Tampereen yliopisto

Niko Rinko

EFFICIENT IEC 61131-3 STRUCTURED TEXT TOOLING IN MODERN DISTRIB-UTED CONTROL SYSTEM

Master of Science Thesis
Faculty of Engineering and Natural Sciences
Examiners: Professor Hannu Koivisto,
University Instructor Mikko Salmenperä
November 2022

ABSTRACT

Niko Rinko: Efficient IEC 61131-3 Structured Text tooling in modern distributed control system
Master of Science Thesis
Tampere University
Master's Degree Programme in Automation Engineering
November 2022

Structured Text is one of the programming languages defined in the IEC 61131-3 standard. The standard defines several programming languages that can be used in automation control software. Programmers have accustomed to use various advanced tools to develop programs efficiently. A good development tool increases the efficiency by making the development both easier and faster.

The goal of this thesis is to find out what is required to develop an efficient development tool for Structured Text language that is integrated into a web-based automation platform, while taking the platform constraints into account. The thesis has two research questions. One aspect is to research what kind of features are useful and required for the tool. Another goal is to find out how the features should be implemented. The requirements were collected via user interviews, examining existing tools and by using developers' experience and opinions to determine most useful code editor features.

As all interviewed persons were inexperienced with Structured Text, the first version of the tool was implemented before the user interviews to spark the discussion. The findings from the interviews were then used to further plan the tool and analyze how useful the implemented features are.

Most of the findings from interviews were biased towards features which make the development easier for beginners who are not familiar with the Structured Text language. For example, syntax documentation was the most desired feature. As the number of conducted interviews is low and the interviewees have such a strong bias, the results are not very generalizable. The implemented system was considered a great step forward and especially the debugging features were very positively received by the interviewees. The implementation supports features that were considered important by the developers. These features are automatic suggestion and completion of keywords, functions and variables, syntax highlighting, and basic syntax error reporting.

The goal was to improve the user experience of Structured Text development inside Valmet's configuration tools and this goal was reached. However, the tool has one major technical flaw in the Structured Text parser. The implemented parser does not support meaningful error messages and a new parser is required to improve the error messages. Despite the limited and biased data, this thesis provides some guidelines of what features are required for an efficient programming tool from the point of a user who is not familiar with the language.

Keywords: IEC 61131-3, Structured Text, automation platform, code editor

TIIVISTELMÄ

Niko Rinko: Tehokas IEC 61131-3 Structured Text ohjelmointi työkalu modernissa hajautetussa ohjausjärjestelmässä
Diplomityö
Tampereen yliopisto
Automaatiotekniikan DI-ohjelma
Marraskuu 2022

Structured Text on eräs IEC 61131-3 standardissa määritellyistä ohjelmointikielistä. Standardi määrittelee useita ohjelmointikieliä automaation ohjausjärjestelmien käyttöön. Ohjelmoijat ovat tottuneet käyttämään erinäisiä kehittyneitä työkaluja kehittämään ohjelmia tehokkaasti. Hyvin suunniteltu kehitystyökalu parantaa tehokkuutta tekemällä ohjelmoinnista helpompaa, sekä nopeampaa.

Tämän työn tarkoituksena on löytää vaatimuksia tehokkaalle Structured Text ohjelmointityökalulle, joka integroidaan web-pohjaiseen automaatioalustaan ottaen alustan tuomat rajoitteet huomioon. Työllä on kaksi tutkimuskysymystä. Yksi näkökulma on tutkia, minkälaisia ominaisuuksia työkalu vaatii ja mitkä ominaisuudet ovat hyödyllisiä. Toinen tavoite on selvittää, kuinka nämä ominaisuudet tulisi toteuttaa. Vaatimuksia selvitetään käyttäjähaastatteluilla, tutustumalla olemassa oleviin työkaluihin ja hyödyntämällä ohjelmistokehittäjien kokemusta ja mielipiteitä hyödyllisistä koodieditorin ominaisuuksista. Koska Structured Text on lähes tuntematon kaikille haastateltaville, työkalun ensimmäinen versio toteutettiin ennen käyttäjähaastatteluja, jotta esimerkki työkalua voidaan käyttää herättämään keskustelua. Haastattelujen tuloksia hyödynnetään analysoidessa työkalun ja projektin onnistumista.

Suurimmaksi osaksi haastatteluista johdetut tulokset olivat painottuneita ja korostivat voimakkaasti aloittelijaystävällisiä ominaisuuksia, joista on hyötyä kehittäjille, jotka eivät tunne Structured Text kieltä entuudestaan. Esimerkiksi kielen syntaksi dokumentaation saatavuus oli selkeästi toivotuin ominaisuus. Koska toteutuneiden haastattelujen lukumäärä on pieni ja haastateltavat edustavat tiettyä käyttäjäryhmää, tulokset eivät ole helposti yleistettävissä. Toteutettua järjestelmää pidetään kuitenkin suurena askeleena oikeaan suuntaan ja erityisesti vianetsintään liittyvät ominaisuudet otettiin käyttäjähaastatteluissa erittäin positiivisesti vastaan. Toteutus tukee muutamaa ominaisuutta, jotka olivat työkalun toteutuksen parissa työskennelleiden mielestä tärkeitä ominaisuuksia. Nämä ominaisuudet ovat avainsanojen, muuttujien ja funktioiden automaattinen ehdotus ja täydennys, syntaksin korostus ja yksinkertainen syntaksivirheiden korostus.

Tavoitteena oli parantaa Structured Text ohjelmoinnin käyttäjäkokemusta Valmetin konfigurointityökaluissa ja tässä työ onnistui. Työkalussa on kuitenkin yksi merkittävä tekninen puute Structured Text parserissa. Toteutettu parseri ei tue virheiden yksityiskohtien raportointia. Rajallisesta ja painottuneesta tutkimusdatasta huolimatta, tämä työ tarjoaa osviittaa millaisia ominaisuuksia kieltä osaamaton käyttäjä kaipaa ohjelmointityökalulta.

Avainsanat: IEC 61131-3, Structured Text, automaatiojärjestelmä, koodieditori

PREFACE

This thesis was made for Valmet Automation Oy in 2022. This thesis has been very educational experience for me with a lot of new subjects to learn. I wish to thank especially Ari Leppäniemi and Jari Tamminen for the opportunity and the generous freedom to work on the thesis without pressure. Special thanks to Ari Leppäniemi for the great feedback on the thesis. I also want to thank Tapio Katajisto, Sergey Khentov, and other team members for helping me with my questions and problems during implementation phase.

I am also grateful to Professor Hannu Koivisto and University Instructor Mikko Salmenperä for great feedback and guidance at the beginning and the end of the process.

Finally, I wish to express my gratitude to my family for supporting me for all these years and encouraging me to finish my degree.

Tampere, 20.11.2022

Niko Rinko

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

AST	Abstract Syntax Tree
CSS	Cascading Style Sheets
CST	Concrete Syntax Tree
DAP	Debug Adapter Protocol
DCS	Distributed Control System
EBNF	Extended Backus-Naur form
FBD	Function Block Diagram
GDB	GNU Project Debugger
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
IEC	International Electrotechnical Commission
IL	Instruction List
IT	Information Technology
JSON-RPC	JavaScript Object Notation Remote Procedure Call
LD	Ladder Diagram
LSP	Language Server Protocol
npm	Node Package Manager
OOP	Object-oriented Programming
OT	Operations Technology
PLC	Programmable Logic Controller
POU	Program Organization Unit
R&D	Research and Development
RAM	Random Access Memory
SFC	Sequential Function Chart
SLSP	Specification Language Server Protocol
ST	Structured Text
UI	User Interface
(Valmet) DNA	Dynamic Network of Applications
XML	Extensible Markup Language

1. INTRODUCTION

Software development is increasingly growing area of business and a huge selection of tools have been developed to make software development easier. These tools are rapidly evolving and improving in the more traditional IT business. However, these tools are rarely developed to support automation specific programming languages such as Structured Text or Function Block Diagram. These languages are defined in standard IEC 61131-3 (International Electrotechnical Commission), which is a commonly used standard for automation control application programming. While the standard defines the languages, it does not take a stand about tools for producing the program.

Modern automation systems are utilizing more and more tried and tested tools from traditional IT business and thus the requirements for automation engineer's skills are becoming more inclined with the skills required from a software developer. As this happens, tools for developing automation software must become more like ones used in traditional software development. However, as automation has a long lifespan and most of existing automation applications can be considered custom from traditional IT standpoint, modern automation engineering tools are required to be somewhere in between traditional programming and automation specific software.

Valmet has a long history in developing and manufacturing distributed control systems for process control in various industries. This includes the programming tools for the systems. The primary method of programming the automation applications within Valmet's automation platform, DNA (Dynamic Network of Applications), is function block diagram. However, some programming concepts, such as while- and for-loops are difficult to create in the Function Block Diagram language. To improve the feasibility of such loop logic in automation applications, Valmet has added support for Structured Text (ST) language. The ST language support is integrated within the function block diagram as a special programmable function block as can be seen in Figure 1.

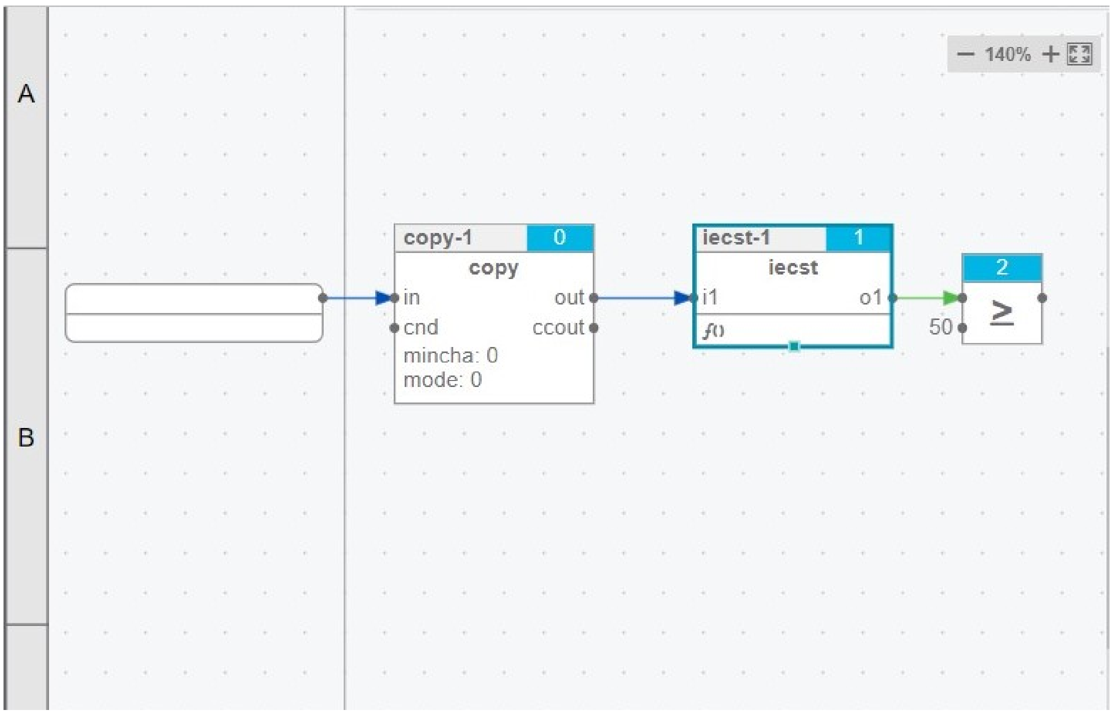


Figure 1. IEC ST function block as a part of function block diagram

The role of ST programming in Valmet DNA is not to replace function block diagrams, but rather provide an alternative, powerful way to extend the functionality of a function block diagram. The ST language and its usage requires some special knowledge about PLC programming and programming in general. To make the use of ST easier, the editing tool should provide helpful features and information to the user.

Valmet has previously implemented the compilation process and runtime for ST code, and a barebones code editor. This existing code editor resembles basic text editor such as Notepad from Microsoft Windows and does not contain any language specific features. This thesis focuses on the code editing user experience and aims to improve the user experience of ST code editing in Valmet's automation platform.

This thesis has two research questions (1) What features are required for efficient IEC ST programming in modern web-based automation platform, and (2) How these features should be implemented while taking the platform constraints into account? The required features are researched via user interviews and investigating existing tools for ST development and considering their features. Research is performed by interviewing end-users about the requirements and wishes for ST development tool. Additionally, existing tools and research for ST programming are studied to get broader picture of required features. Limitations for implementation are mostly pre-determined by existing platform. Therefore, limitations mostly only affect the choices made in implementation. ST development tool is first implemented based on few persons' opinions, showcased in the interviews, and afterwards evaluated against the findings from the interviews.

Chapters 2 and 3 consist of mostly theoretical background. These chapters include the brief overview of the IEC 61131 and especially its part 3, which is about programming languages for PLC (Programmable Logic Controllers). Languages defined in IEC 61131-3 are all briefly introduced in the chapter 2. All IEC 61131-3 languages share a decent amount of common concepts and definitions, some more important ones for this thesis of those are also introduced. Chapter 3 contains introduction and overview of modern code editor or IDE (Integrated Development Environment) features. And especially how they are implemented. Chapter 3 also introduces the basics of programming language parsing. Some of the modern code editor features are implemented into the ST tooling in this thesis using the implementation methods introduced in chapter 3. Concepts, such as a language server are crucial to understand the implementation of ST tooling. Already existing development tools for ST and previous research of subject are introduced in chapter 4. Chapter 4 also introduces other ST related projects that could be useful for this thesis. Interview methods are addressed in chapter 5, with the results of interviews. These results are further processed in chapter 6, where requirements and platform limitations are defined based on data received from interviews and other research methods. Chapter 7 consists of implementing ST tooling fulfilling requirements and complying with constraints. Implementation is only the first iteration of tooling, and it is not necessary to implement all the features in first iteration. Chapter 8 consists of evaluation of the meaningfulness and importance of identified requirements and constraints and how well implemented system fulfills requirements and complies with constraints. Final chapter 9 is about conclusions of this thesis.

2. IEC 61131-3

2.1 Brief overview of IEC 61131

IEC 61131 is a large standard or series of standards for PLCs containing guidelines for hardware, communication, software, and safety.

IEC 61131 is a widely used standard in automation. Instead of being called a standard, IEC 61131 is sometimes called a collection of standards [7]. As IEC 61131 is so large standard, it is currently divided to 10 parts which are updated individually. These parts are:

1. General Information
2. Equipment requirements and tests
3. Programming Languages
4. User Guidelines
5. Communication
6. Functional Safety
7. Fuzzy control programming
8. Guidelines for the application and implementation of programming languages
9. Single-drop digital communication interface for small sensors and actuators (SDCI)
10. PLC open XML Exchange Format [7]

Most encountered part is the third part, which is very important for this thesis as well. The third part focuses on the programming languages and defines the programming languages used in PLC programming. Hanssen states the standard aims to create specification so PLC manufacturers, programmers and users could understand each other better and create programs that could more easily be used in other PLC devices. The standard is more like a set of guidelines rather than absolute truth manufacturer's should follow to the letter. [6, Ch. 5.1] While the standard aims to make the PLC programs and devices more uniform, the manufacturers often have their own specializations in the programming language. This essentially makes the programs incompatible with other manufacturer's devices.

IEC 61131-3 defines both textual and graphical languages, two textual and three graphical languages. Textual languages are Structured Text (ST) and Instruction List (IL). Standard's graphical languages are named Function Block Diagram (FBD), Ladder Diagram (LD) and Sequential Function Chart (SFC). This thesis focuses mostly on the ST language, which is explained better in chapter 2.3.

The IEC 61131-3 standard has already three published versions with 4th edition currently in progress with estimated publication date of 30th July 2024. Original 1st edition was published in 1993. The standard was first revised in 2003 with the publication of 2nd edition. [24] The second edition fixed inconsistencies and contradictions of the standard and included some of the proposed enhancements and revisions. The 3rd edition released in 2013 is fully backwards compatible with the 2nd edition of the standard but includes a lot of additions and enhancements to the standard, most notably the object-oriented programming concepts such as classes, namespaces and interfaces.[25] The another textual language of the standard, IL, has been deprecated from the standard starting from the 3rd edition [26].

2.2 Common elements in IEC 61131-3 languages

All five languages defined by the standard share some features and architecture. One major shared feature is the program structuring into Program Organization Units (POUs). Each POU is a single compilation unit and the topmost program structure. All applications written in IEC 61131-3 languages consist of POUs. Tiegelkamp's book about PLC programming tells POU's can be compiled independently and linked together by compiler [7, Ch. 2.2]. As POUs are independent modules of code, they can be easily reused by other programs. There are three different POU types: Program, Function Block and Function. Program-type POU contains the main program code, global variables, and acts as a backbone of the program. Function Blocks slightly resemble objects in object-oriented programming (OOP) in a sense that they are functions which can also have a state. According to Tiegelkamp [7], function blocks are the most commonly used POU type. Functions are the most restricted and the simplest POU type. Function has input parameters and returns output parameter but does not have a possibility to define state variables that would persist between function calls.

All POUs share general structure of having variable declaration part and main code part. In dedicated PLC programming tools, these parts are often separated as own dedicated windows/code editors. Declaration part may contain interface variables, which are accessible from outside of POU and local variables which are internal to the POU. Code part of the POU contains all the functional code: variable assignations, calculation, loops, and conditions. Not every variable type is permitted in all POU types. Local variables and basic interfaces of types VAR_INPUT, VAR_OUTPUT, and VAR_IN_OUT are usable in all POU types. Global and external variables are more restricted and only usable in PROGRAM-type POUs.

All languages share common data types and most standard functions and function blocks work similarly in all languages. Data types of IEC 61131-3 languages can be divided to two categories: elementary types and derived types. Elementary types are basic, built-in data types, which are the smallest building blocks of the language structures. Derived types are user defined types, which can be anything from complex multi-dimensional structures to simple alias for elementary data type. Tiegelkamp has a very informative table of elementary data types which is replicated below in Table 1.

Table 1. *Elementary data types of IEC 61131-3 [7, Ch. 3.4]*

Boolean/stringbit	**Signed integer**	**Unsigned integer**	**Floating point (Real)**	**Time, date, and character string**
BOOL	INT	UINT	REAL	TIME
BYTE	SINT	USINT	LREAL	DATE
WORD	DINT	UDINT		TIME_OF_DAY
DWORD	LINT	ULINT		DATE_AND_TIME
LWORD				STRING

Meaning of the first letters: D = double, L = long, S = short, U = unsigned

2.3 Structured Text

IEC 61131-3 defines textual programming language Structured Text. ST is a high level programming language derived from Pascal, other high level textual programming language [26]. Structured Text is a strongly typed language, which requires every variable to have a static type. Structured Text language consists of statements which end with a semicolon. In a sense, Structured Text is a list of statements which are executed in order in a similar fashion to all popular textual programming languages. These statement lists are encapsuled within POU declarations' code section. ST also supports common conditional statements and loops like WHILE and FOR. ST resembles modern popular programming languages more than any other IEC 61131-3 language. An example code written in ST is shown in program 1. In a world where automation and more traditional IT are constantly closing in on each other, ST can become a great common language between IT and OT (Operational Technology) programmers.

```
PROGRAM example
   VAR_IN
         A: BOOL;
   END_VAR
   VAR_OUT
         X: BOOL;
   END_VAR
   IF (A)
         X := A;
   END_IF
END_PROGRAM
```

Program 1. *Structured Text example*

2.4 Function Block Diagram

Function block diagram is a graphical programming language, which consists of blocks with input and output ports and wires connecting these blocks together. These wires can be split, and a single output can be split to multiple inputs. Blocks and wires form a network of blocks, which represents the program logic. Essentially the wires represent the data flow in the program, while the function blocks are operations done to the data. One great benefit to the FBD (Function Block Diagram) language is its function blocks' reusability. It is very modular by design and allows for non-programmers to write complex logic more easily.

Figure 2 is an example of FBD language syntax originally made via CoDeSys

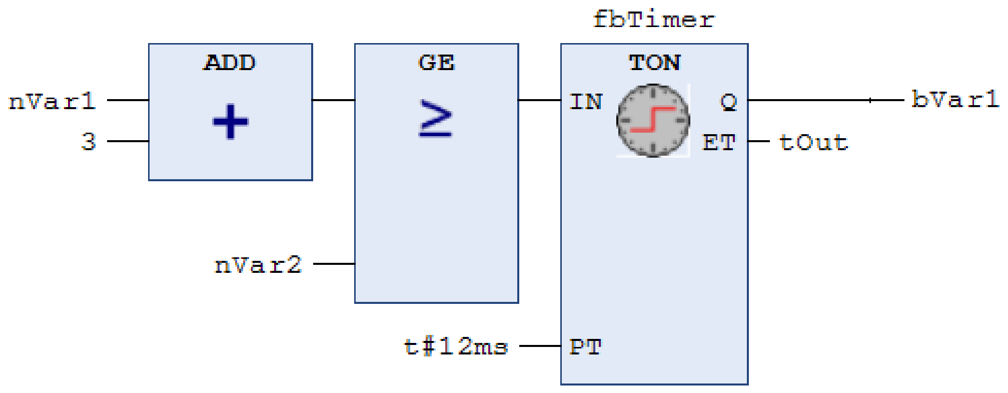


Figure 2. Function Block Diagram example [27]

For distributed control systems using FBD, there exists another standard called IEC 61499. The standard currently has three parts, 1: Architecture, 2: Software tool requirements and 4: Rules for compliance profiles [28]–[30]. Part three has been withdrawn in

2008 [24]. The IEC 61499 aims to standardize the distributed control system (DCS) architecture by defining an application model using function blocks. The IEC 61499 extends the FBD language defined in the IEC 61131-3.

2.5 Other languages in IEC 61131-3

The other three languages of the standard are of little importance for this thesis. They are however briefly introduced in this chapter for the sake of being crucial part of the standard.

2.5.1 Instruction list (IL)

Instruction list is marked as deprecated language in the standard's third edition IEC 61131-3:2013 [26]. Instruction list resembles assembly language and is not very similar to modern programming languages such as JavaScript or Python. This makes it hard to understand and read, while ST is more easily understandable and allows all the same functionality. Hanssen has stated IL is efficient regarding required computing power [6, Ch. 5.2.4]. This is one of the few benefits of the language compared to the other IEC 61131-3 languages. IL can possibly be still found in old control applications as it is one of the original languages of the standard. CODESYS, one popular PLC programming IDE (Integrated Development Environment), has stated they no longer maintain IL support but it can still be used in CODESYS [31].

Program 3 is an example of a very simple IL program. Essentially program 3 is a simple AND-statement, which in pseudo code could be written as

```
If(A & B){
   X = true;
}
```

Program 2. *Program 3 in pseudo code*

```
LD A       LD A
LD B       AND B
ANB        ST X
ST X
```

Program 3. *Instruction list example [32]*

2.5.2 Ladder Diagram (LD)

LD is another old language, and very popular one as it is a graphical language which makes PLC programming easier to people who are not familiar with more traditional textual programming. According to Yi and Hangpin LD is the most widely used language among the standard's languages. They also state that LD being a graphical language the code is always transpiled to instruction list first before PLC can run it. [22] However de Sousa and Catalao describe there being three distinct approaches to running code on PLC. Compiling the code to assembly or machine code, using a virtual machine, or running a code interpreter. De Sousa's IEC 61131-3 compiler MatIEC first compiles other languages to ST before compiling ST into C code. [16]

Figure 3 shows the simplified compilation workflow from graphical IEC 61131-3 language to executable machine code. MatIEC does not compile the code into directly executable native code, but C code. Generated C code can then be further compiled into native code by the platform specific C compiler. For example, GNU Compiler Collection (GCC) can be used to compile native x86-64 code from the C code.

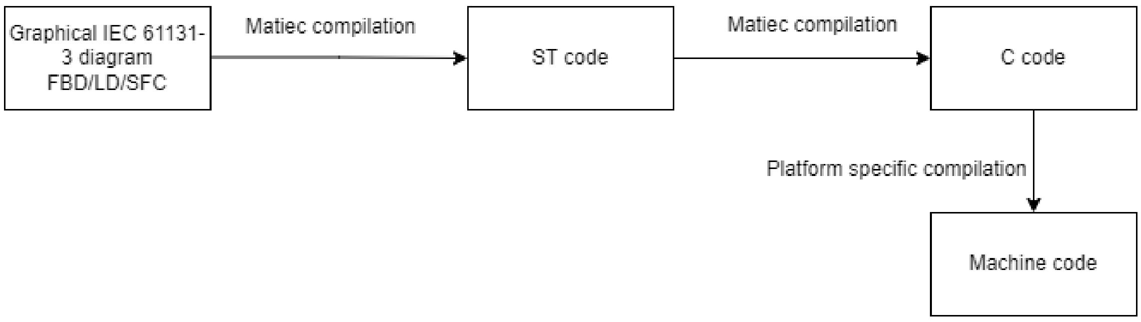

Graphical IEC 61131-3 diagram FBD/LD/SFC
Matiec compilation
ST code
Matiec compilation
C code
Platform specific compilation
Machine code

Figure 3. Simplified compilation workflow using MatIEC

So transpiling, or source-to-source compiling, to IL is not the only way, especially now that IL has officially been deprecated. Arguably, MatIEC is more of a transpiler rather than compiler, since it produces source code in another high-level language rather than code which is ready to be executed.

LD is primarily used to program Boolean variables with symbols resembling electrical relays [7, p. 147]. In LD, programs are drawn from top to bottom, left to right and the end product sort of resembles ladder. Vertical lines evaluate all incoming horizontal lines from left with OR and copy the result to all outgoing horizontal lines to right [7, p. 149]. The horizontal lines contain the function blocks and operators which form the program logic. Example of a simple LD program can be seen in Figure 4.

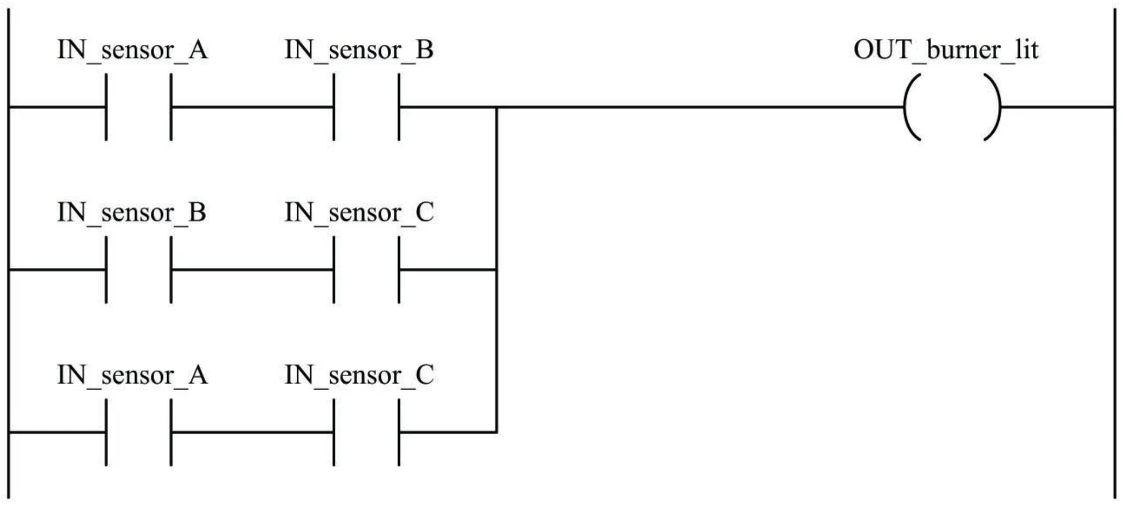


Figure 4. Ladder Diagram example [33]

2.5.3 Sequential Function Chart (SFC)

SFC is useful for programming process or manufacturing sequences. Tiegelkamp states SFC was made to divide a complex program into smaller units which can be parallelized [7, p. 169]. SFC diagrams run step by step. Each step has functions which define what is done while the step is active and transitions, which define when step is finished and next one should activate. Figure 5 contains an example of an SFC program.

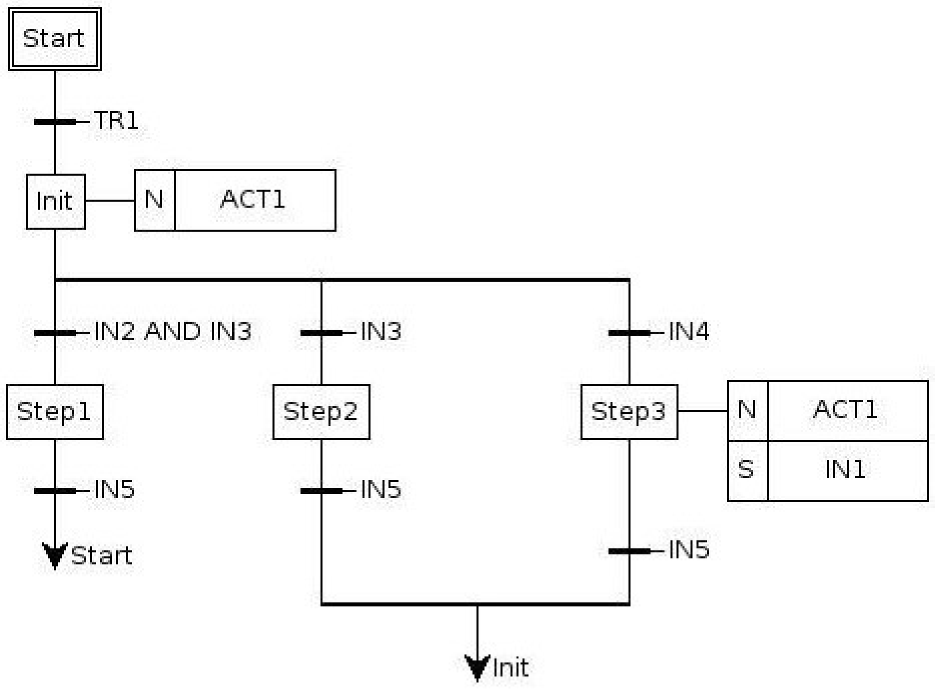


Figure 5. SFC example [20]

3. MODERN CODE EDITOR FEATURES

Modern code editors and IDEs have numerous features to help programmer create code efficiently. Features such as code completion suggestions, intelligent syntax error messages, snippets, syntax highlighting are very common in modern code editors. Sulin et al. have studied source code editors augmenting code with additional annotations and came to conclusion nearly all source code editors have some sort of augmentation support. Most of the visualizations were done with color or icons, with color being clearly the most popular.[18] Author's experience with programming IDEs also supports this claim, as the syntax highlighting is almost always done with colors. With code editors having so clear common de facto standard about the tooling style, one question arises whether similar style should be applied also to the ST language tooling instead of taking inspiration from more automation-oriented tools. One way to research this is to ask users if they would feel comfortable using tool with features styled as such.

Developing a feature rich tooling for a programming language also requires deep understanding of the language, similarly to compilers. Compilers and language tooling however differ in some ways. While for compilers it is acceptable, although sub-optimal, to stop parsing in case of error, this is not acceptable for a code editor's parser. When compiling, the process cannot proceed unless each stage produces valid output and any error aborts the compilation process. When editing the code, user expects advanced editor features such as autocompletion to work even when the code has errors. In other words, code editor's parser must have some sort of error recovery and produce sensible output even when the input contains errors. For compilers it is also beneficial to show all errors instead of just the first one, but the error recovery is not as important for parser designed for compiler than it is for editor.

One goal of this thesis is to find a way to create these modern code editor features into a code editor integrated into automation platform. To do this, one must first understand the basics how these features are or can be implemented. Almost all modern code editor features require parsing and some sort of analysis of the source code. The analysis can be roughly divided to two categories, syntactical, and semantical analysis.

3.1 Syntactical analysis

Modern code editors generally have syntactical analysis to be able to highlight or otherwise inform user about the syntax errors in the input. This requires programming the knowledge of code syntax which can be a daunting task if the language is complex and very context dependent. In other words, a lexer and parser for the language is required. This is also stated in the introduction of paper on syntax error reporting and recovery in parsers using specific type of grammars [12]. Parsing techniques are widely researched subject with hundreds of papers released and dozens of tools developed. However, due to the complexity of the problem, no tool is perfect, and every technique appears to have some sort of drawback. There is a large book written which is said to summarize over 700 papers on parsing [5].

In a book about compiler design [3], the goal of syntactical analysis is described as conversion of given source code into intermediate representation which is then used for other analysis and/or compilation. In the book, the syntactical analysis is referred with term “front-end” which is nowadays more commonly associated with web development. The syntactical analysis generally consists of at least two stages, lexical analysis, and parsing. In lexical analysis the character stream is mapped to list of tokens according to specific tokenizer rules. In a sense the tokenizer rules define the vocabulary of a language, all the keywords and identifiers. Essentially lexical analysis stage is about splitting the entire input into elementary language concepts such as keywords or identifiers. Fictional example of a result produced by lexer is pictured in Figure 6.

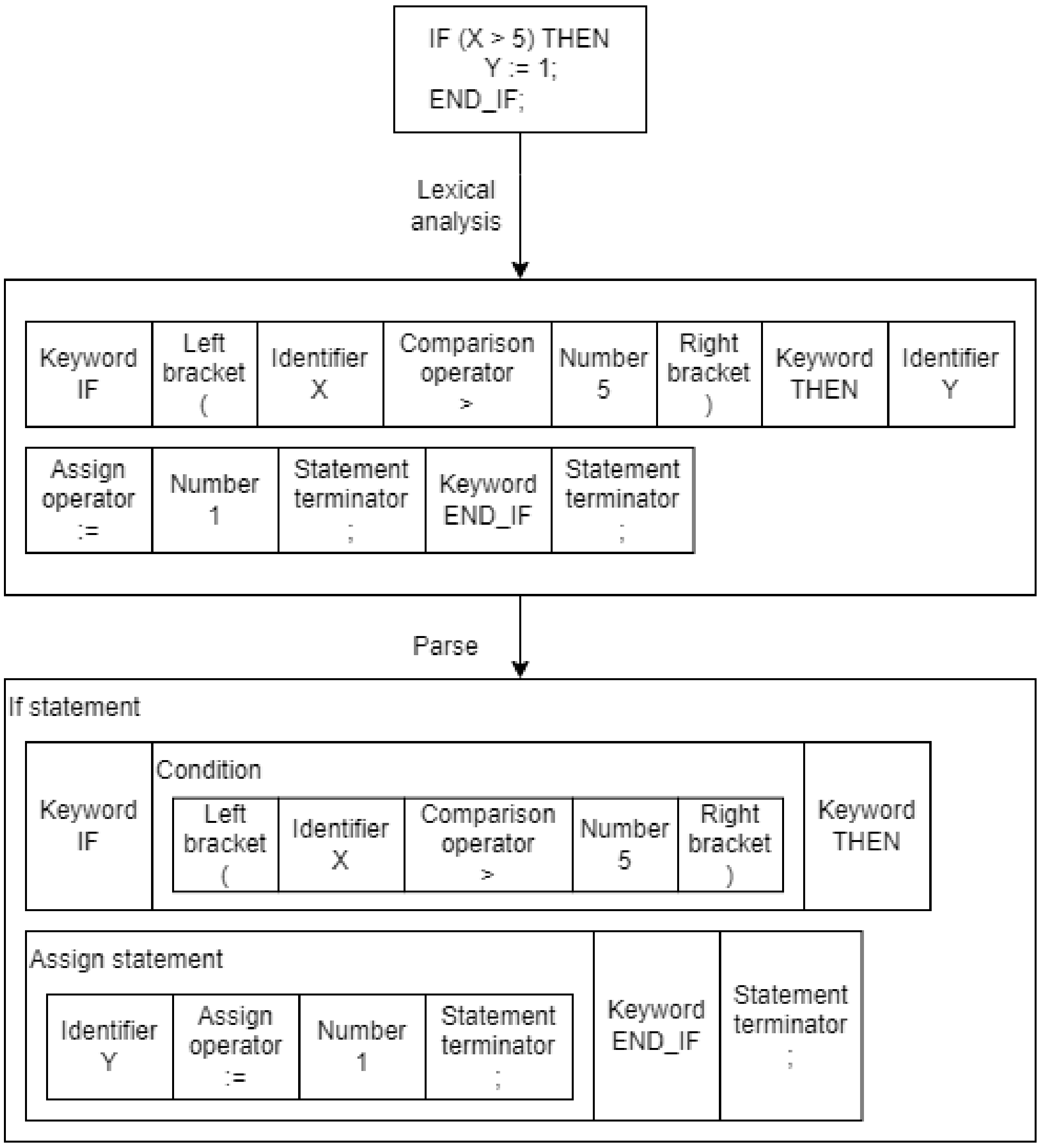


Figure 6. Example of lexer and parser workflow

Parsing is the next stage where these tokens are combined according to grammar rules producing higher level language constructs such as if-statement or variable declaration. Example of this can also be seen in the Figure 6, where the result of lexical analysis is parsed by parser.

Parser usually produces these in a hierarchical tree-structure representing the used language concepts from higher level structure down to elementary structures such as identifier. Example of this can be seen in Figure 6 where the construct named expression is very high level and generic construct which is specialized by nested rules.

The tree directly produced by parser without any additional optimization is called Concrete syntax tree (CST) and it contains all the nodes exactly like parser identified them based on grammar [34]. Figure 7 depicts an example CST for the input of “return a +2;”.

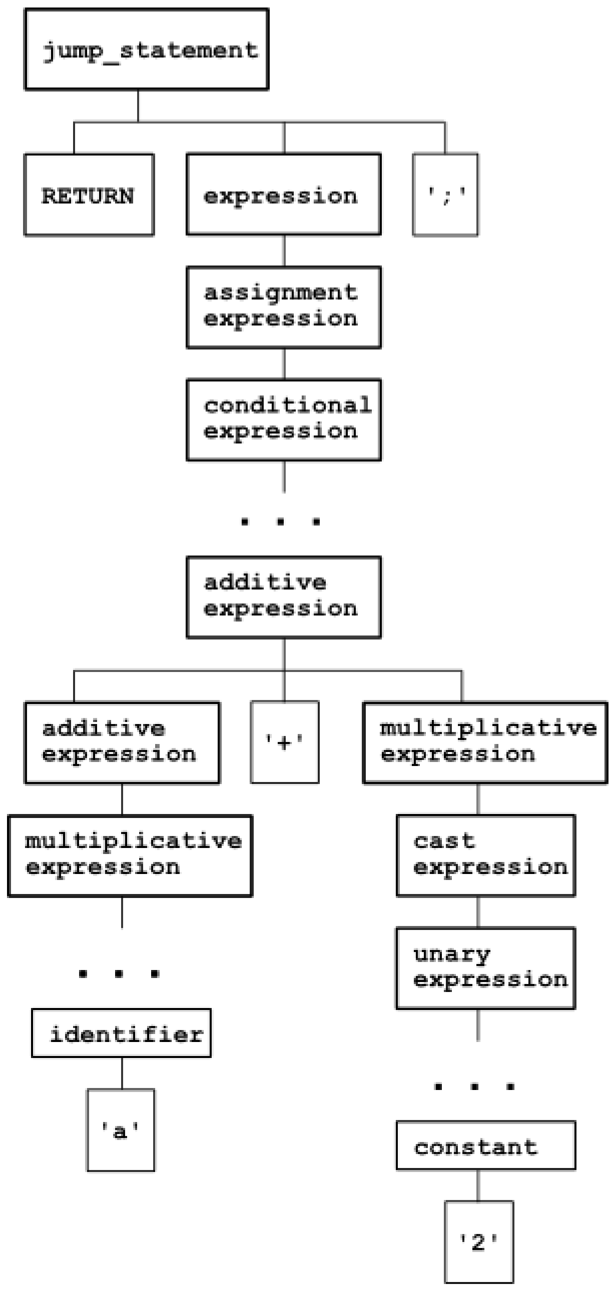


Figure 7. Concrete syntax tree for "return a + 2"[34]

The CST can however be optimized by removing unnecessary higher-level nodes from the tree. For example, if tree has an if statement node, it is unnecessary to have higher level node indicating it is a statement. This kind of reduced or optimized tree is called AST(Abstract Syntax Tree) [34]. Difference between the CST and AST is that CST includes every matched grammar rule in the nodes, while the AST is trimmed, and only meaningful data is kept. This makes AST faster and easier to walk through and analyze. CST naturally contains the same information, and it can also be used for more in-depth analysis, but usually using AST as basis for further analysis is easier. Creating AST can be a complex task but due to the mentioned benefits it often is worth the effort.

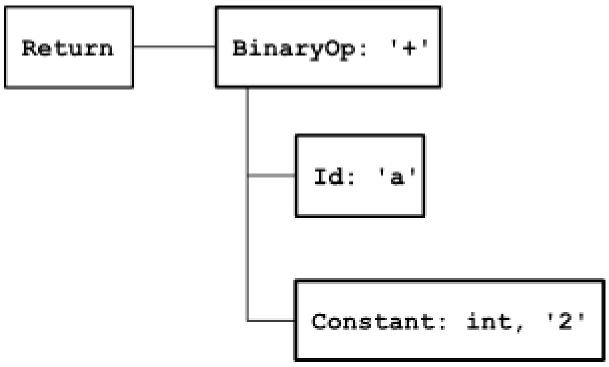


Figure 8. Abstract syntax tree for "return a + 2"[34]

In Figure 7 there is a concrete syntax tree for simple return statement “return a + 2”. Figure 8 shows an abstract syntax tree for the same input. When comparing the Figures 7 and 8 it is obvious the concrete syntax tree is significantly larger than the abstract syntax tree. When considering these examples were done with a single statement, one can imagine how much larger CST is compared to well-trimmed AST when using entire source code file with hundreds of statements as the input.

The parsing is quite straightforward when the input is valid and conforms to the grammar, but in the case of code editing, the code is more often invalid than valid. This makes the error recovery features of the parser highly important as the tooling features depend on the produced syntax trees. The errors in input can be generally recovered in two different ways, inserting a token or removing a token [19]. The error recovery can become quite complex task and while there are multiple methods for it, none fit for every situation. The complexness stems from the questions *What should be inserted/deleted?* and *How many tokens should be inserted/deleted?* It is difficult to figure rules for recovery which would fit for all possible syntax errors. In paper by Quieiroz de Medeiros et al. they developed an algorithm for automatic syntax error recovery and reporting. The new algorithm managed to report most of the errors excellently, but still the evaluated rate of excellently reported and recovered errors is 64% at best. [12] This highlights the complexness of the generalized error recovery and reporting.

3.2 Semantical analysis

While syntax analysis focuses on input being correct grammar wise, it does not analyze if the input is meaningful outside of conforming to the grammar. Semantic analysis is executed to check if the code has a sensible meaning. Usually, semantic analysis methods require analyzing the code from wider point of view than in syntactical analysis. One common type of semantic analysis is type checking. For ST type checking, the correctness of variable types must be checked from two sections of the code. Variable types are defined in the variable declaration part and when these defined variables are used, the analysis must check variable declarations to figure out if the types of two variables match. As an example, code may be syntactically correct and having variables a and b, of type int and string respectfully. Now if there is a statement where a is assigned the value of b, it is syntactically correct but semantically invalid. When the analysis encounters the assign statement, it must confirm the two variables have matching types by checking the declaration part. Of course, this does heavily depend on the language used, but ST is strongly typed and does not allow this kind of type mismatching. Below program 4 shows example of an ST program which is syntactically correct but semantically incorrect. Integer cannot be assigned to boolean in ST. This is not however found in syntactical analysis since *identifier := identifier* is valid syntax.

```
PROGRAM example
   VAR_IN_OUT
         IN: INT;
         OUT: BOOL;
   END_VAR
   OUT := IN;
END_PROGRAM
```

Program 4. *Syntactically correct but semantically incorrect ST program*

De Sousa has published a paper [15] on type checking for ST language and implemented type checking in the MatIEC compiler. Generally, compilers for strongly typed languages implement type checks.

Another analysis method is to differentiate between variables with same names residing in separate scopes. For example, two variables both named x can be differentiated to be separate semantic entities if they for example are defined in different functions. This kind of analysis can be used to implement “rename symbol” feature to the code editor. This is more advanced than simple find and replace functionality, since the code may have multiple variables with same name, but they are semantically different due to the scopes of code. Program 5 shows short example code in ST how variable names can be re-used in different scopes. Both FUNCTION and PROGRAM have their own variable scopes and the variable IN is declared in both. These IN variables are different although they have the same name and type. Simple find & replace functionality would change all occurrences of the variable, but with some scope analysis the refactoring can be limited to affect only semantically identical variables.

```
FUNCTION func: INT
   VAR_INPUT
         IN: INT;
   END_VAR
   func := IN + 1;
END_FUNCTION

PROGRAM main
   VAR_IN_OUT
         IN: INT;
         OUT: INT;
   END_VAR
   OUT := func(IN);
END_PROGRAM
```

Program 5. *Example ST program demonstrating the re-use of variable name in different scopes*

One interesting use case for an AST is introduced in paper by Liang et al: usage of an custom, more information rich, AST to automatically locate bugs in source code based on deep learning algorithm [10]. The algorithm takes source code and bug reports as input and parses and compares both to identify likely locations which cause the bug. This kind of advanced analysis is not common in popular code editors or IDEs. However, it is interesting idea and is a good example how advanced analysis is possible with properly constructed AST and modern advanced technology.

3.3 Language Server Protocol

The actual analysis of the code is often done in its own process in the background to not block the editor UI. The output of the analysis still must be compatible with the editor API to visualize the results. Microsoft attempts to solve the problem of having to make an adapter for every language's analysis software by defining Language Server Protocol [35]. LSP (Language Server Protocol) aims to define universal protocol for language analysis result format and the requests for analysis. The separate analysis software, called language server by the protocol, must implement the protocol interface, and communicate adhering to the rules of the protocol. The internal implementation of the server is not defined or constrained by the rules of the protocol and can be freely designed by the developer. However, in practice the rules of communication guide the implementation to use specific data structures. Language server using the protocol could be used in any code editor which implements the language server protocol's client interface. And this also enables any code editor using LSP to use any language server available.

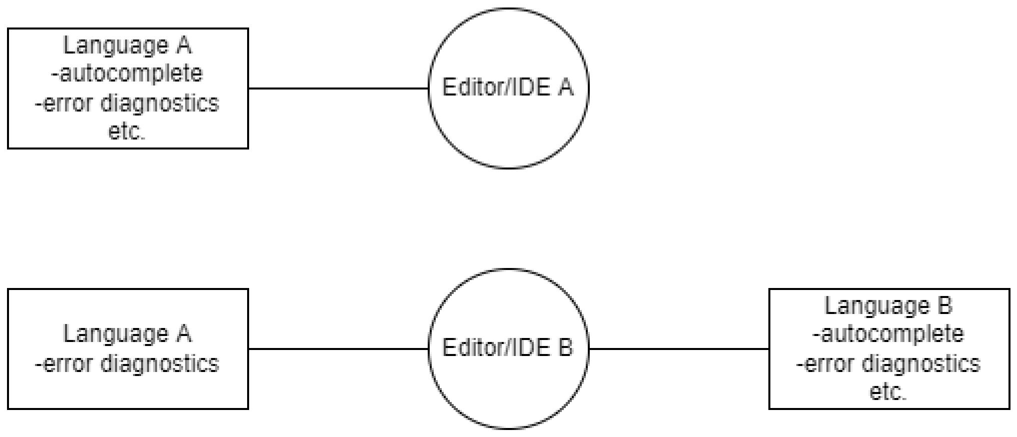


Figure 9. Language specific tooling without LSP

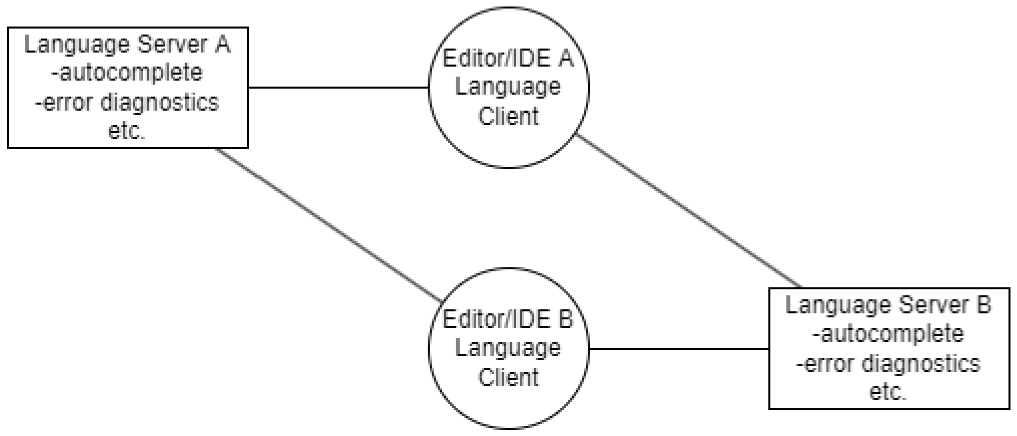


Figure 10. Language specific tooling with LSP

Figures 9 and 10 above show examples of simple situation where there are two editors and two languages. In Figure 9, Editor A has implemented services for language A, but does not support language B. Editor B has implemented services for language B and some services for language A, but the feature set is smaller than in the editor A. The Figure 10 shows the main benefit of LSP when used by all editors and language specific tooling, both editors can use the feature rich language servers, increasing the functionality of both editors.

On the other hand, using a protocol defined and maintained by third party, the protocol may restrict usability of bleeding-edge technology like the previously mentioned analysis of bug reports to locate the code causing the bug. Kjær Rask et al have published a proposal to fix this downside of the LSP. They propose another standard or extension to the LSP specification to standardize LSP extensions called Specification Language Server Protocol (SLSP). According to Kjær Rask et al. the downside to extending LSP is the necessary initialization and synchronization messages which can be sometimes a bit over-complicated to implement for a simple language feature which does not require synchronization.[8] The extension of a specification is always a difficult problem since both clients and servers must support the extensions and the more there are extensions, the less there are benefits from the original idea of LSP. This SLSP is step in the right direction, but it does not seem generic enough to completely solve the issue. Kjær Rask et al. themselves state the generalization of the protocol for all languages is left for future work [8].

3.4 Debugging

Modern IDE’s often offer support for debugging program and mapping debug data to the source code. The goal of a debugger is to allow developer to see the state of the executed program and control the execution. Debuggers are generally separate programs which are integrated seamlessly into IDEs. Generally, debuggers are language specific and support only single programming language. For example, pdb for Python, or jdb for Java.

As with the language servers, Microsoft has developed a protocol called Debug Adapter Protocol (DAP) to connect external debuggers to IDE’s user interface.[36] The idea of DAP is to separate debugger front-end or user interface from the actual debugger implementation and allow the use of a generic debugger user interface with multiple debuggers. DAP’s website explains it is unlikely older debuggers add built in support for DAP and instead adapters for them are recommended.[36] Figures 11 and 12 show the differences how the debuggers are connected to IDEs and editors without and with DAP, respectfully. The main goal and benefit of DAP is to re-use existing debuggers and provide an easy way to connect a single debugger to multiple development tools.

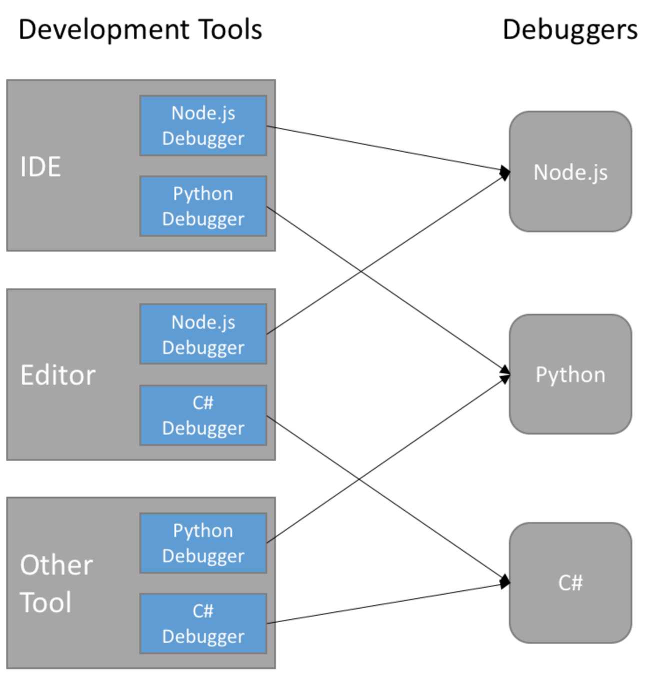


Figure 11. Debugging without DAP[36]

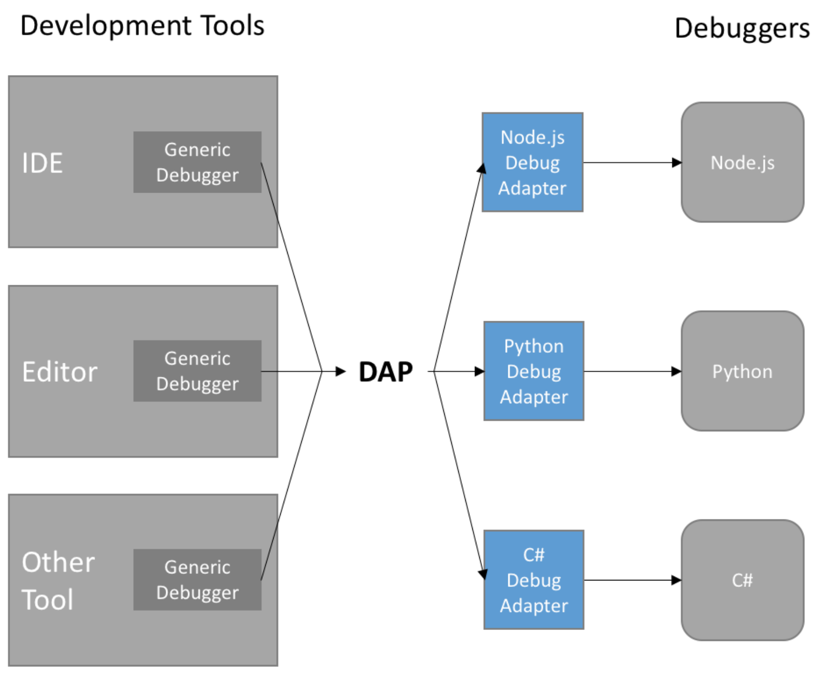


Figure 12. Debugging with DAP[36]

One of the most widely known debuggers is the GDB (GNU Project Debugger). GDB supports several programming languages [37], most notably C++ and C, but not the ones specified in IEC 61131-3. The debuggers like the development tools for IEC 61131-3 languages are generally proprietary software. The proprietary but free, cross-platform PLC IDE CoDeSys has a built in debugger for these languages [38]. Most PLC tools appear to offer some sort of simulation mode, where the application can be debugged even without the physical hardware.

One way of implementing debugger for embedded system, which is somewhat close to a PLC, is presented in a paper by Dolinay et al. where a debugger is implemented for an Arduino board. The debugger is implemented as a GDB stub. Essentially Dolinay et al. made a program library which listens and executes commands from GDB and sends the requested data to debugger front-end which is running on user’s computer.[4]

Modern web browsers also have integrated debuggers for debugging JavaScript, CSS (Cascading Style Sheets) and HTML (Hypertext Markup Language), the major languages for web. As debuggers are common in programming, it is worth investigating if a debugger exists for the language before starting development of own debugger. In addition to source code debugger, modern browsers include various other development tools such as performance monitor or network request logs. Figure 12 below is a screenshot of Google Chrome’s DevTools as of version 105.

4. EXISTING TOOLS AND RESEARCH FOR STRUCTURED TEXT DEVELOPMENT

Most ST parsers and static analysis tools are proprietary software owned by PLC manufacturers. While the open-source culture has been prevalent in IT, OT systems are still generally only used by companies and there are much less open-source versions of OT systems. There however are a few open-source automation targeted software and as the IT and OT continue to converge, there probably will be more open-source software for automation in the future.

ST language has at least one unofficial extension called poST, which is introduced in paper by Zyubin et.al. One major addition feature of poST is the addition of states into the ST language. [23] States make some concepts easier to program. It is a bit unclear if the poST language is supported by any open-source development tools. The paper states however it is an extension of ST and a translator between poST and ST has been developed [23]. Since the extension defines new constructs, also existing language tools must be updated to support this kind of extensions. If there would have been open-sourced tooling support for poST, it could have been a great example for web-based ST tooling.

4.1 MatIEC

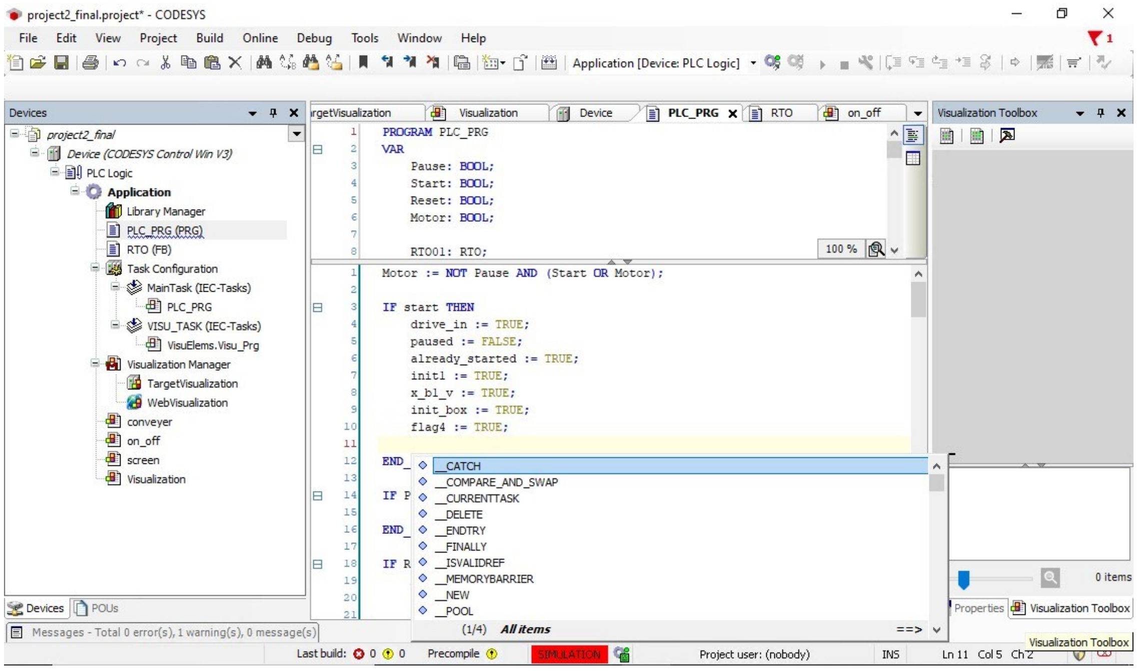
MatIEC is an open-source compiler for all the languages defined in IEC 61131-3 standard. The compiler compiles code written in standard's languages into C code. The Bitbucket repository has a good technical readme file about the compiler's functionality and high level internal architecture. [16] The C code can be further compiled into native binary with any C compiler. With MatIEC, PLC manufacturer can support all IEC 61131-3 languages with developing or using an existing C compiler for the target architecture.

MatIEC is very interesting software for this thesis as compilers generally must include lexical analysis, syntax analysis and some semantical checks such as type checking [3, Ch. 1.2], [11, p. vi]. These analysis and checks are needed for code editor's language tooling as well. Unfortunately, for Valmet's purposes the MatIEC's license is quite restrictive despite being open-source software and these analysis stages cannot be separated from the rest of the compiler without also distributing the modified source code.

MatIEC compiler has built in lexical and syntax parsers and semantical analysis as stated by MatIEC developer de Sousa [17]. According to de Sousa [17] MatIEC's semantic checker was implemented at later date compared to lexical and syntactical analysis stages. He also states MatIEC does not do complete semantical analysis, but the compiler will be extended in the future to include code style verifications as well. It is difficult to analyze how much of these checks have been implemented just by viewing source code when the subject of compilers is not familiar. MatIEC however seems to be in semi-active development in de Sousa's repository [16]. As of August 2022, latest feature seems to be CoDeSys syntax compatibility implemented in April 2021.

4.2 CoDeSys

CoDeSys is a closed source, free, manufacturer independent PLC programming IDE. Harwell's paper [13] summarizes the problem CoDeSys attempts to solve: PLC manufacturers often have differences in the implementations so that IEC 61131-3 programs developed in one manufacturer's software is not compatible with another manufacturer's devices. CoDeSys can support multiple manufacturer specific extensions to the standard and adapt code automatically from one target device to another. Thus, CoDeSys further helps the standardization of PLC programming from the practical standpoint. Harwall also mentions the difference in the look and feel of the different programming environments may make it difficult to change vendors, while CoDeSys offers universal tool for most common manufacturer's systems [13]. According to Hanssen [6, Ch. 14.1] there are over 250 hardware manufacturers using CoDeSys to program their PLC devices. CoDeSys also includes a runtime for testing the program without PLC hardware, which is helpful for the programmers. Figure 13 shows a screenshot of the CoDeSys' UI (User Interface)



```
PROGRAM PLC_PRG
VAR
    Pause: BOOL;
    Start: BOOL;
    Reset: BOOL;
    Motor: BOOL;

    RTO01: RTO;
```

```
Motor := NOT Pause AND (Start OR Motor);

IF start THEN
    drive_in := TRUE;
    paused := FALSE;
    already_started := TRUE;
    init1 := TRUE;
    x_bl_v := TRUE;
    init_box := TRUE;
    flag4 := TRUE;

END_

IF P

END_

IF R
```

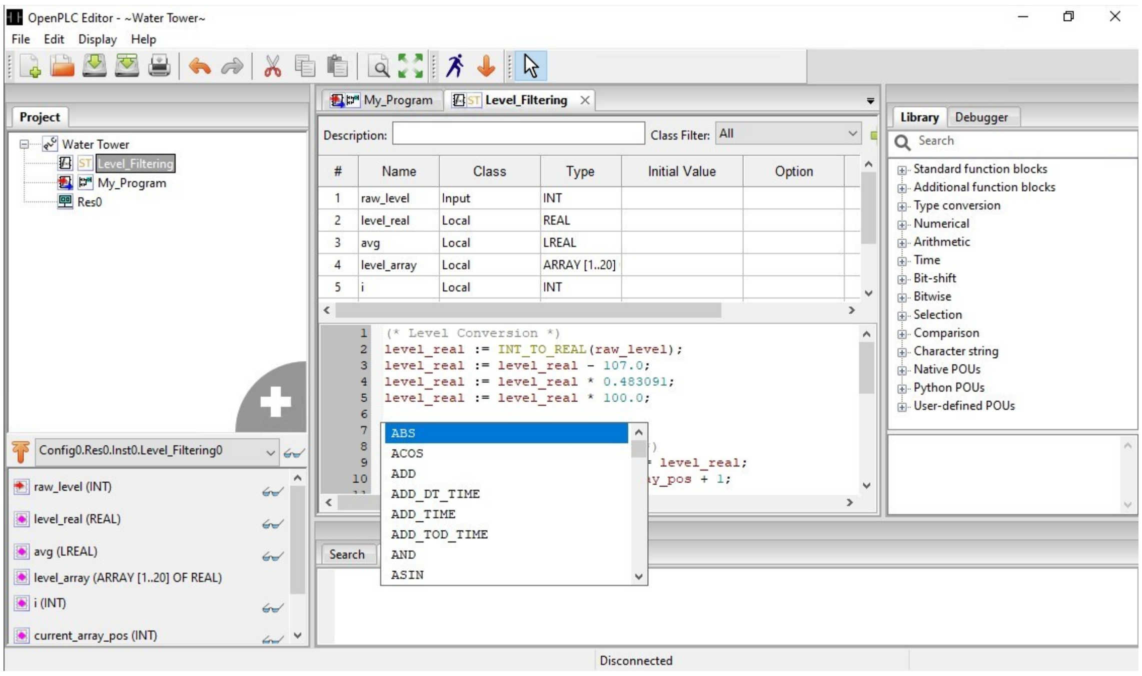
```
__CATCH
__COMPARE_AND_SWAP
__CURRENTTASK
__DELETE
__ENDTRY
__FINALLY
__ISVALIDREF
__MEMORYBARRIER
__NEW
__POOL
```

Figure 13. CoDeSys development system

While CoDeSys does what is expected from the ST tool, it is not feasible option for the Valmet's use case as CoDeSys offers much more features than required and the program is quite heavy and difficult to run in a web environment.

4.3 OpenPLC

OpenPLC is another manufacturer independent PLC programming IDE, but unlike CoDeSys, OpenPLC is entirely open source [39]. OpenPLC allows to run PLC programs on more diverse hardware via OpenPLC runtime. In the paper authored by Alves et al. [14] they demonstrate OpenPLC can be run on custom hardware, providing a low cost alternative to PLC vendors. The documentation [40] tells the OpenPLC runtime is written in C, so any target platform which has an existing C compiler can run OpenPLC to run programs written in IEC 61131-3 languages. Another research by Alves and Morris focused on the cyber security and validation of OpenPLC as an alternative for commercial PLC products. They mentioned OpenPLC performance was satisfactory when compared to 4 different commercial PLCs. [1] The OpenPLC project uses MatIEC as the compiler [1]. Figure 14 shows the user interface of the OpenPLC IDE.



#	Name	Class	Type	Initial Value	Option
1	raw_level	Input	INT		
2	level_real	Local	REAL		
3	avg	Local	LREAL		
4	level_array	Local	ARRAY [1..20]		
5	i	Local	INT		

```
(* Level Conversion *)
level_real := INT_TO_REAL(raw_level);
level_real := level_real - 107.0;
level_real := level_real * 0.483091;
level_real := level_real * 100.0;
```

Figure 14. *OpenPLC IDE*

4.4 Vendor specific tools

Most PLC manufacturers have their own proprietary programming tools, for example Siemens' Simatic STEP 7 [41] or Beckhoff's TwinCAT 3 [42]. This makes it more difficult to program variety of PLC devices from different manufacturers and encourages users to vendor lock their PLC systems to single vendor to ease development. PLC manufacturers who take this approach are mostly bigger companies with resources to implement superior interoperability, support, and user experience if all the devices are from them. This is of course desirable outcome for the PLC manufacturer, but reduces the effectiveness of the IEC 61131-3 standard in practice since the vendor often implements additional features not specified in the standard to gain customer's favor. Vendor locking is opposite goal compared to OpenPLC's and CoDeSys' approach.

4.5 Small open-source projects

Besides the MatIEC compiler, there is at least one open source program IEC-Checker [9] designed for static analysis of ST programs. IEC-Checker is said to have some checks based on PLCOpen organization's guidelines [43] for PLC development in addition to some other checks such as "Declaration analysis for derived types" [9]. This kind of check is interesting, as the MatIEC developer de Sousa has published paper pointing out the ambiguities related to derived variables datatypes [17]. In the paper, de Sousa points out the standard does not clearly state how complex derived data types' equality should be evaluated when deriving from already derived types.

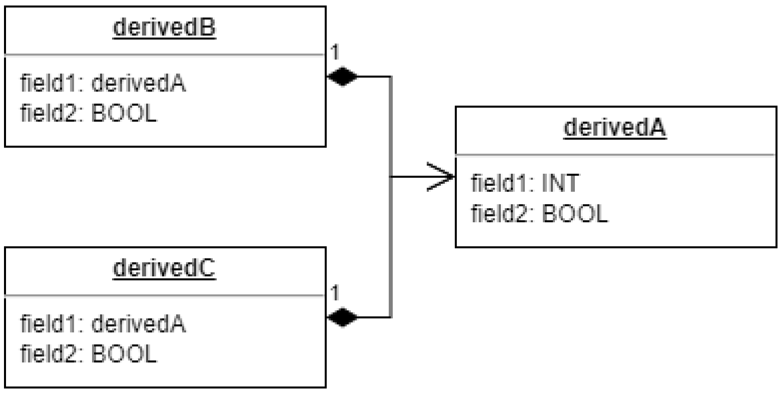


Figure 15. Example of ambiguous derived types

Figure 15 represents a simple example scenario where two types, derivedB and derivedC, are derived from type derivedA. This by itself is not an ambiguous situation, but what should result when checking equality between derivedB and derivedC? Both have identical fields, but they are declared as separate types.

This raises the question, how valid or thorough can the check be for data type declarations for derived types, if the standard itself is ambiguous?

There is also another open source IEC 61131-3 compiler named Echidna [2]. It is combined compiler and runtime in a similar although more tightly coupled way compared to OpenPLC. Echidna repository page states to support compiling and running code written in Instruction List language on any hardware with supported C compiler. As compilers generally must have syntactical analysis, Echidna should also contain parser for the IL language. However, this thesis focuses on the ST, and Echidna it is not usable for the ST tooling.

5. INTERVIEWS

Interviews are chosen as major research method to gather insights and opinions about the new tool for ST development. Results are analyzed further in chapters 6 and 8. In chapter 6, requirements for the tool are derived from the interview results and in chapter 8 the implementation is analyzed how well it fulfills these identified requirements. The analyzed findings from the interviews should answer the research question: “What features are required for efficient ST development tool?”. While originally interviews were planned to be conducted early in the project, they were postponed as latter part of the project. This choice was done to get more meaningful results as it is expected the ST development is rather unfamiliar to most of the interviewees and it is beneficial to have some baseline tool as an aid. Since the ST support has not had proper development tool previously in Valmet’s system, it is unlikely to find experts on the subject. It is easier to find issues and lack of features from existing tool rather than realize them purely by discussion. Therefore, the first iteration of the tool was developed first based on few person’s opinions about the matter.

5.1 Interview structure

The interviews were conducted as open-ended interviews to allow relatively free discussion about the experiences and thoughts about the ST editing and debugging. The goal of the interviews is to pick up the most important ideas and thoughts how the user would want to use the tool. Structured and semi-structured interviews could limit the field of discussion too much since the questions could easily steer the discussion to specific points of view. On the other hand, open-ended interviews can give too much freedom and the interview results become impossible to generalize due to questions being too different. This is a risk which was accepted and must be considered when analyzing the results.

Target is to interview 4 persons from different backgrounds. Some who have previous experience with writing ST, some have general programming experience but not with ST, some have no programming experience at all. Preferably there would be more interviews from each group, but it is especially difficult to find people who have prior experience with ST programming. The interviewees are all Valmet personnel. External interviewees are not conducted since it would be difficult to find suitable candidates and the tool itself is integrated into Valmet’s automation platform which is in internal development. There is an increased risk some classified details about the system would accidentally leak to external users if external users would be interviewed. The group of interviewees is very small, but it is expected for answers to be quite similar and hence the saturation point of new ideas is reached quickly even with a small target group. Additionally, the target of interview is rather small and well-confined feature. Because of this, the number of opinions is expected to be small and the opinions to correlate between each other strongly.

In research by Weller et al. [21] the probing technique is emphasized to greatly enhance the results even with small amount of interviewees. In probing technique more clarifications or follow-up questions are asked after initial response to a question. This is especially important in open-ended interview where it can be sometimes difficult to get solid answers immediately. The questions and especially follow-up questions are altered according to the interviewee's experience about the ST development. On persons who are more accustomed to ST development, answers and follow up questions can be more technical and focus more on the specifics of the language. More familiar interviewees may also have experience about other existing tools for ST development and may have some bias towards specific way of working.

Main goal of the interviews is to gather opinions and ideas about the state of the tool and future development ideas for ST development.

Interview questions:

1. How would you describe your experience regarding ST development?
2. What features and assist would you expect or hope code editor to offer?
3. What features would you expect or hope ST test mode/ debugger to offer?
4. What do you think are the user's needs and use-cases for the ST editor?

After these initial questions, allow interviewee to briefly test the implemented tool or showcase the current implementation to spark discussion.

5. What do you think about these editor features?
6. What do you think is missing? Now that you see the editor, do you have additional expectations for the editor?
7. What do you think about these debugging features?
8. Should the debugger offer something else? Is something missing?
9. Anything else to add?

5.2 Interview results

In the end, only three official interviews were conducted. Additionally, some unofficial conversations included the same topics which were discussed in interviews. The number of interviews is less than originally planned. This can cause ideas and opinions to look more popular than they are. Also, the amount of ideas may be a bit smaller and there is less overlapping between the ideas.

This chapter summarizes responses received from the interviews. The responses are anonymous and listed by the topic. The interviews were not recorded word by word. Instead, the responses were summarized after the interview. Asked questions differ slightly between the interviews and the exact questions asked were not recorded and follow-up questions were unfortunately not recorded either during the interviews. As all the interviewees are Valmet personnel with relations to the R&D (Research and Development) department, the answers may be a bit biased towards a specific way of working.

1. **Prior experience regarding ST programming?**

   Each interviewee had slightly different experience. One person responded to have once programmed ST code with the previous tool version but had not much experience outside of that. Another interviewee had no experience about ST or programming altogether. Third interviewee had also no experience with ST and very slightly about textual programming. He however had experience about logic programming.

2. **What features and assist would you expect or hope code editor to offer?**

   The first feature the interviewees usually came up with was the availability of the documentation or some other form of syntax help in the editor. Especially programmers who were inexperienced with the ST wished for the syntax documentation. One interviewee stated the user interface must be as clear as possible and understanding what features it supports and how to use them should be obvious to the user without ever seeing the user interface before. More specifically, the UI (user interface) should be familiar in a way that UI elements should be familiar and more importantly, if familiar UI elements are used, they should work like the element works in other tools.

3. **What features would you expect or hope ST test mode/ debugger to offer?**

   One idea or wish was to have offline debugging, or simulation. Essentially some sort of way to test the functionality of the program without the actual runtime environment.

4. **What do you think are the user's needs and use-cases for the ST editor?**

   Interviewees had different thoughts about the use cases. One use case was the copy and adaptation of existing logic written in ST to Valmet's automation platform. Other point of view was that ST is used for functionality which is not easily creatable with function block diagrams, such as for-loops. The answers are wildly different, some users immediately think of re-using entire control programs, other users consider using this only when necessary. Overall, the use case is not clear to anyone. Some interviewees even noted they don't know the use cases of the tool before even asking this question.

After these initial questions, the new editor and debugger was showcased or given to interviewee to test it out.

5. **What do you think about these editor features?**

   Interviewees mostly focused on the validation and syntax error highlighting, which are new and most visible features. Opinions were quite positive, although the initial assumption about syntax errors was that the editor would advise how to fix them. Outside official interviews, the ability to validate code in the editor was found out to be very useful while writing the code. Also, the validation feature received an improvement suggestion for the UI, which does a poor job of informing if the validation is running still or if it produced the same message than before.

6. **What do you think is missing? Now that you see the editor, do you have additional expectations for the editor?**

   Idea about a conversion function for example ST programs written originally in CoDeSys came up after explaining the idea to write the boilerplate code such as interface variable definitions. The syntax used by CoDeSys differs slightly from the syntax defined by the standard. For example, in CoDeSys the semicolons are not required after the END_IF but in the standard, they are required.

   Another kind of similar idea was to generate the interface code automatically, since with current implementation it must be defined twice.

   For users, it is annoying to write similar code repeatedly and instead it would be nice to have some selection of common snippets of code. Similar but still slightly different idea would be to have templates for slightly larger pieces of code, such as a POU skeleton.

7. **What do you think about these debugging features?**

   The debug prints are welcome addition as before the debugging was done mostly by trial and error by adding additional ST block interface variable and writing to it. The debugging features were considered great step into the correct direction. The debugger does what interviewees expect it to do.

8. **Should the debugger offer something else? Is something missing?**

   Generally, interviewees were positively surprised about the offered functionality. One further suggestion was to be able to insert debug print commands to the code while in debugging mode. Usually debugging requires the user to start following the execution from a point where the logic is surely still working correctly towards the point where the result is unexpected. As the debug prints are the best way to figure out how the program logic works and the point where the user is interested constantly changes, it would be beneficial to be able to use debug prints like breakpoints.

   When observing the use of the debugger by interviewee, the order of code execution was not immediately clear to the user and the debugger does not have any kind of indication what is executed when. With common programming language debuggers in IDEs, the ability to execute code step-by-step gives this information. The debugging becomes a complex task quickly as the ST code is divided to more and more POUs.

9. **Anything else to add?**

   The reusability and modularization of the code came up in discussions which is not directly coupled to editor, but editor features must support code which is split to different “files” or modules. In discussion there were mentions about pros and cons of both having everything in one file or splitting the code into reusable modules. The answer which is better is not clear as there is not a clear picture what is the role of ST programming in Valmet’s automation platform. On other hand, the modularization could be done entirely outside of the code editor by dividing the ST code into blocks and connecting the blocks together with “wires” which are used in function block diagrams. With this approach, the ST code editor would only ever be used to edit single block’s code and the organization of ST code would be left to the function block editor.

Outside of the interviews, the users were observed to be struggling with ST syntax, which further shows the importance of documentation help to be readily available while writing ST code.

6. REQUIREMENTS AND CONSTRAINTS

This chapter lists and justifies identified requirements. Some of the requirements originate from the interview results, while some have been defined directly by the project goal. Some implementation constraints, such as the requirement to make the tool run within the web-based Valmet DNA tools, create additional requirements and constaraints.

6.1 Identified requirements

The project goal creates some requirements by itself. The goal is to improve the user experience of ST programming within Valmet's automation platform's configuration tool and create some way for user to verify the ST program works as intended. Following requirements are set as the starting point for the project.

- Tools must comply with the IEC 61131-3 standard
- The editor must be compatible with the existing MatIEC compiler
- Editing must happen in Valmet's web-based configuration tool
- Editor should be syntax-aware and provide syntax highlighting
- Editor should provide syntax error diagnostics or in other words, meaningful error messages
- User should be able to verify the correctness of ST program to some extent

Rest of the requirements are defined via user interviews. Some requirements have come up in both initial project requirements and user interviews. Most of the ideas were considered as requirements. Following list of requirements was created by combining the ideas mentioned in interviews and removing those ideas that do not greatly improve the user experience.

- Language documentation should be easily available
- User should be able to modularize source code and easily reuse code that has been saved in a library.
- The interface between the function block diagram's IEC ST block and contained ST code should be automatically generated
- The user interface should be intuitive for the user

- Editor should suggest fixes for syntax errors
- Debugger should allow step-by-step debugging
- Inserting debug print commands in the middle of a debug session

Syntax documentation was clearly the most desired feature which was not implemented in the first iteration of the tool. This was a bit surprising initially as such documentation usually is not integrated into the editor when considering the programming languages used in IT, such as JavaScript or C++. However, in automation related programming tools, such syntax documentation is often available. This and the fact that the language is unknown to nearly all the interviewees, explains, why the feature is required.

The second most desired feature is the ability to re-use ST code or otherwise make it easier to write common code that is often required. The feature seems to be useful to the users, as it is generally recommended to not repeat code or reinvent the wheel so to speak. Instead, often used snippets could be saved to a shared library of code snippets and browsed within an editor. Without such feature, users likely will create their own libraries of ST code stored externally, for example in text files on local machine. While this kind of feature is not necessarily a requirement for efficient development tool, it certainly would help.

Previous two wishes were brought up in all interviews, next ideas were less common and all of them are not important enough to be classified as requirements. The requirements derived from the interviews are largely biased towards the features useful for beginners. There is a chance more experienced ST programmers would wish for different features. Most of the wished features were either focused on learning the ST programming or having the editor automatically do as much of the work as possible. Out of these the latter would most likely be appreciated also by more experienced programmers.

Having an intuitive user interface is very important and an obvious requirement for efficient ST development tool. Also, the ability for editor to report detailed syntax errors or even fixes is seen as an obvious feature of the tool. Both are requirements for the tool.

Related to same reasons the re-use of code is desired, the automatic generation of external interfaces is wished. The interface must be defined both on the function block level and in the ST code. The wish is to define it only in one location and the other would be automatically updated. This would make the editing experience more efficient and thus is deemed as a requirement.

One wish, which did not qualify as a requirement was to have conversion from alternative ST syntax. While it could be useful in some situations, it would require some work to make a reliable converter and the benefits are estimated to be too small for the work. In this conversion case, interviewee thought about copying code over from some other ST development tool which uses slightly differing syntax. Another unlisted wish was to have offline debugging without controller. While this would be beneficial in some scenarios as well, the technical and architectural choices make this difficult to implement and the use case has a workaround by creating a temporary virtual controller for debugging. Thus, it is deemed to not be important enough to qualify as a requirement.

6.2 Constraints

The biggest identified constraint is that the ST editor must be integrated into Valmet's automation platform, Valmet DNA (Dynamic Network of Applications), in some way. DNA uses primarily function block diagrams as a language to program automation applications. In the new web-based Valmet DNA Configuration Environment, the ST code is embedded within a Function Block Diagram as a special function block. Essentially the function block accepts ST code as configuration and has typical function block input and output interfaces which can be referenced within ST code. The function block diagrams are executed in cycles, and as the FBDs can be connected to each other, pausing the execution is technically difficult. This creates constraints to ST debugging as the ST code is executed as a part of an FBD. As the ST code editing starts when the user navigates to the specific function block within a diagram, it is desirable for ST code editing to happen in the web-based function block editor for smoother, more uniform user experience.

As the tool is web-based, there are additional constraints related to web and browser context. The editor should not heavily rely on client machine's software and instead run entirely within browser. Also, some client machine's resources such as the local filesystem are much more restricted in browser environment. Some cache files or local storage for the website could be utilized but the majority of filesystem is off-limits. Therefore, the system should rely mostly on RAM (Random Access Memory).

The requirement to be compatible with the existing MatIEC compiler creates a constraint for the IEC 61131-3 standard's version. As MatIEC is based on the standard's 2nd edition, the tooling must match with the 2nd edition's syntax and features.

In automation the product lifecycle is often long, even longer than 10 years. This discourages the heavy usage of libraries and npm (node package manager) packages. The less there are dependencies, the less there are components to monitor and maintain. After 10 years there is a big possibility most packages active today are not maintained anymore. Unmaintained packages possibly contain security threats, and either no one is fixing the found vulnerabilities or the maintenance of these packages falls to the user of the packages. It is not forbidden to use libraries or npm packages, but the use case should first be considered if it is easy to implement by oneself.

7. IMPLEMENTATION

This chapter consists of technical implementation details and planning and the first iteration of the implemented system. Due to the size of the scope for the tooling system and limited time for the project, only a subset of planned features or simplified versions of the features were implemented.

7.1 Selected technologies, libraries, and existing components

As the tooling system was chosen to be integrated into Valmet's Automation platform's configuration environment, it affected technology selection and feasible ways to engineer a system for intelligent editor assistance. As the environment is web-based and the end-product is used via web interface, some programming languages were more suited to the task than others. JavaScript and Typescript, which is a superset of JavaScript and often referred as "JavaScript with types"[44], are both common in web applications and optimal programming language for this thesis. However, this slightly reduces the possible external libraries and applications for code editing to be used in the system, especially as most editor software are designed as standalone desktop software. Typescript was however chosen as the primary programming language since the platform's configuration environment that wraps the ST tooling is browser based and mostly written in Typescript thus making integration much easier.

The platform's configuration environment has previously implemented code editor for ST language using Monaco Editor as base, but the previous implementation does not have any assist features enabled and the editing experience is similar to writing plain text to an input box. This implementation could however provide a great base for implementing the assisting features.

The tooling system was chosen to be implemented as language server which is run in a Web Worker. Web Workers run in the background in the same browser instance, each worker as a dedicated thread. Web Workers are a way to run JavaScript in browser without interfering with the user interface [45].

The ST editor creates and owns the Web Worker instance which runs the Language Server. With Language Server running in dedicated Web Worker, the Monaco Editor and the Language Server exchange messages using the Web Workers API of the browser.

The Language Server and Monaco Editor communicate via the Language Server Protocol. The Monaco Editor was previously chosen to be used in other tools within the platform and there was no interest in changing the editor unless necessary. Therefore, primarily approaches which include the Monaco Editor were considered. Monaco Editor is used in popular multi-language code editor Visual Studio Code and includes many useful features for code editing.

For the ST parser, parser generator Lezer was chosen as it is JavaScript based and thus fits well into the web environment. Lezer has been developed for use in another code editor, CodeMirror [46]. Being developed for code editor, Lezer has crucial features for a parser used in editor environment, such as error recovery or incremental parsing. Unfortunately, generated parsers nearly always have poor error messages, especially when combined with error recovery and Lezer is no exception. Some parser generators or libraries allow the customization of error messages, but Lezer does not have such feature.

The user interface wrapping the editor is created using React as the main framework. Main reason for using React is the existing web-based configuration environment which uses React. Using a different framework for a small integrated feature does not make sense if the same framework can be used for both.

7.2 Implementation plan

Tooling system implementation plan includes the development of language server for ST language, improved user interface to accommodate the new features, and new debugging related functionality to the web-based code editor integrated into the Valmet’s automation platform. The plan initially was not extremely detailed and focused more on the larger picture rather than specifics. The initial plan specified that the ST editing experience should be improved from the user’s standpoint. The platform had existing barebones editor for ST without assist features, such as syntax highlighting or autocomplete. Additionally, the platform had existing compiler and a separate runtime for executing the ST code. The primary plan was to extend the functionality of these existing components instead of replacing the editor with third party tool.

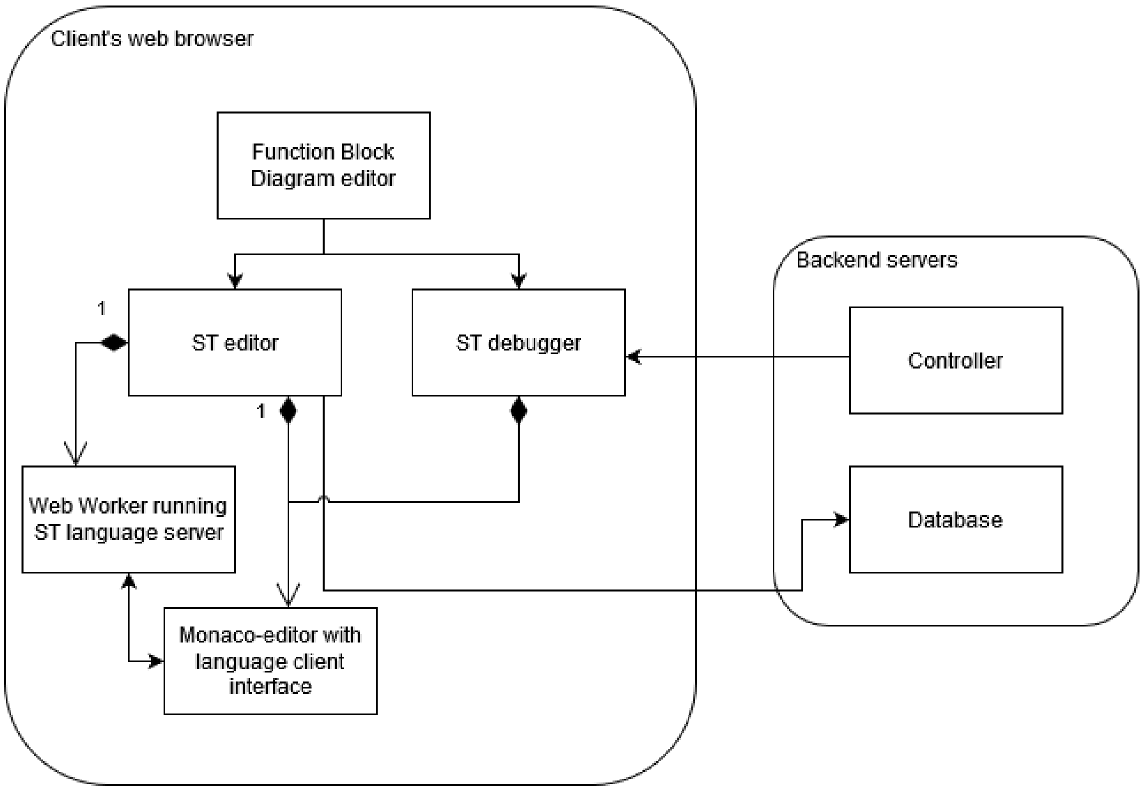

Client's web browser
Function Block Diagram editor
ST editor
ST debugger
1
1
Web Worker running ST language server
Monaco-editor with language client interface
Backend servers
Controller
Database

Figure 16. Simplified overview of the system

The ST editor and debugger are planned to be web components to be integrated into existing web-based function block diagram editor. Figure 16 shows the basic architecture of the tooling system. The ST editor is planned to consist of two main components, the code editor Monaco which can be extended to be Language Server Protocol compatible and the Language Server. The Language Server is explained in more detail in the next chapter 7.2.1. The editor saves the ST code into the backend's database for persistent storage. The backend is also responsible for the compilation of the code. The debugger component fetches the runtime data from the controller, which is executing the compiled ST code. In this first iteration, the debugger component does not utilize the Language Server. In the future, some more advanced "go to" features could be implemented in the debugger component by utilizing the Language Server.

The entire system is web-based, and the actual editor runs entirely on the client machine's web browser. First the web server serves the webpage with the function block diagram editor, including the code for the ST editor. The ST editor starts the ST language server in dedicated Web Worker and initializes the Monaco editor as child component of ST editor. The Monaco editor and the Language Server communicate directly with each other, while the ST editor communicates with the backend.

7.2.1 Language Server

Language Server is an actual implementation of the features specified by Language Server Protocol. Language Server does not have to support all features of the protocol as the interface gives the possibility to specify which of the protocol's features are supported. These capabilities can be easily expanded if needs arise. Client and server share their list of supported features at the start of session as part of the handshake. Only the features supported by both are used. New ST language server's first features are autocompletion and syntax error highlighting.

Initial plan for Language Server is to support two types of autocompletion, static and dynamic with basic context awareness. Static autocompletion would not strictly require language server as it is just a list of reserved keywords in ST language. Dynamic autocomplete is used to identify and list user defined variables and functions. For dynamic autocomplete language server with a parser for the ST language is required. Language Server has access to the information where user's document cursor is and can deduce the context and filter suggestions based on the information. Language Server Protocol supports wide variety of requests and features and not all of them are applicable to the Valmet's environment. Protocol supports for example "go to definition" requests, where the language server is given a small subrange of the document which contains the symbol in question, for example a function call. Language server then finds the file and position where given function is defined and responds with location of the requested definition in the source code.

7.2.2 Debug functionality

For debugging, true step-by-step execution control is difficult to implement to the automation platform where real-time operation is critical to the system and the idea was postponed until deemed necessary. Instead of real execution control, the plan is to implement simplified variable reader functionality. This simple "debugger" reads program variables after each cycle ST program has executed and visualizes variables state at the end of execution. The actual runtime execution is not controlled by debugger, thus mitigating the issue of debugging affecting time-critical processes running on the same controller. In this approach, user has no control over execution and can only observe the status of the variables. In addition to reading variables, user can use the debug print function to print values to the log during execution. The debug prints are written to separate log buffer during the code execution and visualized to the user after the execution cycle has finished. This could be used for example to log the value of an index variable in a for loop for each iteration during the execution.

One idea for further development is the history functionality where the application states are recorded and can be visualized so that it resembles debugging in popular IDE's such as Visual Studio. Included features should include at the very least "go to next step" and the list of variables at that state. With larger or heavily looping/recursive programs, this would likely result in heavy memory consumption. Another possibility is to execute the code in a dedicated simulated environment instead of using the actual runtime with other applications running. With simulated environment it would in theory be feasible to control the execution step by step.

7.3 First iteration

The implementation started with the creation of new language server. To aid development of new language server, a standalone Monaco Editor was set up as prototype environment. Monaco Editor is also used in the target system, and it was assumed the stand-alone editor would behave like the target system in many ways. Monaco Editor was setup by using browser-esm-webpack-typescript-react sample package from Microsoft's monaco-editor repository's samples. Both language server and language client must implement the protocol and unfortunately Monaco does not support language servers out of the box. Hence, Monaco must be extended with a language client implementation. There is an existing npm package called monaco-languageclient which implements language client interface for Monaco Editor. In addition, transport layer connection between server and client must be created and provided to the LSP connection. LSP does not take a stance what kind of transport method should be used as long as it can transport JSON-RPC (JavaScript Object Notation Remote Procedure Call) messages. Couple of transport methods were experimented with, WebSocket and Web Worker. Web Worker seemed to be quite uncommon in language servers but is very suitable for code editor running entirely in web browser. To run the Language Server, it is likely Web Workers would have been used anyways, so it makes sense to use the built-in messaging in Web Workers. Vscode-languageserver npm package has helper classes for connecting language servers via Web Worker. With connection mostly built by combining existing packages, next step is to implement actual language server features. First iteration of the server is quite limited in functionality as most of the development time is used to develop a Structured Text parser.

For the parser, multiple parser generators and libraries were searched and evaluated. Perfect parser generator was not found for the purposes of this thesis but compared to writing a custom parser completely from scratch, parser generator still seemed favorable to get something working in a reasonable timeframe. Parser generators generate the parser from a grammar which specifies the keywords and structures of the language. For example, if-statement could be one language structure. The chosen parser generator, Lezer, has its own notation for grammar but it closely resembles extended Backus-Naur form (EBNF). The grammar for ST parser is created by converting the grammar definitions from the standard [47] to Lezer's notation. While this requires quite a lot of handwork, it is still relatively simple as the formats are very close to each other in the first place. Additionally, assuming the standard's grammar specifications are correct and comprehensive, the resulting grammar file is very close to complete and less prone to errors compared to writing it from the scratch.

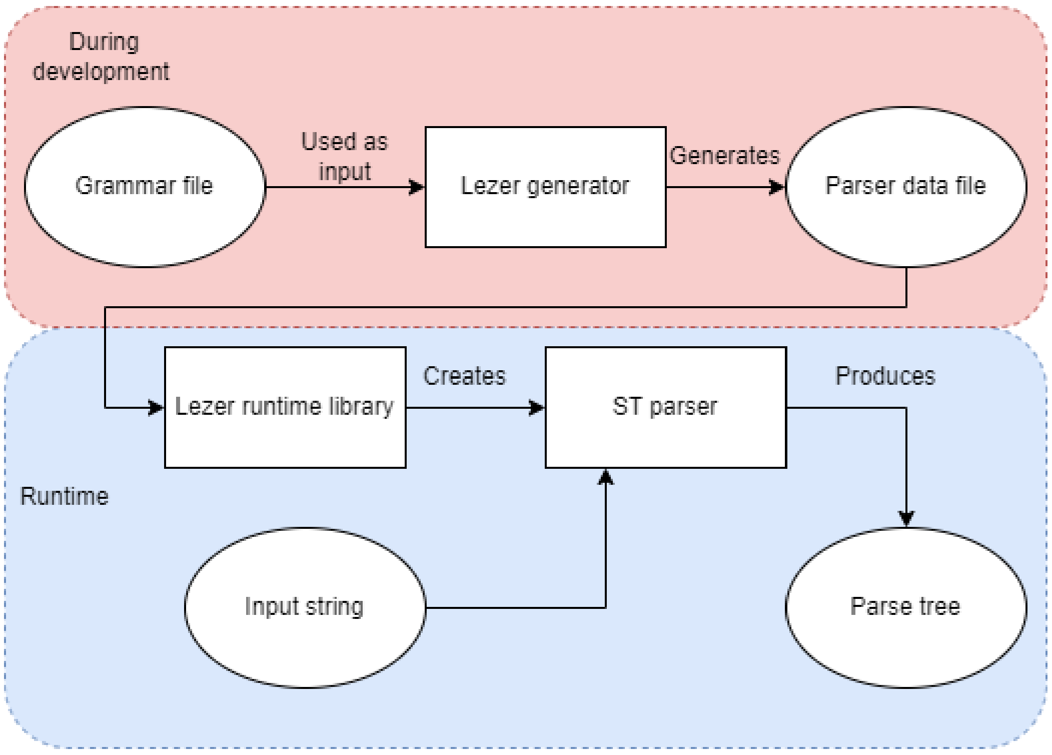
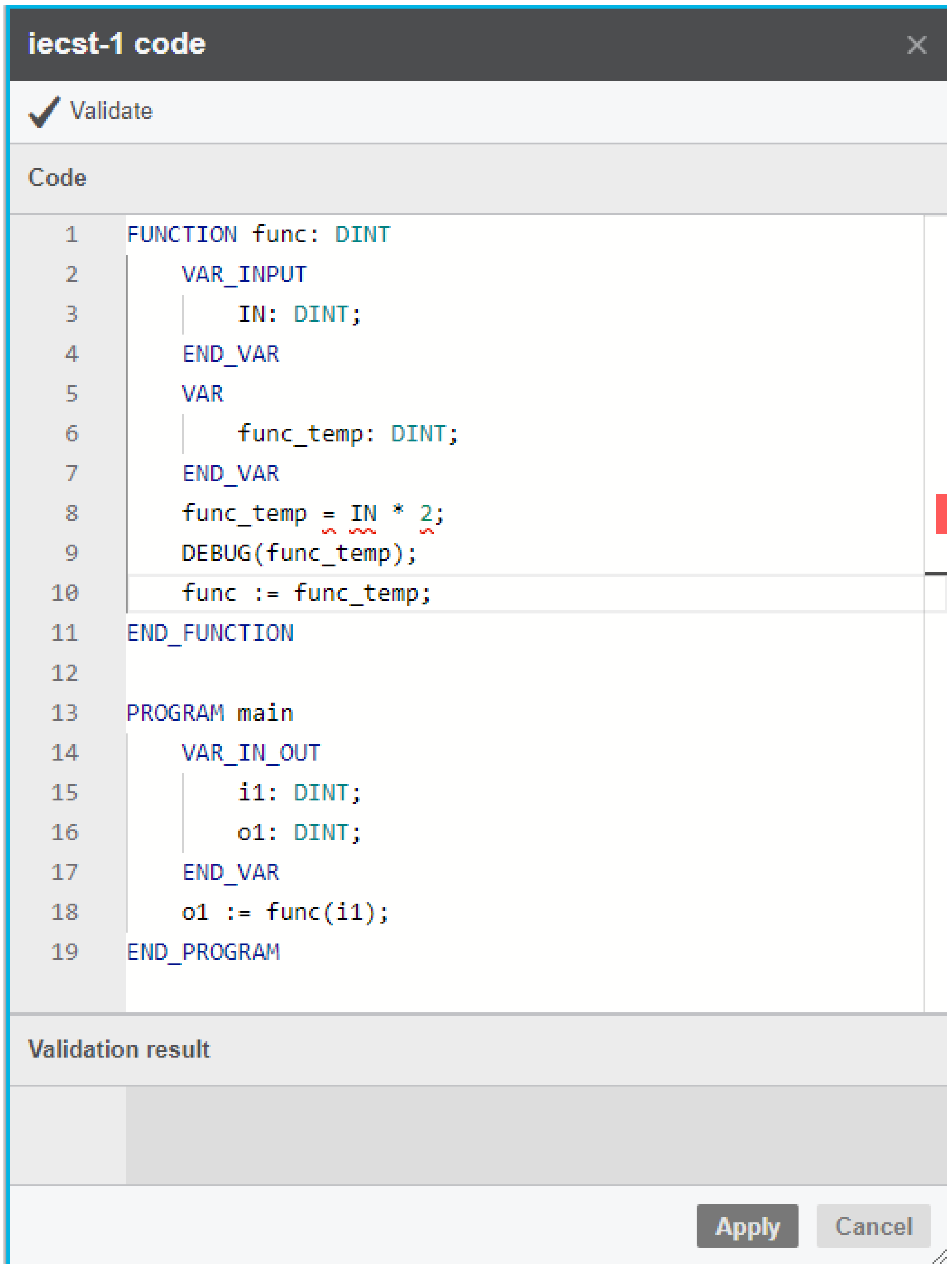


Figure 17. Lezer workflow

Figure 17 depicts the flow from grammar file to fully working parser using Lezer. Lezer generates the parser from grammar as input data for the Lezer parser runtime. In application code, which in this case is the Language Server, the Lezer runtime is merged with the parser data file. As a result, the library returns a fully working parser as a JavaScript object. The input to be parsed is then passed to this object. The parser also has various helper functions and methods to set settings and fetch or iterate parse result. The parse result is a tree structure of nodes containing some data about the language structure the node represents. The essential data of the node is the type of the node or in other words, which language structure it represents. A node could for example be a block of code like function definition or a keyword such as RETURN. Nodes often have several child nodes in several depth layers forming the parse tree. At its core, parse tree is a representation of the entire input string as language structure nodes. Another essential piece of data in nodes is the position of the node in the original input. For example, node has from- and to-properties which are 0-based position indices of original input string, thus giving the exact location of the substring that was identified to be of the node's type. With these information, most modern code editor features can be implemented. Parse tree can be seen as a basic level of computer understanding source code structures.

With parser developed, next step is to take the parse tree into use and provide language features. One easy feature is to report syntax errors. Parser adds special error-node to the parse tree which signifies syntax error. As nodes have the position information, this can be used to map syntax errors to source code producing the commonly seen red squiggly underline in the editor view. The accuracy and specificity of the errors produced are heavily dependent on the quality and robustness of the parser. While Lezer is very robust with syntax errors and has built-in error recovery, its capabilities to report meaningful error messages are minimal. Due to this, the first implementation of language server is unable to tell what kind of error is encountered and can only show the location of the erroneous node. Figure 18 shows how the editor highlights syntax errors found by the parser. In the example code in Figure 18, there are reported syntax errors on line 8. There the assignment operator is missing colon. Parser however marks entire line after the operator as invalid. This example shows how the error location is not always clear, but the error usually is located within one symbol from the first syntax error in the sequence of errors. In this case, the first syntax error points to operator, which is the actual location of the error. In the case of a function missing return type, all variable declarations are shown erroneous. In such case the actual error location is missing syntax just before the variable declarations, thus the parser marks the variable declarations invalid as they are unexpected before return type. Unfortunately, the parser does not tell any information about what kind of error it has encountered, and the user must guess what is wrong.

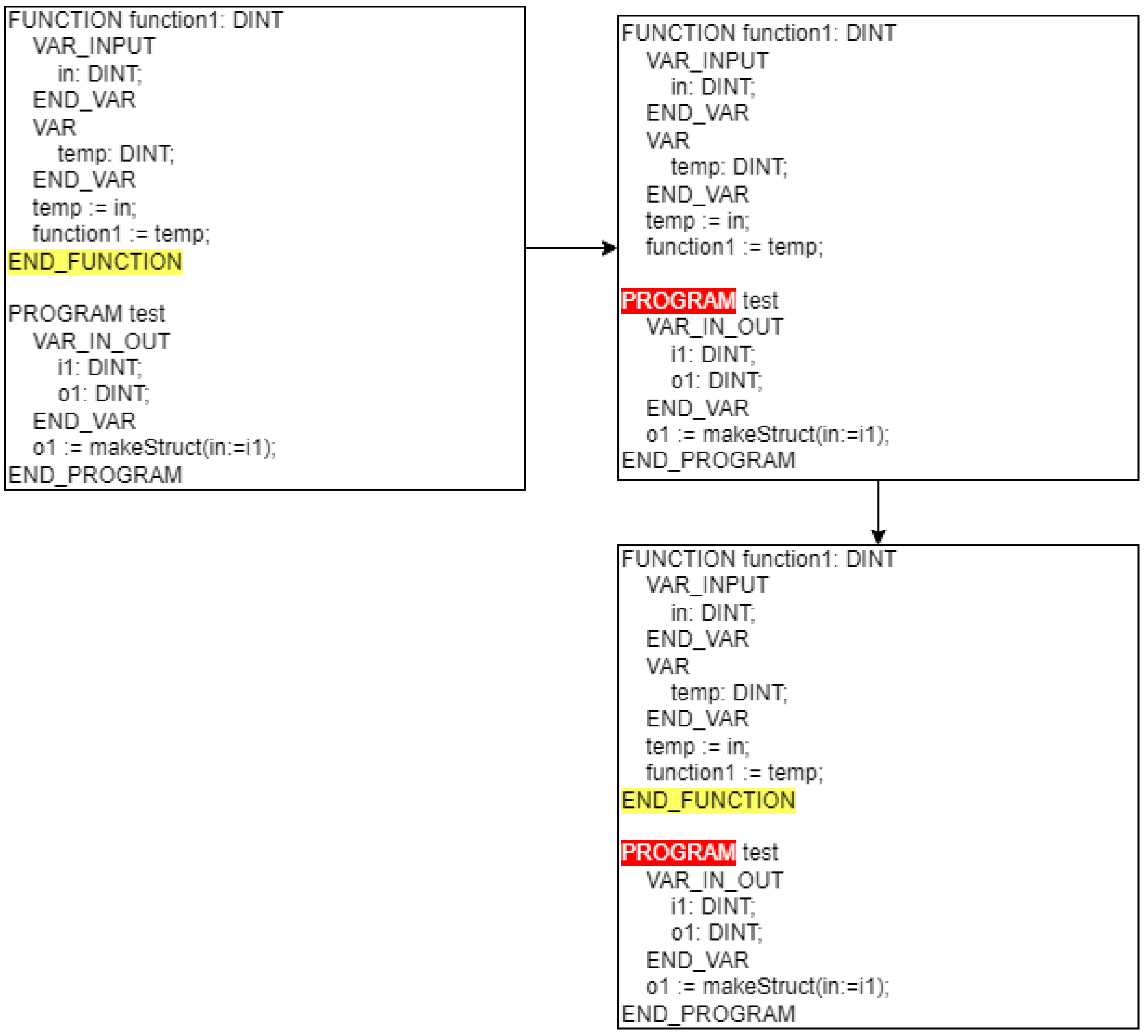


```
FUNCTION func: DINT
    VAR_INPUT
        IN: DINT;
    END_VAR
    VAR
        func_temp: DINT;
    END_VAR
    func_temp = IN * 2;
    DEBUG(func_temp);
    func := func_temp;
END_FUNCTION

PROGRAM main
    VAR_IN_OUT
        i1: DINT;
        o1: DINT;
    END_VAR
    o1 := func(i1);
END_PROGRAM
```

Figure 18. ST editor syntax error highlighting

Despite this downside, parser generator was used to create a proof of concept. To improve the error messages, custom handmade parser would be beneficial.

One benefit of using Lezer is its built-in support for incremental parsing which reduces the amount of work required to parse the document after changes significantly. The incremental parsing however required quite a lot of extra logic to extend support for incremental parsing for the analysis of declared variables and functions. Essentially, the analysis is required only for the part of code which can be affected by the change. Due to the structure of IEC 61131-3 languages, a single change can only affect single POU at least in the context of analyzing which variables or functions have been declared. Hence a scope tracking system was implemented to split the source code into sections of POUs and running analysis only for the section which contains the change. There is however one more thing to consider: change of code may change also break the POU in a way parser does not recognize it anymore which may lead to the corruption of scope tracking occurring as duplicate or overlapping POUs. As a solution, the analysis range is widened to include also the next POU if situation where POU ceases to exist is detected.



```
FUNCTION function1: DINT
  VAR_INPUT
    in: DINT;
  END_VAR
  VAR
    temp: DINT;
  END_VAR
  temp := in;
  function1 := temp;
END_FUNCTION

PROGRAM test
  VAR_IN_OUT
    i1: DINT;
    o1: DINT;
  END_VAR
  o1 := makeStruct(in:=i1);
END_PROGRAM
```

```
FUNCTION function1: DINT
  VAR_INPUT
    in: DINT;
  END_VAR
  VAR
    temp: DINT;
  END_VAR
  temp := in;
  function1 := temp;

PROGRAM test
  VAR_IN_OUT
    i1: DINT;
    o1: DINT;
  END_VAR
  o1 := makeStruct(in:=i1);
END_PROGRAM
```

```
FUNCTION function1: DINT
  VAR_INPUT
    in: DINT;
  END_VAR
  VAR
    temp: DINT;
  END_VAR
  temp := in;
  function1 := temp;
END_FUNCTION

PROGRAM test
  VAR_IN_OUT
    i1: DINT;
    o1: DINT;
  END_VAR
  o1 := makeStruct(in:=i1);
END_PROGRAM
```

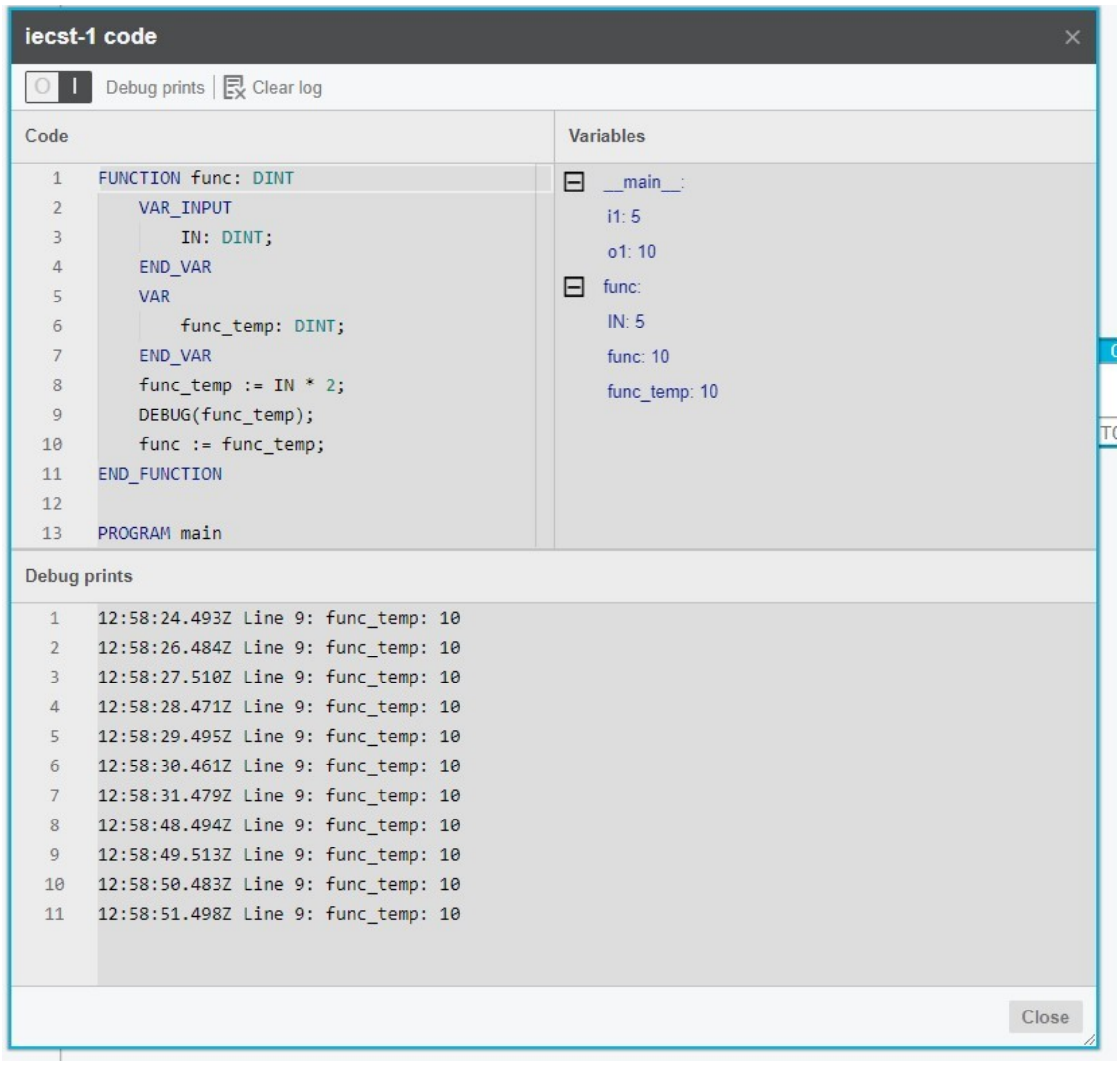
Figure 19. Example of situation which requires wider analysis

In Figure 19, first the POU function1’s closing statement is removed, this correctly leads to error being detected by the parser, which is highlighted as red in the Figure. The error is in the next POUs opening as it is unexpected token. Now because of the incremental parsing is based on changed POU scopes, re-adding END_FUNCTION keyword does not remove the error, since it is in the next POU outside of the change range. *POU function1* gets re-parsed and analyzed in all the steps of Figure 5, while *PROGRAM test* is not re-analyzed at all. This is one example of situations where this rule is extended to re-run the analysis for also the following POU.

7.3.1 Debugging and validation of ST code

In addition to language server implementation, some basic debug and code validation features are implemented. For debugging, most of the work was done by a colleague working with the ST runtime. With his additions, data about the most recent variables values and debug print logs are available. ST editor can fetch this data periodically via a subscribe mechanism. In addition to fetching the data, it must be visualized to the user. The new user interface for visualizing this debug information is showcased in Figure 20. The variables are visualized in a resizable panel located right of the source code. The list of variables shows only the variables that have been used during the latest execution cycle. For example, if there is a function which is not always executed, the variables from it are not shown when the function hasn't been executed. Function called only when a condition is true could be one such case when not all functions are executed.

The debug prints work slightly differently compared to the debug variables. Whereas debug variables show the most recent execution cycle's values for the variables after the execution, debug prints can be used to save variables into log in the middle of the execution. Additionally, debug print log keeps the history of last 1000 prints so that the prints can be read and analyzed. Debug print log is fetched similarly from the runtime as the debug data for the variables. The log history is only kept locally in the browser's memory and is cleared when the debugging session ends. User can also toggle the debug print log writes off and/or clear the log manually if necessary. The log window has an auto scroll function which automatically scrolls the log to the bottom, where the newest log messages appear. The auto scroll feature is disabled when the user manually scrolls upwards and is re-enabled when the user scrolls manually to the bottom. The auto scroll feature was inspired by the need to read the logs for a while but on the other hand when looking for the newest data, the constant manual scrolling is annoying.



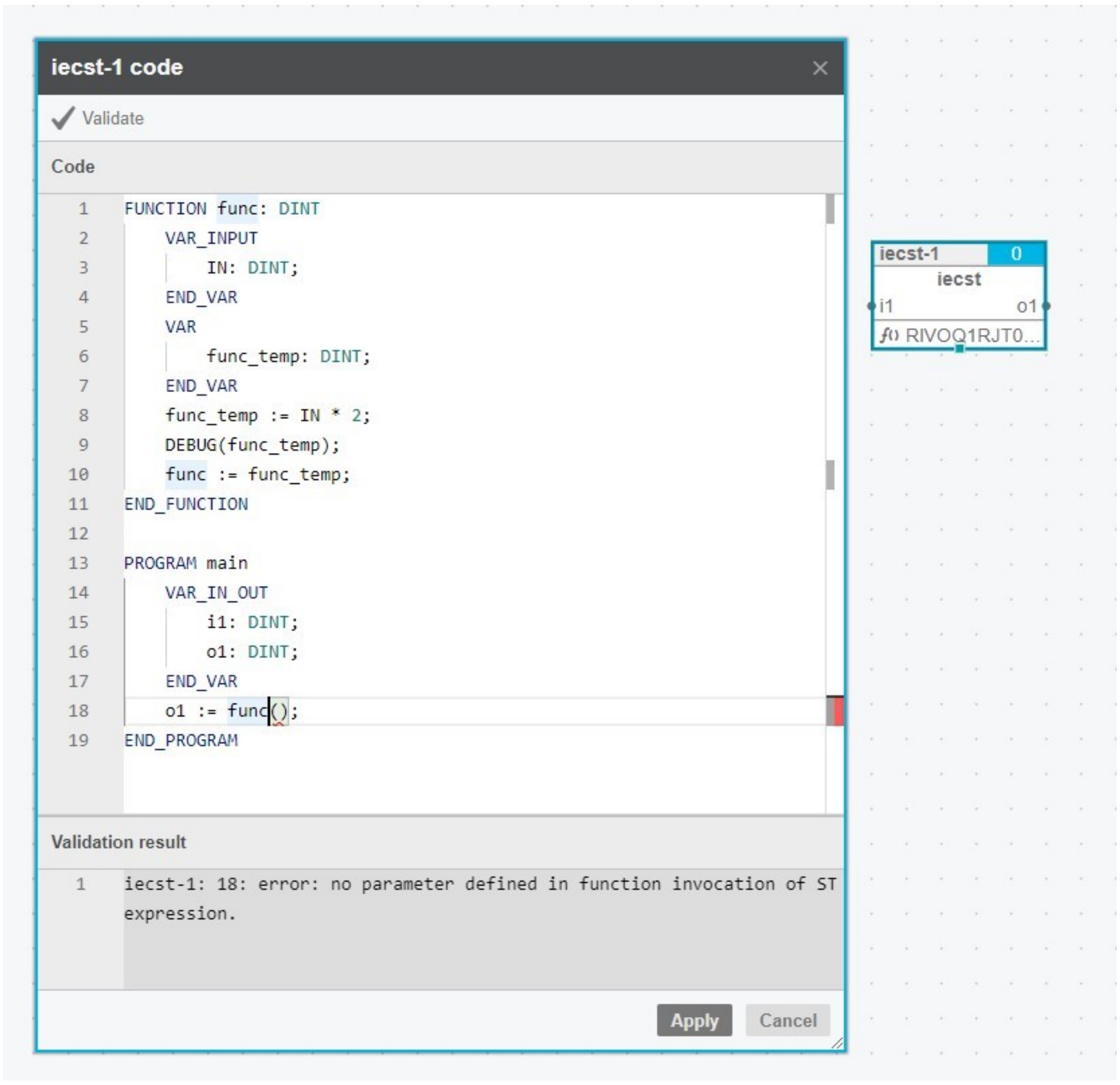
```
FUNCTION func: DINT
    VAR_INPUT
        IN: DINT;
    END_VAR
    VAR
        func_temp: DINT;
    END_VAR
    func_temp := IN * 2;
    DEBUG(func_temp);
    func := func_temp;
END_FUNCTION

PROGRAM main
```

```
__main__:
  i1: 5
  o1: 10
func:
  IN: 5
  func: 10
  func_temp: 10
```

```
12:58:24.493Z Line 9: func_temp: 10
12:58:26.484Z Line 9: func_temp: 10
12:58:27.510Z Line 9: func_temp: 10
12:58:28.471Z Line 9: func_temp: 10
12:58:29.495Z Line 9: func_temp: 10
12:58:30.461Z Line 9: func_temp: 10
12:58:31.479Z Line 9: func_temp: 10
12:58:48.494Z Line 9: func_temp: 10
12:58:49.513Z Line 9: func_temp: 10
12:58:50.483Z Line 9: func_temp: 10
12:58:51.498Z Line 9: func_temp: 10
```

Figure 20. Test mode/debugger view

The code validation was previously run as part of the validation of the entire function block diagram with potentially a lot of unrelated content. This validation method is not changed but an additional smaller scope validation is added to the ST code editor which validates only the ST code without the other contents of the function block diagram. Being able to validate the ST code on its own makes checking the correctness of the code easier and faster as the user does not need to leave the editor or validate the entire function block diagram. The new edit view with validation support is shown in Figure 21.



```
FUNCTION func: DINT
    VAR_INPUT
        IN: DINT;
    END_VAR
    VAR
        func_temp: DINT;
    END_VAR
    func_temp := IN * 2;
    DEBUG(func_temp);
    func := func_temp;
END_FUNCTION

PROGRAM main
    VAR_IN_OUT
        i1: DINT;
        o1: DINT;
    END_VAR
    o1 := func();
END_PROGRAM
```

Validation result

```
1   iecst-1: 18: error: no parameter defined in function invocation of ST
    expression.
```

Figure 21. Edit view

The errors shown by the validation result panel are the errors reported by the compiler MatIEC. While the messages can sometimes be a bit cryptic, they are however the most accurate errors available for this iteration of the ST editor tool. The Language Server, as stated before in chapter 7.3, can only indicate the starting point of error but not the nature of the error. The MatIEC's checks are more sophisticated and can provide better location and explanation of the error. The downside to MatIEC's error reporting in an editor use is that it is a compiler and compiling is notoriously heavy operation. Running MatIEC repeatedly on code changes would waste processing power and with the compiler residing in the server-side, also create large number of unnecessary requests and load to the server. Therefore, the validation is implemented as a manually invoked operation.

8. EVALUATION

This chapter consists of analysis and evaluation of how well the identified requirements support the goal of improving user experience of ST programming and how well implementation fulfills the requirements. The interviewees experience and the effect of that experience or lack of it is taken into account while analyzing the identified requirements.

8.1 Identified requirements and constraints

Requirements identified from project goals were quite general, which can be more beneficial than having precise and specific requirements. Based on the discussions, the nature of the tool and the actual use cases for the tool are not completely clear. Mainly it is unclear if users will use the ST to program relatively small programs performing some function that would be difficult to do with function blocks or if the users use ST to import control logic from external sources. These use cases affect the prioritization of features of the tool. With large programs, modularization of ST source code and the performance of the analysis stages of language server are emphasized. With small user written functions, the prioritization shifts to the language server features such as syntax suggestions, autocomplete and function parameter signatures. Having specific requirements with unclear use cases would be detrimental to the quality of the tool as the specified features could easily be unnecessary.

Interviews were planned to identify some of missing features. The lack of experience about ST development makes the ideas and requirements derived from the interviews biased towards features useful for the beginners. Interviews also provide additional or alternative points of view how the tool could work. Despite low number of interviews, they produced few requirements which are estimated to greatly improve the user experience. Especially for users who are not familiar with ST language. The bias towards help for beginners is shown by the popularity of features such as syntax documentation and ability to use premade code snippets or templates. The interview results raise the question, should the tool be targeted for beginners or more experienced developers, or can the tool target both audiences effectively? These questions remain unanswered for now, as the needs for experienced ST developers are still somewhat unknown.

The requirements are missing some commonly used features in IDEs such as symbol refactoring and go-to shortcuts. These features are estimated to greatly improve the user experience. It is interesting they did not come up in the interviews. Exact reason, why these common IDE functionalities are not mentioned in interviews is difficult to guess. This may be because they were taken as granted or then the inexperience of ST programming affected the results here. On other hand the interview questions may have guided interviewees thoughts in a way where traditional IDE did not come to mind.

Overall, there are not many constraints identified, which raises the question are there some important constraints not identified. Most constraints are derived from the choice of developing an integrated web-based tool and integrating the tool into automation platform with a long lifecycle. Missing requirements and constraints make estimating the success of implementation more difficult.

Table 2. below lists the requirements, and implementation status alongside estimated importance for the goal of improving user experience. Out of the initially identified requirements, the implementation fulfills all but one initial requirement. As the interviews were conducted late in the project, most of the requirements identified via interviews were not implemented.

Table 2. *Requirements and implementation status*

Requirement	**Details**	**Source**	**Implemented**	**Importance**
Tools must comply with the IEC 61131-3 standard		Initial requirement	Yes	Necessary
The editor must be compatible with the MatIEC compiler	The compiler uses 2nd edition of the standard	Initial requirement	Yes	Necessary
Editing must happen within the Valmet's web-based configuration tool	Due to architecture constraints, it is recommended for the editor to be completely web-based.	Initial requirement	Yes	Necessary
Editor should provide syntax highlighting		Initial requirement	Yes	High importance

Editor should provide syntax error reporting	Editor should mark the syntax errors in the editor and provide diagnostics what the error is.	Initial requirement	Partial, the editor marks syntax errors but does not provide diagnostics	High importance
User should be able to verify the correctness of ST program to an extent	User can run the compilation and receive the possible errors. Additionally, user can debug the code during execution and observe variables	Initial requirement	Yes	Medium importance
Language documentation should be easily available	A link to documentation within the editor	Interviews	Yes	High importance
User should be able to re-use code	Some sort of library with templates. Editor could also suggest snippets	Interviews	No	Medium importance
The external interface for the ST block should be automatically generated	Quality of life feature, reduces manual work.	Interviews	No	Medium importance
Editor could automatically convert common ST code into compatible form	Refers to situation where user would copy and paste code from external sources.	Interviews	No	Low importance

Debugging offline without runtime controller		Interviews	No, workaround exists	Low importance
The user interface should be intuitive to use		Interviews	Yes	High importance
Editor suggests fixes for syntax errors	While it would be nice, having descriptive error messages should be enough	Interviews	No	Low importance
Step-by-step debugging or ability to insert debug print commands to code while debugging	Useful but technically difficult feature to implement	Interviews	No	Medium importance

8.2 Implementation

The implementation prototype, which was created in few weeks, was reviewed, and discussed with a small group of software engineers and deemed feasible. Afterwards the implementation for the first iteration of new ST tooling system was developed and integrated to the configuration environment. In hindsight, the development could have benefitted from user feedback before the first working version is done. On the other hand, since there are very few users who have any experience with ST programming, the feedback could have been very shallow and not provide much value outside of developer's ideas. That was the main reason feedback interviews were pushed to later stage after the ST editor is usable.

One of the most important improvements to the ST editing was the detection and visualization of syntax error. Unfortunately, the diagnostics for syntax errors are poor due to the minimal error reporting capabilities of the used parser library Lezer. Otherwise, the developed ST language server checks the syntax conforms to the IEC 61131-3 2nd edition, which is supported by MatIEC, and marks anomalies as syntax errors. The editor's validation function runs the compiler to validate the code, fulfilling the requirement for user to be able to confirm code's correctness partially. Rest of the requirement is fulfilled by the debugging features, which allow the user to debug the program to an extent via debug prints and live variable panel.

Other improvements to editor include the implementation of syntax highlighting, suggestions for variables, keywords and functions and autocomplete for the suggestions. All of these are huge improvements to the starting point, which was essentially a plain text box. Syntax highlighting helps user to distinguish keywords from identifiers and function calls at first glance. Suggestions and autocomplete support the popular requirement of syntax help by providing list of possible keywords and variables at a position. The autocomplete additionally makes writing code faster and lowers the number of typographical errors as just couple of letters is often enough to narrow suggestions to the correct word.

The choice for the initial feature set fell on the developers as they have the most experience with programming with textual languages and have strong opinions which features are important in a code editor. Due to time limitations, only few most important features were implemented.

The syntax documentation was one feature which was deemed very important in the interviews and the implementation for it was started immediately after identifying its importance. The syntax reference material about the language structures, datatypes, operators, and standard functions was created based on the IEC 61131-3 standard's 2nd edition. This material is used as a base for the official user documentation, which is linked to the editor for easy access. Some users who read the material, told that it really helps the development of ST code as a beginner. This feature is the clearly most wished missing feature from the editor based on the interviews and the feature greatly adds value to the tool.

The debugger was found very successful part of the new ST development tool and the features implemented were the ones that were also wished for. However, when observing the use of the tool the users seemed to have a bit trouble following the execution of the code. The common debugger feature is to execute the code step by step and when asked about this, the interviewees agreed immediately that would be very appreciated feature. Step-by-step debugging unfortunately is also very difficult to implement technically. On the other hand, it would make the debugging much easier and again improve the value of the tool. The technical difficulties, work required and received benefit should all be carefully estimated.

Overall, the implementation already greatly improved the user experience of ST programming, and the language server has still huge amount of potential. Language Server

Protocol supports a huge variety of features used in IDEs and the implemented Language Server gives the base to build features upon.

8.2.1 Technical evaluation

The internal code structures for the Language Server are mostly focused on parsing and selecting meaningful data from the concrete syntax tree. The lack of meaningful error messages at the parser level is a major caveat of Lezer and an unfortunate overlook while evaluating the options for a parser generator. The choice to use a parser library was influenced by the timeframe of this project and the lack of experience regarding parsers. If handwritten parser would have been selected as the initial approach, there is a chance other useful features such as the debug prints could not have been made in this timeframe. The parser of the Language Server should eventually be replaced with better suited parser. Likely custom handcrafted parser would be the best choice. One early idea was to keep the parser library as decoupled from the rest of the language server as possible to ease the replacement of the parser. In practice this is not easily seen in the code as most of the code is directly related to parsing and interpreting the parse results. As the parse tree must be read with the methods in the Lezer runtime library, there isn’t very easy way to decouple the Lezer generated parse tree from other functionality. One considered method was re-creating the parse tree as own custom data structure. This is redundant work and was not deemed a good idea at the time. Instead, the functionality uses Lezer’s helper functions and methods and when the time of rework comes, most of the code must be rewritten. It is recommended to fix design issues earlier rather than later as the cost of the issue becomes much higher the further development proceeds. In this case, the rewrite of the language server’s parser should take priority before extending the feature set of it.

One major benefit of Lezer is the incremental parsing feature. The performance of the parser is much less affected by the size of the document. In fact, during performance profiling, there was a significant difference in processing time when parsing few thousand lines of code. Time taken to parse few thousand lines was less than a second with incremental parsing, while full document parse took few seconds. However, the performance boost from the incremental parsing may not be as important as initially thought. The parsing is relatively fast operation with small documents and the added complexity due to the incremental parsing may not be worth it if the users generally write files which are under thousand lines long. Considering the scenario where the user writes code without incremental parsing, the analysis is constantly run on the entire code, which is much easier to implement but uses more system resources. If running the analysis takes for example one second, it still is fast enough for the user to likely not notice difference between running the analysis for the entire code versus partial analysis. However, if there are additional analysis steps in the future and the users work with larger code files, then the importance of the incremental parsing becomes more and more relevant. Incremental parsing may be one feature which could be omitted from the future replacement parser if incremental parsing is deemed as overly complex feature and users are expected to write relatively small programs.

The general design from architectural standpoint for an entirely browser-based editor was a great success. The web workers are perfect for relatively independent sub-programs such as language servers. Some modern code editors work entirely online in a web browser and as such there exists open-source tools and libraries which can be used to easily embed a fully working code editor into a web page. As the editor is designed to be used in a web browser from the start, integrating it into Valmet's configuration environment was a breeze. As the editor is browser based, besides making the integration requirement much easier to fulfill, it also makes the debugging side much easier to implement, as the configuration environment has existing mechanisms to read data from the runtime and the editor can utilize these same mechanisms.

The benefits of the language server protocol are very small in this use case as there aren't a lot of different languages used in the first place. However, it is a solid foundation if there will be need for other languages in the future. The editor client is easily re-usable for another language, especially if a fully featured language server already exists for the language. In that case, supporting new language is not much more complex from just defining language configuration and launching the appropriate language server in a web worker.

The autocomplete feature implementation includes static list of keywords and additionally the user defined variables and functions. While the feature works, there is room for improvement. The autocomplete could also suggest user defined datatypes and the snippets mentioned before. The user defined variables are collected from the parse tree after the parse is finished by iterating specific sections of the tree structure.

The debug functionality is mostly implemented in the ST code runtime. The runtime development was not the target of this thesis and therefore only the front-end for the debugger is evaluated. The implementation works well, and the editor should be quite responsive even with larger amount of data. This is largely supported by design where only the changed values are updated in the editor's user interface's list of variables. The debug print log is limited to 1000 entries, so the memory won't run out even if user runs debugger for extended periods of time. It is extremely unlikely anyone would need to have more than 1000 debug prints visible at the same time. The step-by-step debugging is technically very difficult to implement but would be very much wished feature. The step-by-step debugging is not only wished for ST but also to the function block diagrams. This feature is more dependent on the runtime and debugger backend, rather than the editor and debugger frontend.

9. CONCLUSIONS

The goal of this thesis was to find out what features are required for an efficient IEC ST development tool and implement a tool with some subset of the features, improving the user experience. The tool is integrated into Valmet's web-based automation platform, which created constraints for the implementation.

The requirements for the tool were researched in a few different ways. Major research method for the requirements are user interviews, which were moved to later part of the project. This was justified by the assumption the interviewees are not familiar with IEC ST and for the results it is beneficial to have some sort of demonstration of the tool. Thus, the implementation was done before the interviews. The second part of thesis, the implementation was designed by researching common design in programming language tools and parsing basics. Early in the project free IEC 61131-3 compliant IDE's, such as OpenPLC, were tried to get better understanding of common features in PLC oriented development tools. In practice, the implementation prototype was created by trying different libraries in attempt to create a proof of concept.

The conducted interviews produced wishes which were then analyzed whether they are significant enough to be requirements. All interviewees were inexperienced with the ST language and had varying experience with programming in general. Thus, the requirements identified via interviews are biased towards features for inexperienced programmers. Unfortunately, there were no ST experts found as interviewees. Experienced ST programmer could have had different opinions about the tool. Some of the requirements were initially given at the beginning of the project, while the rest were identified via interviews near the end of the project. The requirements identified in the end inclined more towards improvement ideas or wishes rather than well specified requirements, but they are nevertheless valuable data to answer the question, what features should the editor include.

When considering the results, it must be noted that in the Valmet's environment the ST language support is not the primary way of programming automation applications and as such, the requirements are different if one would for example create a system mainly focused on the ST. The requirements list is not exhaustive and obvious requirements such as “The editor must allow saving the code to the configuration environment” are not listed. Features which help to write correct syntax came up most frequently during the interviews. This was also in line with author's own estimates for important features. Other category of requirements focused on editor automatically doing part of the work, such as automatically defining ST function block’s interface in ST code. Majority of the requirements can be divided into these two categories.

The implementation was a great success and provides a great example how language specific tooling can be implemented in completely web-based environment. The use of the Language Server Protocol makes supporting additional languages as easy as finding suitable Language Server and connecting the editor to it. As the most important requirements revolved around the syntax help, the new tool has autocomplete, syntax suggestions, syntax highlight and readily available ST syntax documentation. These features greatly improve the user experience which was one of the main goals of this thesis. However, there is still a lot of potential for the language tooling which is not utilized. The implementation is thus evaluated to be a major step to the correct direction, rather than finished and polished tool.

Technically the implementation has one unfortunate drawback. The used parser generator Lezer was not entirely suited for the purpose and made implementing the proper error messages very difficult. Other technical solutions worked perfectly and provide a good implementation example for web-based code editor language specific tooling. The choice to use language server and run it in a dedicated web worker works extremely well in a web environment. The downside to using such web-based environment is the lack of directly compatible language servers. In theory, any textual language could be supported by the editor by using suitable language server. This opens a lot of possibilities to extend the language support in the future.

To develop the Structured Text language server further, there are several features the Language Server could support. ST language generally contains quite a lot of repetitive and strict structures and would benefit quite a lot about automatic generation of these repeated structures. The implementation done in this thesis implements one feature to aid automatic generation, autocomplete suggestions. Further ideas for this are larger snippets, automatic boilerplate generation and templates. Another point of research is to investigate how the parser should be improved. One option is to rewrite the parser completely by hand. For that option, the amount of work required, and possible algorithms should be investigated. Another option is to investigate alternative parser libraries, but as was seen in this thesis, their features are rarely ideal for code editor’s purposes. And most of, if not all the parser related code must still be rewritten.

As most of the publicly available language servers are designed to work in a native desktop environment, one point of research could be a general adapter for use of language servers in web environment. This would make addition of new languages to the editor very easy.

Once the tool has been used for some time, user feedback could be beneficial to improve aspects which were not identified during this thesis due to lack of Structured Text experience.

REFERENCES

[1] T. Alves and T. Morris, “OpenPLC: An IEC 61131–3 compliant open source industrial controller for cyber security research,” *Comput. Secur.*, vol. 78, pp. 364–379, 2018.

[2] R. Casey, “Echidna,” 24-Jul-2022. [Online]. Available: https://github.com/61131/echidna. [Accessed: 15-Aug-2022].

[3] K. Cooper, *Engineering a compiler*, 2nd ed. Amsterdam: Elsevier, 2012.

[4] J. Dolinay, P. Dostalek, and V. Vasek, “Arduino Debugger,” *IEEE Embed. Syst. Lett.*, vol. 8, no. 4, pp. 85–88, 2016.

[5] D. Grune and C. J. H. Jacobs, *Parsing Techniques: A Practical Guide*. New York, NY: Springer New York, 2007.

[6] D. H. Hanssen, *Programmable Logic Controllers: A Practical Approach to IEC 61131-3 Using CoDeSys*. New York: John Wiley & Sons, Incorporated, 2015.

[7] K.-H. John and M. Tiegelkamp, *IEC 61131-3: Programming Industrial Automation Systems: Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Aids*, 2. Aufl. Berlin, Heidelberg: Springer-Verlag, 2010.

[8] J. Kjær Rask, F. Palludan Madsen, N. Battle, H. Daniel Macedo, and P. Gorm Larsen, “The Specification Language Server Protocol: A Proposal for Standardised LSP Extensions,” *Electron. Proc. Theor. Comput. Sci.*, vol. 338, pp. 3–18, 2021.

[9] G. Komarov, “IEC Checker,” 05-Aug-2022. [Online]. Available: https://github.com/jubnzv/iec-checker. [Accessed: 08-Aug-2022].

[10] H. Liang, L. Sun, M. Wang, and Y. Yang, “Deep Learning With Customized Abstract Syntax Tree for Bug Localization,” *IEEE Access*, vol. 7, pp. 116309–116320, 2019.

[11] T. Æ. Mogensen, *Introduction to Compiler Design*, 1st ed. 2011. London: Springer London, 2011.

[12] S. Queiroz de Medeiros, G. de Azevedo Alvez Junior, and F. Mascarenhas, “Automatic syntax error reporting and recovery in parsing expression grammars,” *Sci. Comput. Program.*, vol. 187, 2020.

[13] Richard C Harwell and L. Sparks Kerry, “IEC 61131-3, CoDeSys standardize control logic: ease control programming across multiple controller platforms using IEC 61131-3-based CoDeSys programming software,” *Control Eng.*, p. 20+, Jan. 2011.

[14] T. Rodrigues Alves, M. Buratto, F. M. de Souza, and T. V. Rodrigues, “OpenPLC: An open source alternative to automation,” in *IEEE Global Humanitarian Technology Conference*, 2014, pp. 585–589.

[15] M. de Sousa, “Data-type checking of IEC61131-3 ST and IL applications,” in *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation*, Krakow, 2012, pp. 1–8.

[16] M. de Sousa, “MatIEC repository,” *Bitbucket*. [Online]. Available: https://bitbucket.org/mjsousa/matiec_git/src/master/. [Accessed: 11-Aug-2022].

[17] M. de Sousa, “On Analyzing the Semantics of IEC61131-3 ST and IL Applications,” in *Advances in Sustainable and Competitive Manufacturing Systems*, Heidelberg: Springer International Publishing, 2013, pp. 559–571.

[18] M. Sulír, M. Bačíková, S. Chodarev, and J. Porubän, “Visual augmentation of source code editors: A systematic mapping study,” *J. Vis. Lang. Comput.*, vol. 49, pp. 46–59, 2018.

[19] S. D. Swierstra and P. R. A. Alcocer, “Fast, error correcting parser combinators: A short tutorial,” in *SOFSEM’99: Theory and Practice of Informatics*, Milovy, 1999, pp. 112–131.

[20] E. Tisserant, L. Bessard, and M. de Sousa, “An Open Source IEC 61131-3 Integrated Development Environment,” 2007, vol. 1, pp. 183–187.

[21] S. C. Weller *et al.*, “Open-ended interview questions and saturation,” *PloS One*, vol. 13, no. 6, 2018.
[22] Y. Yan and H. Zhang, “Compiling Ladder Diagram into Instruction List to comply with IEC 61131-3,” *Comput. Ind.*, vol. 61, no. 5, pp. 448–462, 2010.
[23] V. E. Zyubin, A. S. Rozov, I. S. Anureev, N. O. Garanina, and V. Vyatkin, “poST: A Process-Oriented Extension of the IEC 61131-3 Structured Text Language,” *IEEE Access*, vol. 10, pp. 35238–35250, 2022.
[24] “IEC 61131-3:2013,” *IEC Webstore*. [Online]. Available: https://webstore.iec.ch/publication/4552. [Accessed: 08-Jul-2022].
[25] “Status IEC 61131-3 standard,” 19-Jul-2018. [Online]. Available: https://plcopen.org/status-iec-61131-3-standard. [Accessed: 07-Nov-2022].
[26] “IEC 61131-3:2013 Programmable controllers - Part 3: Programming languages,” IEC, 2013.
[27] “Beckhoff Information System - English.” [Online]. Available: https://infosys.beckhoff.com/english.php?content=../content/1033/tc3_plc_intro/135987851.html&id. [Accessed: 23-Aug-2022].
[28] “IEC 61499-1:2012,” *IEC Webstore*. [Online]. Available: https://webstore.iec.ch/publication/5506. [Accessed: 23-Aug-2022].
[29] “IEC 61499-2:2012,” *IEC Webstore*. [Online]. Available: https://webstore.iec.ch/publication/5507. [Accessed: 23-Aug-2022].
[30] “IEC 61499-4:2013,” *IEC Webstore*. [Online]. Available: https://webstore.iec.ch/publication/5508. [Accessed: 23-Aug-2022].
[31] “CoDeSys Development System,” *CODESYS*. [Online]. Available: https://www.codesys.com/products/codesys-engineering/development-system.html. [Accessed: 08-Jul-2022].
[32] “Why is the instruction list (IL) language for PLCs falling out of favor?” [Online]. Available: https://www.motioncontroltips.com/why-is-the-instruction-list-il-language-for-plcs-falling-out-of-favor/. [Accessed: 23-Aug-2022].
[33] “Ladder Diagram (LD) Programming | Basics of Programmable Logic Controllers (PLCs) | Automation Textbook.” [Online]. Available: https://control.com/textbook/programmable-logic-controllers/ladder-diagram-ld-programming/. [Accessed: 28-Jul-2022].
[34] “Abstract vs. Concrete Syntax Trees - Eli Bendersky’s website.” [Online]. Available: https://eli.thegreenplace.net/2009/02/16/abstract-vs-concrete-syntax-trees/. [Accessed: 17-Aug-2022].
[35] “Official page for Language Server Protocol.” [Online]. Available: https://microsoft.github.io/language-server-protocol/. [Accessed: 22-Aug-2022].
[36] “Debug Adapter Protocol.” [Online]. Available: https://microsoft.github.io/debug-adapter-protocol/overview. [Accessed: 12-Sep-2022].
[37] “GDB: The GNU Project Debugger.” [Online]. Available: https://www.sourceware.org/gdb/. [Accessed: 01-Sep-2022].
[38] “Testing and Debugging,” *CODESYS*. [Online]. Available: https://help.codesys.com/api-content/2/codesys/3.5.17.0/en/_cds_struct_test_application/. [Accessed: 01-Sep-2022].
[39] “OpenPLC – Open-source PLC Software.” [Online]. Available: https://openplcproject.com/. [Accessed: 12-Aug-2022].
[40] “1.5 Installing OpenPLC Runtime on Microcontrollers,” *OpenPLC*. [Online]. Available: https://openplcproject.com/docs/installing-openplc-runtime-on-arduino-and-other-platforms/. [Accessed: 12-Aug-2022].
[41] “PLC programming with SIMATIC STEP 7 (TIA Portal),” *siemens.com Global Website*. [Online]. Available: https://new.siemens.com/global/en/products/automation/industry-software/automation-software/tia-portal/software/step7-tia-portal.html. [Accessed: 15-Aug-2022].

[42] "TwinCAT 3 Engineering," *Beckhoff Automation*. [Online]. Available: https://www.beckhoff.com/en-en/products/automation/twincat/te1xxx-twincat-3-engineering/te1000.html. [Accessed: 15-Aug-2022].

[43] "Software Construction Guidelines," *PLCOpen*, 12-Jul-2018. [Online]. Available: https://plcopen.org/software-construction-guidelines. [Accessed: 15-Aug-2022].

[44] "Official page for TypeScript." [Online]. Available: https://www.typescriptlang.org/. [Accessed: 20-Sep-2022].

[45] "Using Web Workers - Web APIs | MDN." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers. [Accessed: 22-Sep-2022].

[46] "Lezer." [Online]. Available: https://marijnhaverbeke.nl/blog/lezer.html. [Accessed: 26-Sep-2022].

[47] "IEC 61131-3:2003," *IEC Webstore*. [Online]. Available: https://webstore.iec.ch/publication/19081. [Accessed: 27-Sep-2022].