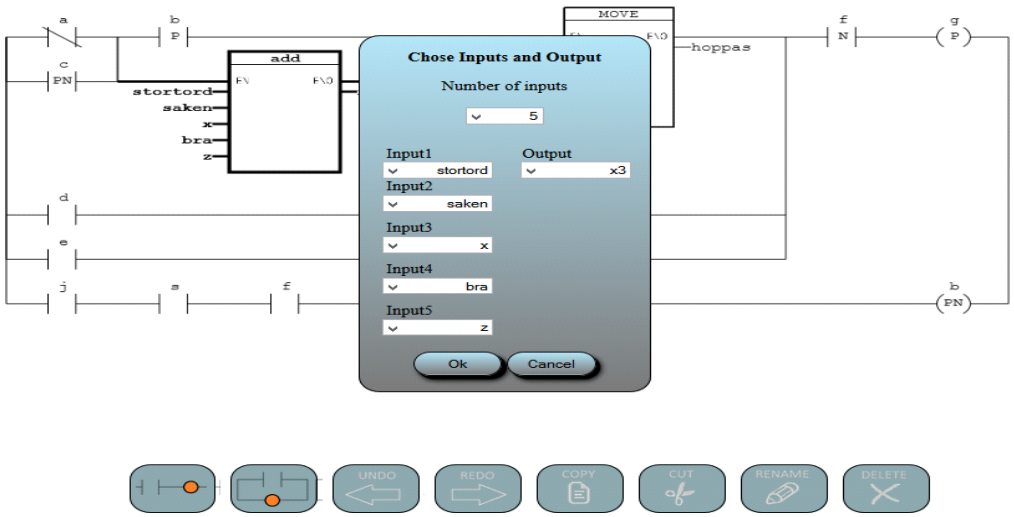

Figure 18 A menu is displayed where the user can select the inputs and outputs

The graphical appearance of the menus background and buttons was designed in the same way as the software's other buttons.

2.7 Buttons

2.7.1 General

For this section [3] [4] [5] [6] has been frequently used.

The first buttons that were implemented where very primitive, and were drawn squares made by the drawer. These buttons were merely placeholders and they were only used for a short amount of time before they were replaced with more elegant buttons. The drawbacks with these primitive buttons were first of all that they could not be styled in a good way. For example the shape of the buttons was hard to customize, and it was impossible to add any gradient to the buttons. When creating buttons you also want the button to change while “hovering” over it with the mouse and this was also an effect that it was felt should be included but could not be achieved in an acceptable fashion at this stage. In some cases one might even want to remove a button after it is pressed, and these features were not possible to achieve in a satisfying way.

A much better way of creating the buttons, was to use the built in button-objects that could be created using HTML5:

<button id=”orbutton” class=”my_button” style=” ……. “
onclick=”myFunction()”></button>

.my_button {

// Common styling for all buttons

}

This is HTML5 code, and is not a part of the canvas window, and therefore gave us a lot more options and features for the buttons. HTML5 is behind every modern website today, and the programmers now had the freedom of styling the buttons, just as one would on any good-looking website.

Creating buttons this way, gave the authors huge advantages, not only for the buttons appearances. The button was given an id, as it would be referred to later on in the code. Each button object also inherited the attributes of the same class, called my_button. In the style field you could give the button even more styling and attributes than was inherited from the class. It is important to understand the difference between the class and the styling field. The class was only written once and was inherited by all our buttons. This means that styling in the class would affect all the buttons in the same way, and therefore the shape and color and all special

effects were written here, since we wanted that for all buttons. The style-field were then custom-made for each button. Changes in the layout and styling that one only wanted to affect a couple of buttons had to be made in the style-field for each button with the desired effect. One example of styling that would be implemented in the style-field would be the size of the buttons, as this is something that is not common for all the buttons. Another attribute that had to be coded in the style-field was the position (coordinates) for each button, since this was very diverse.

When creating a button one also has to specify what code that should be executed once it is pressed. When a user presses the button, that function will start executing no matter what is currently being executed. In the example above, myFunction() would be executed, when the button is pressed.

2.7.2 Create a button

The first task was to decide the size of each button. After experimenting the fixed size for the buttons was decided to be 156x120 px and 156x56 px (depending on the screen size). This would give the buttons the same width for good appearance, and symmetry. Since the authors wanted a picture on each button, representing the element that would be created if you pressed the button, these had to be created before anything else could be done. In Microsoft Paint each picture was drawn with high precision, and this was something that was very time-consuming, but had to be done. This can be seen in figure 19. Once all 40 buttons were drawn with the correct size, it was time to give the pictures a gradient, which was also done in MS paint. Once that was done, we had created the 40 different pictures. These pictures would become the face of the finished buttons (see figure 20).

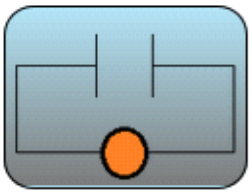

Figure 19 A picture made in paint

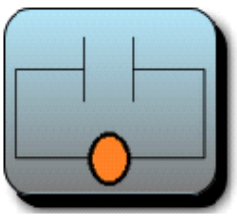

Figure 20 The finished button-look after adding some more effects in HTML5

2.7.3 Presentation and improvement

B&R Automation wanted the gradient and style of the button to be easy to change in the code. The whole purpose of the ladder-editor is that other companies should use it, and they might want to have their own design on the buttons, to match their logo. As the buttons were created now, the companies would have to edit every single button in MS paint and change the color, one by one. This wasn't efficient enough, and the authors were asked to implement a way to change the color and style for all buttons with just a little editing in the code, instead of editing the pictures themselves.

After some research it was found that this was possible. Instead of giving the pictures a color and gradient in the Paint-program, they could just be saved as transparent (no background as seen in figure 21). Each company could then give the buttons any background by editing the class,my_button, which is inherited by all buttons as mentioned above. A few examples of this can be seen in figure 22.

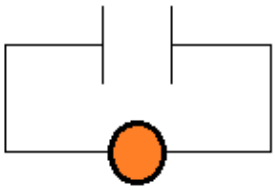

Figure 21 A transparent picture made in paint

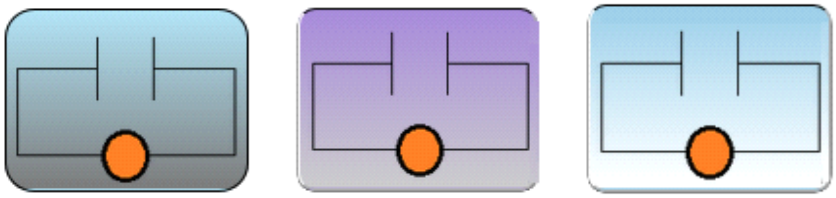

Figure 22 Different stylings in HTML5

2.7.4 Layout

One thing that B&R were clear about from the very beginning was that this editor had to be easy to use. It wasn't trivial to decide which buttons should be shown at which time, and where they should be displayed. A button with id=orbutton, could easily be made invisible/visible with the commands:

```
document.getElementById("orbutton").hidden=true;
document.getElementById("orbutton").hidden=false;
```

A lot of testing was required and the first approach was to give the user two simple options in the beginning, and only display those two buttons, to avoid confusion. The option would be if the user wants to place the next element in "or"/ "and", as seen in figure 23.

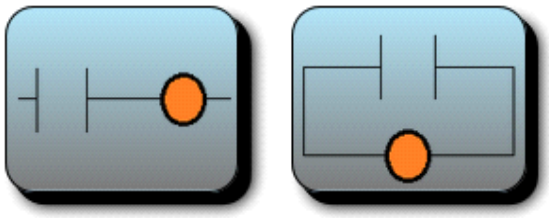

Figure 23 The two buttons that determine the structure of the next added element

Once one of the buttons was pressed, and the user had decided how the next element should be placed, the two buttons were hidden, and the possible elements to add would be displayed (see figure 24).

Figure 24 Different elements to choose from

Once one had chosen an element and placed it in the circuit, these buttons were hidden, and the main buttons in figure 23 were displayed again and everything would start over. This approach was a huge improvement, but after a conversation with B&R it was decided that the button structure should be further modified for optimal look and user experience.

2.7.5 Final touches

The first improvement that was made was sub-categories. This was needed since the number of elements kept increasing and users might have trouble to find the elements they were actually looking for.

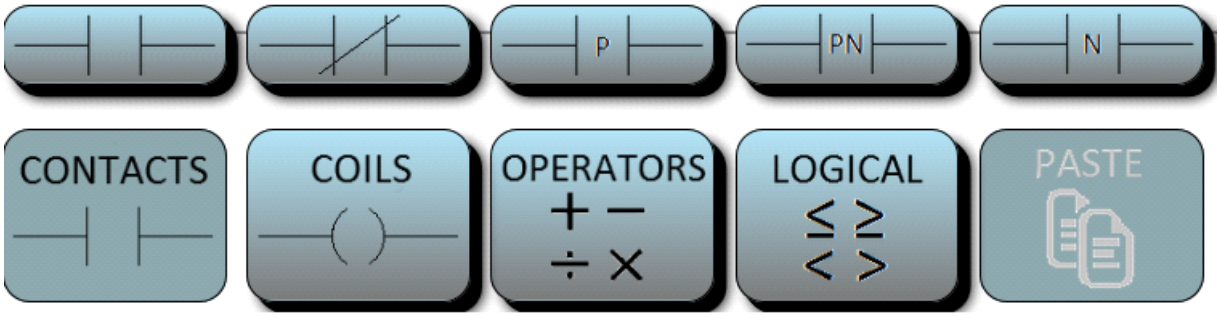

Figure 25 Showing the different elements in the sub-category ”Contacts”

In figure 25 the four big buttons (156x120px) each representing a sub-category. As we can see Contacts is pressed, and five different contact-elements are displayed (156x56px). If the element that the user is looking for, is not in this sub-category, one can for example press the Coil-button and see if the desired element can be found there (see figure 26).

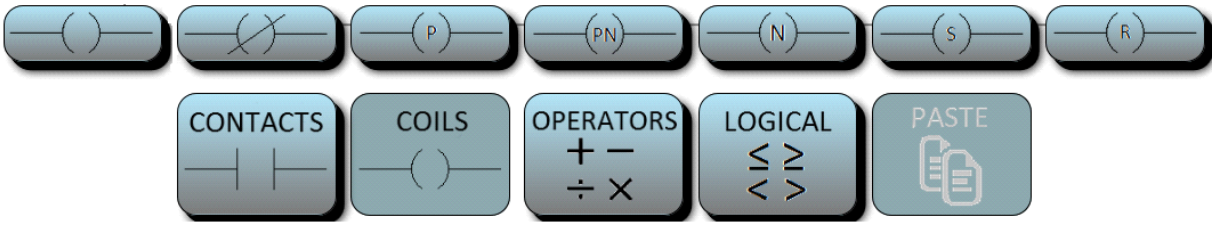

Figure 26 Showing the different elements in the sub-category ”Coils”

There are a few operators and those can be found in this sub-category(see figure 27). In the beginning the buttons had pictures with the sign on it (+, - , * , /), but it was later on agreed that just the text would be enough.

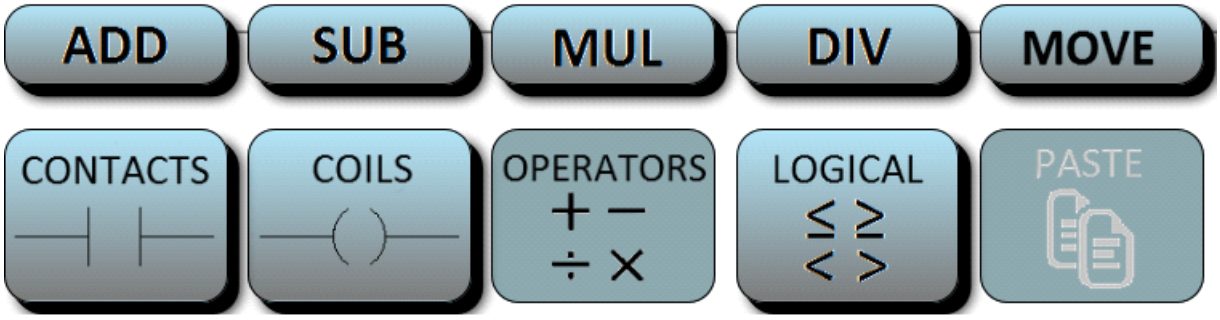

Figure 27 Showing the different elements in the sub-category ”Operators”

The last sub-category is the logical one, and as shown in figure 28 there are a few comparison functions. The letters are abbreviations for the mathematical term. For example GE stands for greater or equal to.

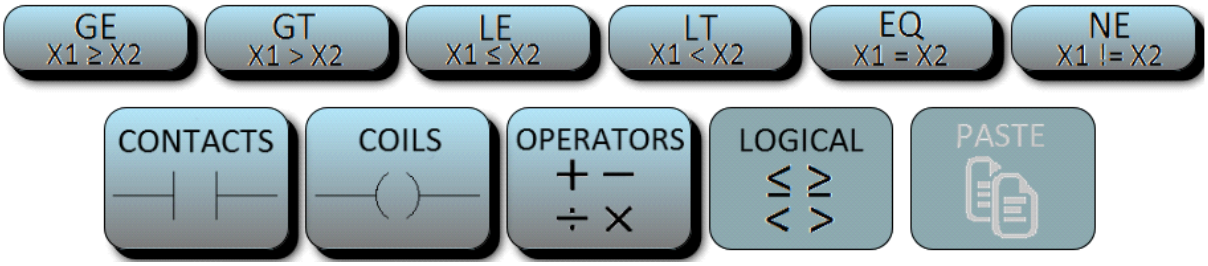

Figure 28 Showing the different elements in the sub-category ”Logical”

The main function that B&R found lacking, was the ability of the user to change his or her mind, after a choice had been made. What if you had pressed “Or” and then when it was time to select what element to use, you would realize that it was actually “And” you needed. With the current structure there was no turning back and you would have to place the element, even if you knew that it wouldn’t be placed the way you wanted. This was solved by displaying the next layer of buttons above the current ones, instead of hiding the old ones. You would now simply press the same button twice to regret your decision. It was also desirable to implement copy, cut, delete, rename ,paste - buttons , like in any document-handler.

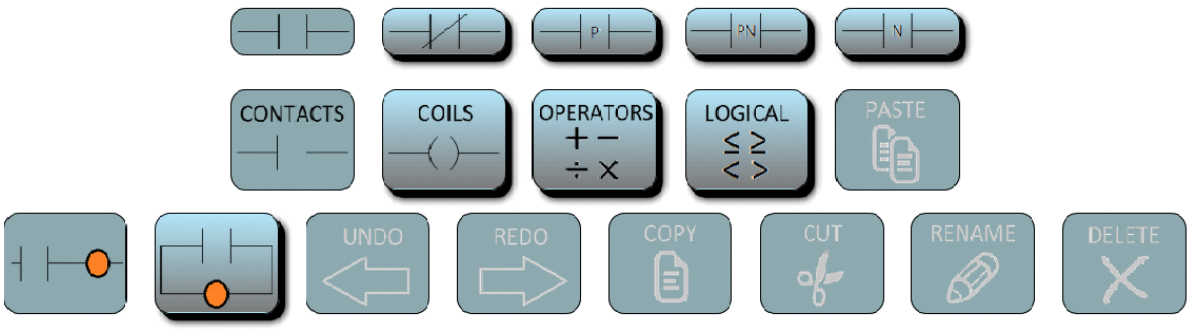

Figure 29 The full set of buttons

As seen in figure 29 a “push-down effect” is added to the pressed buttons, to see which you’ve chosen. If you press a button that has already been pressed, it will get back to normal and the layer of buttons above will be hidden. A gray version was also made for some buttons, and this was to indicate if it was possible to press the button at a specific time, or not. For example in figure 29 you can’t press “UNDO” since it is gray. However, if you would add any element the Undo-button would light up, and it would now be able to be pressed, since there is something to undo (see figure 30). Similar terms regards the other gray buttons, but the requirements for lighting up a button is different for every button.

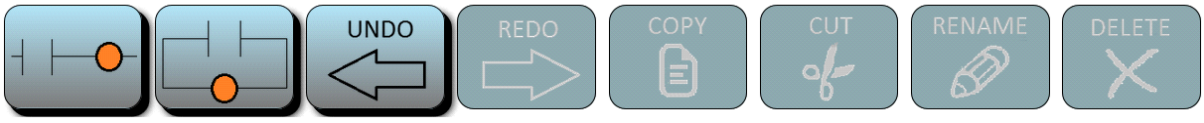

Figure 30 The “undo-button” is made available since there now is an action to undo

2.8 Dividing the code into different files

When programming all the code was stored in one index file. This, while convenient when working on it, was not easy for an uninitiated to browse through, as the various functions where spread all over the file in no apparent order. Hence the index file was divided into many different files to give a better overview. In the end the program was divided into the following files:

Ourcss: A file was made that contained the graphical data of the buttons.

Drawingfile: The file containing the code for drawing the ladder program on the canvas.

Makeclickables: The file dedicated to recording the co-ordinates on the canvas where one could click (such as the components, as well as the circles used when adding new components).

Changeladder: Here the code for changing the ladder program was included. So it would contain the functions to add and remove components to the ladder diagram.

Buttonscode: Here the code connected to each of the buttons was included.

Meny: This file contains the code for the menus used for changing the properties (such as name) for a component.

Modes: This file contains the code used to ensure that the correct buttons are visible depending on what the user has clicked.

Utils: This file was dedicated to the general utility functions (such as finding a component in the system code), that were used at many places and did not fit in any of the other files.

With this subdivision it became simpler to find the different parts of the code.

2.9 Communication with Automation Studio

With the editor complete, the next step was to establish communication with B&R's Automation's own software. This communication was done via HTTP requests, where the editor program would initiate communication with a B&R Automation Studio program which was being run simultaneously. This communication was used both for receiving a list of variables from Automation Studio as well as for saving and loading the ladder program.

2.9.1 Automation Studios

Automation Studios is B&R's existing program for communicating with their PLCs. It contains programs that allows the reading and writing of variables from their machines. There are possibilities to write and run several different programming languages, including C (with its own library for a few functions which differs somewhat from standard C).

The programs run by Automation Studios are generally run cyclically. That is to say they are run at predetermined time intervals (for example at every second).

When implementing the communication with Automation Studios for the ladder editor, the necessary code that Automation Studios would run was written in C.

2.9.2 Receiving a list of variables

When adding components to the ladder program, one would choose from a list of variables when selecting the properties of the component. So for instance when adding a contact, one would select a variable from a list of boolean inputs to be the input of this contact.

This list of variables was of course dependent on the process the ladder program was made for. Hence it would be managed by B&R's Automation Studio. So there would be a file containing the list of variables that Automation Studio would on a request from the ladder editor read from and then send the list to the ladder editor. This list would contain the variables available for contacts, coils etc. The list was written in a text file (.txt) and had the following format:

[coils]

Valve1=TestLadder:a

Valve2=TestLadder:b

Valve3=TestLadder:g

Conveyor1=TestLadder:f

Conveyor2=TestLadder:es

[contacts]

Sensor1=TestLadder:a

Sensor2=TestLadder:b

Sensor3=TestLadder:c

[inputs]

input1=TestLadder:h

[outputs]

output1=TestLadder:k

output2=TestLadder:j

As can be seen the variables are defined in the format (Valve1=TestLadder:a). "Valve1" here is mainly for the users benefit, denoting to what physical output the variables refers. "Testladder:a" however, denotes that this is variable 'a' and that it is used by the Automation Studio program TestLadder. This notation is used by Automation Studio to find the variable. This is done for two reasons. Firstly, it allows the program to ensure that the variable exists and is of a correct type (for instance if the variable is a string it should not be sent to the ladder editor). Moreover when sending the variables to the editor, the Automation Studio program was programmed to generate a list, saving the variable sent, along with the id number Automation Studio had given them. This list links the variables used by the editor to the variables used by Automation Studio, which can be used when making an interpreter for the editor.

When the ladder editor receives the list of variables (sent as a single long string) from Automation Studio it will use this to update the lists used when changing the ladder components properties, using commands available from the JavaScript string class to interpret the text from Automation Studios [12] [13].

2.9.3 Saving and Loading the file:

When implementing save and load of the system, the fact that the system was described in a format usable by Json was taken advantage of. When the user clicks on the save button, the system would be converted to a Json string and sent from the ladder editor to the Automation Studios program, which would write the string to a .txt file. The load button worked in a similar fashion. The ladder editor would sent a request to Automation Studio, which would read the string saved in the .txt file and send it to the editor. The editor would then convert the string back to a JavaScript object and replace the current system object with this object. The system would then be redrawn, so the change would be reflected graphically as well.

3. Results, discussions and conclusions

The result was a user friendly, intuitive and multifunctional ladder-editor. The editor was tested widely both by the authors and at B&R. Very large systems were created, with different structures and the editor passed them all. The editor was also tested in all the common browsers such as Google Chrome, Firefox, Safari, in all cases with correct performance.

Something that was always at the forefront was that the editor should be user friendly and intuitive. While this is something of which success is difficult to asses, most users seemed to quickly be able to grasp how to use the editor. Moreover, functions such as the undo button makes it more forgiving to an inexperienced user, and allows for the user to experiment with the editor. Hence we determined that no further instructions should be necessary for a user to work with the editor.

The way the editor is implemented, with the user simply clicking where he or she wishes to add a component, has many advantages. As the program itself handles the drawing of the appropriate lines, this ensures that the ladder diagram will always appear neat, with clear straight lines. This can be compared to many other ladder editors in where the user draws the lines, which often leads to messy and unclear ladder diagrams which are difficult to edit.

While a wide array of components was implemented, there is obviously always the possibility of more components that may need to be added. The editor was programmed so that it is relatively simple to expand the number of components that the editor contains.

Indeed, the fact that small changes to the editor should be possible to make quickly and easily was something ensured while programming. For instance the buttons all derive their appearances from a single css-class, hence it is trivial to change the buttons color scheme.

There are a few limitations to the editor however. For instance, as discussed in the relevant section, every possible ladder program cannot be implemented, as some unintuitive ones cannot be described in the format we use to describe a ladder program. Moreover, there are a few limitations in how it was implemented, for example that functions can have a maximum of five inputs. However, these limitations can generally be expanded quite easily, if necessary.

Another limitation worth mentioning is that the editor does not work on Internet Explorer. This because the web browser does not contain many of the functions that were used. However, this was seen as a minor issue, as use of Internet Explorer is

limited, and going down, and it was initially specified that the editor did not need to work with Internet Explorer.

4. Future work

While the editor itself is quite complete there is still the matter of an interpreter. There was not time to complete this task in this master thesis project, but it is something that needs to be done, perhaps in another thesis project. The interpreter would interpret the program of the ladder editor, updating variables as needed. So for instance if one had the following diagram (see figure 31).

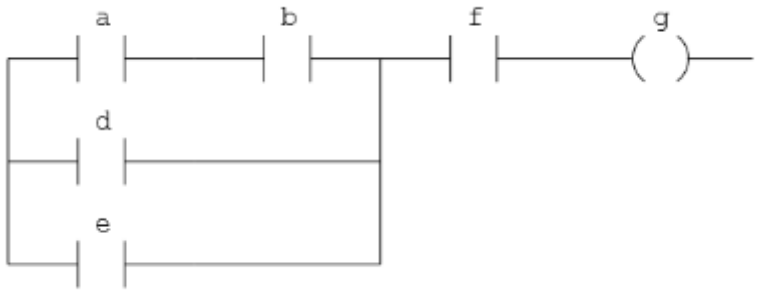

Figure 31 The test-system used in the beginning of the report

The interpreter would look at the values of a, b, d, e and f from the process, and update the value of g appropriately.It would keep doing this at every time interval. It would likewise have to work for function blocks, updating the variables connected to them as appropriate, hence the interpreter can be said to run the program.

The interpreter would have been written in C in the Automation Studio program. While not much work has yet been done on the interpreter, the general structure envisioned was that one would use the string that was sent from the editor when saving the ladder diagram. This string would then be converted to a structure similar to the JavaScript object, but in C. Then recursive functions would probably be needed, going through the C object and checking the values of the contacts, going forward through the object until one reached a coil. It would then use the data it had acquired to check if the coil should be true or false.

Interpreting function blocks would be more complicated. However, this can probably be implemented within the recursive structure, by when reaching a function, the program tests if the function should be activated or not in a similar manner that coils are checked. The code for each function activation would then of course have to be implemented, keeping in mind that many functions (such as add) can have multiple inputs.

5. References

All websites were reachable at 11:25 CET 07/07/2014

[1] http://www.w3schools.com/HTML/HTML5_intro.asp

[2] http://www.w3schools.com/tags/default.asp

[3] http://www.w3schools.com/cssref/css3_browsersupport.asp

[4] http://www.cssbuttongenerator.com/

[5] http://buildingfirefoxos.com/building-blocks/buttons.HTML

[6] http://HTML.net/tutorials/css/lesson1.php

[7] http://www.w3schools.com/HTML/HTML5_canvas.asp

[8] http://www.w3schools.com/js/js_intro.asp

[9] http://stackoverflow.com/questions/8773921/how-to-automatically-scroll-down-a-HTML-page

[10] http://www.w3schools.com/js/js_arrays.asp

[11] http://www.w3schools.com/jsref/jsref_obj_array.asp

[12] http://www.w3schools.com/js/js_strings.asp

[13] http://www.w3schools.com/php/php_string.asp

[14] www.stackoverflow.com

[15] http://simonsarris.com/blog/510-making-HTML5-canvas-useful

[16] http://www.w3schools.com/tags/tag_select.asp