Stepladder

Easy to use displacement diagram editor

Mark Bekooy
Arend-Jan van Hilten
Nathan van der Kamp
Robin Kouwenhoven
Mathijs de Wolf

Delft University of Technology

TU Delft

Stepladder

Easy to use displacement diagram editor

by

Mark Bekooy
Arend-Jan van Hilten
Nathan van der Kamp
Robin Kouwenhoven
Mathijs de Wolf

at the Delft University of Technology.

Project duration:	April 20, 2020 – July 3, 2020	
Thesis committee:	Dr. Myrthe Tielman,	TU Delft, supervisor
	Thomas Overklift,	TU Delft, coordinator
	Roland Hollaar,	De Techniekschool

An electronic version of this report is available at `http://repository.tudelft.nl/`.

TU Delft

Preface

We, a group of five Computer Science and Engineering students at Delft University of Technology, wrote this report as part of our Bachelor End Project. Our Bachelor End Project took place in the last quarter of the third year, just before completing the bachelor program. The goal of the Bachelor End Project is to work for a business, where a student can research a topic the business is interested in and create a software product for them. This report describes our experience and result of researching and creating the Stepladder software for De Techniekschool.

This report is written for readers who we assume to have nearly no knowledge in the field of pneumatics. Essentially, just like we did not know much about the topic before starting this project. The problem and our analysis of the problem are described as clearly as possible at the start of this report in order to make sure that it is comprehensible for anyone. In addition, Appendix D explains crucial information about pneumatics and ladder logic for anyone to understand.

Readers who are primarily interested in the problem that our product solves and how it fits into context, can best read chapters 1 and 2. For those who are primarily interested in how our product looks, chapter 4 is best suited. Chapter 6 is for those who would like to know how our Stepladder software is actually implemented.

We would like to thank Myrthe Tielman for being a great supervisor and all the helpful weekly meetings. We are also thankful to Roland Hollaar for providing us this project and the explanation of pneumatics in the first two days of this project together with Jan Asmus. Furthermore, we are grateful for Jan Asmus, Marko Versteeg and Jesse Knobbe from De Techniekschool for giving us feedback on the application prototypes which lead to making the final product even better.

Delft, 26 June 2020

Mark Bekooy
Arend-Jan van Hilten
Nathan van der Kamp
Robin Kouwenhoven
Mathijs de Wolf

Summary

Working with pneumatics can be done using a Programmable Logic Controller (PLC), which commands the individual actuators. To start, the movements of each component are described in a displacement diagram. From this diagram, ladder logic can be extracted, which is a set of conditions with corresponding actions. These actions are executed when the conditions hold. This ladder logic is the programming language of PLCs and makes it possible to control pneumatics. De Techniekschool provides courses on these techniques and pneumatics.

This report describes the research, design and implementation of a software application to aid students in their education of pneumatics. The main goal is to allow users to create a displacement diagram and then generate the corresponding ladder logic based on this diagram. This will allow users to understand how this conversion works and help them in understanding PLC programming.

In the research phase, various designs were evaluated. The implementation of the compiler was thought out and possible designs for the Graphical User Interface (GUI) for creating the diagram were explored. In the case of the GUI, the final implementation differs from the initial chosen design. This is because, during the project, it was discovered that the design did not work as well as intended, thus the switch was made. To guarantee the correct behaviour of the product, each individual component was tested on its own with unit tests.

Besides a compiler and the ability to draw displacement diagrams, features of Stepladder include the option to save and open work, and the option to undo and redo actions. It is also possible to show the signals which the PLC will receive and send. These signals already play a vital role in the conversion from displacement diagram to ladder logic and the generated signals can help to understand this conversion.

Desirable features which are not yet implemented are the support for multiple languages and improvements to the accessibility of the application. Also, a more advanced compiler is desired, because the current compiler does create ladder logic according to a valid technique, but it is not the optimal solution. For learning purposes, an optimal conversion may be desired. Proper user testing is also desired to verify the design and whether the application is easy to use.

To conclude, the final product allows the user to create displacement diagrams and generate ladder logic. This can be used to study the conversion needed in PLC programming. The final product meets the design goals set in the design process. Besides the design goals, the product also satisfies the initial requirements. There are some features the client could implement but these are not required.

Contents

Preface i

Summary ii

1 Introduction 1

2 Problem definition and analysis 2
2.1 Problem definition . . . 2
2.2 Problem analysis . . . 2

3 Design 3
3.1 Ethics . . . 3
3.1.1 Trust . . . 3
3.1.2 Privacy . . . 3
3.1.3 Accessibility . . . 3
3.2 Goals . . . 4
3.3 Requirements . . . 4
3.3.1 Non-functional requirements . . . 4
3.3.2 Must have . . . 4
3.3.3 Should have . . . 5
3.3.4 Could have . . . 5
3.3.5 Won't have . . . 5
3.4 Architecture . . . 6

4 Product 7
4.1 User Interface . . . 7
4.2 App bar . . . 7
4.3 Devices and diagram . . . 8
4.4 Signals . . . 9
4.5 Settings . . . 9
4.6 Shortcuts . . . 10
4.7 Notifications . . . 11

5 Process 12
5.1 Group division . . . 12
5.2 Development methodology . . . 12
5.2.1 Scrum . . . 12
5.2.2 Tracking progress . . . 13
5.3 Communication . . . 13
5.3.1 Communication with coach . . . 13
5.3.2 Communication with client . . . 13
5.3.3 Communication with group . . . 13
5.4 SIG . . . 13

6 Implementation 14
6.1 Frameworks . . . 14
6.1.1 Electron . . . 14
6.1.2 Vue . . . 14
6.1.3 Vuetify . . . 14
6.1.4 I18n . . . 15

6.2 User Interface choices. 15
6.2.1 App bar 15
6.2.2 Devices and diagram 15
6.2.3 Signals 17
6.2.4 Settings 17
6.2.5 Notifications 17
6.3 Compiler 18
6.3.1 Manually. 18
6.3.2 Automatic 20

7 Code quality 23
7.1 Continuous integration 23
7.2 Testing 23
7.2.1 Static analysis 23
7.2.2 Dynamic testing 24
7.2.3 Testing utilities. 24
7.3 Code review. 25

8 Reflection on Ethics 26
8.0.1 Trust 26
8.0.2 Privacy 26
8.0.3 Accessibility 26

9 Discussion and recommendations 27
9.1 Discussion 27
9.1.1 UI Mocks. 27
9.1.2 Code style 27
9.2 Recommendations 27
9.2.1 User testing 27
9.2.2 More advanced compiler. 27
9.2.3 Multiple languages. 28

10 Conclusion 29

A Original project description 31

B Project info sheet 32

C Feedback SIG 34
C.1 Code duplication 34
C.1.1 Compiler. 34
C.1.2 User Interface 34
C.2 Large units 35
C.2.1 Compiler. 35
C.2.2 User Interface 35
C.2.3 Vuetify plugin 35
C.3 Complexity 35
C.3.1 Compiler. 35
C.3.2 User Interface 36

D Context 37
D.1 Pneumatics 37
D.2 Ladder logic. 37
D.3 Displacement diagram 38

E Checks 40
E.1 Errors 40
E.2 Warnings 40

1 Introduction

Many industries in the world use components such as cylinders and motors to control their industrial production line. These components can be operated by air, other gasses or fluids. It is thus of importance that people are educated on these topics. De Techniekschool, who commissioned this project, does exactly that. They teach people how to use machinery and components with air in their course on pneumatics.

De Techniekschool was established in 2009 [1]. They focus on educating practical engineering and technologies, from electrics and electronics to forklift certificates and control technology. In their course on pneumatics, De Techniekschool explains the fundamentals. The main part of the course is understanding how pneumatics work. This movement can be described in displacement diagrams. These diagrams are in general converted manually to ladder logic and code that controls the pneumatic components or relays logic.

The purpose of this project is to allow users to create such displacement diagrams and then automatically transform them into ladder logic, in such a way that the user can understand the ladder logic and learn how to create it themselves. The relays logic can easily follow from the ladder logic.

This report elaborates on the development process from initial design to completion. In chapter 2 the problem that is solved is described. This problem is also further analysed in relation to the goals of De Techniekschool. Chapter 3 describes the design goals of the product as defined in the preliminary research phase. It also continues on the ethical implications of these choices. Chapter 4 presents the final product. It uses descriptions and pictures to show how the product works. In chapter 5, the development process is reviewed. In chapter 6, the design of the product from the research phase, and how the actual implementation differs from the initial design, are discussed. Then, chapter 7 describes the methods and utilities used to ensure high code standards. Furthermore, chapter 8 is dedicated to reflecting on the ethical implications of the product. Chapter 9 reflects on pitfalls encountered during the project and discusses recommendations towards De Techniekschool. Then the conclusion, where the initial goals are reviewed to see if they have been achieved, can be found in chapter 10. To conclude this report, Appendix D gives some extra context about the field of pneumatics and ladder. This is not necessary to read the report but may help in understanding the underlying concept.

2 Problem definition and analysis

In this chapter, the goal of this project is specified. In section 2.1, the problem is defined to give a clear understanding of what needs to happen in the project. Section 2.2 elaborates on the problem and how De Techniekschool is planning to use the final product.

2.1. Problem definition

Pneumatics is the technology of using air to operate mechanical machinery. This machinery is controlled by the direction of airflow. The airflow can be controlled with a Programmable Logic Controller (PLC). In order to educate on the use of these PLCs, De Techniekschool decided to develop their own. Besides their custom PLC, De Techniekschool also developed Ladderino which is used to program their PLC. The purpose of Ladderino is to transform ladder logic into code that controls the PLC.

The creation of ladder logic can be difficult. To aid students even further in the learning of PLC programming, De Techniekschool came up with the idea of creating an application which can create ladder logic from a displacement diagram. A displacement diagram shows the movement that each component makes over time. Creating such a diagram is much easier than creating ladder logic by hand.

The goal of this project is to create an Integrated Development Environment (IDE) called Stepladder which allows users to create a displacement diagram. The application should then transform this created displacement diagram in ladder logic such that it can be used by Ladderino. This translated ladder logic is used for educating people on how to convert it themselves.

2.2. Problem analysis

De Techniekschool wants to use the product to educate on the conversion from displacement diagrams to ladder logic. To educate is, therefore, the main purpose of the program. The aim is to have users use the generated ladder logic to study how the conversion is done and how they can do it themselves, not to replace the manual conversion of displacement diagrams in ladder logic.

De Techniekschool is also developing teaching material for lessons on VMBO and MBO schools. They want to use both the final Stepladder product and Ladderino, the program that handles the logic that is created by Stepladder, to support with understanding the material. This enforces the statement made in the paragraph above, since it will not only be used to educate in their own lessons, but also on a national scale in schools.

Since the project entails to create an IDE, it is important that it will be user friendly. It must be easy for users to interact with the software and perform its main task, namely, creating displacement diagrams. To result in a usable product, human-computer interaction principles have to be taken into account. It has to be designed with ease of use in mind. This means that it must not be complex and relatively simple in design. This is further elaborated on in section 3.2.

3 Design

In this chapter, the design of the application will be discussed. First of all, ethics are considered. Second, the goals of the design are stated. Third, the requirements of the design are listed. Finally, a brief overview of the architecture of the application is included.

3.1. Ethics

To minimise any harm done by the product, its ethical consequences have to be considered and it must be ensured that the design conforms to ethical principles as much as feasible. Trust, privacy and accessibility aspects will be considered.

3.1.1. Trust

Users should be able to trust the application to work correctly. Therefore, the application should be thoroughly tested with automated and manual tests to ensure that features work correctly, not only when they are added, but also after other changes are made. The codebase should at least have a test coverage of 80% of all statements, lines, branches and functions, though we will aim to cover as much as possible.

3.1.2. Privacy

An important part of software ethics is respecting the user's privacy. The application does not need to handle any personal data or usage information about the application. Therefore, it should only store and save the information required to describe displacement diagrams and Ladderino files. This means that it will be impossible for our application to violate a user's privacy.

3.1.3. Accessibility

The application should be accessible to as many potential users as possible. This means that the user interface should be easy to understand and use, but also that it should account for people with disabilities like colour blindness or ones that prevent them from using a mouse or keyboard.

Colour blindness

To accommodate for colour blind users, the application should use clear contrasting colours as much as possible. Furthermore, it should not rely on the colours of UI elements. Colour coding can be used to improve the clarity of the UI for users that can differentiate the used colours normally, but should be used in conjunction with clear text or icon labels for the elements.

Motor and dexterity impairments

Since some users with motor and dexterity impairments might have trouble operating a mouse precisely, it would be advantageous if the application can be controlled with the keyboard alone. On the other hand, some users might be unable to use a keyboard, so the application should also be entirely usable by mouse. Text fields do require keyboard input, but an on-screen keyboard can be used for these. Some text fields are used for entering numbers. These fields can also allow the user to change the value by clicking an up or down arrow to increment or decrement the number in order to avoid typing.

Dyslexia
The application will not require the user to read large amounts of text but nonetheless, dyslexic users should be accounted for. Research has shown that serifed fonts and italics decrease reading speed significantly as opposed to sans-serif and roman font styles. [2] Therefore we will not use serifed or italicised fonts in our application.

Language
Another important aspect of accessibility is language. Initially, the application will be made in English, but if there is time remaining, setting up localisation and translating the app to Dutch would be helpful, since most of the users of the app will likely be Dutch.

3.2. Goals

The product should be able to generate ladder logic from a displacement diagram for Ladderino to use. Ladderino must then be able to generate code for the PLC of De Techniekschool from the generated ladder logic. The users will use the generated ladder logic to learn how to convert a displacement diagram into ladder logic by hand. In this section, three design goals are discussed, which the product should fulfil.

The first design goal is the correctness of the generated ladder logic. When converted into code for the PLC, the machinery connected to the PLC should do exactly what is defined in the displacement diagram. Any ambiguities that can result from the translation of the displacement diagram into ladder logic should be resolved. These ambiguities are inherent to the translation process and are not a result of bad diagram design, which will be further discussed in section 6.3.

The second design goal is that using the product should be easy to learn. The product will be used by people who work or are going to work in the engineering branch to learn to program PLCs. It can be assumed that the user has experience using computer programs. As the focus should lie on learning to program PLCs, learning to use the product should not be difficult. The use of the product should be intuitive, such that using it will go almost automatically. After being familiarised with the product for four hours, the user should know how to use the program independently.

As the main purpose of the product is to let the user study the generated ladder logic to find out how the displacement diagram can be converted, the last design goal is that the generated ladder logic should be understandable for the user. If the generated ladder logic is unreadable, the user cannot learn anything from it. To make the generated ladder logic readable, it should use easily recognisable patterns. What these patterns are and how they can be used, can be found in section 6.3. These patterns should even be used when this compromises the speed of the product. Readability is more important than performance, as the purpose of the product is to teach the users and not to control actual machinery.

3.3. Requirements

To prioritise the work that has to be done for the project, the features are split up using the MoSCoW method [3]. For the Minimum Viable Product (MVP) the features listed in the "must have" and "should have" category need to be implemented. Without these features, the MVP is not complete. The features listed in the "should have" category, should be implemented, but are less important than the features in the "must have" category. The "could have" features should be seen as bonus features and they do not need to be implemented and could be done by De Techniekschool after the project. Next, the list of requirements follows, beginning with non-functional requirements.

3.3.1. Non-functional requirements

- When the user gives input, the program has to respond or show an indication of progress within 400 ms
- Only sans serif typefaces and no italics will be used

3.3.2. Must have

The following requirements have to be implemented for the MVP:

- The user must be able to make a displacement diagram
- The user must be able to compile the diagram into ladder logic (Ladderino)

- The user must be able to choose between single and double acting cylinders
- The user must be able to add a start button or have continuous execution
- The user must be able to open and save diagram files

3.3.3. Should have

The following requirements should be implemented at the end of the project, but are less important than the must have requirements:

- The user should be able to use motors with encoders and switches
- The user should be able to add counters that allow sections to repeat any number of times
- The user should be able to add timers that allow the execution to be paused for a certain number of milliseconds
- The user should be able to undo and redo actions

3.3.4. Could have

The following requirements could be implemented, but are not necessary:

- The user could compile the diagram to Codesys, another Ladder logic IDE
- The program could be able to give suggestions to the user
- The program could explain the Boolean formulas in the ladder output
- The user could be able to add safety features, like emergency button, safety button and protective cover
- The program could be able to show examples
- The user could be able to use a debug mode with highlights on what is happening

3.3.5. Won't have

The following requirements are not going to be implemented in this project:

- The user will not be able to follow a walk-through

3.4. Architecture

The application consists of two main components, the user interface and the compiler. In the interface, the user can enter all the information required to define a displacement diagram. This information is stored as a JavaScript object and passed to the compiler which has to check if it is valid. If there are errors in the diagram, these are shown to the user. Else the diagram can be compiled and saved to a .laddr file. Figure 3.1 shows the architecture of our application.

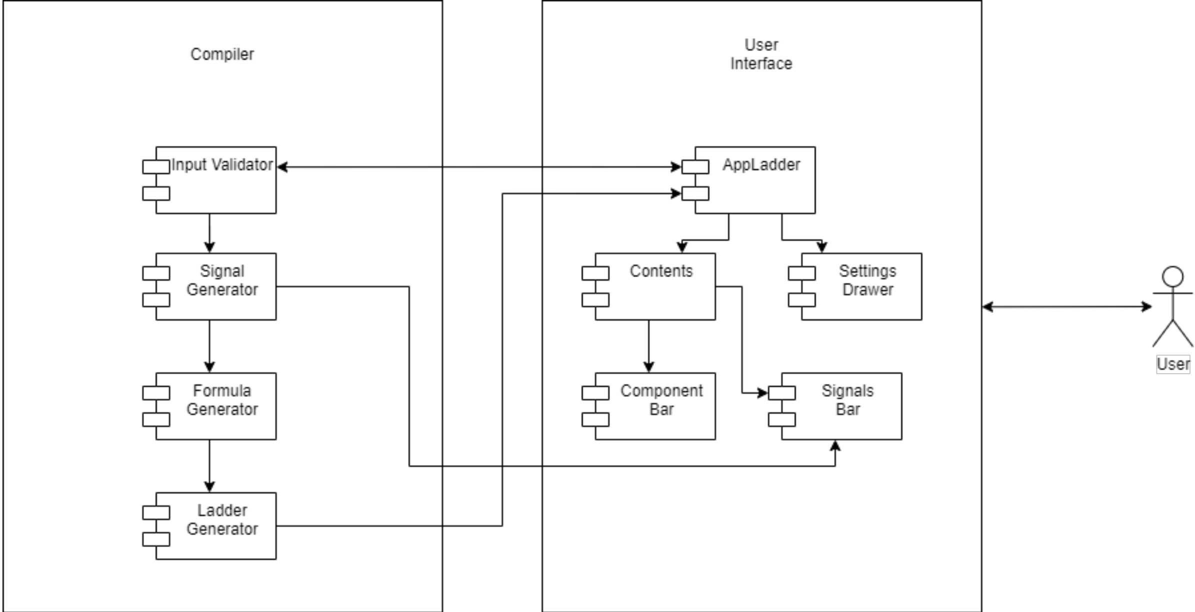


Figure 3.1: Application architecture

4
Product

The final product is the Stepladder application. Its layout and features are presented in this chapter.

4.1. User Interface

The User Interface (UI) is split into multiple parts that are consistently shown across the whole application. These parts can be seen in figure 4.1 and are explained in the sections hereafter.

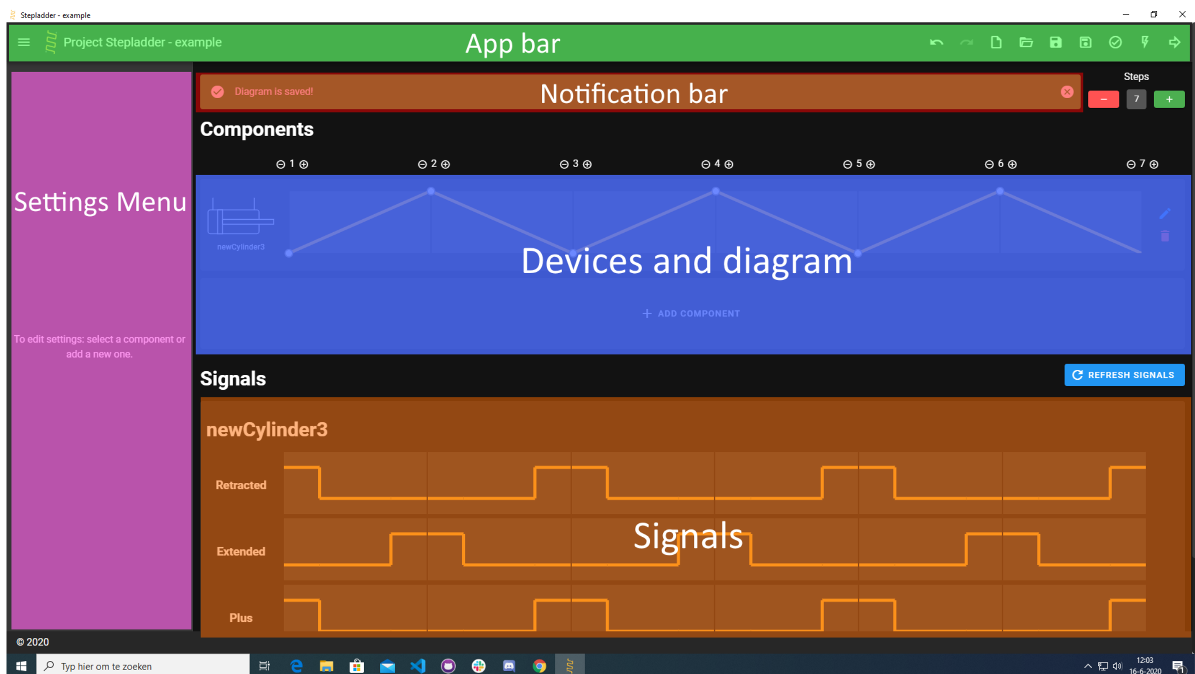


Figure 4.1: An overview of the different components of the UI

4.2. App bar

Figure 4.2: The app bar

The app bar is the bar with buttons that is always at the top of the screen. The app bar will show the user what file is currently being edited and it has the main buttons of the application. The button icons are the same as

in Ladderino to be consistent and all the buttons have a small tooltip that explains the functionality of that button. The buttons are also ordered in a logical way. It starts with the undo-redo buttons, which are greyed out when they are not applicable. Those buttons are available when a user wants to undo a previous action or actually redo something that was first undone. Next are the “new file”, “open file”, “save file” and “save file as” buttons, which is the same as, for example, Microsoft Office Word has them ordered. After the file operation buttons are the Stepladder specific buttons. These are ordered in the order that a user would use them. First is the “Validate diagram” button, that shows a window with warnings and errors of the drawn displacement diagram, see figure 4.3. After that is the “Generate signals” button, to generate the signals, which are shown beneath the drawn displacement diagram. The final button is the “Compile” button. This button compiles the diagram to ladder logic and automatically opens Ladderino to help the user.

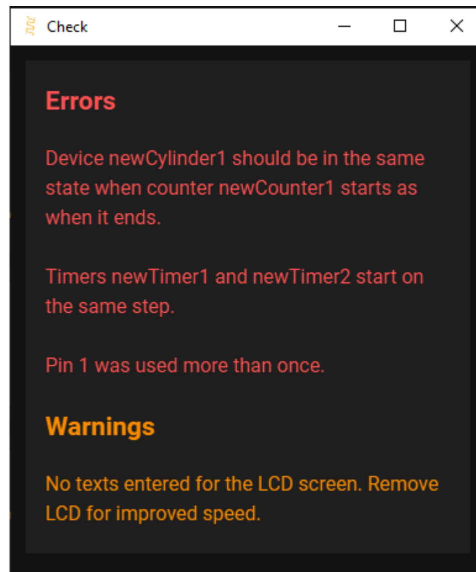


Figure 4.3: Check window with errors and warnings.

Clicking the Stepladder specific buttons invokes the compiler, which is not shown. It performs several tasks, which are done in a background task to improve speed and keep the UI responsive. The workings of the compiler are explained in section 6.3.

4.3. Devices and diagram

The devices and diagram part of the window is the most important part to look at as a user. The whole view is shown in figure 4.4.

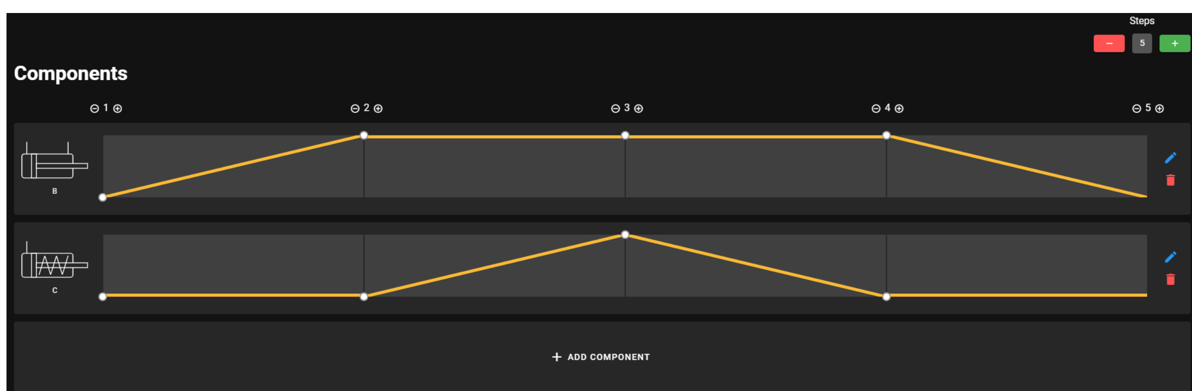


Figure 4.4: The devices and diagram part

This part of the window consists of one or more components, that each has an image and a line split up into multiple steps that can be moved to control the movement of the component. The image is a drawing of the type of output device and has a tooltip with the details of the component as shown in figure 4.5.

Changing the value of a step of a device is should be easy. The user can move the handle of the slider up and down and a tooltip will show what value is currently selected. This is shown in figure 4.6.

After the movement diagram, two buttons are located on the right side. These two buttons are edit and delete. The delete button will spawn a popup asking the user if they are sure to delete the component. Because a single little button to edit the component is not user friendly, any click on the dark-grey part will open the settings menu. It is easy to add or remove a step in the middle by clicking on the + or – at the top. Also adding or removing a step at the end can be done by clicking on the corresponding green and red button.

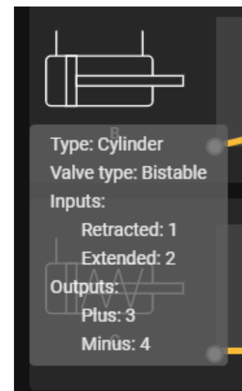


Figure 4.5: Tooltip of a cylinder with a bistable valve

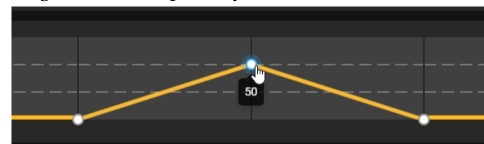


Figure 4.6: Slider of a motor with encoder with multiple states and tooltip

4.4. Signals

After creating a movement diagram, the user can generate the input and output signals of the components. One example of generated signals is shown in figure 4.7. These signals can easily be regenerated with the “Refresh signals” button, which is disabled when there are no changes. They are based on the diagram and change when the diagram changes and are an indication of the inputs, outputs and variables of the PLC.

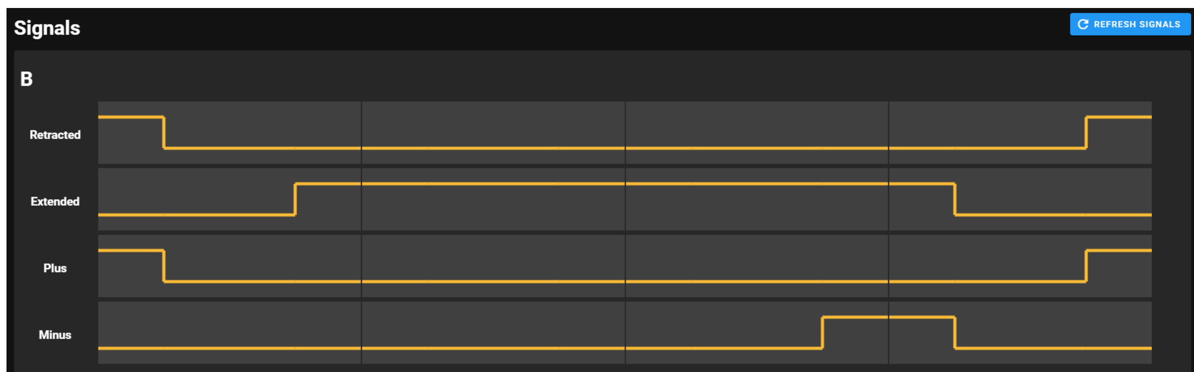


Figure 4.7: The signals part

4.5. Settings

To change the name or pins of a device, the user can click on the edit button and the settings view will pop up on the left side. These settings are different for each device and only the required input fields are shown. An example of a double-acting cylinder with a bi-stable valve is shown in figure 4.8. The fields are ordered in a logical manner. First the name of the device, then the inputs and outputs and after that the additional settings. When a user enters a value that is not allowed, the settings pane will alert the user that it is not allowed and the OK button will be disabled and greyed out. An example of an invalid input with an alert is given in figure 4.9. When a user clicks on the OK button, the settings pane will hide and a notification will be shown that the component is updated.

Figure 4.8: The settings part

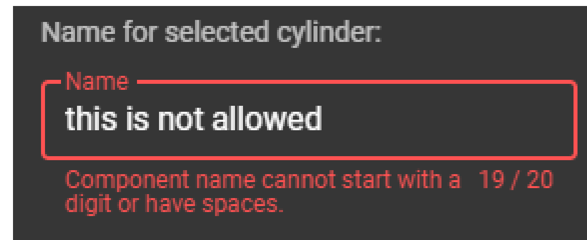


Figure 4.9: Alert of incorrect value

4.6. Shortcuts

To help the user perform actions more quickly in the application, shortcuts have been added. The shortcut combinations and corresponding action are all standard combinations in Windows [4]. This way the user does not have to learn new shortcuts. The shortcuts are shown in table 4.1.

Shortcut	Action
Ctrl + s	Save file
Ctrl + shift + s	Save file as
Ctrl + n	New file
Ctrl + o	Open file
Ctrl + i*	Check diagram
Ctrl + u*	Compile diagram
Ctrl + z	Undo
Ctrl + shift + z	Redo
Ctrl + y	
Enter	Apply settings
Enter	Hide settings
=	Add step
+	
-	Remove step
_	
0	Reset Number of steps

Table 4.1: Available shortcuts in Stepladder

The shortcuts with a * annotation are not usual combinations, but they are also in use in Ladderino and the Arduino IDE, so to keep it consistent, these key combinations are added.

4.7. Notifications

To indicate what is happening and what the application did, the notification bar is added. The notification bar is always visible, but the colours and text are only shown when there is a notification. To clearly communicate the intent and meaning of the alert, the common alert colours [5] are used as shown in the graphic in figure 4.10.

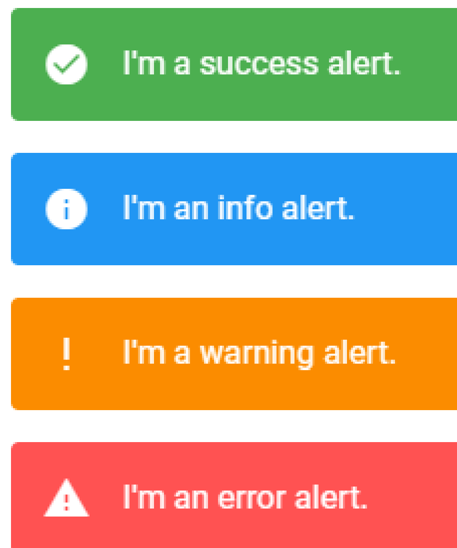


Figure 4.10: Notification colours

5
Process

In this chapter, the process of developing the product is explained. The process towards the final product involved four important aspects. These were the group division, the used development methodology, the communication and the SIG feedback. These will all be explained next.

5.1. Group division

During the project, the team was split into two groups. One group focused on creating the compiler part of the product (back-end), while the other group focused on creating the graphical user interface (front-end) of the product. Halfway, the compiler was almost finished. The group was therefore merged back into one group. However, when there was a task related to the compiler, the team members from the compiler group still performed this task.

To be able to work efficiently with this setup, a format was defined at the start of the project in which the back-end accepts the displacement diagram and the front-end stores the displacement diagram. In addition, the back-end's main functionality was finished first, before linking it to the front end. These two measures made it possible to work in two separate groups without having to wait on tasks that had to be completed by a team member from the other group.

5.2. Development methodology

For this project, it was decided to use Scrum, which will be explained first. After this, the way of tracking progress is explained.

5.2.1. Scrum

With Scrum, the project is divided into sprints [6]. For this project, sprints of two weeks were used. This means that in total there were five sprints. The first sprint was spent on research. A crash course pneumatics was given to the team by De Techniekschool and possible solutions were formulated. The next three sprints were spent developing the product. The last sprint was spent on the final report.

During the first sprint, an initial set of requirements was formulated. These were prioritised using the MoSCoW method as can be read in chapter 3. These requirements were then divided over the next three sprints. The second sprint was spent on developing the check functionality of the compiler, as well as the signal and command generation. In addition, the sprint was spent on implementing the ability to draw displacement diagrams and the ability to add components. The third sprint was spent on finishing all the must have and should have requirements. These requirements together formed the minimum viable product. The fourth sprint was spent on improving the application and finishing tasks that were not finished before.

This division of tasks was not final. At the beginning of all development sprints, requirements that were not implemented yet and had the highest priority became a possible candidate for implementation in that sprint. For every candidate requirement, tasks that had to be performed to implement the requirement were discovered. This was followed by estimating the amount of time every task would possibly take. Tasks were then added to the sprints backlog, a list of tasks to perform that sprint. If there were enough tasks picked for everyone to have enough work for two weeks, no additional tasks were added to the sprints backlog.

The tasks were not always directly assigned to a team member. It often happened that one team member was spending significantly longer on a task and another team member significantly shorter. Therefore, it worked well for the team to assign tasks that were best done by a specific team member to that team member and leave the other tasks unassigned. When a team member was done with his tasks, he assigned himself to a new task.

5.2.2. Tracking progress

De Techniekschool provided the team with a GitLab server for the project. For tracking the team's progress, this GitLab server was used. This was done at several levels.

At the highest level, milestones were used. For every sprint, a milestone was created. This allowed the team to track if sprints failed or succeeded. In addition, it allowed for a visual representation of the progress inside the sprint.

At a lower level, issues were used. At the beginning of each sprint when all tasks for that sprint were defined, an issue was created for each task. Each task was assigned the milestone of that current sprint. In addition, the tasks were assigned an estimated time for completion. This allowed the team to track the progress for separate tasks.

At the end of each sprint, the milestone of that sprint was closed and any incomplete issues were moved to the next sprint. A meeting was held to discuss the resulting product of the sprint. In this meeting, problems that occurred during the sprint were also discussed.

5.3. Communication

In this project, three types of communications are relevant. Communication with the coach, communication with the client and communication between group members.

5.3.1. Communication with coach

Communication with the coach was mainly done through weekly meetings. With the Coronavirus measures in place, it was not desirable to hold these meetings in person. These meetings were, therefore, held with Jitsi, a video conferencing tool.

5.3.2. Communication with client

At the start of the project, two meetings were held with the client at De Techniekschool. During these meetings, a crash course in pneumatics was given. In addition, the requirements of the project were discussed. There were no additional meetings held with the client. Instead, communication with the client was done indirectly. One of the team members spoke to the client regularly. If there was some question from the team's side, it was asked to this team member. The team member then asked it to the client, and came back with an answer. If there was feedback or a feature request from the client's side, it was also told to this team member. The team member then told the whole team.

5.3.3. Communication with group

Inside the team, every Monday, Wednesday and Friday a meeting was held. For every meeting, a shared document was made with a list of things the team members wanted to discuss. This made sure that the meetings were fast and efficient. In addition, it served as a reminder for team members that they wanted to ask or discuss something. During the meetings, all points from this list were discussed. However, before doing so, the current tasks that the team members were performing were discussed. Sometimes tasks took longer than was expected and a plan could be made to help the person with performing the task.

5.4. SIG

At the end of the third sprint, the code from the project was sent to SIG. SIG then analysed the code using certain metrics. During the fourth sprint, these metrics were used while refactoring the product to improve the code quality. What this feedback was and how it was resolved can be found in appendix C.

6 Implementation

The aim of this chapter is to explain the implementation of the final product. First, the used frameworks in the front end are explained. This is followed by an explanation of the implementation of the user interface. Finally, the implementation of the compiler is explained.

6.1. Frameworks

A software framework is useful as it “provides a foundation on which software developers can build programs for a specific platform” [7]. Predefined classes, functions and other functionality can be used without having to program it yourself, making frameworks more time-efficient, allowing best practices and overall streamline the code base. For these reasons, we looked at several frameworks and picked a few which suited our needs.

6.1.1. Electron

As the main framework, it was decided to use Electron[1]. This is a framework that uses HTML, CSS and JavaScript in conjunction with Node.js to create applications which can both be used on web pages as well as stand-alone desktop applications. The reason why this framework was chosen is because the client has a possible desire to run the application on their website in the future and their current program Ladderino is also written in Electron making it easier to maintain and implement and easier to develop for multiple operating systems.

6.1.2. Vue

Electron uses JavaScript for the interactivity of the application, which allowed for the usage of a JavaScript framework for making standard behaviour easier to implement. For JavaScript, there are several frameworks like React, Angular and Vue.

Some research showed that, although each framework has its advantages and weaknesses [8], it is ultimately to the user’s preference. Within the group, there was already some knowledge on Vue[2] and its component-based structure seemed ideal to have everyone work on a separate piece of the application. This is ideal as later on, the code can easily be combined to create our application. For these reasons, Vue was chosen as the UI framework.

6.1.3. Vuetify

To make creating a user interface easier, there are several frameworks and libraries that provide styled elements that can be used to more quickly develop a good looking application. Since the application will use Vue, some time was taken to look into frameworks that could provide styled Vue components. The ones most common were Vuetify, BootstrapVue and Buefy[9]. Vuetify[3] ultimately looked best as it has a large collection of components and the material design theme looked good to the client.

[1] https://www.electronjs.org/
[2] https://vuejs.org/
[3] https://vuetifyjs.com/

6.1.4. I18n

The initial language for the application is English, because Ladderino is currently also in English. To accommodate for Dutch and other language speaking users, i18n[4] was used. I18n is a common plugin used to enable internationalisation by using translation files. This plugin was chosen because it is the most used internationalisation plugin. To start on internationalisation, only some texts were added to the language file, but more can be added easily.

6.2. User Interface choices

This section will discuss the choices that were made for the user interface, which itself is described in section 4.1.

6.2.1. App bar

The app bar, see figure 4.2, has a hamburger menu icon on the left side to display to the user that they can also manually open the settings drawer. Next to this hamburger icon is the Stepladder logo. This logo is included as it provides a reason for the diagram lines to be the yellow/gold-ish colour. It also functions as a brand image which can influence the users to relate the logo with this stepladder application. The app bar also includes the currently opened file, so users know what they are editing.

Grouped on the right side are the buttons that provide specific functions, as described in chapter 4. These buttons are in a certain order. This order represents the logical workflow as much as possible. To begin working on a displacement diagram, a new or existing diagram needs to be opened. During the process of designing the diagram, the diagram will be saved. When the diagram is finished, the diagram has to be checked first. Therefore, the check button is first. After checking, the signals have to be generated for being able to compile the diagram. After generating the signals, the diagram can be compiled. In addition to performing the button's own functions, the Stepladder specific buttons perform the functions from the preceding buttons. This means that the check button only performs the check, the generate signals button performs a check and generates the signals and the compile button checks the diagram, generates the signals and compiles the diagram.

6.2.2. Devices and diagram

The main objective of the application is to easily be able to create displacement diagrams. Creating these displacement diagrams should, therefore, be as intuitive and user-friendly as possible, so everyone, even those with a little bit of knowledge on pneumatics, is able to generate these types of diagrams.

The potential options for drawing these diagrams were as follows. Either the user could drag & drop tiles, the user could draw by clicking on canvasses, the user could actually draw on the canvasses, or the user could draw diagrams by adjusting sliders.

Drag & drop

One of the possibilities was a drag and drop system. In this option, the possible step transitions are to the left and the user is able to drag each one onto the diagram. An example is given in figure 6.1. The reason behind this idea is that it is similar to the way Ladderino works, because that uses a drag and drop system as well. This would give the user uniformity in the tools they are using.

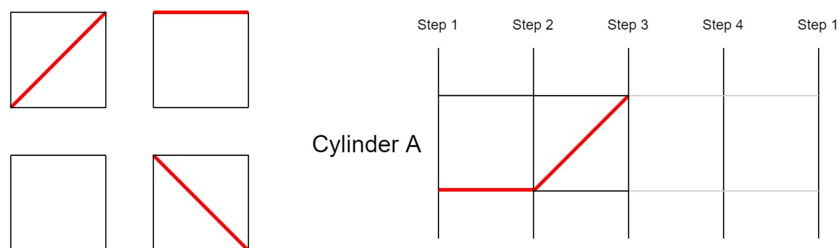


Figure 6.1: Example of drag and drop system for diagram drawing

[4]https://www.npmjs.com/package/i18n

Clickable
The second option is a variation on the drag and drop system. The user can click on a square resulting in a change to the next one. These squares are on a rotation as described in figure 6.2. The benefit is that the user does not have to drag each element to its position and can just click each element to its desired state. The downside from this system is that it can be frustrating when you accidentally click one time too often and you have to cycle all the way back. When the user wants to draw a signal that is high, he or she has to click several times for each step, which takes a lot of unnecessary and annoying clicks.

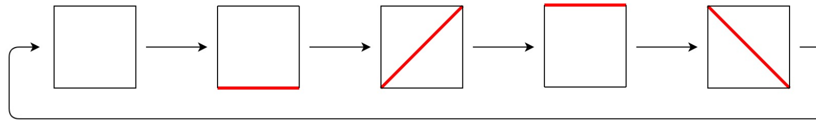


Figure 6.2: Rotation of diagram states in the clickable solution

Drawable
A third option was to let the users draw the diagram themselves. This is shown in figure 6.3. The user can connect points on the diagram, meaning it is easier to draw the diagram, since the user does not have to drag each element one by one. It also gives greater flexibility to the program when it comes to motors as they have multiple states they can be in. The drawable diagram is, however, much harder to implement as it needs to be highly responsive and give immediate feedback to the user.

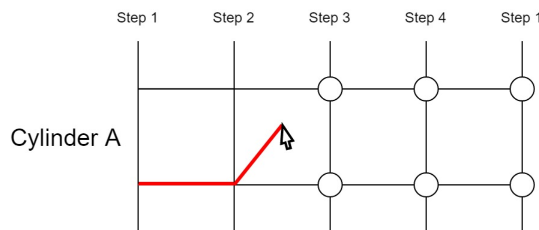


Figure 6.3: Example of the drawable diagram

Sliders
The fourth option was to let the users draw the diagram by pulling sliders up and down. The only interaction that is then needed is just simply dragging the sliders. This is very basic and simplistic, making it a great option for usability. Figure 6.4 shows the slider design with large handles that the user can click. This design also assures that all diagram lines are always connected.

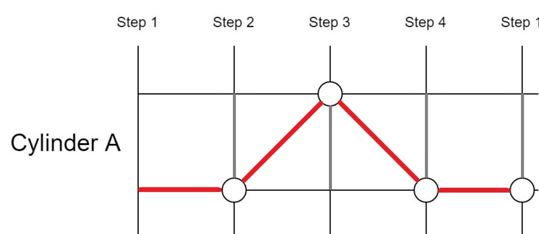


Figure 6.4: Example of the sliders diagram

Choice
The first option of drag & drop strives for uniformity between both programs of De Techniekschool pleasing Jakob's law which states that users will take expectations, they have built on an other site, to yours [10].

The second option of clicking suffers from the fact that the time required is dependent on how many times a user has to click. This still means that the user needs to go over every square and has to click several times per square. In the first prototype of the application, clicking was implemented as it was easiest to develop, but it did indeed need too many clicks, which was the reason why it got dropped.

The third option of drawing on canvasses aims to be easier in use. This is inspired by Fits's law which states "The time to acquire a target is a function of the distance to and size of the target"[11]. Meaning that the time

that is required for an action is influenced by the distance a user has to travel. Because of this law, the dragging & dropping option was not even considered and initially we chose this third option to be implemented. This implementation worked well for the first couple of weeks until a better solution was thought of.

The final option of using sliders to draw the displacement diagram is ultimately what we ended up with. Compared to drawing on canvasses, our initial pick, it provided several advantages. First, the lines are always connected and thus the user cannot draw the diagram incorrectly anymore. Second, sliders are easier for adjusting the diagram compared to redrawing a part of the diagram. Finally, drawing on canvasses had quite complicated code, which did work, but had the occasional bugs. Sliders are more simplistic and made our codebase less complex.

6.2.3. Signals

The signals in the UI are underneath the components as this is common practice regarding displacement diagrams. The signals are grouped per component with the component name at the top in its own bar and the signals are presented with a yellow/gold-ish line similar to the drawn displacement diagram. We chose this layout to be similar as it gives recognisably to the user, design-wise it is good-looking and it is simplistic overall, making it not too confusing. Furthermore, the signal name is also included on the left side, to indicate which signal belongs to which line, which also leads to a better understanding of the generated signals. The user ultimately has to learn how pneumatics works and these signals are a major part of it.

6.2.4. Settings

The settings are grouped in a collapsible sidebar on the left side of the application. Here the settings per component always start with the name related to the component and then it is followed by the settings to fill in the pins. Underneath this, various other settings can be found which differ per component type. The settings bar is tried to be kept as compactly as possible, since we wanted to assure that there would not be a scrollbar in the settings. A scrollbar had shown up once during the development and we disliked it very much, thus we crammed the settings closer together to make sure it would not show up again.

6.2.5. Notifications

The notifications are taken from Vuetify.js as those were good-looking, informative to the user and overall very flexible. Most notifications in the stepladder application show up for only 2 seconds as that is long enough to read them and not be annoying. They are also dismissable, so the user always has the option to close the alert at any time. Just a couple of notifications which contain more text show up for 4 seconds, as they would close before the user could have read the whole text. And regarding the text in the notifications. They are kept as short and accurate as possible.

6.3. Compiler

In this section, the architecture of the compiler is discussed. To make the compiler, it is important to know how the "compilation" is done manually. This will be explained first. Next, a way to automatically compile displacement diagrams into ladder logic is explained. This conversion consists of multiple stages and each is explained separately.

Besides the normal devices like pneumatic cylinders and motors, timers and counters have to be supported. In this context, a timer adds a delay, to ensure that a step takes at least a certain amount of (milli)seconds. These timers can, for example, be used to have a continuous motor run for some amount of time, like a shredder. A counter is a special component that repeats a set of steps for a certain number of times. This is, for example, useful for a machine that drills holes every 5 cm. It can do the drill steps, the move steps and repeat them.

6.3.1. Manually

Converting a displacement diagram to ladder logic or a relay schematic is mostly done manually. This is because it consists of clear steps. To illustrate the steps, a displacement diagram is shown in figure 6.5 (the "•" means "and"). The steps to create a ladder logic diagram are as follows:

1. Draw the movements of the cylinders.
2. Add a start signal and other buttons.
3. Draw all the signals of the end sensors (b0, b1 ... d0, d1).
   _0 is high when the cylinder is retracted, _1 when it is extended.
4. Check that there are no steps with exactly the same states of cylinders/switches (except first and last step).
   If two or more states are equal, add a memory unit with different values at the same steps, this way the steps are different from each other. The memory unit can be high or low. In the example, G1 is added to make step 3 different from step 5, and step 2 different from step 6. Add memory units until no more steps are the same.
5. Draw the signals of the output. Bi-stable valves can be powered very shortly (0.2s, but this can be longer). Mono-stable valves must be powered the whole time the cylinder has to be in the alternative position.
   Every bi-stable valve needs two outputs (X+ and X-) to make the cylinder move both ways. A mono-stable valve needs a single output.
6. Find the logic formulas for the output as functions of the signals of the switches and memory units.
7. Create ladder logic with the logic formulas.

This is the best and the most accurate method and it can also be used to create relay diagrams, which allow for using relays instead of PLCs.

With ladder logic, a step counter can be used, which can greatly decrease the difficulty of finding the logic formulas. This method keeps count of the current step as an integer. The counter is increased when the conditions (or changes) from the previous step are met and the step counter is one lower than the current step. For Step 4 in the example, this would result in the formula: $d1$ && $counter == 3 \rightarrow counter = 4$.

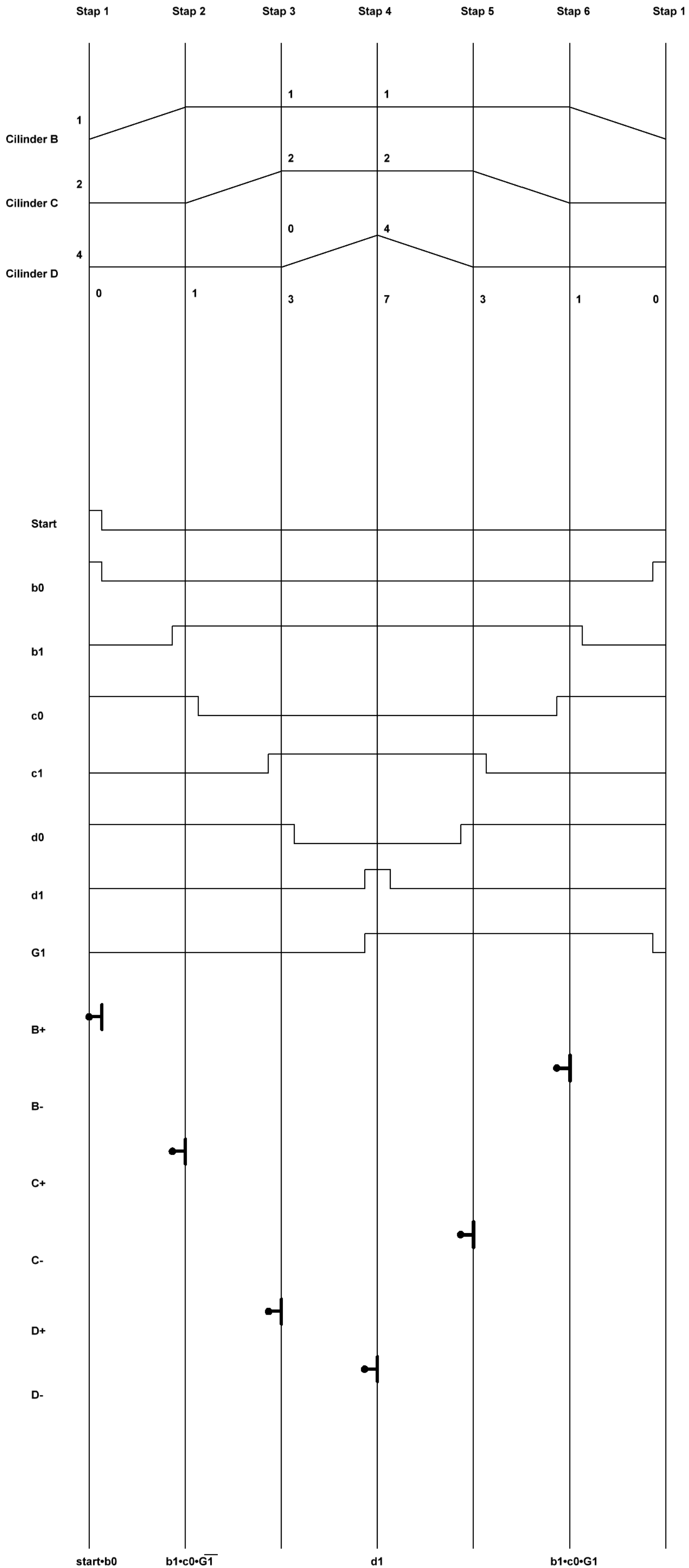


Figure 6.5: A displacement diagram

6.3.2. Automatic

In this section, it is explained how converting a displacement diagram to ladder logic can be done automatically. The first step in this conversion is to check whether the displacement diagram is valid. If it is not valid, it is useless to try to convert it. Next, as in the manual case, signals need to be generated. These signals help in generating the formula which is done next. With these formulas, the ladder logic can be generated.

Moreover, an explanation of all individual stages is given. This explanation uses the displacement diagram described in figure 6.6. In this diagram, cylinder B represents a double acting cylinder with a bi-stable valve and cylinder C is a single acting cylinder with a mono-stable valve. The movement is the same as in figure 6.5, but without cylinder A.

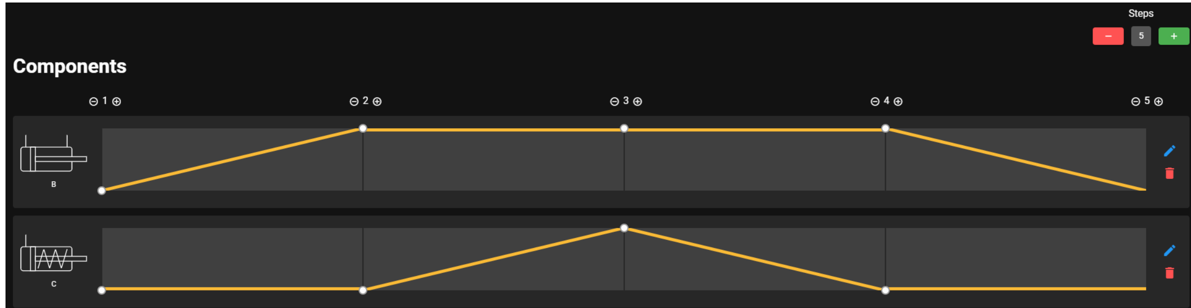


Figure 6.6: Displacement diagram used in explanation

Check

First, the compiler will need to check if the input displacement diagram is valid. This allows the other parts of the compiler's logic to purely focus on converting the valid diagram into ladder logic. Some requirements for a diagram to be valid are:

- The start and end of the diagram should have the same state (since the sequence repeats).
- The start and end of each counter should have the same state.
- Each input and output pin can only be used once.
- If counter A starts first and then counter B starts, then B must end before A ends.
- All devices have the same number of steps.

These are not the only requirements that will be checked in this step. There are also a lot of technical details that do not matter when drawing a diagram on paper, but are important when compiling the diagram to actual code. For instance names (for devices, buttons, etc.) cannot be reused and numbers must fit in 16 signed bits (a limitation of the PLC). The full list of checks can be found in appendix E.

Signal generation

When the diagram is checked, the signals of the inputs and outputs need to be generated. This is mostly for the user, such that the user can see why a certain formula is generated, but also for generating the formulas, because the signals of the inputs are used. The output signals are not used in the compiler, because each device type has its own compilation process and output types. Because the devices take some time to react to an input, each step gets 2 values. One value is on the line. The other value is between two steps. Just before and just after a step, the value is the same, but between steps, the values can differ. To show it in the UI, the "on" is 25% before and 25% after the line, leaving 50% for the "middle" value.

Each type of device has a different signal generation function, because every device type has different inputs/outputs and how they should be read/powered. For instance, a bi-stable valve has 2 inputs and 2 outputs, adding up to 4 signals, whereas a mono-stable valve has 2 inputs, 1 output and set/reset signals, adding up to 5 signals. Also, a motor with end-switches has a different output signal behaviour than a bi-stable valve, even though they have the same behaviour and look almost the same in the output.

For counters, some extra signals are shown, namely, the increment signal and the reset signal. The increase of the counter can be done at every step in the sequence, but to make it consistent, it will be done at the last step of the counter range, when the conditions are met and the counter has not repeated yet. This is

to allow nesting of counters, because when there are multiple counters with the same end step number, all the counters would be increased when the counter increase is triggered on the last step. A counter will be reset when the counter is done and the starting step is done. This is to allow counters with the same start step to work correctly.

For the example drawn in figure 6.6, the generated signals are shown in figure 6.7 where the "Retracted" and "Extended" signals are the end switches of the cylinder and "Set" and "Reset" are the signals for the latch that controls the "Plus" output signal.

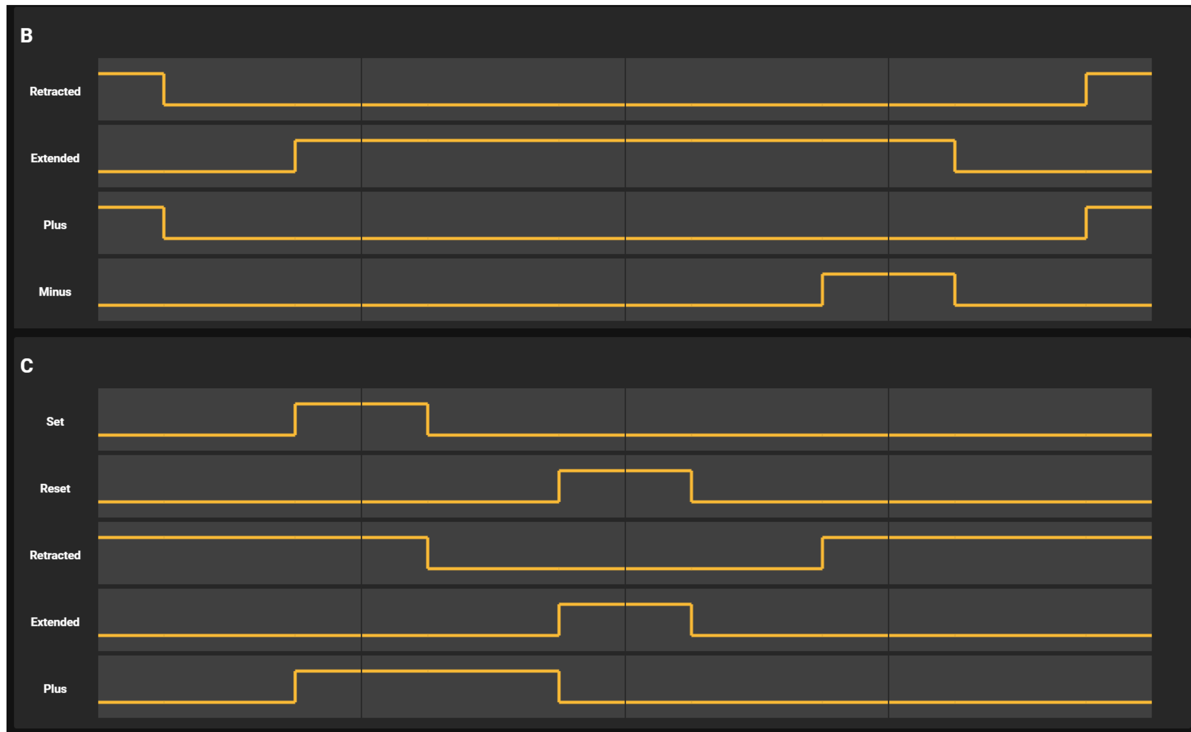


Figure 6.7: Signals used for explanation

Formula generation

After the diagram has been checked for correctness and the signals are generated, the diagram has to be converted to logic formulas. For the MVP, it was chosen to use counters to control the current step. For every component, one or more formulas need to be generated. A formula will be a sum of products together with an output. For example, $((a\&\&b)||(c\&\&d))$. These formulas will not be reduced in Stepladder, to make it understandable for the user. However, reducing and optimising will be done by Ladderino. For every step where an output needs to change, the counter (optionally with the input that will change) corresponding to the step will be used in the formula for that step. For every step, the counter needs to change. The formula for this will be *(list of changed inputs ANDed)* $\&\&count = step - 1$. Counter increment uses a variable put into a trigger to make sure it always triggers, but also that it does not trigger multiple times. When a counter is not yet done, the step counter just resets to the start of the counter. Timers will only need the *step count* == *step*. The output (done variable) of the timer will then be used in increasing the step count.

The example generates the following formulas:

```
v_counter==0 • B_0 → o_B_p
v_counter==3 • B_1 → o_B_m
set: v_counter==1 • C_0 reset: v_counter==2 • C_1 → latch o_C_p
B_0 • !B_1 • C_0 • !C_1 • v_counter==-1 → set v_counter = 0
B_1 • v_counter==0 → set v_counter = 1
C_1 • v_counter==1 → set v_counter = 2
C_0 • v_counter==2 → set v_counter = 3
B_0 • v_counter==3 → set v_counter = -1
```

The first two formulas are for the bi-stable valve. When the counter is at 0 and the cylinder is still at the 0 position, then the solenoid of the valve needs to be energised because the valve will stay at the pushing position. The same holds for the retracting (0_B_m) output.

The third formula is for the output of the mono-stable valve and uses a latch. The other 5 formulas are for controlling the step counter. The first step formula is to make sure all the cylinders are in the correct position before moving.

In the first step, only the B cylinder moves, so the end condition for that step is only reading the end switch of that cylinder.

Code generation

After the generation of the formulas, the components are no longer needed and only the formulas are used. This is to separate the two parts of the compiler and make it possible to interchange a part or add a device without changing the whole compiler. The Ladderino code generator gets a list of formulas, and for each formula, it adds a new rung. This rung will have an output block and one or multiple lines with input conditions for the output. For each input and output of each device, a corresponding input or output is generated for the ladder logic and they will be used for the conditions and actions blocks. To allow the user to learn what the compiler did and learn how to do it themselves, each rung gets a comment with a readable text that explains what the set of conditions is.

The generated ladder logic of the example can be seen in figure 6.8.

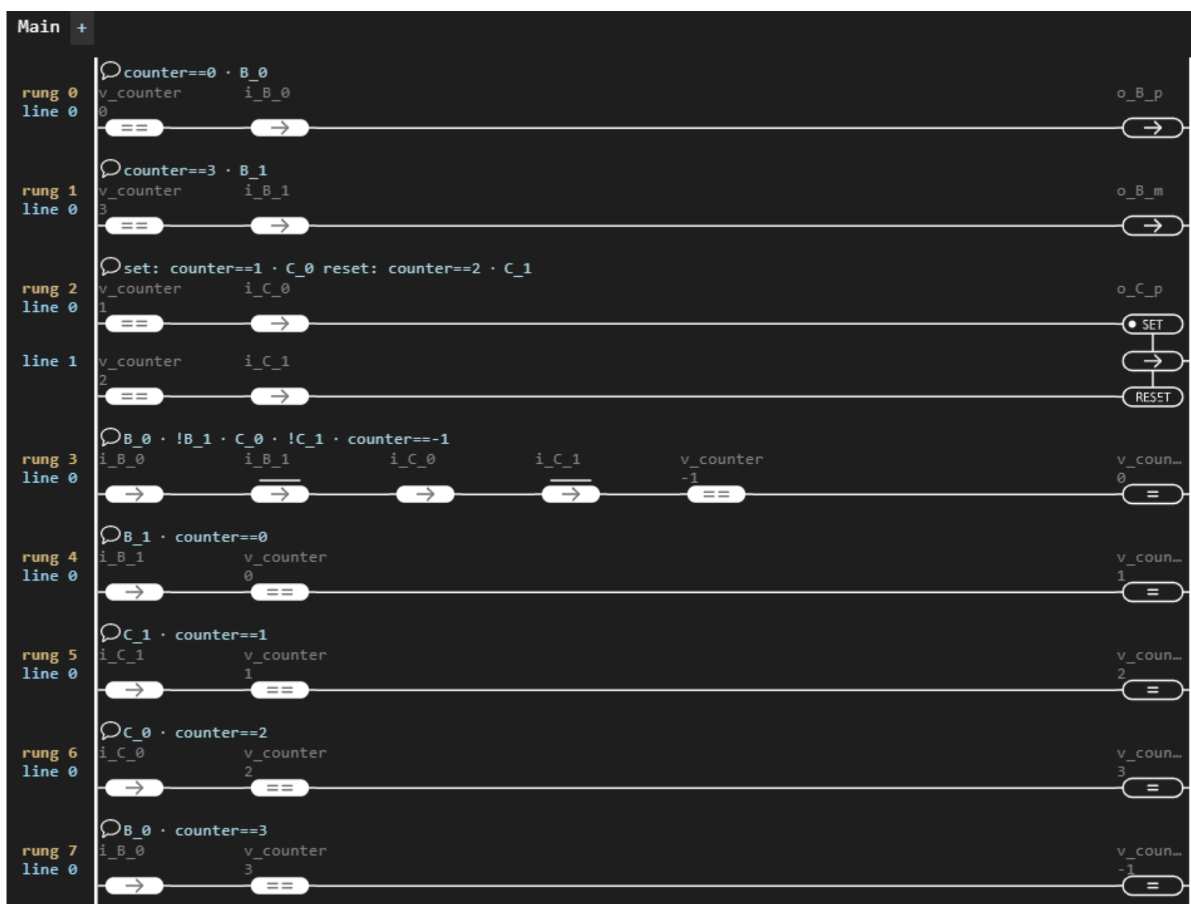


Figure 6.8: The generated ladder logic

7 Code quality

For improving the quality of the code, different methods were used. In this chapter, these methods are discussed. First, the use of continuous integration is explained. Next, the strategy used for testing the code is considered. Last, the use of code reviews is elaborated on.

7.1. Continuous integration

An important part of quality control for this project was the use of continuous integration. Continuous integration made sure that all tests were automatically run each time a change was made to the code. In addition, it made sure that every team member adhered to three rules. For the continuous integration, GitLab CI was used.

The first rule was that every function or component should be documented. By enforcing this, important information about functions and components was always available to all team members. This avoids misusing functions or components caused by a team member that misinterprets the functionality of the function or component.

The second rule enforced that the static analysis phase should not result in warnings or errors. What this phase involved is explained in the next section. In short, it checked for certain bad coding practices and anti-patterns. By not tolerating any warning or error, the team members were forced to rethink the use of this bad coding practice or anti-pattern.

The third rule that was enforced made sure that a certain test coverage percentage was achieved. This rule helped to ensure that every team member put effort into testing their functions or components. A threshold of 80% was enforced for line coverage, statement coverage, function coverage and branch coverage. The team found that this coverage threshold was easy to achieve, while still making sure almost all code is covered by some tests.

7.2. Testing

Testing was done in two phases. The first phase was the static analysis phase. This phase analysed the code without running it. The second phase was the dynamic testing phase. This phase tested the code by executing it.

7.2.1. Static analysis

Static analysis analysed the code to ensure it conforms to a specified style. This resulted in the use of a uniform code style, which made sure the code was readable. Later in the project, it was found there was a stricter recommended style by the developers of Vue[1]. Instead of only following the essential Vue style, the static analysis phase was adapted to also check for the essential, strongly recommended and recommended Vue styles.

[1] https://vuejs.org/v2/style-guide/

7.2.2. Dynamic testing

Dynamic testing was done using automated test suites. These test suites were run automatically by the GitLab CI. Writing the tests was done in two parts. One part tested the back-end and the other part tested the front-end. For both parts, the coverage thresholds mentioned before were enforced separately.

The back-end was tested using unit tests. In addition, one of the team members went to De Techniekschool a few times to test the compiler with actual hardware. Different arbitrary test setups were created, and different diagrams were then drawn and translated by the compiler. It was then tested if the output from the compiler was correct by putting the ladder logic on a PLC, using Ladderino, and observing if the test setup performed as expected.

The front-end was also tested using unit tests. Mainly the logic of state management was tested with these tests. The actual visible layout of the user interface was tested by manual inspection.

Integration tests and system tests were not performed. There are two reasons for this. First, there were problems with writing these tests. A setup was made to run the tests, but when writing the actual tests, they failed. This could have been solved by doing more research. However, the second reason was that the project is relatively small. It could be tested very fast manually. This lead to the decision to not put more time in getting this to work, as there was a big chance the time that would have been spent on getting it to work and writing the tests would have been more than the time that was spent on manual testing and extra debugging.

7.2.3. Testing utilities

For testing the code, different frameworks and libraries were used.

ESLint

ESLint[2] is a static analysis tool that analyses the code without running it. It can check, for example, if the code was correctly formatted and also fix it automatically if it was not. It can also check if the code adheres to the earlier mentioned Vue style guide.

Karma

Karma[3] is a test runner. It can run tests in multiple environments, such as browsers or Electron. For this project, it was used to run Mocha tests in an electron environment.

Mocha

Mocha[4] is a JavaScript testing framework, which provides the ability to write tests for the created code to check whether functions have the expected functionality and perform correctly. This testing is done by means of assertions which test whether a condition is fulfilled or not.

Chai

To make the assertion statements more readable, the assertion library Chai[5] is commonly used. Chai allows three different assertion styles to be used for writing clearer assert statements. These three styles are “should”, “expect” and “assert”. This project uses the “expect” style as we believe that this is the most comprehensible and straightforward form. Instead of assertions such as `assert.equal(foo, 'bar');`, they can be written as `expect(foo).to.equal('bar');` with Chai.

Sinon

Sinon[6] provides standalone test spies, stubs and mocks for JavaScript. Spies can be used for monitoring and verifying function behaviour in tests. Stubs allow modifying function behaviour and replacing difficult to test code and mocks are fake methods. Sinon thus enhances the testing procedure and allows for higher-quality tests.

Istanbul

Istanbul[7] is a test coverage reporting tool. It instruments JavaScript code with line counters, such that it is possible to track how well the unit-tests exercise the code. It takes line, statement, branch and function coverage into account and displays what percentage of code per file is covered.

[2] https://eslint.org/
[3] https://karma-runner.github.io/latest/index.html
[4] https://mochajs.org/
[5] https://www.chaijs.com/
[6] https://sinonjs.org/
[7] https://istanbul.js.org/

7.3. Code review

For this project, all code was first reviewed before merging it with other code. These reviews reduced the chance that a team member forgot about something. For example, it could be that some important test cases were not written. When the code was then reviewed by other team members, they checked if all important test cases were written. If not all of them were written, the team member was instructed to write them. In addition, it helped with getting a good design for the project. Both the actual looks were reviewed as well as the architecture of the project.

To be able to do the code reviews, all functionality was implemented in its own git branch. When the functionality was finished, a merge request was created on GitLab. Before merging the request, the code was reviewed by at least two other team members. When all feedback was resolved, the functionality was merged.

All code was first merged in a branch called dev. This branch was never removed. At the end of each sprint, the dev branch was merged into the final branch named master. Before doing this, the project was manually tested by all team members. To make sure no team member violated this strategy, the master and dev branches were protected for pushing changes to directly. Changes could only merge into these branches through merge requests.

8 Reflection on Ethics

In this chapter, reflection is given on how well the Stepladder application conforms to the goals set in chapter 3.

8.0.1. Trust

The code of the application has achieved a test coverage of, on average, 95% in all categories, which is better than the minimum goal of 80%. It is in practice not possible to know whether an application is completely bug-free, but we believe our application is quite reliable.

8.0.2. Privacy

As predicted, the application does not require any personal information and thus it does not collect or store any. Therefore the application respects the user's privacy.

8.0.3. Accessibility

Colour blindness

The colour theme of the application is mostly grey-scale and the colours that are used clearly contrast with the background. We have tested that the contrasts are clear using www.color-blindness.com[1]. In places where colour coding of UI elements has been used, like the buttons for adding and removing steps being red and green, there is clear labelling with symbols or text. One possible improvement that could still be made is to allow the user to select a colour scheme of choice for these kinds of elements.

Motor and dexterity impairments

The application is almost entirely accessible through the keyboard except for the sliders that control the displacement lines. This is due to the Vue library used for these components not supporting tab navigation. These sliders could be replaced with Vuetify sliders, which do work with this kind of navigation. However, we have not had the time to do this. The application is usable entirely with a mouse, with the exception of text boxes for entering names and pins. There, users that are unable to use a keyboard will need to use the on-screen keyboard built into their operating system.

Dyslexia

We have chosen the font "Source Sans Pro" because it is an open-source sans-serif typeface that is well suited for user interfaces. It is a simple and clean font that we think is easy to read.

Language

We have not had time to implement localisation or translate our app to Dutch. Because most Dutch high school graduates have at least basic knowledge of the English language and the application contains many icons, we think that this will not be a very large problem, though it is something that could be improved.

[1] `https://www.color-blindness.com/coblis-color-blindness-simulator/`

9 Discussion and recommendations

In this chapter, the main mistakes and future improvements for this project are discussed.

9.1. Discussion

During the project, we got to bring a lot of the skills that we learned in our Bachelor into practice. There were some things that we profited from, such as the importance of planning and effective and frequent team meetings. There were also some hurdles along the way which we would do differently in future projects. The main mistakes which could have been prevented are described below.

9.1.1. UI Mocks

Although we did make a couple of mocks of potential UI designs, they were not detailed and only used as a general guide. The mocks did help with getting everybody on the same page regarding the layout of the application. However, because we were eager to start, we did not create a detailed mock. This caused a number of design changes in the middle of the project. These changes could have been prevented if we devised a more detailed mock at the beginning of the project.

9.1.2. Code style

Another problem, we found out halfway through the project, was that we had not discussed all parts of the code style we would use. We had agreed to write JavaScript in a certain style but we had no clear guide on how to write the JSDoc. Also, line endings were a problem where one person committed using LF and the other using CRLF. This marked all lines in a file as changed instead of the actual changed lines, which made the merge requests unreadable.

In future projects this is definitely something that needs to be discussed beforehand, as it is only a small decision, yet makes the review process a lot easier and faster and the code style more uniform.

9.2. Recommendations

Since we only had six weeks to develop the application, there are still some potential additions that can be made. Possible future development will be described in the next sections.

9.2.1. User testing

Before the application can be deployed and used as teaching material on schools, it could be beneficial to perform user tests. Due to the short time frame of the project, there was no time for an extensive usability test. In the process, the program was validated by employees of De Techniekschool. This gave some valuable insights into the usability of Stepladder. However, the employees of De Techniekschool already know about pneumatics, so it would be valuable to test the product with people that are less experienced in that field.

9.2.2. More advanced compiler

Currently, the ladder code generated by Stepladder makes use of a counter to distinguish between steps. Although this is a valid solution, it is also quite basic. A more complete solution would be to define each step

as a condition and create ladder logic from there. This method is also described in section 6.3.1. This would not only make the generated ladder logic more readable, but it would also fit better with the course material from De Techniekschool.

The system was designed with this intention in mind. Instead of generating additional counter constraints that are added to make each step unique, the formulas of each step need to be compared. If there is a duplicate state, both states can be made unique with the help of a variable.

9.2.3. Multiple languages

In the initial research phase, it was decided that we would develop the application in English. This choice was made since Ladderino is also in English. If the application will be used in schools, it might be more appropriate to have everything in Dutch. This means that the app should support multiple languages.

To account for this we already made use of a package called i18n. This package makes it easy to add more languages. For each language, a dictionary can be defined which translates certain words or keys into other words or sentences. Currently, such a dictionary is only defined for the English language, but it is easy to add another language by adding another dictionary. When this is done, there must also come a way to change the current language. Once this is done, it is very easy to support as many languages as needed.

10 Conclusion

In this report, a program was proposed and produced for automatically translating a displacement diagram into ladder logic for Ladderino to use. This program will be used for educating people to program PLCs. The main goal of the program is, therefore, not translating a displacement diagram in the best possible way to ladder logic. Instead, it is translating a displacement diagram to ladder logic in a way that can teach people how to do this themselves. From this, it follows that the product should meet three design goals. First, the product should generate correct ladder logic. Second, users should be able to learn how to use the product in four hours. Third, the generated ladder logic should be readable by the user.

These goals were achieved by formulating requirements and implementing them. These requirements were categorised by priority using the MoSCoW method. Requirements were put in one of the following categories: must have, should have, could have, won't have and non-functional requirements. All requirements in the must have and should have category have been implemented, as well as the requirements in the non-functional category. The requirements in the must have category involved the ability to draw a displacement diagram, opening and saving this diagram and compiling the diagram to ladder logic for Ladderino to use. The requirements in the should have category involved the ability to add other components than only cylinders, such as motors, counters and timers. In addition, there was a requirement about the ability to undo and redo changes made to the displacement diagram. Requirements in the could have category were not implemented. These were not implemented, because there was not enough time. These requirements involved the ability to compile to other programs, giving suggestions to the user and explaining the generated ladder logic. The requirements in the won't have category were also not implemented, as these were not supposed to be implemented. This involved following a walkthrough through the application.

To realise the product, there were two parts that needed to be developed: the compiler and the user interface. The compiler consists of four stages. First, the compiler will check whether the displacement diagram is valid. Next, the compiler will generate signals. With these signals, the compiler generates formulas. Last, these formulas are converted into ladder logic. For this conversion, an approach with counters was taken. The counter will increase at every step of the displacement diagram, making sure that each formula is unique.

For the user interface, there were four approaches discussed for drawing a displacement diagram. The first approach involved a drag-and-drop system. The possible lines in the diagrams are dragged from a menu into the diagram. This approach is similar to how Ladderino solves the problem of drawing ladder logic. The second approach involved a clickable canvas that draws the lines for the diagrams by repeatedly clicking. While this can be easily implemented, the user would have had to click a large number of times, which is not ideal. The third approach involved directly drawing the diagram on the screen. This is more intuitive as it resembles drawing the diagram on paper. The fourth approach involved using sliders, which can be used to slide a step to a new state. This approach is used in the final product.

In conclusion, the developed product accomplished the design goals. All requirements from the must have and should have categories are implemented, as well as the non-functional requirements. It would have been better to perform user tests while developing the program. In addition, the client could add a more advanced compiler to the product as well as add support for multiple languages.

Bibliography

[1] De Techniekschool. (2020). De techniekschool dienstverlening, [Online]. Available: https://detechniekschool.nl/over-ons/over-ons/ (visited on 04/29/2020).

[2] L. Rello and R. Baeza-Yates, "Good fonts for dyslexia", in Proceedings of the 15th International ACM SIGACCESS Conference on Computers and Accessibility, ser. ASSETS '13, Bellevue, Washington: Association for Computing Machinery, 2013, ISBN: 9781450324052. DOI: 10.1145/2513383.2513447. [Online]. Available: https://doi.org/10.1145/2513383.2513447.

[3] H. van Vliet, Software Engineering: Principles and Practice, 3rd ed. Chichester (UK): Wiley, 2008.

[4] Microsoft. (n.d.). Keyboard shortcuts in windows, [Online]. Available: https://support.microsoft.com/en-gb/help/12445/windows-keyboard-shortcuts (visited on 06/25/2020).

[5] Vuetify. (Jun. 2020). Alerts, [Online]. Available: https://vuetifyjs.com/en/components/alerts/ (visited on 06/22/2020).

[6] K. Schwaber and J. Sutherland, "The scrum guide-the definitive guide to scrum: The rules of the game", scrum.org, Jul. 2013.

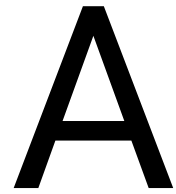
[7] Tech Terms. (Mar. 2013). Framework definition, [Online]. Available: https://techterms.com/definition/framework (visited on 05/04/2020).

[8] Web Development Team. (2019). Angular vs. vue vs. jquery vs. react vs. ember, [Online]. Available: https://kruschecompany.com/ember-jquery-angular-react-vue-what-to-choose/ (visited on 04/30/2020).

[9] Shaumik Daityari. (2020). 15 of the most interesting vue ui component libraries for 2020, [Online]. Available: https://www.codeinwp.com/blog/vue-ui-component-libraries/ (visited on 06/25/2020).

[10] Laws of UX. (n.d.). Jacob's law, [Online]. Available: https://lawsofux.com/jakobs-law.html (visited on 05/08/2020).

[11] Laws of UX. (n.d.). Fitts's law, [Online]. Available: https://lawsofux.com/fittss-law.html (visited on 05/08/2020).



Original project description

Project stepladder
a displacement diagram to Ladder IDE

This project is proposed by TU student Arend-Jan van Hilten, to get info and join this project, send an email to a.j.vanhilten@student.tudelft.nl

The goal of this project is to create an IDE for displacement diagrams (bewegingsdiagrammen). These diagrams are used in industrial automation to describe the sequential movement of actuators like motors and pneumatic or hydraulic cylinders within a production cycle. Industrial automation software is usually designed based on these diagrams. Although there is software to simulate industrial automation, there is no software suite to automatically generate automation software based on a displacement diagram.

De Techniekschool is developing a whole new training practicum for industrial automation. This IDE is meant for VMBO/MBO students that want to learn about the fundamentals of industrial automation. For this purpose the IDE should be easy to use, to improve their learning. The diagram should be compiled to a ladder logic diagram, which originates from the relay diagrams. These ladder logic diagrams are then compiled to Arduino code in another application that the displacement IDE should work with. To make it better for the users, the ladder logic should be as simple as possible to let the user learn what the compiler did and to learn them how to do the conversion. No prior knowledge of industrial automation (ladder logic, pneumatics, displacement diagrams) is needed, De Techniekschool has facilities to get you up to speed.

The IDE should:

- able to write a displacement diagram
- drag-drop/easy usable with a mouse
- easy to use interface
- compile the diagram to ladder logic (Ladderino)
- check the diagram for errors and warnings
- preferably be written for Electron (nodejs)
- allow single and double acting cylinders
- allow motors with encoders and switches
- start buttons/continuous execution

Extra optional features:

- Compile to Codesys, another ladder logic IDE
- Give suggestions
- explanation of the boolean formulas in the output ladder
- safety features
- examples
- debug mode with highlights on what is happening

B
Project info sheet

Title: Stepladder
Client: De Techniekschool B.V.
Date of presentation: 03-07-2020

Displacement diagrams are a notation of the movement of components in industrial machinery. They are commonly used in pneumatics. They are normally drawn on paper and manually converted to logic that drives the components. De Techniekschool tasked us with building an application in which the user can draw these types of diagrams and which can automatically convert the diagrams to ladder logic. This is intended for use in schools, to help students understand how to create this logic themselves.

Challenge: The challenge of this project was to make creating displacement diagrams as easy for the target audience as possible and to build a working compiler to compile the diagram to ladder logic.

Research: Our research focused on how to make an IDE that is easy to use for every kind of user to draw displacement diagrams. Therefore, we looked at several different ways of drawing the movement of a component. Furthermore, in order to educate people, our research also focused on making the compiler generate clear and comprehensible ladder logic.

Process: We developed our application in a Scrum style workflow, with sprints of two weeks. The work was split up into the UI and compiler parts and each member had a task in one of those parts.

Product: The product is an IDE for users to create a displacement diagram which can then be compiled into ladder logic.

Outlook: The product will be used on various schools to teach students how to create and use displacement diagrams and convert them to ladder logic.

Members:

Mark Bekooy
Interests: Full-stack development, entrepreneurship, fitness
Contributions: UI layout, signals, alerts, dialogues, Vue and Vuetify integration
Arend-Jan van Hilten
Interests: Embedded systems, electronics, pneumatics and Volvo classic cars
Contributions: Product owner, compiler creator and tester
Nathan van der Kamp
Interests: Back-end development, computer simulation, CG
Contributions: Compiler input validation and output file generation, UI sliders and undo-redo system
Robin Kouwenhoven
Interests: Operating systems, electronics and boats
Contributions: UI, saving and opening
Mathijs de Wolf
Interests: Entrepreneurship, big data, rowing
Contributions: UI, integrate UI with backend, counter and timer component
All members worked on the reports and presentation.

Contact person:	Arend-Jan van Hilten		arendjan18@gmail.com
Client:	Roland Hollaar	De Techniekschool	roland@dewerkunie.nl
Coach:	Myrthe Tielman	TU Delft EWI, Interactive Intelligence	M.L.Tielman@tudelft.nl

The final report for this project can be found at: `https://repository.tudelft.nl`

C

Feedback SIG

The feedback from the Software Improvement Group on the code is split up into multiple parts. The most important topics of what is changed and what is not, are discussed here.

C.1. Code duplication

The largest number of 'violations' of our code was code duplication. Most duplication was in the compiler section.

C.1.1. Compiler

The duplication in the compiler was primarily correct duplication (no false positives), but because of the structure of our code, it was not doable and deemed unnecessary to remove the duplication. This is because the compiler is built with 2 files that have classes defined in them with all their functions and settings. One example is `devices.js`. This file is a big object with an object for each type of device, which has the following setup:

```
typename: {
    steps:[],
    name:'',
    outputs:{...},
    inputs: {...},
    generateSignals: function() { ... },
    getSignals: function() { ... },
    formula: function(getChange) { ... }
}
```

The most duplication violations come from two or more devices with the same setup of signals, because this is for some (like `cylinder with bi-stable` and `cylinder with mono-stable` devices) almost the same. It was decided to keep these duplications, because it is not needed to change these functions (maintainability) and would need functions with branches to compensate for the little differences in the signal generation.

The file with the other duplication was `units.js`. This file defines all the input and output types, which in the end get compiled to ladder logic. These include types like `pinFalseCond` or `triggerUpCond`. Almost all duplication comes from the `init` function, where the pin and other settings for that type are set. It is possible to change it to one `init` function, but that would require branching and require a function that is able to handle all types of inputs and function overloading is not available in JavaScript, so that would be a function introducing a large complexity.

C.1.2. User Interface

The UI had no duplication other than one configuration object for the saving and opening files functions (in 2 files). We did not change this, because that would introduce importing the configuration from a new file or dependencies on the other file just to improve one metric.

C.2. Large units

The Software Improvement Group also tested for large units, being large files or large functions. Next two sections are about how the violations were removed or why they were kept.

C.2.1. Compiler

Another big part of the violations were unit size violations. Two of those files were again `devices.js` and `units.js`. These were flagged because of their length (±800 LOC and ±180 LOC). The only way to reduce the size would be to divide the files into multiple files, resulting in 9 and 17 more files. This would remove the size violations, but would only be done to remove the violations and not increase the maintainability. Because of this, it was decided to not change this code.

Another part of the compiler with duplication was the `generateLadder.js` file of the compiler. This file gets a list of formulas and generates a Ladderino file. Two functions in this file are `createRungOutput` and `createCondition`. These two functions had big switch statements to select the correct type of Ladder code (object). We had originally chosen to work with these two types of adding functionality to a type (units with function property vs function with a switch on unit type) to investigate the best way to code. Because these switch statements added too many lines of code and complexity to the functions in `generateLadder.js` it was decided to move the code in the switch cases to the `units.js` file, increasing its length, but decreasing the `generateLadder.js` length and complexity.

To reduce the unit size of some other functions, functionality that did not need to be in the same function was split up into multiple functions. One example of this is the `checkInput` function in `checkInput.js`. This function checks for all kinds of faults in the diagram and originally it had some functions, but also some checks in the main function directly. To reduce this, the separate functionality was refactored in a new function like `checkDevicesStepCount`. This also reduced the complexity of the function.

C.2.2. User Interface

The UI had a few violations in this category. The following three were the most important and an explanation of each is given.

Index.js

This file is the start of any Electron application. This file defines which windows to create with what .html file and the communication between the windows. This file was (when handing it in the first time) 140 lines with no function longer than 30 lines. Because this file does not need to be changed often, it was decided to keep the file as is.

Opener.js

This file read a file and processes it. The function that was flagged did the processing. This was done in that single function without calling any other function. This function was refactored by creating sub-functions for every step and moving functionality specific to a device or type of device to a new file (`processCylinder.js` for example). This also decreased its complexity.

C.2.3. Vuetify plugin

The last UI file with too many lines is the Vuetify plugin file. This file initializes the Vuetify plugin (see section 6.1.3). This file is too long because of initializing all the images in the UI. This is one object of 80 lines and moving it would only change the place of the violation, so it was decided to keep it as is.

C.3. Complexity

The complexity was the most important metric in our opinion, because a higher complexity makes it harder to maintain than a higher code duplication or size. The biggest improvement was made in this part.

C.3.1. Compiler

In section C.2.1 a few complexity violations were addressed.

`devices.js` had in some functions the same if statement twice. This was originally because the two parts were used for generating different signals (inputs and outputs), but to reduce complexity, the if statements were merged, resulting in a lower complexity with the same functionality.

The `converter.js` file had a function with multiple for loops (6) with an if-elseif... statement, which was moved into separate functions, one for converting all the devices and one for all the other types of components like buttons and timers. This is enough to get the complexity down in our opinion, because the single if-elseif... statement is with a for loop in a separate function and the others are simple for loops with no branching.

C.3.2. User Interface

The UI had some complexity violations, the main one being the `opener.js` file, as already discussed in section C.2.2. The other violations of the UI were some functions with multiple (disjoint, non-nested) if statements. These were dealt with by refactoring the separate if statements into their own function.

D Context

In this section, crucial information about pneumatics, to understand how the product works, is explained. What pneumatics is, will be explained first. This is followed by an explanation of ladder logic. Last, what a displacement diagram is will be explained.

D.1. Pneumatics

Pneumatic devices are mechanical components driven by air pressure. The most common pneumatic components are pneumatic cylinders. These cylinders can be made to extend and contract by applying air pressure to an input port on the cylinder. This is usually done by electronically switching solenoid valves to control the flow of air. There are two types of pneumatic cylinders, single-acting and double-acting cylinders. Single acting cylinders require air pressure to move one way and will return automatically by a spring. Double-acting cylinders have two air inputs, one to extend and one to retract the cylinder. The valves used to control these cylinders also come in multiple variations. The most common, and relevant for our application, are mono-stable and bi-stable valves. Mono-stable valves fall back to one out of two output states when not receiving power but bi-stable valves stay in the position they are in. Mono-stable valves are often used with single-acting cylinders, but it is not a requirement.

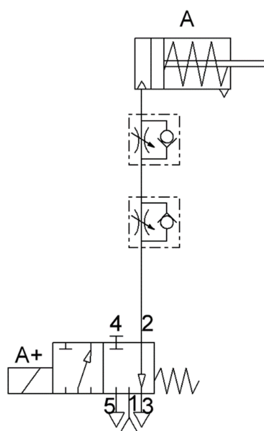


Figure D.1: A schematic drawing of a single acting cylinder with a 5/2 electric powered mono-stable valve

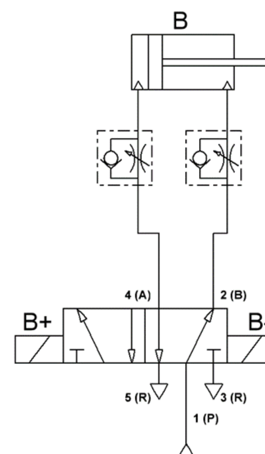


Figure D.2: Schematic drawing of a double acting cylinder with a 5/2 electric powered bi-stable valve

D.2. Ladder logic

Originally pneumatic cylinders were controlled using 24V/230V relays. To operate a whole production plant, whole closets were filled with relays. When Programmable Logic Controllers (PLC) arose, the designers of the original machines wanted to code the PLCs just like the original relays as they already knew how those worked. The new programming language that was developed was ladder logic. In essence a ladder logic

diagram is a relay diagram rotated 90°. An example of a relay diagram is shown in Figure D.3. A ladder logic diagram with the same functionality is shown in Figure D.4.

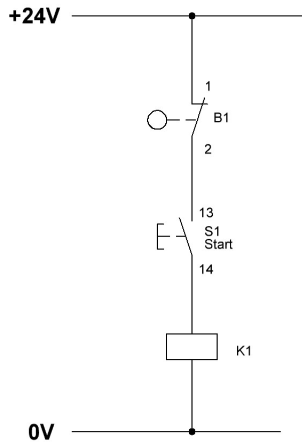


Figure D.3: Relay diagram. When B1 is not pressed and Start is pressed, relay K1 will turn on

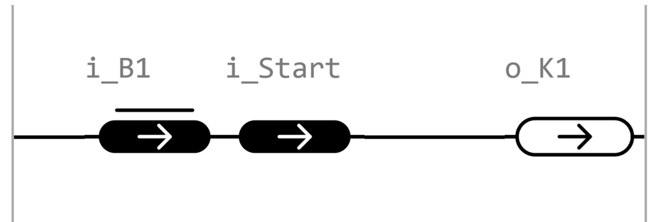


Figure D.4: Ladder logic describing the same circuit

A ladder logic diagram consists of multiple lines (rungs), each with one output. Each rung has multiple input conditions, which can be put in series for an "and" function or put in parallel for an "or" function.

D.3. Displacement diagram

A displacement diagram is a diagram where the state and movement of actuators are described. The diagram is used to represent the different movements and gives an overview of how the system needs to behave. As an example, figure D.5 describes the movement of three cylinders. Each step is a different state and when a state is reached it will continue to the next state. Figure D.6 describes the actual movement of the cylinder step by step.

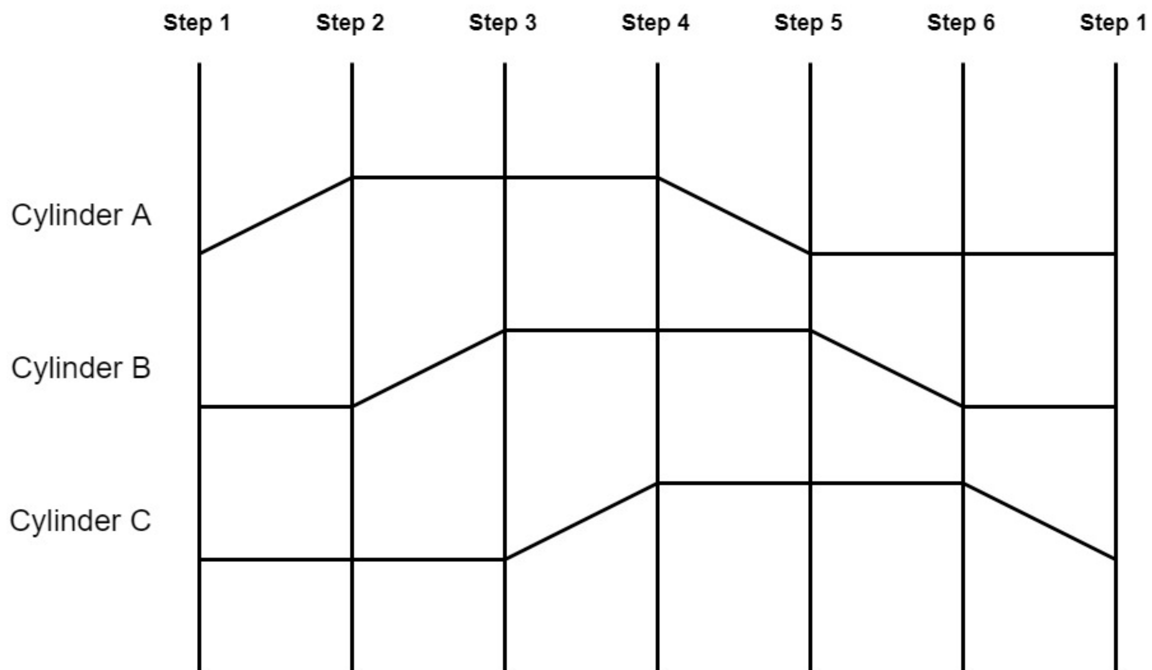


Figure D.5: Displacement diagram of 3 cylinders

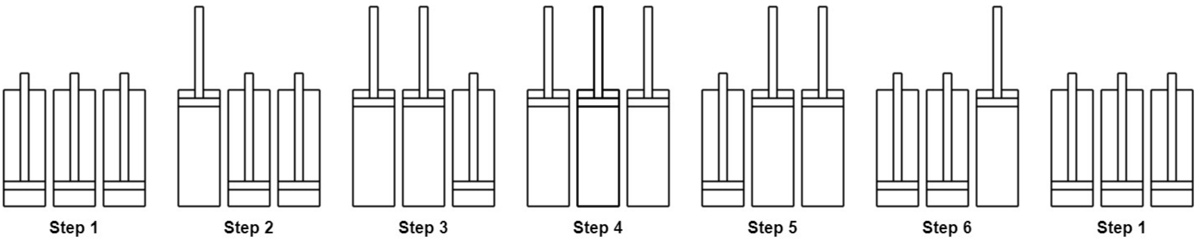


Figure D.6: Movement of cylinders in figure D.5

These steps can be seen as a set of conditions. When the conditions of a certain step are reached, it will perform an action to reach the next one. Each cylinder has two sensors, one at either end (_0 and _1), which will turn on when the cylinder is in that state. This enables the PLC to know about a state of a cylinder. In the example below the step and conditions are not as interesting because each step has a different set of conditions, but when there are two or more steps with the same conditions it becomes a bit more difficult. In these cases, a memory unit has to be added to differentiate between the two steps.

E

Checks

The compiler checks the diagram on errors and warning. A diagram with warnings can be compiled, but a diagram with errors cannot and will not be compiled.

E.1. Errors

The possible errors the compiler checks for are:

- Diagram must have more than 1 step.
- Diagram cannot have more than 32767 steps.
- All devices must have the same number of steps.
- All devices must have the same beginning and start state.
- All step values must be valid for the device it belongs to.
- Encoder motor must have more than 1 possible position.
- Device names must be unique.
- Device names cannot start with a digit and can only contain letters, numbers and underscores.
- A counter must repeat between 0 and 32767 times.
- A counter must enclose at least 2 steps.
- A counter cannot start before step 0 or end after the number of steps.
- Devices must be in the same state at the beginning of the counter as at the end of the counter.
- Counters cannot be intersecting, only enclosed.
- 2 counters cannot start and end at the same state, only 1 equal (start or end) is allowed.
- Timer start cannot before step 0 or after the last step.
- Two timers cannot start on the same step.
- Only one LCD screen can be added.
- Input/Output pins cannot be empty.
- Input/Output pins cannot be the same.

E.2. Warnings

The possible warnings that the compiler checks for are:

- Same LCD text on consecutive steps.
- LCD text length longer than 20 characters.
- No LCD texts entered.