MINISTRY OF EDUCATION AND RESEARCH OF REPUBLIC OF MOLDOVA
TECHNICAL UNIVERSITY OF MOLDOVA
FACULTY OF COMPUTERS, INFORMATICS, AND MICROELECTRONICS
DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATICS

Report

on Formal and Automatic Languages

Theme: A DSL for PLCs

Mentor: lect. univ., Irina Cojuhari
Students: Irina Racovcena, FAF-212
Roman Trandafir, FAF-212
Vladislav Frunze, FAF-212
Vlad Ursu, FAF-212
Inga Paladi, FAF-212

Chișinău, 2023

Abstract

The purpose of this report is to explore the potential of using domain-specific languages (DSLs) for programmable logic controllers (PLCs) in industrial automation. By providing an example of a grammar for a DSL, this report demonstrates how DSLs can be used to create efficient, maintainable, and flexible programs for industrial automation. The proposed grammar includes essential elements such as input, memory, and output variables, logical operations, and program structure.

DSLs for PLCs offer several advantages over traditional programming languages, including improved readability, reduced complexity, and easier debugging. They also allow engineers and technicians with limited programming experience to create sophisticated programs that can meet the specific needs of industrial automation applications. The use of DSLs can also lead to improved productivity, as engineers can focus on the automation process rather than the intricacies of programming languages.

The report concludes that DSLs offer significant potential for the development of PLCs in industrial automation, and further research should be conducted to explore the use of DSLs in specific applications. The proposed grammar can be used as a starting point for developing DSLs for industrial automation applications, and additional work is needed to refine and extend the grammar for more complex automation tasks. The use of DSLs in industrial automation has the potential to revolutionize the industry, and this report provides an important foundation for future research and development in this area.

Content

Introduction

Programmable Logic Controllers (PLCs) have been an indispensable part of the manufacturing industry for decades, effectively controlling processes in sectors such as automotive, chemical, food, and beverage. PLCs can manage multiple operations, including material handling, quality control, and machine control.

The traditional programming language for PLCs is the ”ladder diagram”. However, as industries evolve, the need for more sophisticated and maintainable programs becomes apparent. Domain-specific languages (DSLs) can address this need. DSLs are programming languages specialized in a particular domain of control systems.

Using DSLs for PLCs can create maintainable, readable, and simpler programs that govern industrial processes. DSLs provide high-level abstractions that help program complex control logic in a more intuitive manner. Additionally, DSLs can reduce complexity, improve maintainability, and enable the development of more sophisticated control systems.

Moreover, DSLs for PLCs can increase productivity and efficiency, leading to cost savings and improved profitability for companies. By making control systems easier to understand and maintain, DSLs can reduce development times and maintenance costs.

One of the main advantages of utilizing DSLs in PLCs is their ability to facilitate code generation, resulting in a significant decrease in the time and effort required for control system development and maintenance. This can prove particularly advantageous for industrial processes on a large scale, where the creation and upkeep of PLC programs can be both time-intensive and costly.

To summarize, DSLs possess the potential to fundamentally transform the industrial control system development and maintenance process. DSLs present an intuitive and natural method for programming PLCs, thereby reducing complexity, increasing maintainability, and enabling the creation of more sophisticated control systems. Given the rising demand for more complex and efficient industrial processes, the significance of DSLs for PLCs is set to escalate, thus, making it an area of great interest for future development.

In this report, it is proposed to realize a domain-specific language that would account for the above points. The idea for the DSL is to be textual in contrast to the ladder diagram, which is graphical. Nevertheless, the DSL will borrow the best features of the ladder diagram like standardization, efficiency, and flexibility.

1 Domain analysis

1.1 Domain analysis and identification of the problem

Domain-specific languages (DSLs) are programming languages that are designed to solve problems within a specific domain or application area. They are typically more expressive and easier to use than general-purpose programming languages, which can make them more productive and efficient for solving certain types of problems.

Regardless, of whether it is domain-specific or not, it should consist of the following main parts. Starting with the concrete syntax which defines the notation with which users can express programs. Types of concrete syntax: be textual, graphical, tabular, or a mix of these. The abstract syntax is a data structure that can hold semantically relevant information expressed by a program. Usually, it is represented as a tree or a graph. It does not contain any details about the notation – for example, in textual languages, it does not contain keywords, symbols or whitespace. The static semantics of a language is the set of constraints and/or type system rules to which programs have to conform, in addition to being structurally correct (with regards to the concrete and abstract syntax). Execution semantics refers to the meaning of a program once it is executed.[1]

DSLs are often used to solve problems in areas such as scientific computing, finance, telecommunications, and web development. They can be used to create simulation models, automate financial calculations, develop network protocols, and more.

One of the main advantages of DSLs is that they provide a higher level of abstraction than general-purpose programming languages. This means that developers can work at a higher level of complexity, which can make it easier to understand and modify code. DSLs can also be more expressive and concise than general-purpose programming languages, which can reduce the amount of code needed to solve a particular problem.

DSLs can be implemented in several ways. External DSLs are languages that are created specifically for a particular application area or domain, and they often have their own syntax, grammar, and toolchain. Examples of external DSLs include SQL, VHDL, and MATLAB.

Internal DSLs, on the other hand, are embedded within a general-purpose programming language and leverage the syntax and semantics of that language to provide a more specialized syntax for the specific domain. Examples of internal DSLs include LINQ, RSpec in Ruby, and jQuery in JavaScript.[2]

When designing and implementing a DSL, it is important to consider the trade-offs between expressiveness, simplicity, and ease of use. A DSL that is too complex may be difficult to understand and maintain, while a DSL that is too simple may not be expressive enough to solve the problem at hand. It is

also important to consider the needs of the users of the DSL, and to ensure that the DSL provides a clear and intuitive way to solve the problem.

1.1.1 What domain will the DSL address?

The DSL (Domain Specific Language) will address the domain of logical and programmable automata, which is related to the field of computer science and control systems. Control systems are a set of mechanical or electronic devices that regulates other devices or systems by way of control loops. It is possible to use logical and programmable automata for implementing such control systems like:

- **Robots**: Robots can use such types of automata for specific tasks like moving in a certain way, and, in general, to do repetitive tasks.
- **Industrial control systems**: In industry, such automata are used for controlling different processes.
- **Embedded systems**: In addition, such automata are used in vehicles, appliances, and medical devices in order to control their behavior.
- **Smart homes**: They can also be used to regulate the states of a smart home like lighting, heating, and cooling.
- **Network security**: Moreover, these automata are also helpful in detecting threats in a computer network.

1.1.2 What is a PLC?

PLC stands for Programmable logic controllers which are now the most widely used industrial process control technology. An industrial-grade computer that can be configured to carry out control tasks is known as a programmable logic controller (PLC). The programmable controller has significantly reduced the amount of hardwiring required for traditional relay control circuits. Easy programming and installation, fast control, network interoperability, ease of troubleshooting and testing, and excellent dependability are further advantages.

The programmable logic controller is intended to work with a variety of input and output configurations, operate over a wide range of temperatures, be immune to electrical noise, and be impact and vibration resistant. Typically, battery-backed or non-volatile memory is where software for controlling and operating industrial process equipment and machinery is kept. A PLC is an illustration of a real-time system since the output of the system it controls is dependent upon the input circumstances.

The PLC was first used to replace relay logic, but due to its wide range of capabilities, it is now utilized in a variety of more applications. A PLC can execute relay switching operations as well as other applications like timing, counting, calculating, comparing, and the processing of analog signals because of the fact that its structure is built on the same concepts as those used in computer architecture.

Over a traditional relay kind of control, programmable controllers have a number of benefits. To carry out a given task, relays must be hardwired. The relay wiring has to be altered or modified as soon

as the system requirements change. In severe circumstances, such as the automotive industry, whole control panels had to be rebuilt because it was not practical to rewire the old panels for each model change. The programmable controller has significantly reduced the amount of hardwiring required for traditional relay control circuits. Compared to other relay-based process control systems, it is compact and reasonably priced. Relays are still a part of contemporary control systems, but logic is rarely applied to them. [3]

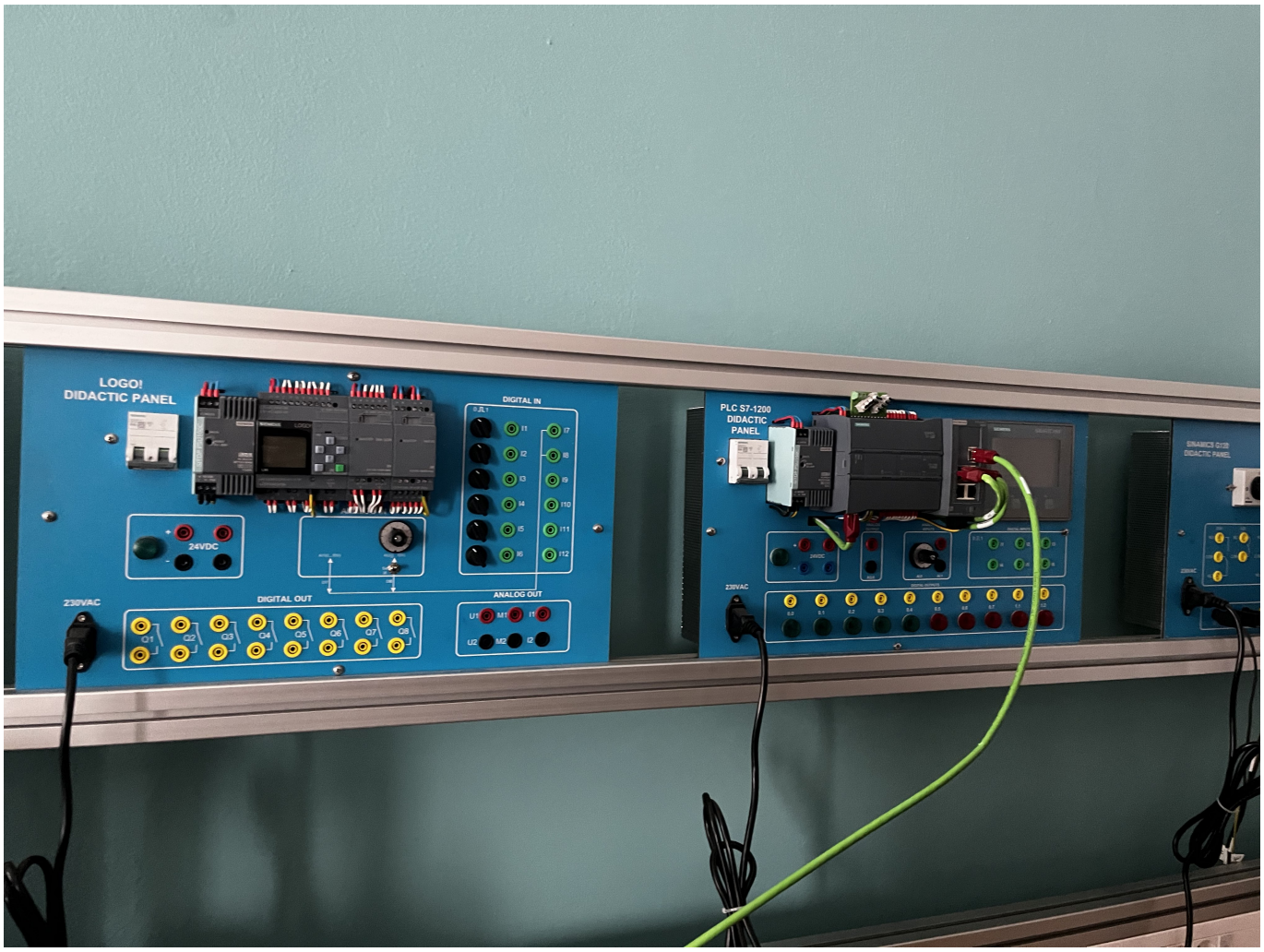

Figure 1. *Programmable Logical Controllers*

1.1.3 Objectives

Setting clear objectives when designing a Domain Specific Language (DSL) for Programmable Logic Controllers (PLCs) is important for several reasons: focus, usability, safety, portability and maintainability. Therefore, the objectives for this project are:

- Creating a Domain-specific language capable of performing basic calculations and operations.
- Design a DSL that should be easy to learn and use. Providing good syntax and semantics, as well as tools and documentation to help users understand and use the language effectively.
- giving the user full control in structuring and constructing login expressions.
- The DSL should be designed to be portable across different PLC platforms and architectures.

- The DSL should be designed to ensure that the control logic it produces is safe and reliable. This may include providing error checking and diagnostics, as well as support for fault tolerance and recovery.

1.1.4 Why or how could this domain benefit from a DSL?

A Domain Specific Language (DSL) can be advantageous to a logical and programmable automaton, such as a robot or computer, in various ways:

- **Abstraction**: A DSL offers a high-level, abstract representation of the work to be completed, enabling users to succinctly and intuitively define complicated procedures. This makes it simpler for people who are not programming specialists to develop programs, debug them, and understand their behavior.
- **Specificity**: DSLs, such as those used in robotics, banking, or web development, are created to address unique issues within a certain field. The automaton may more easily carry out complicated actions precisely and efficiently by employing a DSL to benefit from domain-specific knowledge and techniques.
- **Interoperability**: This allows it to integrate the automaton into a bigger system since DSLs can be constructed to interact with other software tools and components. This can also streamline the process of integrating capabilities and technologies and make it simpler to share information and work with other systems.
- **Improved Performance**: Because a DSL is tuned for a particular domain, the automaton may carry out complicated operations more quickly and effectively. Compared to utilizing a general-purpose programming language, this can result in better performance and shorter development times.

To sum up, using a DSL can make it simpler to operate and program a logical and programmable automaton, which can enhance its performance, efficiency, and compatibility with other systems.

1.1.5 What problem will be solved by the proposed language?

Domain Specific Languages (DSLs) are specialized programming languages that are designed to solve specific problems in a particular domain. DSLs provide a high-level, intuitive syntax and a set of abstractions and tools that are tailored to the needs of the domain, making it easier for developers to solve problems and achieve their goals. A DSL for Programmable Logic Controllers (PLCs) can be used to simplify and streamline the process of programming PLCs, reducing the amount of manual coding required and improving the accuracy and quality of the PLC code.

PLC programming is the process of creating programs that control and automate industrial processes using PLCs. A PLC is a compact, rugged device that is used to control and automate a wide range of industrial processes, from simple processes like turning on and off a motor to complex processes like coordinating the operation of multiple motors, sensors, and actuators.

PLC programming requires a good understanding of the process being controlled, as well as a deep understanding of the PLC hardware and software. Programmers must also have a solid understanding of

the I/O (input/output) signals and data types used by the PLC, as well as the communication protocols used to transfer data between the PLC and other devices. PLC programming can be done using a variety of programming languages, including ladder diagram, structured text, and function block diagrams. The choice of language depends on the PLC manufacturer and the specific requirements of the application.

Overall, PLC programming is an essential aspect of industrial automation and control systems engineering, as it enables the development of efficient and reliable control systems that can improve the quality, safety, and profitability of industrial processes.

The PLC DSL provides a simplified, high-level syntax that is easy to read and understand, making it straightforward for programmer to express their ideas and solutions in a concise and expressive way. The DSL also includes various tools and features that can automate various aspects of the PLC programming process, such as error checking, code generation, and debugging. After the code is generated, it is compiled and downloaded to the PLC, where it runs on the PLC's microprocessor to control and automate the industrial process. The PLC code can be tested and debugged using various tools and techniques to ensure that it meets the requirements and behaves as expected.

1.1.6 Comparative analysis

For logical and programmable controllers, there are in fact a number of additional DSLs (Domain-Specific Languages). Here are a few illustrations:

1. Programmable logic controllers employ the textual programming language Structured Text (ST) (PLCs). It is created to be simple to understand and write and is based on the Pascal programming language. This language gives the possibility to operate with inputs and outputs, using different statements such as for, while, if, and case.

```
1  IF #start = 1 THEN
2      //comment
3      "Max_nr" := #Array[0];
4      FOR #i := 1 TO 10 DO
5          // Statement section FOR
6          IF #Array[#i] > "Max_nr" THEN
7              "Max_nr" := #Array[#i];
8          END_IF;
9      END_FOR;
10 END_IF;
11
```

Figure 1.1.1 - Structured Text (ST)[5]

2. PLCs employ the graphical programming language called Function Block Diagram (FBD).A Function Block Diagram (FBD) is a graphical programming language that is widely used in industrial control systems. It is used to design, visualize, and document the structure and behavior of a control system. In an FBD, the control system is represented as a set of interconnected functional blocks, each of which performs a specific logical or arithmetic operation. These blocks can be combined to form

more complex functions, which can then be further combined to create a complete control system.

Each block in an FBD has input and output connections, and these connections can be connected to other blocks or external devices. The blocks can also contain variables, which can be used to store and manipulate data.

FBDs are supported by many programming languages used in industrial control systems, including ladder diagram, structured text, and sequential function charts. They are widely used in industries such as manufacturing, oil and gas, and utilities, where they are used to control and monitor a variety of processes, including production lines, power generation, and water treatment.

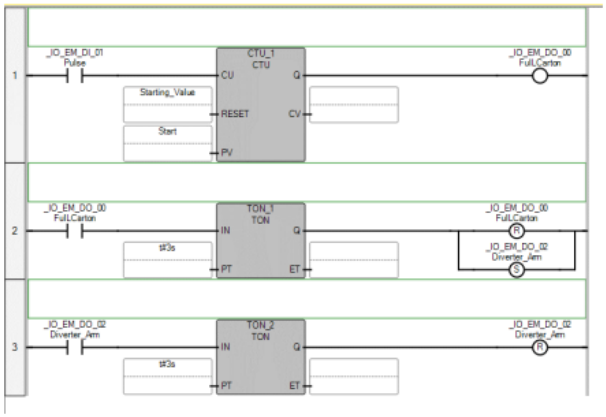

Figure 1.1.2 - Function Block Diagram
[3]

3. Ladder diagram (LL) - This is another graphical programming language used in PLCs. Ladder diagram is a graphical programming language used to program industrial control systems, particularly Programmable Logic Controllers (PLCs). It is based on the electrical wiring diagrams used in the control of machinery and electrical equipment, where the connections between various components are represented by lines and symbols.[4]

   In Ladder diagram, the control system is represented by a ladder-like diagram consisting of horizontal rungs and vertical rails. Each rung represents a specific logical or arithmetic operation and is connected to power and ground rails on either side. Ladder Diagram is a widely used programming language in industrial control systems because of its simplicity, ease of use, and visual representation of the control system. It is particularly useful for control systems that involve discrete operations, such as assembly lines, conveyor systems, and packaging machines.
4. Instruction List (IL): This is a simple textual instruction. A low-level textual programming language used in PLCs is called Instruction List (IL). It is a textual language that represents the control system as a sequence of machine-level instructions, each of which performs a specific operation.

   In IL, each instruction is represented by a mnemonic code and one or more operands, which specify

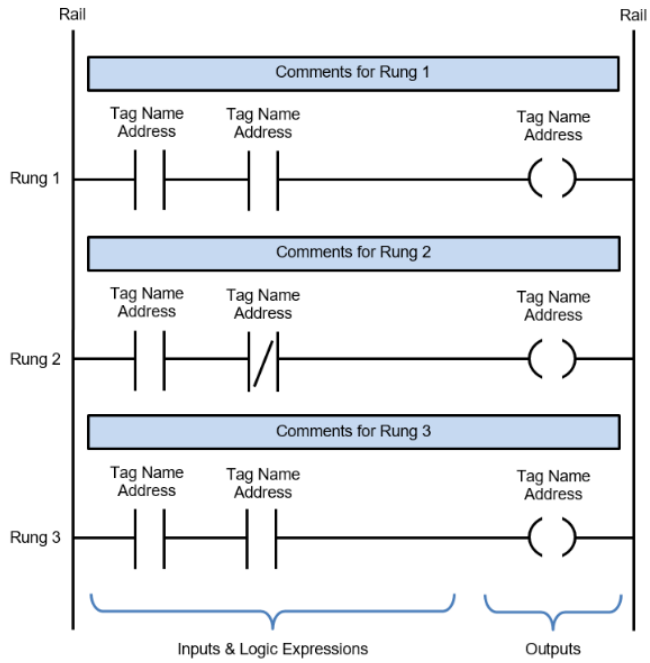

Figure 1.1.3 - Parts of a Ladder Diagram
[6]

the input and output variables of the operation. The variables can be input signals from sensors or switches, output signals to actuators or indicators, or internal memory locations used to store data or program status.

IL is a very compact and efficient programming language that can be used to program complex control systems. It supports a wide range of arithmetic, logical, and bit manipulation operations, as well as advanced functions such as loops, jumps, and subroutines.

In contrast to previous languages, my language is focused on natural language programming and is designed to be simple to learn and use. Moreover, a variety of programming paradigms, such as imperative, functional, and object-oriented programming, are supported. Furthermore, it offers built-in assistance for typical programming tasks like data manipulation and control flow, which can make programming easier.

1.1.7 Target groups for the DSL

Given that the language will mainly be meaning to work with control hardware present in almost all aspects of life, it is safe to say that engineers of every domain could interact with the final product of this project.

- **Robotics Engineers**: The engineers specialized in programming robotic systems, capable of replacing human labor.
- **Industrial Engineers**: Mainly, control engineers, automation technicians, and plant managers, concerned with all the complicated machinery and processes that put big factories into motion. These posts concern themselves with PLCs, as well as other, various automation systems made to control/-

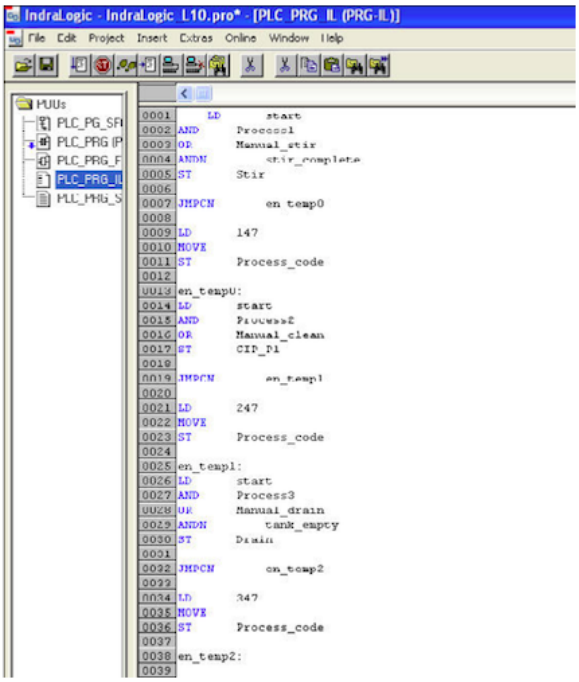

Figure 1.1.4 - A sample instruction list (IL) program[7]

monitor industrial processes.

- **Systems Engineers**: The engineers that install and program everything from car system, to smart homes.
- **Electrical Engineers**: System integrators that create and implement control systems for various fields of industry (ex. manufacturing, automotive, aerospace, etc)
- **Academic researchers and students**: The future engineers, who require specialized languages, for studying and developing control systems and algorithms.
- **Small business entrepreneurs**: Entrepreneurs who perhaps cannot yet afford specialized engineering departments, but still require the tools for customizing their products.

Generally speaking, this DSL could prove itself useful to both individual and companies that need the capability to create and take care of logic systems.

2 Language overview

2.1 Basic computation. Computational model

The basic computation of the DSL is in executing a control logic program, which is a sequence of instructions that imposes how the system will behave in response to the inputs and outputs.

The computational model suggests the following elements:

- **Inputs and Outputs**: A PLC receives certain inputs from different devices like sensors. In response, it will output a sequence of instructions for other devices.
- **Logic blocks**: Logic blocks are the functions and operations which are applied to the input data to create a certain output that will control the devices which are needed.
- **Wiring and Connections**: In order to connect the logic blocks, some connections has to be put between them in order to determine the correct flow of data.

2.2 Basic data structures. Creating and manipulating data

The basic data structures in the DSL are the following:

- **Inputs**: Input data is the data collected from the sensors and other devices like switches.
- **Outputs**: The output data are the signals that are generated by the control logic program and sent to the output devices, like motors or valves.
- **Intermediate variables**: Intermediate variables suggest the data which is gradually generated when the control logic program is running.

Each of the above data structures can contain the Boolean data type. In addition, for such components as the timer, there can be the integer data type.

2.3 Control structures

DSLs can support a variety of control structures, depending on the needs of the domain. However, some common control structures found in DSLs include:

- **Conditionals**: DSLs can provide constructs for conditionals such as ”if-else” statements to execute a certain set of instructions based on the evaluation of a condition.
- **Loops**: DSLs can provide constructs to execute a set of instructions multiple times based on a condition, such as ”while” loops, ”for” loops, and ”foreach” loops.
- **Switches**: Switches: DSLs can provide constructs to select one of several options based on a value, such as ”switch-case” statements.
- **Exception Handling**: DSLs can provide constructs for handling errors or exceptions that may occur during the execution of the code, such as ”try-catch” blocks.

Users can specify or manipulate the control flow in a DSL using the syntax and constructs provided

by the DSL. For example, a user can use an ”if-else” statement to execute a set of instructions based on a condition in the DSL, or they can use a ”while” loop to execute a set of instructions repeatedly until a condition is met.

The syntax for controlling flow in a DSL is typically designed to be intuitive and easy to understand for users in the domain for which the DSL was created. This allows users to write code that is more expressive and easier to read and maintain, especially compared to writing code in a general-purpose language that may not be optimized for a particular domain.

2.4 Inputs and Outputs

There are two main types of inputs: data inputs from devices and machines, and data inputs that are human-facilitated. The input data from sensors and machines are sent to the PLC. Inputs can include on/off states for things like mechanical switches, buttons, and encoders. High/low states for things like temperatures, pressure sensors, and liquid-level detectors, or opened/closed states for things like pumps and values. Human-facilitated inputs include button pushes, switches, and sensors from devices like keyboards, touch screens, remotes, or card readers.

Outputs are the physical actions or visual results that are based on a PLC logic in response to those inputs. Physical outputs include starting motors, turning on a light, draining a valve, and turning the heat up or a pump off. Visual outputs are sent to devices like printers, projectors, GPSs, or monitors.

2.5 Error handling

Many electronically controlled equipment are set to move in different directions in industrial sites. Without a programmable logic controller (PLC), which converts commands into language that machines can comprehend, none of this would be feasible.

A user will press a button on a machine to initiate a certain command. Yet, without programmable code, the machine itself is unable to comprehend this. As a result, a code is required to activate the device. A central processing unit that serves as a middleman houses the PLC code. After the PLC converts the user's button press into a command the machine can comprehend, the command is carried out.

PLCs have become more complicated since they were first created in the late 1960s.[5] The code is now set up to perform several different difficult jobs. PLCs work continuously to convert ongoing sequences of commands for the corresponding machinery as a result. PLCs are complicated, but they may also malfunction.

Module failure, power outages, and poor network connectivity are often cited causes of PLC control system failure. Other causes of PLC failure include dampness, electromagnetic interference, and overheating. Factory engineers must examine their systems to ensure that PLC control system faults don't go out of control.

Indeed, a PLC is built to function in challenging conditions. But, even the most sophisticated con-

trolling code may encounter issues if it is exposed to brownouts, blackouts, or other conditions that might affect the circuitry physically. Indeed, a PLC is built to function in challenging conditions. But, even the most sophisticated controlling code may encounter issues if it is exposed to brownouts, blackouts, or other conditions that might affect the circuitry physically. To communicate errors to the user in PLC programming, there are several techniques that can be used:

- **Error codes**: One common approach is to use error codes to identify the specific type of error that has occurred. These codes can be displayed on the operator interface or logged to a file for later analysis.
- **Alarms**: Another method is to use visual or audible alarms to alert the user to the presence of an error. These alarms can be triggered by specific conditions, such as a sensor failure or a communication error.
- **Diagnostic messages**: PLC programming languages often provide diagnostic messages that can help users identify the source of an error. These messages can provide information about the specific line of code that caused the error or suggest possible solutions.
- **Program flow control**: Program flow control is a technique that can be used to handle errors in a more structured way. By using error-handling routines and exception-handling, programs can gracefully recover from errors without crashing the system or causing data corruption.

Overall, strong error-handling algorithms that can recover from faults in an organized and predictable manner must be included together with clear and succinct user communication if error management is to be successful in PLC programming.

2.6 Implementation plan

For implementing a domain-specific graphical language, among the first actions taken should be defining a visual syntax and the semantics of the language, as well as providing a concrete implementation for rendering and executing real programs in the created language.

A more defined, and simple, step-by-step plan would look like this:

- **Defining the visual syntax**: Constructing the specific visual objects that will represent the constructs for the language, including but not limited to: icons, shapes, connectors, and other programmable components of the language.
- **Defining the semantics**: Defining the logical meaning of the visual syntax elements, and their programmable concepts.
- **Implementing the developed concepts**: Creating a visual editor for developing programs, and allowing any user to specify the properties of any visual element. As this is a graphical DSL, it implies putting the program together by dragging and dropping the syntax elements on a canvas and linking them with connectors.
- **Rendering the DSL**: Developing an engine that renders on a display the created program. This engine

should be able to generate code in some target language (Java, C, etc.), or even directly execute the program.

- **Testing and refining**: Testing thoroughly the DSL, refining visual syntax, refining the language, adding new elements, all this necessary, as well as getting some feedback from users both inside and outside the developing team.

3 Language Design

When it comes to writing a program for a programmable logic controller (PLC), there are a few grammar guidelines that can help ensure the code is clear, concise, and easy to understand:

1. Use descriptive names for variables and tags: Naming conventions should be clear and consistent. This helps ensure that others who read the code can quickly understand what each variable or tag represents.
2. Variables are declared in the inputDeclarations, outputDeclarations and memoryDeclarations sections. As their naming implies, the inputDeclarations and outputDeclarations contain the input variables and and output variables respectively. The memoryDeclarations is used for specifying which coils within the RAM memory module are being utilised.
3. Use comments to explain complex logic: PLC programming often involves complex logic and calculations. Using comments can help make the code more readable and understandable.
4. Use proper indentation and spacing: Proper indentation and spacing can help make the code more readable and easier to follow.
5. Use appropriate programming structures: PLC programs can use a variety of programming structures, including ladder diagram, function blocks, and structured text. Choose the appropriate structure for the task at hand and use it consistently throughout the program.
6. Keep it simple: PLC programming should be as simple as possible. Avoid using complex logic or advanced programming techniques unless they are necessary to achieve the desired outcome.
7. Test and debug your program: Finally, test your program thoroughly to ensure it works as expected. Debugging is an important part of programming, and it is essential to take the time to identify and fix any issues before the program is deployed in the field.

3.1 Development of PLC DSL

The second chapter has described the steps on how to design and implement grammar for graphical domain-specific language, and tools that were used. The DSL design includes several stages. First of all, definition of the programming language grammar L(G) = (VN, VT, S, P):

- S - is a start symbol;
- P – is a finite set of a production of rules;
- VN – is a finite set of non-terminal symbols;
- VT - is a finite set of terminal symbols.

The rule to start a language grammar is to engage the parser first. Any rule in the grammar can act as a start rule for the following parser. Start rules don't necessarily consume all of the input. They consume

only as much input as needed to match an alternative of the rule. For example, consider the following rule that matches one or a few tokens, depending on the input:

<program> ::= <start> <programName> <inputDeclarations> <
outputDeclarations>
<memoryDeclarations> <logicStatements> <endProgram>

Where < *program* > is the start symbol. Non-Terminal symbols are: V_N =program, start, endProgram, programName, identifier, alpha, alphaNum, digit, inputDeclarations, outputDeclarations, memoryDeclarations, var, x1, x2, ram, CN01, CN02, CN03, CN04, coils, coil, logicStatements, statement, logicOp

The terminal symbols are V_T =BEGIN, END, INPUT, OUTPUT, , RAM, CN01, CN02, CN03, CN04, AND, OR, XOR, NOT, I, Q, M,< *alpha* > (all alphabetic characters), < *digit* >(all numeric digits). The second step is to define the reference grammar.

<>	used to indicate non-terminals
::=	used to indicate a production rule
BEGIN	(in **bold** font) means that **BEGIN** is a terminal i.e., a token or a part of a token.
|	separates alternatives
""	used to indicate terminals

< *program* >::=< *start* >< *programName* >< *inputDeclarations* >< *outputDeclarations* >
< *memoryDeclarations* >< *logicStatements* >< *endProgram* >
< *start* >::= "**BEGIN**"
< *endProgram* >::= "**END"**
< *programName* >::=< *identifier* >
< *identifier* >::=< *alpha* >< *alphaNum* > ∗
< *alpha* >::= "**a**"|"**b**"|...|"**z**"|"**A**"|"**B**"|...|"**Z**"
< *alphaNum* >::=< *alpha* >< *digit* >
< *digit* >::= "**0**"|"**1**"|"**2**"|"**3**"|...|"**7**"
< *inputDeclarations* >::= "**INPUT**" < *var* > "**.**" < *var* >"**;**"
< *outputDeclarations* >::= "**OUTPUT**" < *var* > "**.**" < *var* >"**;**"
< *memoryDeclarations* >::= "**RAM**" < *ram* > "**.**" < *ram* >"**;**"
< *var* >::= "**I**" < *x*1 >< *x*2 > |"**Q**" < *x*1 >< *x*2 > |"**M**" < *x*1 >< *x*2 >
< *x*1 >::= "**0**"
< *x*2 >::= "**0**"|"**1**"|"**2**"|"**3**"|"**4**"|"**5**"|"**6**"|"**7**"
< *ram* >::=< *CN*01 > "**.**""**AND**" < *coils* > | < *CN*02 > "**.**""**OR**" < *coils* > |

$< CN03 >$ "**.**""**XOR**" $< coils >$ | $< CN04 >$ "**.**""**NOT**" $< coils >$

$< CN01 >$::= "**CN01**"

$< CN02 >$::= "**CN02**"

$< CN03 >$::= "**CN03**"

$< CN04 >$::= "**CN04**"

$< coils >$::=$< coil >$ "**.**" $< coil >$

$< coil >$::=$< var >$

$< logicStatements >$::=$< statement >$ "**.**" $< statement >$

$< statement >$::=$< var >$ "**:=**" $< logicOp >$

$< logicOp >$::=$< var >$ "**AND**" $< var >$

| $< var >$ "**OR**" $< var >$

| $< var >$ "**XOR**" $< var >$

|"**NOT**" $< var >$

It can be noted that the language allows for up to 16 bits of memory, as there are 8 input bits and 8 output bits.

3.1.1 Coils

In this context, for the PLC, the coil is a digital output signal that is manipulated by the program. These coils are connected to RAM modules which are used for performing logical operations on the input and output variables. In the DSL grammar the coil syntax is defined as such:

$< coils >$::=$< coil >$ "**.**" $< coil >$

$< coil >$::=$< var >$

Coils are therefore defined as a variable $< var >$ which in turn starts with 'I'(indicating input variable),'M'(indicating memory variable), or 'Q'(indicating output variable) and is followed by a two-digit number that helps in specifying the specific coils. The $< coils >$ rule in the grammar uses the dot to specify the input and output coils for a RAM module, the first coil before the dot being the input, and the one after being the output.

3.1.2 Contacts

Contacts are variables used in the $< ram >$ rule for indicating which coils are being used, and they are specifically defined as such:

$< CN01 >$::= "**CN01**"

$< CN02 >$::= "**CN02**"

$< CN03 >$::= "**CN03**"

$< CN04 >$::= "**CN04**"

3.2 Code example

A presentation of a code, that was constructed using the DSL described in this paper is shown in Fig. 2.2.1:

```
BEGIN
v
INPUT
I 011. Q 011;
OUTPUT
Q 011. M 011;
RAM
CN01.AND I 011. Q 011 . CN04.NOT Q 011. I 011;
I 011 := NOT I 011
END
```

Figure 3.2.1 - DSL code example

This code snippet is written in a hardware description language (HDL) and defines a simple combinational circuit that takes an input signal I 011 and produces two output signals, Q 011 and M 011. The circuit also uses a RAM module with a 1-bit address input and 1-bit data input and output.

The RAM module is used to store the current value of Q 011, which is updated on each clock cycle based on the current value of I 011. The CN01.AND and CN04.NOT gates are used to control the write and read operations of the RAM module, respectively.

The last line of the code snippet toggles the value of I 011, which could be used to simulate the input changing from 0 to 1 or vice versa. Overall, this code snippet demonstrates the use of an HDL to describe a simple digital circuit and highlights the importance of understanding the underlying hardware implementation when designing such circuits.

3.3 Derivation Tree

A derivation tree or parse tree is a graphical representation that illustrates the way in which strings in a language are being derived, taking into consideration the rules of the grammar. Creating a parser for

Figure 3.3.1 - Parsing tree for the program example

a Domain Specific Language (DSL) for Programmable Logic Controllers (PLCs) is essential because it allows engineers and programmers to write code in a more human-readable and intuitive format. Rather

than writing low-level machine code or using graphical interfaces, DSLs allow users to write code in a more natural language, making it easier to understand and modify. Fig. 2.3.1 illustrates the derivation tree obtained from an example of code, by using ANTLR as a tool for parsing.[3]

Conclusions

In conclusion, Programmable Logic Controllers (PLCs) are essential components in modern industry for controlling manufacturing processes. They are easy to program, fast, interoperable, and can operate in a wide array of conditions. Furthermore, the most commonly used programming language for PLCs is ”ladder logic”, but as the industry evolves, more complex and maintainable programs are needed. Domain-specific languages (DSLs) provide high-level abstractions that can simplify the programming process, reduce programming time, and increase maintainability. DSLs for PLCs can be designed with a grammar that includes essential elements such as variables, logical operations, and program structures. By utilizing a DSL for PLCs, the programs created can be more maintainable, efficient, and flexible compared to those created with ladder logic. Although relays are still used in contemporary control systems, PLCs have significantly reduced the amount of hardwiring required for traditional relay control circuits. PLCs have become essential for replacing relay logic in many applications. As a result, with the potential benefits of DSLs for PLCs, it is likely that their demand will increase as industry continues to evolve, and they will become an increasingly vital component in industrial automation.

Bibliography

[1] Markus Völter, *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*, CreateSpace Independent Publishing Platform, 2013, p.26

[2] Domain-Specific Languages Guide. Accessed March 22, 2023. https://martinfowler.com/dsl.html#:~:text=A%20Domain%2DSpecific%20Language%20(DSL,as%20computing%20has%20been%20done.

[3] Frank D. Petruzella, *Programmable Logic Controllers 4th Edition*, McGraw Hill, 2010, p.2

[4] The reasons Why PLC Control System Fail. Accessed February 17, 2023. https://gesrepair.com/programmable-logic-controller-failure/

[5] Matthew JOURDEN "Function Block Diagram Programming with PLC Tutorial"[Online] Accessed February 17, 2023. https://www.brightonk12.com/cms/lib/MI02209968/Centricity/Domain/517/Tutorial%2006%20Function%20Block%20Diagram%20Programming%20with%20PLC%20Tutorial.pdf