FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

# An LLVM Based Compiler for the IEC 61131-3

**Tiago Catalão**

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Mário Jorge de Sousa

July 30, 2020

# Resumo

O objectivo deste trabalho é dotar o compilador MatIEC, para as linguagens de programação definidas na norma IEC 61131-3, com a capacidade de gerar directamente código binário e executável em vez de o traduzir para ANSI C.

Das cinco linguagens de programação definidas nessa norma IEC 61131-3, este projecto foca-se apenas nas linguagens Structured Text (ST) e Sequential Function Chart (SFC). O novo gerador de código irá ser escrito recorrendo ao projecto LLVM, uma infraestructura para desenvolvimento de compiladores, e posteriormente integrado no código-fonte existente do compilador MatIEC.

O desempenho do novo compilador será então comparado com o do anterior tradutor recorrendo a testes específicos para os programas executáveis gerados para ambas as linguagens de programação suportadas. Esta comparação será também feita com um outro compilador proprietário, não baseado no MatIEC, compatível com a norma IEC 61131-3: o ambiente de desenvolvimento Codesys. Isto permite obter uma melhor ideia do desempenho do novo compilador, pois a comparação é feita com um compilador muito usado em aplicações reais.

O desenvolvimento deste projecto levou à submissão de um artigo *Work-in-Progress* para a vigésima quinta edição da conferência internacional anual *Emerging Technologies and Industrial Automation* (ETFA) 2020 — em Português, “Tecnologias Emergentes e Automação Industrial” — organizada pelo instituto IEEE.

# Abstract

The purpose of this project is to furnish the MatIEC compiler for the IEC 61131-3 languages with the ability to directly generate binary code instead of translating the source code to ANSI C.

Of the five IEC 61131-3 languages, the focus will land on the Structured Text (ST) and Sequential Function Chart (SFC) languages. For this, a new code generator will be written making use of the LLVM compiler infrastructure project and integrated into the existing MatIEC codebase.

The new compiler will then be compared against the previous MatIEC code translator in a series of specific tests to measure the performance of the generated executables for both of the supported languages. This comparison will also be made with a completely different compiler, the proprietary IEC 61131-3 environment Codesys to give an idea of how the new compiler performs compared to a compiler that is widely used in real applications.

The development of this project resulted in the submission of a Work-in-Progress paper for the IEEE's 25th Emerging Technologies and Factory Automation (ETFA) 2020 international conference.

# Acknowledgements

I would like to begin by thanking Professor Mário Jorge de Sousa for guiding me through this at-first-implausible venture. His explanations simplified what seemed daunting and beyond comprehension, easing the development of this work.

To my family, my parents and sisters, for their support during the research and writing process.

To all my friends, which directed my academic life culminating in this project; without you, I would have finished my degree one year earlier, but with the sensation of something missing. To those that questioned my interest in compilers and programming languages, and their design and implementation, regarding it as unnatural or just poor taste; thank you for reinforcing my motivation to complete this project, even if in an unintentional way.

A special thanks to my friend Rafael for taking his time to help me despite needing to write his own dissertation at the same time.

I would also like to thank my dog Stitch, for his companionship and tirelessly taking me for a walk every two hours while writing this dissertation; those breaks were very much needed.

Finally, a very special thank you to my girlfriend Raquel, for your invaluable help, patience, and ability to put up with me throughout the duration of this project. Thank you for the moments when I thought I would not be able to complete this project and you gave me the needed confidence.

Your repeated proofreading also revealed quite a few mistakes, even when you surely could not stand to read another word of this document.

Tiago Catalão

*"Perfection is achieved, not when there is nothing more to add,
but when there is nothing left to take away."*

Antoine de Saint-Exupery

# Contents

# List of Figures

# List of Tables

# Abbreviations

| | |
|---|---|
| ABI | Application Binary Interface. |
| ANSI | American National Standards Institute. |
| API | Application Programming Interface. |
| AST | Abstract Syntax Tree. |
| CPU | Central Processing Unit. |
| CST | Concrete Syntax Tree. |
| FBD | Function Block Diagram. |
| FPGA | Field Programmable Gate Array. |
| GCC | GNU Compiler Collection. |
| GNU | GNU's Not Unix. |
| GPL | General Public License. |
| IC | Integrated Circuit. |
| IDE | Integrated Development Environment. |
| IEC | International Electrotechnical Commission. |
| IEEE | Institute of Electrical and Electronics Engineers. |
| IL | Instruction List. |
| IR | Intermediate Representation. |
| ISA | Instruction Set Architecture. |
| ISO | International Organization for Standardization. |
| LD | Ladder Diagram. |
| LLVM | Low Level Virtual Machine. |
| LTO | Link-Time Optimisation. |
| PLC | Programmable Logic Controller. |
| POU | Program Organisation Unit. |
| SDK | Software Development Kit. |
| SFC | Sequential Function Chart. |
| SSA | Single Static Assignment. |
| ST | Structured Text. |
| XML | eXtensible Markup Language. |

# Chapter 1

# Introduction

Programmable Logic Controllers (PLCs) were introduced in the 1970s[1] and were rapidly adopted by the industry to replace earlier means of process automation such as hard-wired relay and timer logic control systems and electronic controllers using logic gates. Those systems were complex to build by hand, difficult to maintain and changes in the control logic were not possible. To allow PLCs to replace these systems, they had to be made robust as they would have to withstand harsh conditions in an industrial environment (temperature, humidity, vibration, electrical noise, dust, etc.) and provide consistent, predictable, and repeatable operation. The major advantage of PLCs over earlier systems relies on the flexibility of defining its operation in software and thus allowing circuit logic modifications quickly through software changes.

## 1.1 Context

Since its introduction, PLCs have evolved a lot from simple microprocessors to full-fledged computers running specialised real-time operating systems with several communication protocol stacks. Different brands made PLCs with different architectures, tooling, development environments, and programming languages which made reusing of code between PLCs of different manufacturers and its co-existence within an industrial setting very impractical or even impossible. The International Electrotechnical Commission (IEC) acknowledged this problem and addressed it by publishing a series of standards, the IEC 61131, comprised of several parts to abstract away from the software developer PLC characteristics such as architecture and operating system. The third part of this standard (IEC 61131-3) describes the programming languages used to develop software for industrial automation within PLCs and is what this project will focus on. Programs written in these human-readable languages must be converted to a format that computers can understand and be able to execute—that is the job of a compiler.

[1]Although the first PLC was created in 1968, the market adoption only began in the 1970s

## 1.2 Motivation

MatIEC is an open-source compiler for the IEC 61131-3 languages whose development was started by Professor Mário Jorge de Sousa. It is used by open-source projects (such as Beremiz) aiming to provide a solution for developers to dissociate from proprietary-licensed PLC software and further allow standard-compliant software re-use and independence from manufacturer specifics. MatIEC compiles programs written in IEC 61131-3 languages to ANSI C, needing a compliant C compiler to generate executable code. This project arises from the desire to achieve direct binary code generation, without having to resort to an external program; this has major advantages over translation to another language.

## 1.3 Aims and Objectives

This project aims to create a compiler back-end for the MatIEC compiler, enabling it to directly generate executable code. For this purpose, the LLVM compiler infrastructure project will be used as it brings several advantages and useful capabilities.

The current MatIEC release complies with the second edition of the IEC 61131-3 standard; a third has been published by the IEC, adding many functionalities. A new compiler front-end for MatIEC conforming to the third edition was started in 2016 by the student Bruno Silva; it consists of a new MatIEC front-end building upon the existing MatIEC project. This project proposes to create a new MatIEC back-end, reusing the current MatIEC front-end. Support for the latest IEC 61131-3 standard version would be desirable, but, due to some incompatibilities, this work cannot be integrated into the newer MatIEC front-end; the LLVM-based MatIEC back-end will thus build upon the existing MatIEC release, supporting only the second version of the standard.

Despite not being possible to use the previous MatIEC front-end project, the project is still a very useful resource as it studies both the IEC standard and MatIEC, possibly easing the understanding of the inner details of MatIEC and issues with the standard itself.

This project will focus on the generation of the intermediate representation of an IEC 61131-3 program to feed to the LLVM compiler framework, which will create an executable program accordingly.

From the five programming languages defined in the standard, support for the Structured Text and Sequential Function Chart languages is the main goal of this work.

## 1.4 Document Organisation

This dissertation is composed of 8 chapters, the first being this Introduction.

The Literature Review is divided into 4 chapters:
Chapter 2 presents an overview of the IEC 61131 standard;
Chapter 3 summarises compiler architectures and generators;
Chapter 4 pertains to the LLVM compiler infrastructure—its architecture, reason, and use;
Chapter 5 explains related work and the state of the art in IEC 61131-3 compilers.

The Methodology is approached in Chapter 6, detailing the proposed solution and its implementation. Chapter 7 is an analysis and discussion of the obtained results.

Finally, Chapter 8 formalises the conclusion and details future work.

# Chapter 2

# The IEC 61131 standard

The International Electrotechnical Commission (IEC) is an organisation that publishes International Standards regarding electrical and electronic technologies, aiming to promote international co-operation within these fields and to certify whether systems conform to the established standards. The IEC cooperates with the International Organization for Standardization (ISO) and Institute of Electrical and Electronics Engineers (IEEE) to define International Standards, which are numbered in the 60000–79999 range[4].

## 2.1 Introduction to the standard

The incompatibility between different vendors' PLCs architectures, programming languages, tooling and development environments which prevented interoperability, code reuse and coexistence created a need for standardisation for industrial automation. This led the IEC to the development of the IEC 61131 standard, clearly defining PLCs programming languages, communication, and software architectures, aiming for international co-operation.

## 2.2 Structure of the standard

The IEC 61131 standard—formerly known as IEC 1131 before the IEC numbering change—was first published in 1992 with two parts, IEC 61131-1 and IEC 61131-2. The third part of the standard was published in 1993, IEC 61131-3; the fourth part in 1995 and the fifth, seventh and eighth parts in 2000; In 2003 parts 1, 2, 3 and 8 were revised; Parts 2 and 4 were revised in 2004 and part 2 later in 2007; Part 6 was created in 2012 and 9 in 2013—when part 3 was also revised; In 2017 parts 2 and 8 were republished and the last part, the 10th, was first published in 2019.

The latest standard, IEC 61131:2020 SER, can be acquired in full in the IEC *Webstore*.

Each part of the standard defines a specific subject regarding Programmable Controllers:

**IEC 61131-1:2003** Programmable Controllers - Part 1: *General information*

**IEC 61131-2:2017** Industrial-process measurement and control - Programmable controllers - Part 2: *Equipment requirements and tests*

**IEC 61131-3:2013** Programmable controllers - Part 3: *Programming languages*

**IEC TR 61131-4:2004** Programmable controllers - Part 4: *User guidelines*

**IEC 61131-5:2000** Programmable controllers - Part 5: *Communications*

**IEC 61131-6:2012** Programmable controllers - Part 6: *Functional safety*

**IEC 61131-7:2000** Programmable controllers - Part 7: *Fuzzy control programming*

**IEC TR 61131-8:2017** Programmable controllers - Industrial-process measurement and control - Programmable controllers - Part 8: *Guidelines for the application and implementation of programming languages*

**IEC 61131-9:2013** Programmable controllers - Part 9: *Single-drop digital communication interface for small sensors and actuators (SDCI)*

**IEC 61131-10:2019** Programmable controllers - Part 10: *PLC open XML exchange format*

The first part of the standard defines the terms used throughout the series of standards and identifies the functional characteristics of programmable controller systems and functional characteristics such as the basic structure of a programmable controller system, characteristics of Central Processing Unit (CPU) functions, interfaces to sensors and actuators, communication, human-machine interfaces, power-supply, and programming, debugging, monitoring, testing, and documentation functions[5].

Another part of the series that is interesting when implementing the IEC 61131-3 is the IEC TR 61131-8:2017, the eighth part, which provides guidelines for the application of the IEC 61131-3 standard and its implementation for programmable controller systems, such as usage of data types, data passing over Program Organisation Units (POUs), object orientation implementation, recursion, namespaces implementation, scheduling, concurrency, and synchronisation mechanisms, communication facilities and programming practices[6].

The working group responsible for the IEC 61131 standard published a new part in April 2019—IEC 61131-10—which deals with interoperability and exchange of programs written in the different programming languages defined in the IEC 61131-3. This further enables co-operation within developers and interoperability between different development environments and promotes software reuse. This is done by adopting a widely used markup language, eXtensible Markup Language (XML), to encode the information contained in programs written in any of the standardised languages in a single format. The translation into this format must not lose information so it can be reversed and result in the same source construction. It should also allow incorrect and incomplete program constructions as it would allow developers to exchange programs that are not yet complete, especially for graphical languages that may have partial or disconnected blocks[7].

For the sake of standards compliance and interoperability with other development environments, it is desirable to add support for this IEC 61131-10 XML format to an IEC 61131-3 compiler—both to import and export constructs and programs written in this portable format. This ensures interoperability with other IEC 61131-3 environments.

## 2.3 The IEC 61131-3

The third part of the series of the IEC 61131 standards defines five programming languages for use with PLCs and their syntactic and semantic rules: Structured Text (ST) and Instruction List (IL) are textual programming languages, Ladder Diagram (LD) and Function Block Diagram (FBD) are graphical languages and Sequential Function Chart (SFC) is a structural language, which may be represented either in a textual or graphical format[1].

The defined languages have distinctive characteristics and target different audiences:

**Function Block Diagram:** Very similar to a wiring diagram based on small scale Integrated Circuits (ICs) and its intuitive design makes it very easy to understand even for non-programmers but makes it hard to write larger programs

**Instruction List:** Low-level, assembly-like, language. Complex control functions and large programs are hard to write and maintain in it

**Ladder Diagram:** Maybe the most used language of this standard, it resembles relay-based electrical circuit diagrams. This also makes it easy to use for non-programmers, but complex constructs are hard to describe with it

**Sequential Function Chart:** Graphical language that describes state machines. Forcing this structure makes it unsuitable for every application as it may add unneeded additional complexity. Very intuitive, but usually executes slower than others.

**Structured Text:** This programming language has a syntax similar to Pascal and allows complex control structures and computations to be written easier than other languages. Being a textual language will likely outperform its counterparts and is suitable for writing large and complex programs. Due to its greater usage complexity, it is not as intuitive for non-programmers

### 2.3.1 Versions

The first edition of the IEC 61131-3 was published in 1993 and later a second edition was made public in 2003 which is a technical revision and cancels and replaces the previous edition.

The amendments to the previous version are:

- Numeric literals
- Elementary data types
- Derived data types
- Single element variables
- Temporary variable declaration
- `RETAIN` and `NON_RETAIN` variable attributes
- Invocations and argument lists of functions
- Type conversion functions

- Functions of time data types
- Extended initialisation facilities for function blocks
- Pulse action qualifiers
- Action control
- Configuration initialisation

The third and last edition was released in 2013 and adds significant adjustments and functionalities to the previous specification. These changes remain compatible with the preceding norms as to preserve backward compatibility, which is of utmost importance for the development of reliable software. This edition also deprecates the IL programming language and discourages its use, but any standard-conforming compiler should maintain the support for the language and issue a deprecation warning. This ensures backward compatibility, as older programs exist and should not need to be rewritten to continue working. Backward compatibility ensures a stable development environment and that a program written complying with a version of the standard will remain valid in the future.

The third version of the standard also added considerable changes to the previous:

- Data types with an explicit layout
- Type with named values
- Elementary data types
- References, functions, and operations with reference
- Partial access to `ANY_BIT`
- Variable-length `ARRAY`
- Initial value assignment
- Type conversion rules, implicit and explicit
- Function—call rules, without function result
- Type conversion functions of numerical, bitwise data, *etc.*
- Functions to concatenate and split of time and date
- Classes, including methods, interfaces, *etc.*
- Object-oriented function blocks, with methods, interfaces, *etc.*
- Namespaces
- Structured Text: `CONTINUE`, *etc.*
- Ladder Diagram: Contacts for compare (typed and overloaded)
- ANNEX A: Formal specification of language elements

And deprecates:

- Octal literals
- Directly represented variables in the body of POUs and methods
- Overloaded truncation `TRUNC`
- Instruction List language
- "Indicator" variable of action block

### 2.3.2 Software model

The IEC 61131-3 defines a software model for PLCs as seen in Fig. 2.1.

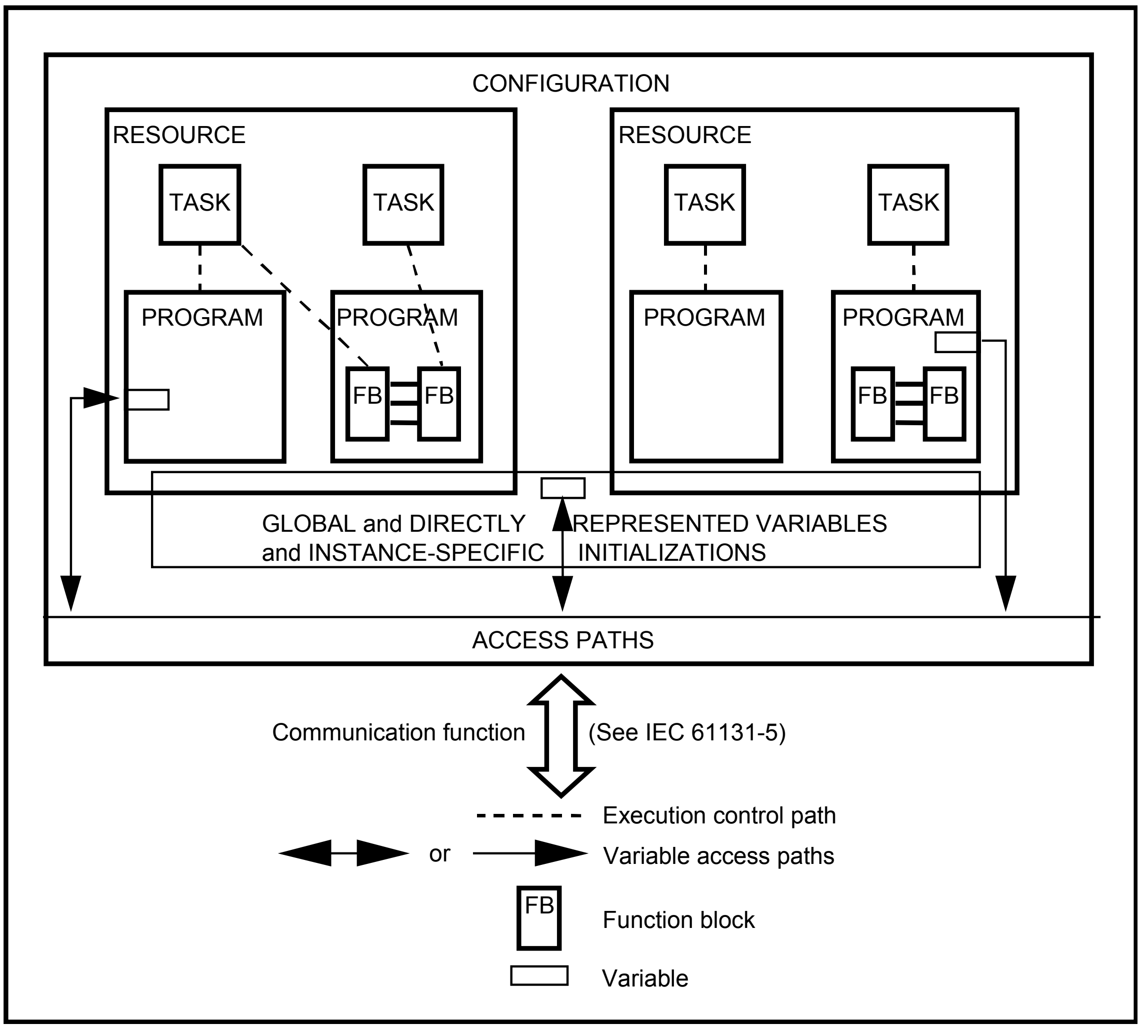


Figure 2.1: Software model (from [1])

The *configuration* is the top level of the software model and represents a *programmable controller system* as defined in the IEC 61131-1. This defines the structure and logic of a PLC system and contains one or more *resource*. A *resource* is a "signal processing function" and its "man-machine interface" and "sensor and actuator interface" functions[1], which is able to execute *programs*. An example of a *resource* in a PLC is a CPU. *Resources* contain *programs* which execution is invoked and controlled by *tasks*. *Tasks* have run-time properties that can be configured to execute a *program* cyclically, periodically, or triggered by the rising edge of a variable and can have different priorities. These software blocks are called Program Organisation Units (POUs).

There are three[1] types of POU: Functions, Function Blocks, and Programs.

[1]In the third edition of the standard, the *Class* was added as a fourth type of POU. This document only focuses on the second edition of the standard; so unless otherwise specified, everything explained pertains to the second edition (from 2003) of the IEC 61131-3 standard.

A Function is a POU with no internal memory or state that accepts input values and returns one value when executed. Since it has no internal state (global variables), the output value entirely depends on the inputs—multiple invocations with the same input values will always yield the same result.

Several standard functions are defined and available for use. A few examples are:

**Numerical functions:** `ABS`, `SQRT`, `SIN`, *etc.*

**Arithmetic functions:** `ADD`, `MUL`, `EXPT`, *etc.*

**Bit-shift functions:** `SHL`, `ROR`, *etc.*

**Bitwise boolean functions:** `AND`, `NOT`, `XOR`, *etc.*

Many more functions are defined: selection, comparison, and datatype specific operation and conversion functions.

Functions can be declared with the construct `FUNCTION...END_FUNCTION`.

A Function Block is a POU with an internal state or memory. Upon execution, it can return one or more values and multiple executions with the same input values can yield different results since a Function Block can maintain an internal state. A Function Block can have multiple *instances*—each instance is a copy of the same Function Block, along with a data structure with its output and internal variables. All instances will perform the same operations defined in the Function Block declaration, but each has its own copy of internal variables, so the invocation of different instances of the same Function Block with a similar input might return different values; but invocation with a similar sequence of inputs, always yields the same results, as Function Blocks provide stability and repeatability.

Several standard Function Blocks are also defined: bistable Function Blocks, edge detection Function Blocks, counter Function Blocks, timer Function Blocks, and IEC 61131-5 also defines communication Function Blocks. An example of an edge detection Function Block is Rising Edge: at each invocation, its input variable is stored in the internal memory and if a change of the input variable relative to the internal state (previous value of the input variable) from a "false" to a "true" value is detected, the Function Block returns a "true" value; otherwise, a "false" value is returned. This example clarifies the need and usefulness of storing an internal state for a Function Block.

Unlike Functions, where the return value is yielded by the expression invoking it, a Function Block return value is exposed by the field `.Q` of a Function Block instance invocation (*fb_instance*`.Q`).

A Function Block is declared with the construct
`FUNCTION_BLOCK...END_FUNCTION_BLOCK`.

A Program is defined as a "logical assembly of all the programming language elements and constructs necessary for the intended signal processing required for the control of a machine or process by a programmable controller system" in the IEC 61131-1[1]. Unlike Functions and Function Blocks, Programs can only be instantiated within a resource—while Function Blocks can be instantiated within a Program or Function Block; and Functions can be declared in any of the three. Programs can also communicate with other Programs, declare global variables and access paths, which allow data exchange between different configurations.

Programs are declared with the syntax `PROGRAM...END_PROGRAM`.

All declared variables, functions, function blocks, programs, *etc.* need to be referenced throughout a POU. That is done by providing those instances with an *identifier*, that is a unique string that refers to a specific object. Identifiers can contain letters, digits, and underline characters ('_') and are case-insensitive. Multiple underlines must be separated by other characters; the underline cannot end an identifier (no trailing underlines).

### 2.3.3 Data types

Data are stored in variables and to correctly represent and interpret them, variables have different types. This standard defines three kinds of data types: Elementary, Generic, and Derived data types.

Elementary data types are shown in Table 2.1.

Derived data types are user-defined and can be enumerated, subrange, struct, or array. Enumerated, structs and arrays are the same constructs as most programming languages provide: enumerated typed variables can only assume one of the specified values in an enumeration list; a struct data type defines sub-elements of possibly different data types (even other derived data types); an array declared variable allocates data storage space for as much variables of the same type as defined. A subrange data type might be unfamiliar as C/C++ languages do not provide an equivalent construct, but several other common programming languages, especially higher-level, do. A subrange data type definition delimits a range of values a variable instance of that type can take, including lower and upper bounds.

From all the programming languages defined in this standard, this project will only focus on textual languages. For graphical languages, many editors and Integrated Development Environments (IDEs) for IEC 61131-3 exist and can generate a textual representation from their graphical editor—usually to ST or the new XML exchange format. The textual languages are ST and IL; SFC also has both a textual and a graphical representation so support for SFC is also possible. This project will not focus on IL despite it being a textual language: it has been deprecated in the newer standard version and is not one of the main choices for industrial automation software development and thus, only the ST and SFC languages will be considered.

All textual languages have common elements: elementary data types 2.1, variables and derived types declaration, and function and function block declaration.

### 2.3.4 Structured Text

ST has a syntax similar to the Pascal programming language and is arguably the most versatile and complex programming language of this standard; it may be the most confusing for non-

Table 2.1: Elementary data types

| Keyword | Data type | Range/Precision |
|---|---|---|
| `BOOL` | Boolean | 0 or 1 |
| `SINT` | Short integer | -128 to +127 |
| `INT` | Integer | -32768 to +32767 |
| `DINT` | Double integer | -2147483648 to +2147483647 |
| `LINT` | Long integer | -9223372036854775808 to +9223372036854775807 |
| `USINT` | Unsigned short integer | 0 to 255 |
| `UINT` | Unsigned integer | 0 to 65535 |
| `UDINT` | Unsigned double integer | 0 to 4294967295 |
| `ULINT` | Unsigned long integer | 0 to 18446744073709551615 |
| `REAL` | Real number | IEC 60559 single-width floating point |
| `LREAL` | Long real | IEC 60559 double-width floating point |
| `TIME` | Duration | Implementation defined |
| `DATE` | Date (only) | Implementation defined |
| `TIME_OF_DAY` / `TOD` | Time of day (only) | Implementation defined |
| `DATE_AND_TIME` / `DT` | Date and time of day | Implementation defined |
| `STRING` | Variable-length single-byte character string | N/A: 8 bits per character |
| `BYTE` | Bit string of length 8 | - |
| `WORD` | Bit string of length 16 | - |
| `DWORD` | Bit string of length 32 | - |
| `LWORD` | Bit string of length 64 | - |
| `WSTRING` | Variable-length double-byte character string | N/A: 16 bits per character |

programmers, but programmers should be familiar with all its constructs and be comfortable writing programs in it. Besides the common elements, ST provides expression and statement constructs.

#### 2.3.4.1 Expressions

Expressions are languages' constructs that yield a specific value when executed. This value has a type and can be any of the elementary data types or a user-defined derived data type.

Expressions are composed of *operands* and *operators*. A single expression can have many operands and operators. Operators have different precedence or priority; meaning that, in an expression, higher precedence operators will be evaluated before lower precedence operators.

The available operators are presented in Table 2.2 from highest to lowest precedence. Operators in the same cell have equal precedence.

An operand can be either a variable, literal, function invocation, or another expression.

Expression evaluation is done by applying an operator to an operand following operator precedence order from highest to lowest until a single value, which is returned to the user. Equal precedence operator evaluation is done in a left-to-right manner, as written.

Table 2.2: ST language operators

| **Operation** | **Symbol** |
|---|---|
| Parenthesization | (*expression*) |
| Function invocation | *function_identifier*(*argument list*) |
| Negation<br>Complement | `-`<br>`NOT` |
| Exponentiation | `**` |
| Multiply<br>Divide<br>Modulo | `*`<br>`/`<br>`MOD` |
| Add<br>Subtract | `+`<br>`-` |
| Comparison | <, >, <=, >= |
| Equality<br>Inequality | =<br>`<>` |
| Boolean AND | `&`, `AND` |
| Boolean exclusive OR | `XOR` |
| Boolean OR | `OR` |

#### 2.3.4.2 Statements

Statements in ST must always be terminated by a semicolon. There are several types of statements defined in the standard: Assignment, Function and Function Block control, Selection, and Iteration statements. A statement can also be empty, being composed only by the terminating semicolon.

**Assignment Statements**

An assignment statement is composed of a variable identifier of any data type followed by the assignment operator, ':=', and an ST expression; it replaces the value of the variable with the result computed by the evaluation of the expression. The data type of the expression result and the variable being assigned to should be the same. An assignment can be done to multi-element variables such as structs. In this case, all elements of the original variable should be assigned the new value present in the expression return value.

A special case of assignment statements is inside functions. An assignment to an identifier equal to the function name will return the evaluated value as the function return value.

### Function and Function Block Control Statements

Function and Function Block invocation is triggered by an expression evaluation, but a control statement may be used to stop Function or Function Block execution early and return to the caller. This can be achieved with the statement '`RETURN;`' and can be useful to stop a Function, Function Block, or Program execution upon a certain condition.

### Selection Statements

A selection statement can divert program execution to a block of statements depending on a condition. Possible selection statements are '`IF`' and '`CASE`'.

The '`IF`' statement will only execute a group of statements only if the evaluation of the condition (a boolean expression) results in a `1` (or '`TRUE`') value. Otherwise, execution will skip the associated group of statements and execute no statement or execute the statement group associated with the corresponding '`ELSE`' keyword, if present. '`IF`' statements can be nested and the keyword '`ELSIF`' is provided to group the '`ELSE`' and '`IF`' keywords as a shorthand.

The '`CASE`' statement selects a statement group to be executed based on a *selector* value. The selector value is the result of an expression evaluation to a value of integer type. The yielded value will select the path of execution from the different statement blocks based on their *labels*. A label is an expression to be evaluated to an integer type. The '`CASE`' statement consists then of a selection expression and a list of statement groups, each one having one or more labels. In the case of the selector not matching any of the evaluated label values, the statement group following the '`ELSE`' keyword is chosen, if present, or none otherwise.

### Iteration Statements

Iteration statements are intended to repeat the execution of a group of statements a specific number of times or until a condition is met. If the repetition is a defined number of times, a '`FOR`' statement should be used; otherwise, '`WHILE`' and '`REPEAT`' statements execute the statements until a given condition is verified.

In the '`FOR`' statement, a given integer variable (control variable) will iterate over a range of values while the statement block is executed in each iteration. The repeated statements must not change the value of the control variable. At the end of each iteration, the control variable is incremented by the defined value, if present, or `1` otherwise. At the start of an iteration, the control variable is tested against the terminating condition (*i.e.* inside the defined range) and the statement block is executed accordingly.

'`WHILE`' and '`REPEAT`' statements operate by repeating execution of the statement block until a specified condition is met. They differ in the order in which condition testing and statement execution are made: condition evaluation is made before repeated statements execution in the '`WHILE`' construct and the inverse order for the '`REPEAT`' statement. Readers familiar with C-derived programming languages will identify these constructs as *while* and *do ... while* loops.

Iteration statements can also be nested and an '`EXIT`' statement is available to terminate an iteration statement early, either before the terminating condition is met or the iteration over the specified range is complete. If the '`EXIT`' statement is used in nested iteration statements, only the innermost loop (where the '`EXIT`' statement is located) will be broken out of, returning execution control to the first statement of the outer loop after the end of the inner loop.

The syntax of the presented ST statements can be found in Table 2.3.

Table 2.3: ST language statements

| **Statement** | **Example** |
|---|---|
| Assignment | *variable identifier* := *new value*; |
| `RETURN` | `RETURN;` |
| `IF` | `IF` *expression* `THEN` *statement list*; `ELSE` *statement list*; `END_IF;` |
| `CASE` | `CASE` *selector* `OF` *range* *label1*: *statement list*; *label2*, *label3*: *statement list*; `ELSE` *statement list;* `END_CASE;` |
| `FOR` | `FOR` *control variable* `:=` *initial value* `TO` *final value* `BY` *increment value* `DO` *statement list*; `END_FOR;` |
| `WHILE` | `WHILE` *condition* `DO` *statement list;* `END_WHILE;` |
| `REPEAT` | `REPEAT` *statement list*; `UNTIL` *condition* `END_REPEAT;` |
| `EXIT` | `EXIT;` |
| Empty statement | ; |

### 2.3.5 Sequential Function Chart

Sequential Function Chart is intended for implementing sequential logic systems, mainly programmed in a graphical representation in a graphical editor, representing a conceptual activity flow between different program *states* in an SFC network. It is composed of *steps*, *transitions*, and *actions* and is analogous to a State Machine, that has states and transitions. In SFC, this *state* is represented by a *step*, that may have associated *actions*. SFC elements are inter-connected with *directed links*.

SFC networks are a very intuitive way of programming sequential control functions and parallel logic since many independent SFC networks can be present at the same time.

Since SFC networks inherently require a state, only POUs of type Program or Function Block can embody SFC elements.

#### 2.3.5.1 Steps

A *step* represents a state in the running program, is always preceded and followed by a *transition* and can be either *active* or *inactive* at any point in time.

Steps are either represented in a textual form with the syntax `STEP...END_STEP` or a graphical representation as in Fig. 2.2. Directed links into and out of *steps* are represented with a vertical line from the top and bottom of the step, respectively, or by a `TRANSITION...END_TRANSITION` construct.

Steps have a boolean *step flag* indicating whether the specific step is active (`1`) or inactive (`0`). It can be accessed by a directed link from the right side of the step (as shown in Fig. 2.2) or textually with the construct `step_name.X`.

The elapsed time since a step has been activated is also available with the construct `step_name.T`, of type `TIME`. Upon activation of a step, `step.T` is reset to the `TIME` literal `t#0s` and once the step is deactivated, its value remains at the last value before step deactivation.

At any point in time, the state of a POU is determined by its active steps and internal and output variables' values. The initial state is represented by the initial value of those variables and a set of the initial steps of the POU—the steps which are initially *active*. Each SFC *network* must have exactly one initial step, but a POU can have many SFC *networks*.

Step and initial step with directed links example:

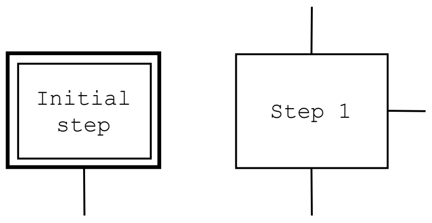


Figure 2.2: SFC step graphical representation

Each *step* can also have associated a number of *actions*.

#### 2.3.5.2 Transitions

*Transitions* control state change, that is, when control from active steps passes to successor steps. It is graphically represented as a horizontal line in the vertical directed links between steps (Fig. 2.3).

A transition has an associated *transition condition* that can be either represented by a ST boolean expression, LD or FBD network or IL instructions. If the evaluation of a *transition condition* yields a `TRUE` value, control flows the successor steps; otherwise, no changes in active steps occur.

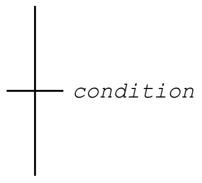


Figure 2.3: SFC transition graphical representation

#### 2.3.5.3 Actions

Steps can have associated *actions* to perform some state-dependent tasks. Actions can take different forms: a single boolean variable, IL instructions, ST statements, LD rungs, FBD networks or even other SFC elements.

Actions must be associated with steps to be performed and are controlled by *action qualifiers*. Steps can have multiple associated actions and different steps can be associated with the same action.

Action declaration can be declared in a graphical form as in Fig. 2.4 and textual form with the construct:

```
ACTION action_name:
    ST statements or IL instructions;
END_ACTION
```

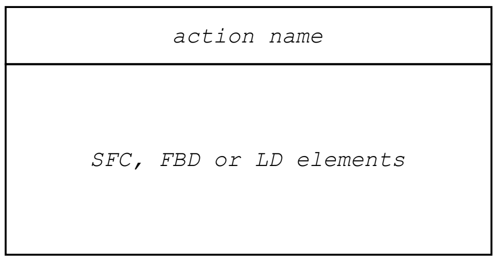


Figure 2.4: SFC action graphical representation

Action associations have *qualifiers* that modify action execution. Several different *action qualifiers* are available, and their behaviour is explained in Table 2.4.

Action association can be done as in Fig. 2.5 or by the construct:

```
STEP step_name:
    action_name(qualifier);
END_STEP
```

Table 2.4: SFC Action qualifiers

| **Qualifier** | **Name** | **Effect** |
|---|---|---|
| None | Null qualifier | same as `N` qualifier |
| `N` | **N**on-stored | executes while associated step is active |
| `R` | Overriding **R**eset | terminates analogous Set execution |
| `S` | **S**et (**S**tored) | executes until analogous `R` qualifier is met |
| `L` | Time **L**imited | ends the execution after a specified time |
| `D` | Time **D**elayed | starts the execution after a specified time |
| `P` | **P**ulse | executes when associated step is activated |
| `SD` | **S**tored and Time **D**elayed | starts Set execution after a specified time |
| `DS` | **D**elayed and **S**tored | starts Set execution if step is active for specified time |
| `SL` | **S**tored and Time **L**imited | starts Set execution and ends it after specified time |
| `P1` | **P**ulse (rising edge) | executes once when associated step is activated |
| `P0` | **P**ulse (falling edge) | executes once when associated step is deactivated |

Action control behaviour is specified in the standard and SFC implementation must be functionally equivalent, albeit the `ACTION_CONTROL` Function Block itself need not be implemented. It is illustrated in Figures 15 -a) and -b) in the standard[1, Section 2.6.4.5] and in Annex A.

**SFC evolution**

The initial state of an SFC network is determined by the initial step, which starts in the active state. All other steps are inactive. A step is activated when all immediately preceding transitions' conditions evaluate to 1. Transition conditions are only evaluated (*enabled*) when all preceding steps are active; transition condition is only then evaluated and if proved `TRUE`, the condition is considered *cleared* and subsequent steps activated. A transition *clearing* deactivates all immediately preceding steps, followed by simultaneous activation of all immediately subsequent steps.

### 2.3.6 Standard inconsistencies

The standard has several inconsistencies between the formal syntax definition in its Annex A and the textual description found in the main body and some semantic ambiguities. Some inconsistencies derive from an incorrect formal grammar definition but are easily understandable and corrected by editing the syntax rules to reflect the intent in the textual description. An example is identifier naming rules: the formal grammar defines them as [1, Annex A]:

```
Letter:       'A'..'Z' | '_';
Digit:        '0'..'9';
Identifier = Letter (Letter | Digit)*;
```

But, from the standard [1, Section 2.1.2]:

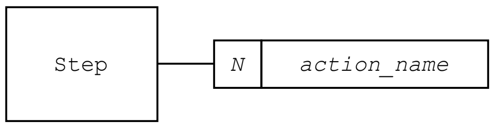


Figure 2.5: SFC action association representation

> "An identifier is a string of letters, digits, and underscores which shall begin with a letter or underscore character. [...] Multiple leading or multiple embedded underlines are not allowed; for example, the character sequences `__LIM_SW5` and `LIM__SW5` are not valid identifiers. Trailing underscores are not allowed; for example, the character sequence `LIM_SW5_` is not a valid identifier."

This definition is not by the formal syntax definition—it allows multiple leading and embedded as well as trailing underscores. This example is a clear bad formal specification and to resolve the inconsistency, it should be updated to:

```
Letter:        'A'..'Z';
Digit:         '0'..'9';
Identifier = (Letter | '_'(Letter|Digit)) ('_' (Letter|Digit))*;
```

Besides syntax inconsistencies, which are not as relevant for this work, the standard also suffers from semantic ambiguities, such as the evaluation order of function parameters, which is not defined. In the case the computation of a function parameter modifies the value of another parameter, its value will be undefined as it depends on the order of parameter evaluation[8]:

```
var1 := foo(bar(in := 33, out => var2), var2);
```

Semantic ambiguities must be addressed, and a specific behaviour must be chosen. Most ambiguities have a clear intended demeanour; in this example, it is correct to either call the `bar` function first, which will modify the variable `var2`, and pass that value as the second argument to function `foo` or store the value of the `var2` variable before its modification by the function and pass the older value to the `foo` function. Intuitively, the first option is the intended behaviour but since the standard does not specify function evaluation order, both are correct.

What matters most in these cases is that behaviour is consistent and well defined and documented.

Many other issues with this standard were already identified in previous work, so it will not be covered here[9, 8].

# Chapter 3

# Compilers

A compiler is a computer program that translates code written in one programming language, the source language, to a target language. Traditionally, the name compiler is used for programs that translate source code from one language to another, lower-level, language (assembly language, object code or machine code) to create a program in a format a computer can understand and execute.

## 3.1 Introduction

Several types of compilers exist: if a compiler is targeting a different architecture than the one it is running on, it is called a cross compiler; a compiler which does the opposite of the traditional compiler definition—that is, translating a lower-level source language to a higher level target language is called a decompiler; a compiler that translates high-level languages into another high-level language is usually called a source-to-source compiler or translator (usually a translator is a compiler that compiles from a high-level language to another high-level, albeit usually slightly lower-level, language—a translator with the same source and target level, such as translating one language to itself with some changes, is called a transpiler). Another interesting use of the term is a compiler-compiler, which is a software tool that generates lexers and/or parsers, some of the parts of a compiler, for a given programming language[10].

There is another type of language processor to transform source programs, an interpreter, which instead of generating an executable program, directly executes the constructions of the source language. Typically, the execution of an interpreter is several orders of magnitude slower than generating and running an optimised executable. Choosing whether a programming language should be interpreted or compiled is often case-specific, weighing the advantages and disadvantages of each option. In a resource-constrained device and one that has real-time requirements, such as PLCs, interpreting the source program instead of executing an optimised binary is unfeasible, so industrial automation software is traditionally compiled to machine code by a compiler.

## 3.2 Compiler structure

Traditionally, compilers are divided into three main stages: analysis, optimisation, and synthesis. The analysis, also called front end, parses source code and checks it for lexical, syntactic and semantic errors and builds a language-specific internal representation of the source code constructs; the optimisation phase, also called middle end, runs over this representation and applies several transformations on it aiming to optimise the end program's running time, resource consumption, *etc*. This phase is usually independent of the source and target languages—it needs only to operate over the internal representation of the source program and does not care what the target language or architecture is; finally, the synthesis or back end generates the target code according to the internal representation for a particular architecture or instruction set. It is responsible for producing correct and optionally optimised code, that takes advantage of the features of the target architecture to ensure the final program will be as performant as possible. This structure is illustrated in Figure 3.1[10].

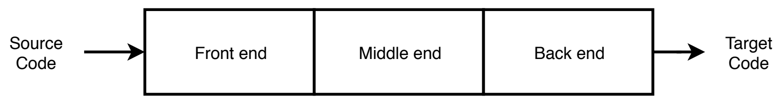


Figure 3.1: Compiler stages

Each of these stages is composed of several parts and compilers implement these in phases, promoting an efficient design and the correct transformation of the source constructs. The whole compiler structure is pictured in Figure 3.2.

### 3.2.1 Lexical Analysis

Lexical analysis (also called scanning, lexing, or tokenization) is the first step of the front end of a compiler and is the process of converting a sequence of characters into a sequence of tokens. The characters from the input source program will be grouped into tokens, or lexemes, which have a meaning within the source language definition[11]. A lexer usually will perform operations such as:

- Remove comments
- Remove redundant whitespace
- Expand macros and pragma directives
- Check indentation in languages with meaningful whitespace (like Python)

The scanner must be concerned with issues such as case sensitivity or insensitivity of the language, if comments in the source program can be nested, whether whitespace and newlines are significant, *etc*.

Lexical errors might arise in this phase, such as characters not allowed in a language, overrunning a word, identifier or line maximum character count, end-of-file within a comment, *etc*.

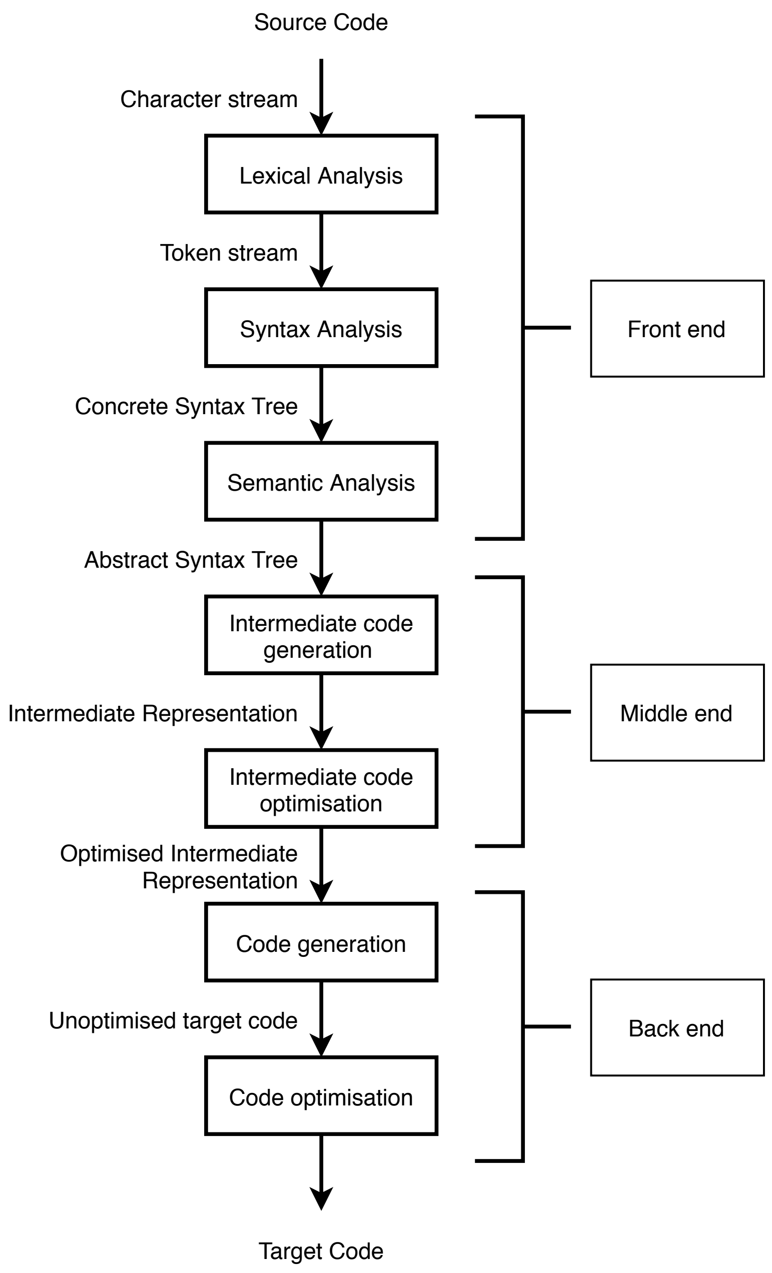


Figure 3.2: Compiler structure

These errors can terminate the execution of the compiler or some error recovery strategies can be employed, such as skipping characters until a valid token is found or adding a missing character to the input string.

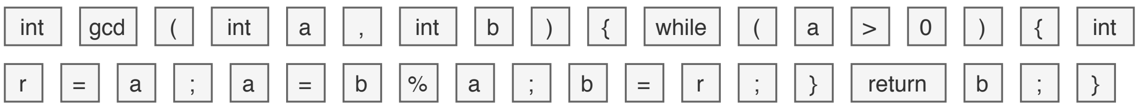
Compilation should still fail to warn the user of incorrect program construction, but ignoring or recovering from errors may be desirable to catch multiple errors at once instead of failing upon encountering the first. This is especially useful when rerunning the lexer is expensive (if the input program is very large) or in integrated development environments, as the lexer is running interactively within a text editor, allowing the user to see multiple errors at the same time and correct them before running the lexer again.

As an example, the lexer for a C programming language implementation would convert this stream of characters:

```
int gcd(int a, int b) {
    while (a > 0) {
        int r = a;
        a = b % a;
        b = r;
    }
    return b;
}
```

Listing 3.1: Euclidian algorithm written in C, adapted from:
https://www.math.wustl.edu/~victor/mfmm/compaa/gcd.c

into this stream of tokens:


int gcd ( int a , int b ) { while ( a > 0 ) { int
r = a ; a = b % a ; b = r ; } return b ; }

It would also identify *gcd*, *a*, *b* and *r* as identifiers and *0* as an integer literal and pass that information to the parser.

### 3.2.2 Syntax Analysis

After lexing, a stream of tokens is converted into a Syntax Tree by a parser, following the input language's syntax rules—that is, its grammar. A Concrete Syntax Tree (CST) is simply a representation of the input program into a data structure that closely resembles the language's grammar rules. It is a one-to-one mapping from the grammar to a tree representation, so reversing this process should result in an almost equal input source code. This is also a good test for testing the correctness of the lexer and parser, as only comments and redundant whitespace should be eliminated from the original input program because they are usually ignored by the lexer and their tokens are never emitted and thus never seen by the parser[2].

Most parsers will often convert the generated CST into an Abstract Syntax Tree (AST): redundant information from the one-to-one mapping will be removed from the Syntax Tree, making it smaller and simplifying operations on it. For example, many languages require a semicolon after an expression. While this is required to be syntactically correct, it adds no meaning to the expression and is useless for the next step in the compilation process: the semantic analysis.

Parsing the C statement

```
return a + 2;
```

according to the American National Standards Institute (ANSI) C specification yields a fairly large CST which has a lot of redundant information that can be omitted as shown in Figure 3.5

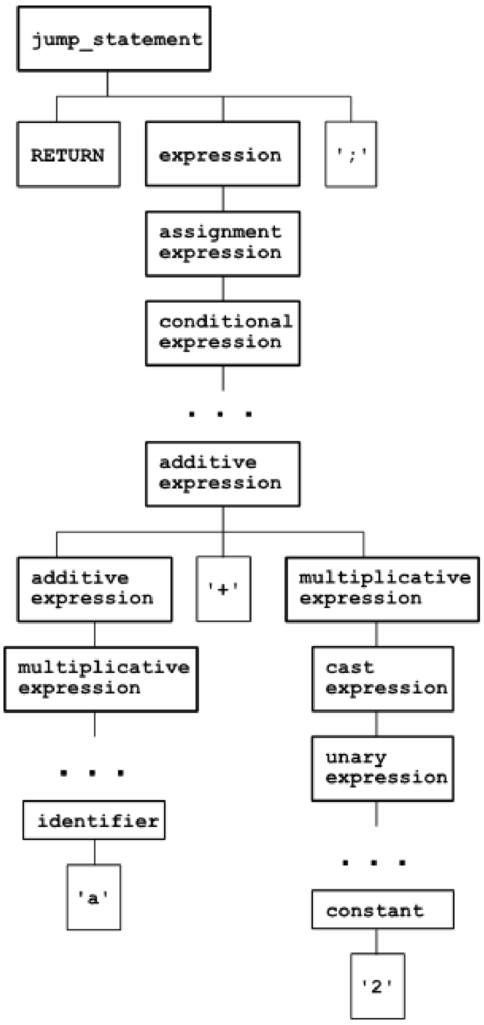


Figure 3.3: Concrete Syntax Tree

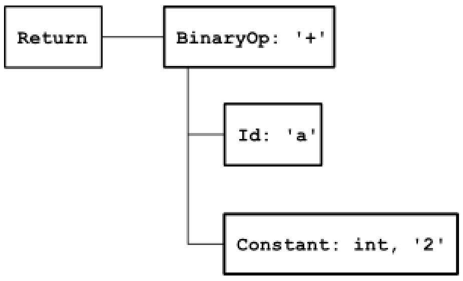


Figure 3.4: Abstract Syntax Tree

CST to AST example transformation [2])

### 3.2.3 Semantic Analysis

During the semantic analysis, several rules of the language are checked, such as type checking, checking whether a variable is used before its declaration, multiple declarations of the same variable, violations of access rules (public, private and protected members, *etc.*), number of arguments passed to functions, *etc*. This type of rule is impossible or very difficult to represent in grammar rules to be caught by the parser and some require context to validate the input so a semantic analysis phase is necessary to ensure program correctness[11].

For example, in Pascal, the **and** operator expects boolean operands but has higher precedence over relational operators, so this example is a semantic error:

x > y **and** x + 5 < 30

Since the **and** expression has higher precedence, it should be evaluated first, becoming

```
x > (y and x) + 5 < 30
```

which might be an error since $x$ and $y$ might not be boolean operands.

To resolve this error, brackets must be applied over the operands as such to produce the intended result:

```
(x > y) and (x + 5 < 30)
```

### 3.2.4 Intermediate Code Generation

Intermediate Code Generation is the first step of the optimiser, or middle-end, of a compiler. Intermediate Representation (IR) is a representation of a program in a data structure and should be independent of the source and target language. Its purpose is to enable optimisations to be done in a source and target language-agnostic way. Using an independent intermediate representation allows us to support different front and back ends within the same compiler and to perform target-independent optimisations in the next phase as shown in Figure 3.6.

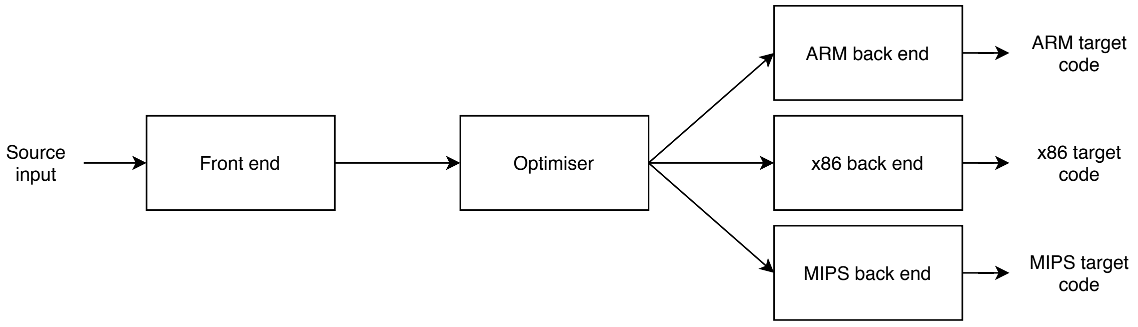


Figure 3.6: Multiple target compiler structure

### 3.2.5 Intermediate Code Optimisation

Intermediate Code Optimisation is the last step of the middle-end of a compiler, where target-independent optimisations are performed to the intermediate representation of the source program. Optimising is a generic term for lowering resource usage by the program, such as time (run-time of the program or compilation time), space (run-time memory usage or size of the executable program—which must be stored in some kind of memory in the device) or more specific computation resources such as file handles, threads, caches, page swaps, *etc*[11].

Usually, optimising one of these will diminish performance in another field. For example, optimising for program size will often reduce program speed or increase run-time memory usage. General or ideal optimisation is not possible or even decidable, it's a trade-off between using more of a kind of resource rather than another, so compilers commonly allow you to prefer optimising for either:

- Program size
- Program speed
- Memory usage

Since optimisation is undecidable, the more aggressive the optimisation performed on the program, the longer the time the compiler will need to run. It's usually done in several passes, with each focusing on a specific improvement. The optimised program must achieve the same result to remain correct and indeed reduce the chosen resource usage.

Some common target-independent optimisations done by compilers are:

- Constant folding
- Algebraic simplification
- Unreachable and dead code elimination
- Loop unrolling
- Tail recursion elimination
- Inline expansion
- Common subexpression caching or elimination
- Replacing expensive operations with equivalent, cheaper, ones

Constant folding and algebraic simplification are simple and common optimisations evaluating expressions at compile-time instead of doing the work at run-time:

```
int x = 12;
int y = 9 - x / 2;
return y * (36 / x + 2);
```

This will be evaluated into:

```
int x = 12;
int y = 3;
return 3 * (36 / 12 + 2);
```

Which can be rewritten as

```
int x = 12;
int y = 3;
return 15;
```

Further optimisation might remove dead code—variables that are initialised but never used, as long as their assignment does not cause side-effects, yielding:

```
return 15;
```

### 3.2.6 Code Generation

The back-end of a compiler is responsible for generating code for the specified target. Since each target usually has its own instruction set and different characteristics, this stage is completely dependent on the target. The generated code must achieve the same purpose of the intermediate representation but correctly using the available instructions for the target.

Compiling the program in the Listing 3.1 with a C compiler (GNU Compiler Collection (GCC)), we obtain the following generated code for the x86_64 architecture, in AT&T syntax, without optimisations:

```
gcd:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    %edi, -20(%rbp)
        movl    %esi, -24(%rbp)
        jmp     .L2
.L3:
        movl    -20(%rbp), %eax
        movl    %eax, -4(%rbp)
        movl    -24(%rbp), %eax
        cltd
        idivl   -20(%rbp)
        movl    %edx, -20(%rbp)
        movl    -4(%rbp), %eax
        movl    %eax, -24(%rbp)
.L2:
        cmpl    $0, -20(%rbp)
        jg      .L3
        movl    -24(%rbp), %eax
        popq    %rbp
        ret
```

Listing 3.2: Unoptimised generated x86_64 code

### 3.2.7 Code Optimisation

The final phase of the compilation is the machine-dependent optimisation of the final program. This requires a deep understanding of the construction of the target architecture to take advantage of all of its capabilities.

Common machine-dependent optimisations are:

- Smart register allocation
- Use of address modes

- Balancing across multiple CPUs
- Instruction selection, considering caches, pipelines, branch prediction, scheduling, alignment, *etc*.

Compiling again the program in Listing 3.1, with optimisations enabled, we get the following optimised code:

```
gcd:
        movl    %edi, %r8d
        movl    %esi, %eax
        testl   %edi, %edi
        jg      .L3
        jmp     .L7
        .p2align 4,,10
        .p2align 3
.L5:
        movl    %edx, %r8d
.L3:
        cltd
        idivl   %r8d
        movl    %r8d, %eax
        testl   %edx, %edx
        jg      .L5
        movl    %r8d, %eax
        ret
        .p2align 4,,10
        .p2align 3
.L7:
        movl    %esi, %r8d
        movl    %r8d, %eax
        ret
```

Listing 3.3: Optimised x86_64 generated code

We can see that the number of instructions in the optimised program is larger than the unoptimised program in Listing 3.2, but the instructions used are different to optimise for execution speed. Speed comparison between the two can be made, with the optimised version being around 15% faster than the original.

## 3.3 Compiler Design, Tools and Generators

To aid compiler development, several tools exist. Lexer and parser generators are available, that given a language's grammar formal definition will produce source code that when compiled performs lexing and parsing for that specific language. These parsers will usually create the AST from the input file and the rest of the compiler will then be able to operate on the AST to continue compilation. Many different such tools are available, perhaps the most used being *flex* (lexer generator) and *GNU bison* or *ANTLR* (parser generators).

An important characteristic of a compiler is the ability to generate meaningful error messages. It immensely helps developers in finding bugs, malformed expressions and invalid syntax, incorrect semantics, *etc.*. A major downside of using parser generators is their poor generated error messages. Since they are very generic, syntax errors are usually hard to understand only based on the error message. The message can be rather cryptic sometimes for a simple statement terminator missing such as a ';' and the received error might appear completely unrelated and the indicated supposedly offending line can be even several lines down the actual source of the error.

For this reason and because generated parsers code is hard to understand and debug, most production compilers use hand-written parsers. Although being initially harder to implement compared to a simple tool that generates the parser, in the long run, it is often easier to maintain and modify conforming to the evolving language needs. Comprehensive and helpful error messages are also a major benefit for developers. The GCC compiler used a parser generator prior to version 5.0[1] for the C and Objective-C languages but has since moved to a hand-written recursive descent parser for the presented reasons.

[1]`http://gcc.gnu.org/wiki/New_C_Parser`

# Chapter 4

# The LLVM Compiler Infrastructure Project

The LLVM compiler infrastructure is an open-source project started in 2000 at the University of Illinois by Vikram Adve and Chris Lattner. Its name originally stood for Low Level Virtual Machine (LLVM) but that meaning has been dropped as it is unrelated to what is traditionally considered a Virtual Machine and confused developers. As of version 9.0.0, it's licensed under the permissive Apache License 2.0[12].

## 4.1 Introduction

It consists of a collection of reusable modular and toolchain technologies, originally designed for the C and C++ programming languages for compile-time, link-time, and run-time optimisations. Because of its language-agnostic intermediate representation, which resembles high-level assembly, it can be used to develop compiler front-ends for any language and back-ends for any architecture. This encouraged the development of LLVM-based front-ends for most of the popular programming languages in use, such as Ada, C#, Common Lisp, CUDA, D, Fortran, Haskell, Java bytecode, Julia, Kotlin, Lua, Objective-C, OpenGL, Ruby, Rust, Scala, Swift, and many other less known languages.

The most prominent projects under LLVM are:

|  |  |
|---|---|
| **LLVM Core:** | Source and target-independent optimiser with code generation support for many CPU architectures |
| **Clang:** | C/C++/Objective-C compiler front-end |
| **LLDB:** | High-efficiency native debugger |
| **LLD:** | Fast linker with support for link-time optimisations |
| **OpenMP:** | Support for OpenMP, a multithreading framework |
| **libc++:** | An implementation of the C++ standard library (a libc implementation is also planned but not yet started) |

**compiler-rt:** Run-time library with implementation of low-level routines for targets that do not have core IR instruction implementations

**polly:** A high-level loop and data-locality optimiser based on integer polyhedra to analyse and optimise memory access patterns of a program

**libclc:** An implementation of the OpenCL standard library, a framework for writing programs that execute across CPUs, GPUs, FPGAs and other hardware accelerators

**klee:** An Execution Engine that implements a symbolic virtual machine to evaluate dynamic paths across a program to prove properties of functions and find bugs, generating test cases for them

## 4.2 Structure

The LLVM Core subproject design eases the creation of compilers for any programming language or adding support for any architecture because of its source- and target-independent optimiser[13]:

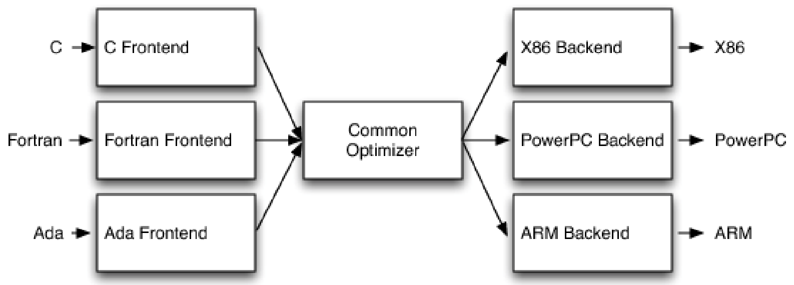


Figure 4.1: LLVM-based compiler structure (from [3])

Decoupling a compiler IR from both source and target languages has major benefits: compiler front- and back-ends are completely independent and can be reused and exchanged. Writing a compiler back-end for an arbitrary architecture does not depend on the target language. As far as the code generator is concerned, the source-language is LLVM IR; likewise, for a front-end, the perceived target-language is also LLVM IR.

Thus, to create a LLVM compiler for any language, such as the IEC 61131-3 languages, only a front-end (syntactic and semantic analyser) needs to be implemented, targeting LLVM IR. And to add support for any architecture to LLVM, only a back-end must be implemented for that particular architecture, generating assembly for the intended Instruction Set Architecture (ISA) from the LLVM IR itself.

Porting a language to a new architecture requires only that the corresponding LLVM back-end be implemented; this will also port every language that generates LLVM IR to that architecture as well. And creating a compiler for a new programming language only requires a front-end to be implemented. By design, the new language will be able to target and run in any architecture already supported by LLVM. The benefits are very obvious and it is not by chance that, in recent years, many new languages were created using LLVM, the most well-known being Rust, Swift (which also has Chris Lattner as one of its creators), Haskell and Kotlin. Another interesting and very recent application of LLVM is Emscripten: an LLVM back-end, compiling IR to WebAssembly (*wasm*) or a subset of JavaScript (*asm.js*), enabling any language that has an LLVM front-end to be run on a web browser.

## 4.3 Use and Tools

Other tools that are part of the LLVM project are of use in the creation of a compiler, such as LLD, a very fast linker that can substitute the more commonly used GNU linker, *ld*, having the benefit of always being a cross-linker that supports all of the common architectures without having to build a cross-linker from its source. It can also be embedded in a compiler or program thus reducing dependencies and by default enables Link-Time Optimisation (LTO), greatly improving the performance of compiled programs.

As an example, compiling the program in Listing 3.1 with link-time optimisations enabled will increase the performance of the program. In fact, the linker can see that the code has no side-effects and does not use the calculated values, so the computation is removed altogether, making the program basically empty. A compiler cannot perform this optimisation as it cannot inspect the whole program at once, only its isolated components, and thus must not make assumptions about the use of the computed values. This is an extreme example, and, on a real program, performance improvement will vary and is likely not as noticeable, but the possible optimisations at link-time can greatly improve performance. These include removal of unused symbols like functions or global variables, reducing program size; and inlining of short but frequently used functions, removing the overhead of a function call.

The LLDB debugger can also be used as a remote debugger: traditional debuggers require the debugger to be run on the target platform which might be unfeasible in a resource-constrained environment or headless devices such as PLCs, in which you do not have a computer screen to inspect the program state and flow inside a debugger. But, with remote debugging, it is possible to debug a program running in another machine (*e.g.*, a PLC) in a more capable computer.

LLVM tools are maturing and having been created to work together makes integration between the different tools seamless. Some systems are starting to move from traditional GNU tools to LLVM's. This includes Clang, a C/C++ compiler, LLVM's *libc*++ (and eventually the *libc* too), the LLD linker, the LLDB debugger and some others. Major examples are Apple Inc.[1], the FreeBSD[2] and OpenBSD[3] projects, which changed their system compiler, *libc*++, and debugger to LLVM's; the Sony Corporation also changed their PS4™ Software Development Kit (SDK) to use Clang/LLVM, claiming massively improved compilation and linking times[4]. Android operating system userspace is also entirely built with Clang[5] and even some phones are also being

[1]https://help.apple.com/xcode/mac/current/#/itcaec37c2a6
[2]https://wiki.freebsd.org/PortsAndClang
[3]https://man.openbsd.org/clang.1
[4]https://llvm.org/devmtg/2013-11/slides/Robinson-PS4Toolchain.pdf
[5]https://lwn.net/Articles/734071/

shipped with a Clang-built Linux kernel, such as the Google Pixel™ 2[6]. LLVM has been steadily growing and improving over the years and already is widely used in production. It is very actively developed and trusted by many corporations and projects and has been proven capable. Most companies using it will contribute back changes, bug fixes, and improvements.

LLVM is written in C++ and also exposes a C++ Application Programming Interface (API) for LLVM IR code generation. To use it, LLVM must be built from source, by downloading it from its git repository at `https://github.com/llvm/llvm-project` and using CMake for configuration and as a building system. CMake supports several generators, such as Ninja, Unix Makefiles, Visual Studio, and Xcode, so it can be used in virtually any system. Debug builds will take a lot of space, but the extra debug information might be useful for compiler debugging purposes (a regular LLVM build will take between 1 and 3 GiB of disk space, while a debug build will need 15-30GiB). Usage of the LLVM API is fairly documented, with user guides and an automatically generated Doxygen reference[14].

Several other API bindings are officially maintained, such as a C API, with a stable Application Binary Interface (ABI)—an important characteristic to guarantee programs written for a specific library version will continue to work with future versions—and a Go, OCaml and Python API. Unofficial bindings for many other common programming languages are also available (Rust, C#, Haskell, *etc.*), but the most complete API is the C++'s.

From the official documentation, as an example, to generate LLVM IR for the simple function returning a multiplication and addition of its arguments in Listing 4.1, several steps are needed:

```cpp
int mul_add(int x, int y, int z) {
    return x * y + z;
}
```

Listing 4.1: Simple C++ function [15]

The LLVM IR equivalent of this function is:

```llvm
define i32 @mul_add(i32 %x, i32 %y, i32 %z) {
entry:
  %tmp = mul i32 %x, %y
  %tmp2 = add i32 %tmp, %z
  ret i32 %tmp2
}
```

Listing 4.2: Equivalent LLVM IR [15]

[6] `https://lkml.org/lkml/2017/11/22/943`

To generate this IR, the API provided functions should be used. The basic building block of LLVM is the *Module*, which is a unit of code, defining global variables, function declarations, and implementations.

First, a new *Module* object should be created:

```
Module *mod = new Module(...);
```

Then, a function representation is created, specifying parameter and return types:

```
Constant *c = mod->getOrInsertFunction(...,
                IntegerType::get(32), // return type
                IntegerType::get(32), // parameter types
                IntegerType::get(32),
                IntegerType::get(32), // end of parameters
                ...);
```

Setting the function’s calling convention:

```
    Function *mul_add = cast<Function>(c);
    mul_add->setCallingConv(CallingConv::C);
```

Create parameter objects and give them names:

```
    Function::arg_iterator args = mul_add->arg_begin();
    Value *x = args++; x->setName("x");
    Value *y = args++; y->setName("y");
    Value *z = args++; z->setName("z");
```

Next, a *BasicBlock* is created, which is simply a container for instructions to execute sequentially

```
    BasicBlock *block = BasicBlock::Create(...);
    IRBuilder<> builder(block);
```

Now, the actual operations can be added to the *BasicBlock*:

```
    Value *tmp = builder.CreateBinOp(Instruction::Mul, x, y,...);
    Value *tmp2 = builder.CreateBinOp(Instruction::Add, tmp, z,...);
```

And finally, the return value of the function:

```
    builder.CreateRet(tmp2);
```

With the function generation done, the only step remaining is to run a *PassManager* and it will generate the intermediate representation of the instructions defined:

```
verifyModule(*mod, ...);

PassManager PM;
PM.run(*mod);
```

A full code listing can be found in Annex B.

Despite a bit daunting at first, the usage of this API is straightforward. Compiling and running the code will generate the IR represented in Listing 4.2. To generate assembly language for a target architecture from LLVM IR, the tool *llc* can be used and then using a native assembler such as GNU *as* to generate object code; *llc* can directly generate machine code with the flag *-filetype=obj* but usually a native assembler generates better assembly for a specific architecture. Creating an executable is then the job of a linker, either LLDB or the traditional GNU *ld*.

LLVM optimisation is done in *passes*. Each *pass* will traverse the program, either analysing the program to obtain information about it or transforming it for optimisation. Each pass will usually focus on a specific optimisation or analysis and depending on the defined optimisation level, different passes can be performed. LLVM already has more than a hundred different passes implemented, and it is possible to write a custom pass with specific intent. Passes allow static analysis of a program and memory, address, and undefined behaviour sanitizers, as an example, which is very useful in debugging a program and ensuring its correctness and standard-compliance. They operate on LLVM IR, so they can also be applied independently of source and target languages.

LLVM's IR is a low-level representation intended to be operated on from every compilation stage. This is needed so it can correctly represent virtually any language's constructs. It is strongly typed and one of its primary features is Single Static Assignment (SSA), which provides type safety. This means that any variable is assigned to exactly once. As a very simple example:

```
x = 1;
x = 5;
y = x - 2;
```

Converted to an SSA form, becomes:

```
x1 = 1;
x2 = 5;
y = x2 - 2;
```

For a human, it is very easy to notice the first statement has no effect, but, for a compiler, this is harder to prove as it requires analysis. This very simple example makes it obvious how this form greatly simplifies analysis and optimisations.

The IR has three representations; in-memory IR, on-disk *bitcode*, and a human-readable assembly-like language, all of them equivalent. The first representation is what using the LLVM API generates, which eventually can be converted to textual IR or *bitcode*.

The reference for LLVM's IR is available on `https://llvm.org/docs/LangRef.html`. It has many different *opcodes* and functionalities, such as: declaring global symbols (functions and variables) with `declare`; return and branching instructions (`ret`, `br`, `invoke`, `callbr`, *etc.*); unary and binary operators (`fneg`, `add`, `sub`, `mul`, `fdiv`, `urem`, *etc.*); logical operators (`and`, `or`, `xor`, `shl`, *etc.*); memory accessing instructions (`alloca`, `load`, `store`, *etc.*); it also has type conversion, truncation, and many other types of instructions which are not as relevant for this project. The official documentation and language reference is very extensive and clear.

# Chapter 5

# MatIEC

MatIEC is an open-source compiler for the IEC 61131-3 languages. The project was started by professor Mário Jorge de Sousa originally for the MatPLC project[1], now obsolete, aiming to create an open-source PLC [16]. It was later integrated into the Beremiz project[2], an open-source IDE for machine automation and conforming to the IEC 61131-3 standard. It includes a graphical editor for SFC, LD and FBD[17].

Another project for an open-source—software and hardware—PLC (analogous to the now-defunct MatPLC) is openPLC[3]. It also uses the MatIEC compiler behind the scenes and the Beremiz core editor, providing a *runtime* for several popular embedded platforms, such as the Raspberry Pi board and Raspberry Pi-based boards and several FreeWave boards; provides also a soft PLC *runtimes* for Linux and Windows machines; it also supports popular slave devices such as Arduino (and Arduino-compatible) and ESP8266 boards.

## 5.1 Introduction

More accurately, MatIEC is a source-to-source compiler or translator/transpiler into ANSI C and is written in C++. It is licensed under the GNU's Not Unix (GNU) General Public License (GPL) v3. The source code can be found in its *mercurial* repository, `https://bitbucket.org/mjsousa/matiec/src/default/`.

It complies with the 2003 version of the IEC 61131-3 (second edition) and supports compilation of textual IEC 61131-3 languages (ST, SFC and IL).

For use of the MatIEC compiler with Beremiz, openPLC and other proprietary IDEs, a graphical SFC programs conversion into its textual counterpart and FBD and LD programs into a textual language supported by MatIEC must be made, either to IL or ST, which will then be compiled into ANSI C code.

[1]`http://mat.sourceforge.net/`

[2]`https://beremiz.org`

[3]`https://openplcproject.com/` — not to be confused with the PLCOpen organisation which creates specifications on top of the IEC 61131-3 standard for industrial automation efficiency

## 5.2 Structure

The compiler is organised in four stages: lexical analysis, syntax analysis, semantic analysis, and code generation. Two target code generators are available, translating into ANSI C or IEC 61131-3 code, producing two different compilers, respectively *iec2c* and *iec2iec*, each with the respective target.

The generated C code must then be compiled to machine code by any standards-compliant C compiler (GCC, clang, Microsoft's MSVC, Intel's icc, *etc.*) to be executed by a PLC or soft-PLC (a software-emulated PLC).

The *iec2iec* compiler generates IEC 61131-3 code very similar to the input; it has a compiler front-end debugging intent and is not meant to be used in production. The generated code helps to debug the lexical and syntactic analysis stages of MatIEC.

The lexical and syntax analysis stages are implemented resorting to the lexer and parser generators *flex* and *bison*, from an adapted syntax definition from the IEC 61131-3 standard. From the input language's grammar definition these tools tokenize the input and construct an AST as specified. This approach to lexing and parsing sometimes generate not-as-friendly and somewhat cryptic syntax error messages; having a custom—hand-written—parser would give more control and extensibility to error handling.

The AST has been implemented as a collection of C++ objects and classes and compiler stages operating on it follow the visitor design pattern. This enables easy addition and removal of stages, like an optimiser stage, or new code generators for any target output language without having to edit the AST classes themselves.

The visitor design pattern is an object-oriented software engineering concept, in which an object structure is separated from an algorithm that operates on it. This allows adding new operations to objects without modifying their structure. This means that instead of having classes implement every possible action on themselves and cluttering the codebase, they implement a *dispatch* function that will delegate the work to a visitor which implements that action on all visitable classes. This makes the addition of visitors (*i.e.*, actions) as simple as implementing that action for the applicable classes instead of needing to implement a function in every class performing it. The downside of this approach is that adding a new visitable class will require modifying every visitor that acts on it, but the action would need to be implemented nonetheless with the same amount of work with other design patterns, so the benefits largely outweigh the disadvantages.

## 5.3 Related work

As of the writing of this document, MatIEC only supports the second edition of the IEC 61131-3 standard, from 2003. Previous work has been made to add support for the newer version (from 2013) by Bruno Silva in 2016 as his MSc's dissertation[18]. It consisted of a new front-end for the compiler, making use of the tool Coco/R, a simultaneous lexer and parser generator instead of the currently used *flex/bison*. However, this is not yet ready for a merge to the main compiler source code.

The work described in this document builds upon the existing compiler structure, based on the earlier version and thus does not support the most recent version. Since the compiler is modular, future work might add support for the newer edition. Since front- and back-end are decoupled, no problems should arise from such a venture, but newer constructs of the language must be implemented in the back-end as well for them to be supported. An example is the object-oriented constructs such as classes, defined for the third edition of the standard but absent in the previous. LLVM IR generation for such constructs must then be implemented; this should not require major changes, if any, to the already implemented work.

There are a few other free IEC 61131-3 IDEs available, such as CODESYS[4] and GEB[5], with integrated compilers but despite being free to download and use, they are not open-source.

MatIEC is then the only open-source IEC 61131-3 compiler[6] ready for use.

Other work to execute IEC 61131-3 programs has been made: compiling the languages to a *bytecode* to be executed in a Virtual Machine[19, 20, 21, 22]; an effort to run these programs in Microsoft's .NET framework's Common Language Runtime (CLR)[23, 24]; and even CPU cores designed to be implemented in Field Programmable Gate Arrays (FPGAs) with IL instructions as their ISA[25, 26]. All these solutions are of a higher level of abstraction and more complex, requiring powerful hardware and more resources to implement IEC 61131-3 languages.

[4]`www.codesys.com`

[5]`www.gebautomation.com`

[6]technically it is a source-to-source compiler or translator but the term compiler is also acceptable and will be henceforth used to refer to MatIEC

# Chapter 6

# Architecture and Implementation

Direct binary code generation has been a long-term goal for MatIEC. It can conceivably be achieved in three ways: writing a code generator for each particular architecture and system we wish to support; generating an already existing compiler's IR and let its compiler back-ends generate object code; or creating an AST with the same structure as a working compiler and feed it to that particular compiler to continue compilation and code generation.

Writing as many compiler back-ends as architecture support is desired requires a lot of work and the generated code would probably be less optimal than it should. Besides the back-ends, an intermediate code optimiser would also have to be implemented since MatIEC does not have one and resorts to a C compiler for optimisations. This makes it clear writing compiler back-ends and optimisers is not the best procedure, despite effectively eliminating MatIEC external dependencies.

The best method is then shifting code generation responsibility to a pre-existing and proven compiler, requiring either its IR generation or a transformation of MatIEC's AST into another with a different structure.

Integrating MatIEC into the GNU Compiler Collection (GCC) has been previously thought of. GCC's compiler back-ends can be used, if either an AST or IR for GCC is generated by MatIEC, but due to the structure of GCC both are daunting tasks. GCC has three different kinds of intermediate representation, none of which are well documented, making such a project unnecessarily difficult. The first generated IR by GCC is called *GENERIC*, a mostly language-independent representation. It is constructed from the AST and tries to mimic a program structure in this IR. It will then be converted into the second IR, *GIMPLE*. GIMPLE is where most optimisations will be performed, a simpler and more restrictive representation than an AST thus requiring another structure to represent program flow, the *Control Flow Graph (CFG)*. Finally, GIMPLE is converted into the Register Transfer Language (RTL) representation which does platform-dependent optimisations, such as register allocations, instruction scheduling, and many other architecture-specific optimisations. To achieve binary code generation with MatIEC, the GCC IR we would need to target is GIMPLE (technically a subset of GENERIC). All GCC front-ends for different languages generate GIMPLE. GIMPLE is not very well documented, and it even has three different representations: *High-level GIMPLE*, which is obtained directly in the conversion from GENERIC; *Low-level GIMPLE* is obtained after "linearizing all the high-level control flow structures of high-level GIMPLE, including nested functions, exception handling, and loops"[1]; and *SSA GIMPLE*, which is what rewriting Low-level GIMPLE into an SSA form generates. This makes obvious the fact that generating binary code resorting to GCC is difficult and limited.

Targeting a compiler's AST (either GCC, clang, or other), also reveals to be a bad solution. As an AST is tightly coupled with the language's syntax and constructs, it is only a viable option if that particular compiler's source language semantics are a superset or very identical to IEC 61131-3 languages, which is simply not the case. The Pascal programming language's syntax is close to ST, but it still has enough different constructs and semantics that targeting a Pascal compiler's AST would be a hard task; since the other IEC 61131-3 languages have a completely different syntax it would only work for ST. Translation to another compiler's AST is also even more restrictive than generating GCC's IR, making it a poorer solution.

Targeting a compiler's IR can be a good solution *if* that IR is not coupled to the source and target languages. The latter is the case of LLVM and the single motivation for the creation of the whole compiler infrastructure project. LLVM's IR is also very well documented and can be generated with the use of LLVM's API instead of directly managing textual files with IR instructions. These characteristics make the use of LLVM the best solution for binary code generation for MatIEC.

This project then proposes to create a new LLVM-based target for the MatIEC compiler, *iec2ll*, for use along with the two existing targets. The new target shall generate LLVM IR code resorting to the LLVM API. A compiler structure diagram showing how the new target should be incorporated and how LLVM is integrated into MatIEC is presented in Figure 6.1. The highlighted block is the new target, referring to this project. The figure also presents how the different projects connect to form a whole and multi-target compiler.

Many transformations and optimisations on generated IR are available, and even analysis; but since IR is so low-level, it is not suitable for correct and complete semantic analysis; for this many languages generate first a higher level IR, in which most analysis is performed: Swift's SIL (Swift Intermediate Language) and Rust's HIR (High-level Intermediate Representation), are example languages taking this approach. MatIEC does this analysis in stage 3 on the AST and due to the relatively low complexity of the languages, there is no need for a higher-level IR for more in-depth semantic analysis.

## 6.1 Methodology and implementation

The internal structure of MatIEC is broken down in 4 stages: lexical analysis, syntax analysis, semantic analysis, and code generation. Stages 1 and 2 occur simultaneously since IEC 61131-3

[1] `https://gcc.gnu.org/wiki/GIMPLE`

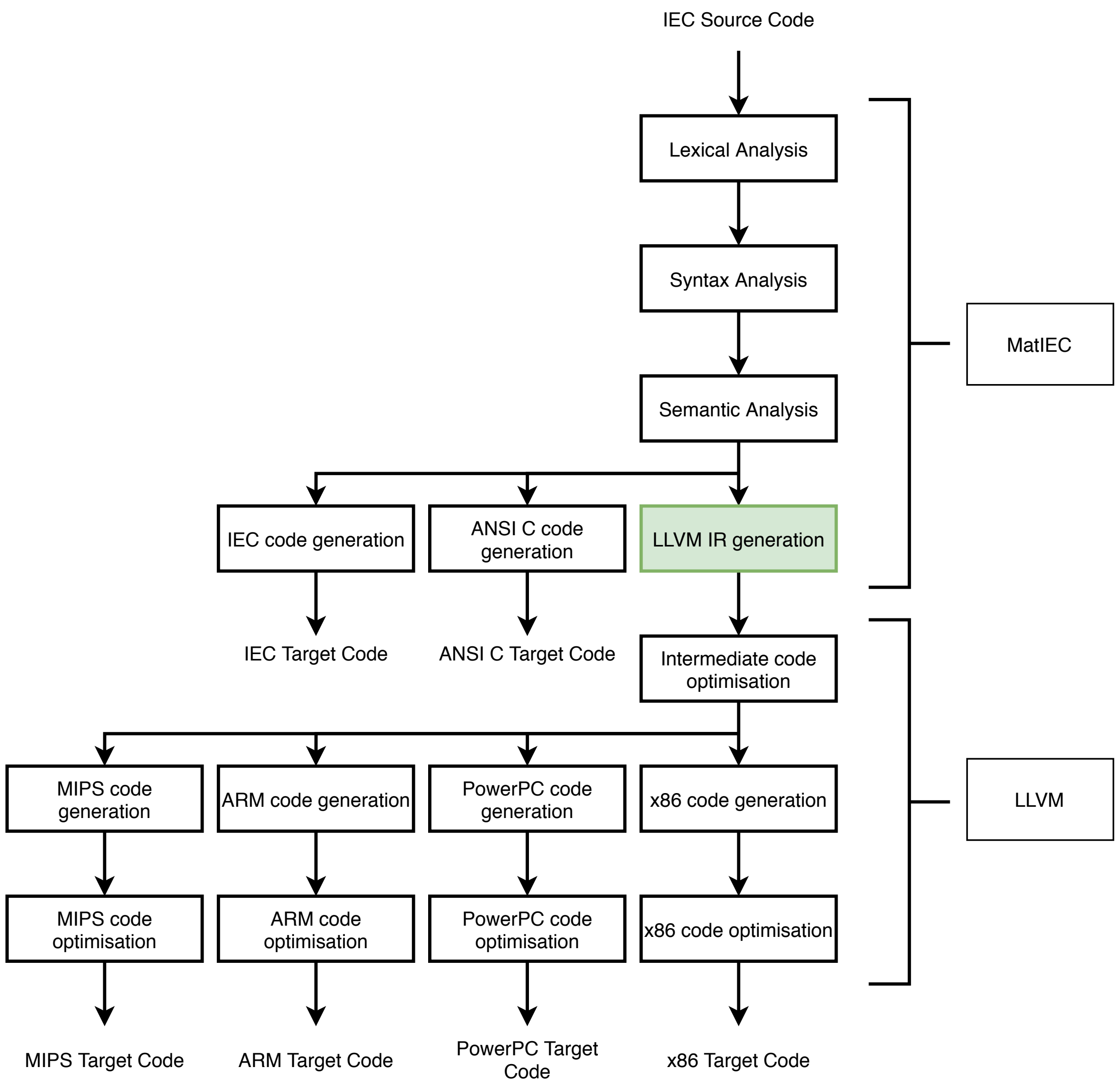


Figure 6.1: LLVM integration into MatIEC

languages are *context-sensitive* and thus requiring *backtracking* for *context-free* parsing[2]. This information passing between stages is only possible if lexing and parsing are done simultaneously and are then implemented as such.

The source-code repository separates code for each stage in different folders in the top-level directory (stage1_2, stage3, and stage4). In the stage4 directory, each different code generator files go into its folder (generate_c, generate_iec, and now generate_ll).

The new code generator will then be needed to be created in the generate_ll folder, inside stage4. The code generator should traverse the AST from stage3 and generate code.

The AST is implemented with several classes, each inheriting from a *symbol_c* base class, which contains information from the parsing stage such as file, line and column for both the start and end of the found symbol, the datatype of a symbol if applicable and a few other useful fields. Class names and which classes should be implemented are taken from the formal syntax definition in the standard. Since MatIEC is implemented with the visitor pattern, all classes define an *accept* function, taking a *visitor* reference as argument. All implemented visitors must define a *visit* function for each type of class it operates on.

*generate_c*, *generate_iec*, and now *generate_ll*, are all implemented as *visitors*. In the beginning of *stage4*, a new visitor is instantiated: either *generate_c*, *generate_iec* or *generate_ll* depending on which compiler is being run; respectively, *iec2c*, *iec2iec*, or *iec2ll*. Then, the visitor only needs to traverse the symbol tree and call the *accept* function of each symbol to execute the correct function. This is a clever use of C++ function overloading capabilities to provide an indirection layer for choosing which code generator should be run and, depending on the current *symbol* being analysed, which function for the code generation itself should be chosen.

In the beginning of the fourth stage for the *iec2ll* the visitor *generate_ll* is instantiated and in the symbol tree root, the accept function is called as:

```
tree_root->accept(*code_generator);
```

Because of its design and despite being a big project, writing a new code generator for MatIEC is straightforward.

Each syntax element in the symbol tree is implemented as a specific class, derived from the base class *symbol_c*, and may have multiple children. The class declaration is not done manually but by very clever macro definition mechanisms, inspired by the GCC codebase. Since each symbol might have several children (references), in the *absyntax/absyntax.def* file, each symbol is declared with the macro function `SYM_REFx()`, in which *x* represents the number of references a particular symbol has. As an example, the *program_declaration_c* symbol (representing a program declaration) is defined as:

```
SYM_REF3(program_declaration_c, program_type_name, var_declarations,
  function_block_body, ...)
```

[2]Formal grammar types and syntax analysis are outside the scope of this work. See the previous dissertation by Bruno Silva on a new front-end for the MatIEC compiler[18]

`SYM_REFx` is an undefined macro and when including the file *absyntax.h*, those macros are first defined. The one presented is as follows:

```
#define SYM_REF3(class_name_c, ref1, ref2, ref3, ...) \
class class_name_c: public symbol_c { \
  public: \
    symbol_c *ref1; \
    symbol_c *ref2; \
    symbol_c *ref3; \
    __VA_ARGS__ \
  public: \
    class_name_c(symbol_c *ref1,\
        symbol_c *ref2,\
        symbol_c *ref3,\
            int fl = 0, int fc = 0, const char *ffile = NULL,
              long int forder=0,\
            int ll = 0, int lc = 0, const char *lfile = NULL,
              long int lorder=0); \
  virtual void *accept(visitor_c &visitor); \
  /* WARNING: only use this method for debugging purposes!! */ \
  virtual const char *absyntax_cname(void) {return #class_name_c;}; \
};
```

After defining macros with the same structure for symbols with a different number of references (from 0 up to 6), the inclusion of the *absyntax.def* file will automatically generate class definitions for every symbol in that file, in this structure, without having to manually declare them.

These macros are used in more than one place with different definitions. Another example is to create the header file for all the *visit* function declarations for the different classes. Because of this design, it is only needed to define the `SYM_REFx` macros as:

```
#define SYM_REF3(class_name_c, ref1, ref2, ref3, ...) \
    virtual void *visit(class_name_c *symbol);
```

Along with the equivalent definitions for the other numbers of symbol references, it is only needed to include again the *absyntax.def* file and any visitor's header file can be declared without manually declaring everything.

Another advantage is that since classes are automatically generated, if a change needs to be done to every class, it must only be done once, in the function macro definition.

To begin the LLVM IR code generation implementation, in the new visitor *generate_ll*, every *visit* function definition for every relevant symbol must then be written. In the program declaration example, this function is:

```
void *visit(program_declaration_c *symbol) {
    ...
}
```

An initial skeleton of such an LLVM IR generator for MatIEC has already been started by Professor Mário. It was started in 2012 but development did not advance much. MatIEC has seen since a lot of improvement relative to the older version where the new *generate_ll* target was started; many of the changes are breaking changes, making the compilation of the LLVM target skeleton for the older version impossible to compile on the newest versions. It also used LLVM release 3.1 (May 22$^{nd}$, 2012), which is very outdated.

Work on the new target started by converting the use of the LLVM API compatible with the newest version, 11.0.0. Many changes were needed, as the LLVM project in 2012 was very recent and the API was very unstable, changing its use with every release.

## 6.2 Common Elements

The first things to implement are the Common Elements as summarised in Section 2.3.2, the different POUs and data types. Programs and Function Blocks are implemented in the same manner as from a code generation point of view they are functionally equivalent. Their internal variables are stored inside a global struct of a type declared with the same name as the Function Block or Program. The code generated will go inside a function declared with the name of the POU and the string '`__BODY`' appended. A double underscore was chosen since the standard explicitly forbids two consecutive underline characters and thus it is invalid for a user to declare a function that might cause a name collision.

### 6.2.1 Program Organisation Units

Functions are implemented as one would expect: the exported symbol has the same name as in the function declaration. The functions receive a set of arguments, explicitly defined in the function declaration, preceded by the *EN* and *ENO* parameters, even if omitted in the declaration. These are mandated by the standard for explicit function control. The body of a function is composed of LLVM's *BasicBlock* construct. *BasicBlocks* are the building blocks of the LLVM API. They are containers of instructions to be executed sequentially, without branching and always with a return instruction ending them or a jump to another *BasicBlock*. Since the body of a function can have loops and branching instructions (if constructs, function invocation, *etc.*), a function body always starts with three *BasicBlocks*: *init*, *main*, and *ret*. In the *init BasicBlock*, local variables loading instruction are created, for arguments, global variables, *etc.* that might be needed within the function; the actual body of the function instructions will be generated in the *main BasicBlock*, having a jump instruction to the next block in its end and the *ret BasicBlock* has a proper return (from function) instruction. This way, execution reaching the end of the *main* block will jump to the last block to return from the function normally. To end a function execution early, it is only necessary to jump to the last block from anywhere within the *main* block.

The Function, Function Block, and Program code generation is very similar; the only difference is that Function Blocks and Programs are exported as global instances of a custom data type representing their parameters and internal variables, while Functions are exported as regular function symbols. Standard Functions and Function Blocks must be available for use anywhere within a program. This standard library must be implemented somewhere; this implementation itself need not be visible to the user, as much as most systems do not provide source code for a libc but a pre-compiled version and header files so a user can include the desired components. Since IEC 61131-3 does not employ constructs similar to `#include` for these languages, standard functions and Functions and Function Blocks should always be available; the way MatIEC does this is by having two *pragma* extensions to the standard (which is allowed by the standard; implementations should ignore *pragmas* they do not understand). A *pragma* has the syntax `{...}` and MatIEC chose the *pragmas* `{disable code generation}` and `{enable code generation}`. The MatIEC compiler always includes the standard library before the compilation of any file; just before including the standard library, a *pragma* disables code generation, re-enabling it just after the processing of the standard library and resuming the compilation of the intended file as usual. Disabling code generation tells the compiler to export function and function blocks without defining them; this is analogous to a function declaration in C, which exports the symbol without defining it. This process exports symbols for all the standard library, ensuring they are accessible for use anywhere in a program. At link-time, the implementation itself is still needed, and one is provided, for a complete executable. The current standard library implementation is done in C; using LLVM enables us to eliminate the external dependencies that are the C standard library and run-time by implementing the IEC 61131-3 standard library in ST itself.

### 6.2.2 Elementary Data Types

Elementary data types are implemented as specified in Table 2.1. Integer types are the easiest to implement as LLVM provides constructs to directly dictate the width and signedness for integers. For floating-point data types (`REAL` and `LREAL`) are defined as IEC 60559 single- and double-width floating-point data types. It is the standard representation of floating-point data types, more commonly recognized as IEEE 754, for most programming languages and LLVM also provides constructs for these under the names of *float* and *double* data types, respectively with 32 and 64 bits of precision.

The standard also defines the data types `BYTE`, `WORD`, `DWORD`, `LWORD`, `STRING` and `WSTRING` to have a specific number of bits of storage reserved (respectively 8, 16, 32, 64 bits and 8 and 16 bits for each character). All these data types can be implemented with LLVM's integers as they do not represent an actual integer number but a memory location of a specified width to be treated as a number in their binary representation. That also works with characters in strings because a character is represented as a number, but is interpreted in a slightly different way in some cases—namely when printing them, which needs a map assigning numbers to characters (which varies depending on the encoding used); the standard defines the minimum supported character set to be a subset of Unicode—the character set also supported by the ASCII encoding.

Despite not explicitly naming the specified encoding of the character set, it is equivalent to the common ASCII encoding[3]. Additional character set support is permitted, if compatible with the Unicode character set. Both `STRING` and `WSTRING` encodings are mandated to follow the ASCII encoding and to support additional Unicode code points, an ASCII-compatible encoding must be used. The most common encoding of this type is UTF-8. It is a variable-width character encoding where each character is represented with 1 to 4 bytes each; 1-byte characters encode the same characters as ASCII. Although the `WSTRING` type is to encode a character in 16 bits, suggesting UTF-16 should be used, that is not allowed by the standard since it is not compatible with the ASCII encoding.

Since each character is supposed to take 8 bits of storage space, the `STRING` type can only encode ASCII or Extended ASCII characters. But with the `WSTRING` data type, using UTF-8, all characters using one and two bytes can be encoded—that is 2048 different characters.

A possible way to encode all of the Unicode character set is to use the UTF-7 encoding, which is compatible with ASCII at the expense of needing several characters to represent higher Unicode code points (but all of them use only ASCII encoding). The drawback of this approach is that calculating the length of a string will compute the number of bytes of storage and not the intended number of characters, possibly breaking existing programs. Many implementations are using UTF-16 in this way, representing characters with either 2 or 4 bytes; it is possible to use UTF-8 in the same manner, with some characters taking 1 to 4 bytes of storage and not following the standard-defined encoding. This should still work with most programs since in a graphical program codepoint translation to characters is done by the operating system or graphics framework; as long as computation with string types is consistent within the compiler, the end result is unaltered.

UTF-8 is probably the most used encoding for applications, the *de facto* standard for some internet protocols, XML, and many other markup languages, so it is probably the best option for correct string implementation—with the limitation that only characters with codepoints using at most 2 bytes can be used with `WSTRING` and only the Extended ASCII equivalent for `STRING` data types; if standard compliance is not strict, it can also be used to encode every Unicode code point.

The `TIME`, `DATE`, `TIME_OF_DAY`, and `DATE_AND_TIME` data types' precision is not defined by the standard. 64 bits of storage for each type was chosen; this ensures a bigger range than with 32 bits as some implementations do. The possible drawback is for architectures which do not have native 64-bit registers and memory accesses, in which 64-bit operations must use extra instructions to perform operations. This is not a serious concern given modern architectures' capabilities since many of them, despite having 32-bit addressing, can perform 64-bit operations on registers and

[3]Character sets and encodings are a complex subject which is not in the scope of this project. Most programmers will be familiar with ASCII—that is what the IEC 61131-3 standard requires

memory with no performance impact. In older architectures, this approach might be problematic as the performance impact of emulating 64-bit operations is perceptible but not as significant as to have a meaningful impact.

Date and time data types implementation using a string representation underneath is also possible and used in some implementations, but this implies a very significant impact on performance for operations on these types. Internally representing them as 64-bit integers (not visible to the user), even when used in architectures which do not natively support 64-bit operations, will have a substantial performance benefit.

Declared variables are stored in a map to be accessible throughout the whole program, called `localvar_map`. They are represented by a LLVM `Value`, which holds the variable name, value, and type (A `Value` can also hold typed literals or expression evaluation results and not only variable declarations).

## 6.3 Structured Text

After the implementation of the Common Elements, Structured Text implementation is almost complete as well, missing only its expressions and statements. The remaining ST constructs are summarised in Section 2.3.4.

### 6.3.1 Expressions

Most expressions are logical or arithmetic operations on data, except for parenthesization and function invocation. Parenthesization expressions should not be seen in this compiler stage. Constructing a tree structure from the input program already takes into account operator precedence. The very structure of an AST will dictate the operands of an expression node and its operator, so this type of expression should not occur. More detailed explanation on the construction and structure of such a tree was given in Section 3.2.2.

A function invocation expression can be implemented in LLVM with the function `CreateCall`, passing it a reference to the intended function to call and a vector of its parameters' values.

Implementing operators is also fairly easy as such constructs are indispensable and present in all programming languages and therefore LLVM provides functions to quickly implement them. Most logic and arithmetic operators have two operands, exceptions being the Negation and Complement operators (see Table 2.2), which have a single operand.

The operators expect an LLVM `Value` for each operand, which is easy to obtain thanks to the visitor pattern: in the tree, expressions classes have members called *l_exp* and *r_expr* representing the left- and right-hand side of an expression (known as *lvalue* and *rvalue* in C and C++). The only thing to do is call the *accept* function of each one, which will call the correct function depending on what exactly each member is (a literal, a variable, another expression, *etc.*). An example for the *and* expression is:

```
void *generate_ll_c::visit(and_expression_c *symbol) {
    Value *l_value = (Value *)(symbol->l_exp->accept(*this));
    Value *r_value = (Value *)(symbol->r_exp->accpet(*this));
      //--- error checking code ---//
    return builder->CreateAnd(l_value, r_value);
}
```

Equivalently to this example, LLVM provides simple-to-use functions for the other operators: `CreateNot`, `CreateOr`, `CreateXor`, `CreateICmpEQ` and `CreateFCmpOEQ` (both for testing equality for integer and *float* types); `CreateFCmpONE` and `CreateICmpNE` (testing inequality); `CreateFCmpOLT`, `CreateICmpSLT`, and `CreateICmpULT` (*less than* operator, for *float* and integer, both signed and unsigned types); similarly to the *less than* operator, functions for testing *greater than* and *less/greater than or equal to* are also available for different types of variables. There is no need to list the other functions as well because the list gets quite extensive, but very similar functions for different data types and signedness for addition, subtraction, multiplication, division, modulus, and negation exist. An exponentiation function does not exist in LLVM; to generate IR for this operator, a function call to the IEC 61131-3 standard library `EXPT()` is made. A function call to the C standard library function *pow* could be made (and indirectly is, as currently the IEC 61131-3 standard library is implemented in C), but that creates a very strict external dependency on a *libc*—if the IEC 61131-3 standard library was rewritten in ST, the unnecessary dependency would remain, only for the implementation of the `EXPT()` function.

### 6.3.2 Assignment Statements

ST's assignment statement implementation is also painless: the *CreateStore* function stores a value into a variable reference. The variable reference can be obtained easily as declared variables are stored in a globally accessible (in this compilation unit) map, `localvar_map`. The value to be assigned is an expression that might require additional computation, but since expressions are implemented elsewhere, it is only needed to *visit* the corresponding implementation with:

```
Value *r_exp_value = (Value *)symbol->r_exp->accept(*this);
```

and passing the obtained `Value *` and variable reference to the appropriate function.

### 6.3.3 Function and Function Block Control Statements

Function and Function Block Control statements (the `'RETURN;'` statement) is only a matter of creating an IR `ret` instruction through the use of the `CreateRet` function and passing the correct value to be returned as an argument. A special case to consider is an early return, one which happens before the actual end of the Function or Function Block. Since LLVM's *BasicBlocks* must not have branching or return instructions in the middle, but only at the end, an early return instruction would be followed by more instruction with the rest of the Function or Function Block code. To prevent this, a *BasicBlock* is created and inserted just after each return statement. If an early return is being processed, the code generation will go on as usual; if the return statement is found at the end of a Function or Function Block, there will be an empty *BasicBlock* at the end of that construct, with only its respective return instruction. This is fine, empty blocks are allowed and very common; the LLVM optimisations passes will remove these and re-order instructions.

### 6.3.4 Selection Statements

The `'IF'` selection statement is not particularly difficult to implement: it has a condition (an expression) that must be evaluated; depending on that condition, control will either flow to the *then* block or to the *else* block (if present—if not, to the end of the statement). Since, again, *BasicBlocks* must not branch in the middle, three new blocks are created: *then*, *else*, and *end_if*. Condition evaluation is just having the expression *accept* the current visitor:

```
...expression->accept(*this);
```

This returns a `Value *` which can be used to create a conditional branch with the function `CreateCondBr`, which receives as arguments the blocks to which control should flow in case the condition is true and false (respectively, the *then* and *else* blocks).

The *else* block has a non-conditional branch instruction to the *end_if* block; if an *else* construct exists and has statements inside, they will be executed and, in the end, control jumps to the end of the *if* block to resume execution; if there is no *else* block, control just jumps directly to the end of the if construct.

The `'CASE'` statement starts the same way as a simple `'IF'`, by evaluating an expression; the result should be used to select a branch according to its label.

In the C language, the similar *switch* statement exists, but with an important difference: the expression and case labels must be numeric. This enables the compiler to implement these as jump tables most of the time; this is very efficient as the expression is evaluated once and control directly jumps to the correct label, without needing to compare the expression equality with every label until one is found, but it only works for numeric expressions. Since the IEC 61131-3 standard allows the `CASE` expression to be a variable of any type, including user-defined types, this approach is not possible and a comparison must be made with each label to select the correct one. A mixed approach can also be made, where numeric expressions are implemented as a jump table and other types with comparisons but currently only the generic implementation is done.

Implementation of this statement is then started by the evaluation of the expression; then, an equality comparison to each label expression is generated, branching into it if the yielded value is `TRUE`. If an `'ELSE'` is present, the code for its internal statements is generated; if execution reaches this block is because the expression did not match any label, which works as intended. At the end of all alternative branches, an unconditional jump to the end of the `CASE` statement is made so execution can be resumed.

### 6.3.5 Iteration Statements

Of all the iteration statements, the `'FOR'` construct is the most complex. First, a reference to the control variable must be obtained from `localvar_map`. Then, the initial value and increment value expressions must be evaluated to a simple `Value *`; this is simply done by *accept*'ing the visitor. An assignment is done to the control variable with the `CreateStore` function.

The evaluation of the end value for the control variable must be done in every iteration as it may depend on variables that change within the loop. The easiest way to implement this is to put it in a separate block and jump to it at the beginning of every iteration; then, evaluate the condition (an equality test as explained before) and depending on the yielded value, either jump to the loop body block or to the end. The loop body block is a list of statements and can just be generated by *accept*'ing the visitor again. Reaching the end of the loop block control jumps to the increment block, which will store (`CreateStore`) the incremented value (addition has been previously explained in the ST expressions) in the control variable.

The end of the loop block is reached if the control variable equality to the final value is verified.

The `'WHILE'` construct resembles the `'IF'` selection statement. Three blocks are created: in the first, the condition evaluation (an expression) is done, as was previously explained; then, the `CreateCondBr` creates a conditional jump depending on the value yielded by the condition, either to the main block or to the end of the loop; the loop body block is a list of statements, generating the IR for these is only a matter or calling the *accept* function with the current visitor, as usual. The main block unconditionally jumps to the condition evaluation block; when the condition is not verified, control will jump to the end of the loop.

The `'REPEAT'` loop is very similar, but only needs two blocks: in the first, the body statements are evaluated as usual (*accept* function); at the end of that block, the condition evaluation is made (again, an expression evaluation); depending on the value of the condition, the `CreateCondBr` function allows us to either jump to the beginning of this block to repeat the execution of the loop body or to the end block, which ends the loop.

The `'EXIT'` statement is somewhat equivalent to the `'RETURN'` statement, but with a minor inconvenience. Firstly, since a branching instruction is necessary, a new *BasicBlock* must be created right after this statement to ensure block structure correctness. The problem with this statement is that since loops are not new functions, they do not affect the call stack. Thus, it is impossible to simply *return* to the caller—there is no caller, we are inside the same function. An unconditional branch to the correct *BasicBlock* is then needed and information about which block control should jump to must be kept. Inside each iteration statement, a variable will then store a reference to its main block—this way, upon encountering an `'EXIT'` statement, the `CreateBr` function unconditionally jumps to the outer loop.

## 6.4 Sequential Function Chart

The implementation of Sequential Function Chart is not as straightforward as ST expressions and statements, it requires a *runtime* besides the steps, transitions, and actions declarations to execute the SFC logic (as explained in Section 2.3.5).

SFC elements are implemented in a new visitor, *generate_sfc_c*. Within a POU, if SFC elements are detected, a *generate_sfc_c* instance is created; to pass control to the new visitor, the *accept* function of the root SFC element is called, passing as argument a reference to the new visitor instead of the current:

```
visitor_c *sfc_generator = new_sfc_generator(...);
symbol->accept(*sfc_generator);
delete_sfc_generator(sfc_generator);
```

The implementation must cyclically execute the actions associated with the active steps and evaluate transitions' conditions to activate and deactivate steps based on the result. A diagram of the SFC logic implementation can be found in Figure 6.2.

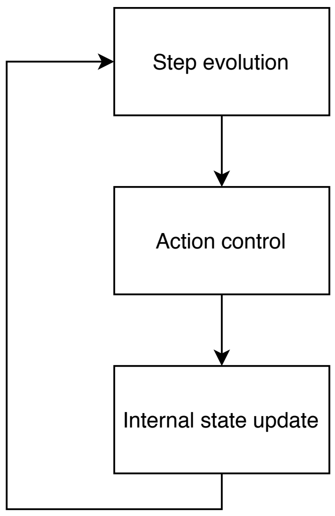


Figure 6.2: High-level SFC behaviour overview

These three stages are repeatedly executed: Step evolution controls which steps are activated at any specific point in time; then, Action control calculates whether an associated action should be run and executes them accordingly; at the end of each cycle, the internal state of the SFC is properly updated, as action control is dependent on some of the last cycle variables' values and thus they must be saved before proceeding to the next cycle.

Step evolution (or SFC evolution) has been explained in Section 2.3.5. It dictates the current *state* of an SFC as it activates and deactivates steps, triggering SFC actions accordingly, and is controlled by the transitions. The algorithm deciding a step state is illustrated in Figure 6.3.

The action control stage needs to perform two distinct tasks: evaluate the Action Control Function Block defined in the standard (the logic only, as the implementation is free to not define the actual Function Block as is in this case—see Section 2.3.5) and, depending on the yielded result, execute the action statements. A diagram demonstrating this behaviour can be found in Figure 6.4.

The last stage in a cycle is then reached. Here, it is only needed to correctly update the internal state of the SFC and its behaviour is heavily dependent on SFC implementation details. This state is mostly needed for correct action control behaviour, as it uses some Function Blocks internally which require a state to be maintained.

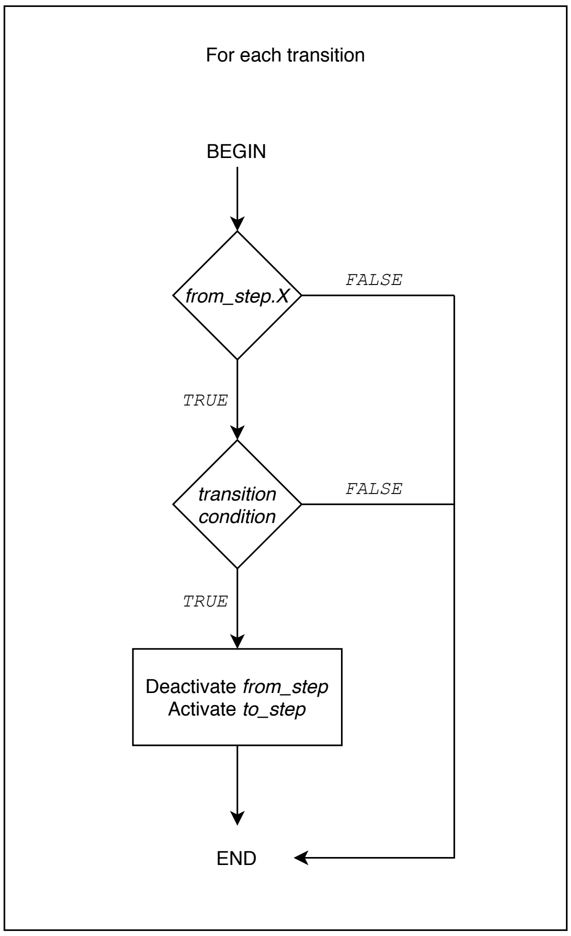


Figure 6.3: SFC step evolution flow diagram

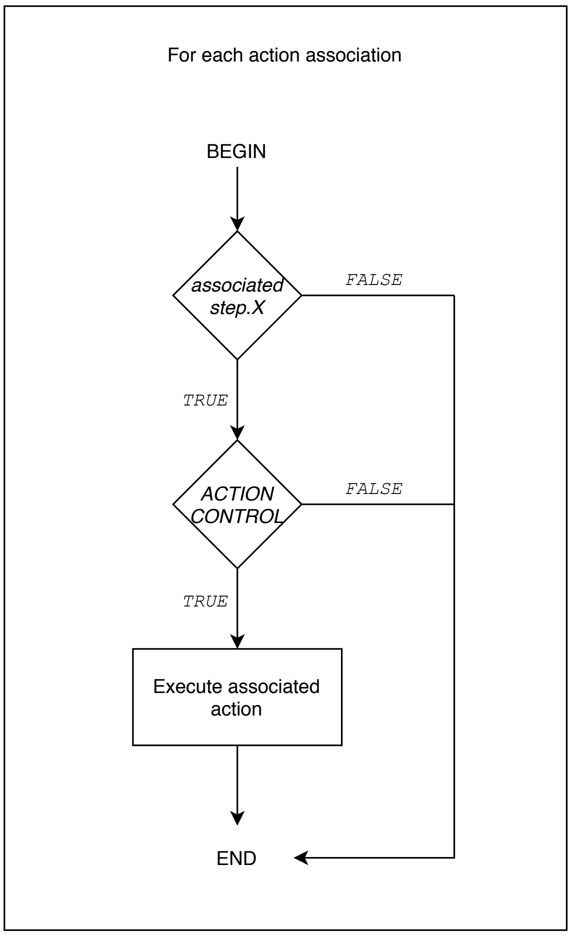


Figure 6.4: SFC action control flow diagram

Declaration of steps does not need any code to be executed (only code generation for the declaration itself), but transitions and actions do. Transitions need to evaluate its condition: generating this is fairly easy as it is just evaluating an expression, but the location *where* this code should be in order to execute the correct logic is the key. The same can be applied to actions; they need also to be declared, but the action execution (statements evaluation) itself must be in a specific place, so the SFC behaviour in Figure 6.2 can be obtained.

Step declaration must expose the step member fields to the user: the active flag (`.X`) and its active time counter (`.T`). This is done by declaring steps as a structure with these fields, a boolean value and a `TIME` value, respectively. Again, since this structure should be available for use, they are added to the declared variable map `localvar_map`.

Transitions do not need any variable declaration of this type, just condition checking code, but actions do. Since an action may be associated with a number of steps, with different action qualifiers, each association needs an action control mechanism of its own. This is necessary because the action control has an internal state, making each association require a different instance of the action control mechanism. In the standard, this is defined resorting to a Function Block called *ACTION CONTROL*. Function Blocks can store internal variables (state), so creating a Function Block instance for each association would allow correct control, but using a Function Block for this effect is more complicated than implementing the logic itself.

The standard explicitly declares the action control need not be implemented by the specified Function Block, just equivalent logic is required. Action association is then declared using an internal structure variable, not visible to the user unlike with steps. It stores 11 boolean values, one for each modifier, plus some more for the required internal state. Inspecting the action control Function Block, we can see some of the qualifiers require other standard Function Blocks such as `RS`, `R_TRIG`, `F_TRIG`, *etc.* These Function Blocks require storing the last value of some of their inputs or other internal variables, which will affect the output value. To implement this, we must store these internal values in the action association instance for later use, for each standard Function Block instance.

Currently, timing-related qualifiers have not yet been implemented, leaving only qualifiers `N`, `R`, `P`, `P0`, and `P1`. This simplifies action control, which behaviour is illustrated in Figure 6.5. For these, 4 additional boolean values are needed to store their internal state; making the action association structure contain 15 boolean variables in total.

An SFC is implemented as a POU, either a Program or a Function Block, and the SFC code generation is done inside that POU. This implies that the SFC logic will be inside a function named after the POU, with `__BODY` appended to it (as explained in Section 6.2.1). This function will be repeatedly called by the runtime and should execute each SFC cycle in each invocation. But SFC requires a state and the creation of some internal variables, which must not be done every time the `__BODY` function is called, otherwise the SFC will be restarted in each cycle. To circumvent this problem, the SFC compilation generates a new function, with the same name as the main one, but with `__INIT` appended. This intends to declare and initialise the required internal variables as global variables to be accessible by the `__BODY` function. This approach has an inherent complication as it breaks the generic interface of a POU: they expose a very simple interface to the runtime calling them, the `__BODY` function. The runtime does not need and should not be aware of what and which types of elements compose a POU, whether it is an SFC network, ST statements, IL instructions, or anything else. But this method requires the runtime to be aware that the specific POU is an SFC and correctly issue the call to the `__INIT` function once and before any of the repeated invocations of `__BODY`. This SFC implementation uses this process to generate the SFC logic, but the inherent issues must be acknowledged. Since this is just an initial

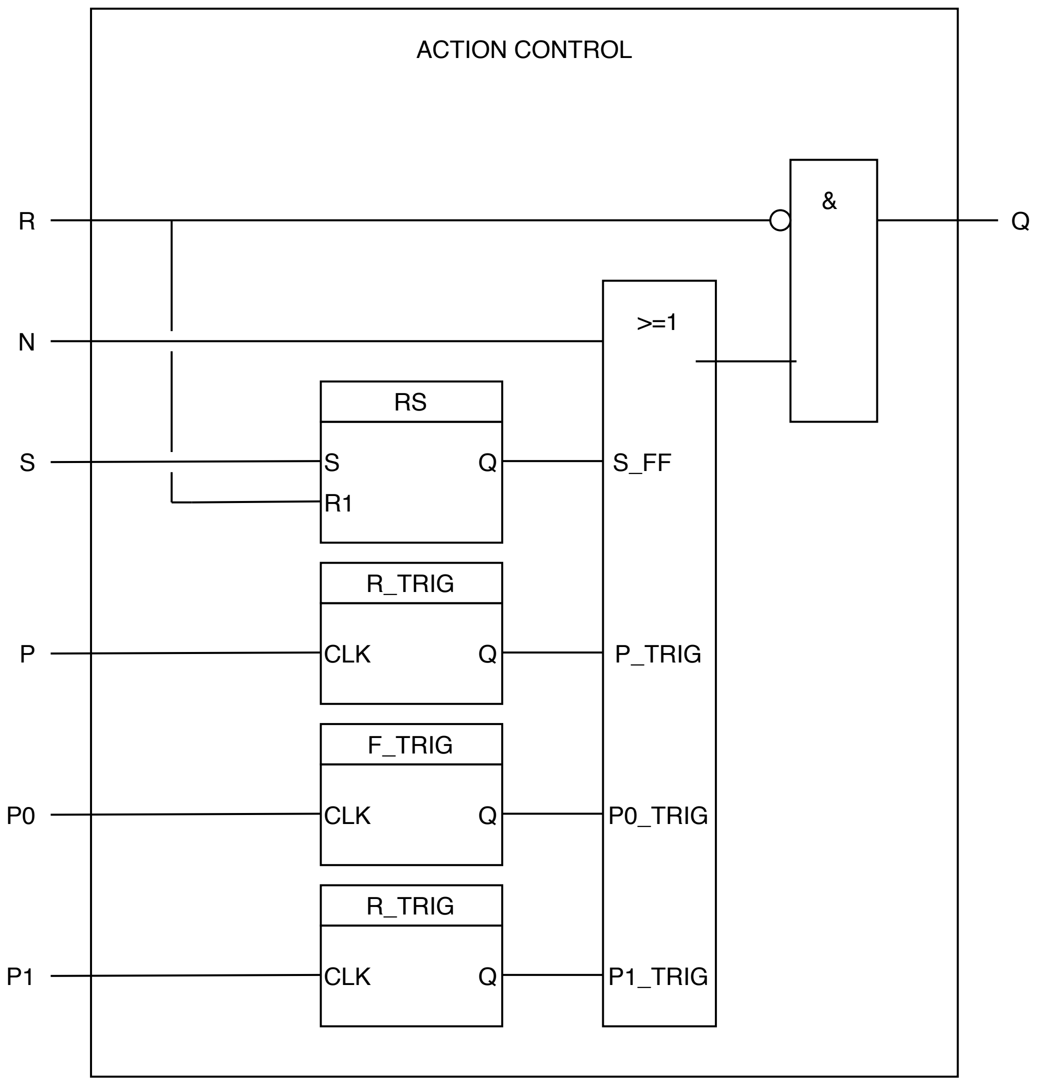


Figure 6.5: SFC action control implementation

implementation and a proof of concept that intends to study and measure the impact of direct code generation, opening the way for a future improved version, this issue is not as significant as it does not affect (positively or negatively) the overall performance and behaviour, it just breaks the generic interface of a POU.

SFC code generation is then done in three compilation passes: the first will create the `__INIT` function and code for internal variables' definition; the second pass generates code for the step evolution and action control SFC stages and, finally, the third pass will generate code for the last stage, updating the internal variables.

The first pass will generate the new `__INIT` function, which will declare and initialise the step and action association structures. For initial steps, the active flag starts *true* and regular steps are initialised with a *false* value. For action association structures, the qualifier flags in the association are initialised to a `1` if used or `0` otherwise. The internal state boolean flags for the standard Function Blocks' initial values are specific to their behaviour, but most of them will be initialised to `0`.

The second pass of the SFC generator visitor will handle the SFC execution logic: code generation for transition condition checking and the action control logic. The `__BODY` function does not need to be created here, as it is created by the POU code generation so it already exists when the *generate_sfc_c* visitor is created; the SFC logic code should be created inside this pre-existing function. Transition condition evaluation is only a matter of evaluating an ST expression as already explained in Section 6.3; if the resulting value is *true*, the previous step should be deactivated and the next step activated, but this only applies *if* the previous step was activated as a deactivated step cannot activate the next even if the transition proves to be *true*.

The algorithm for step evolution to implement is presented in Figure 6.3 and can be achieved by the equivalent pseudo-code:

```
for every transition:
    if previous step active:
        if transition condition = true:
            deactivate previous step;
            activate next step;
```

The actual implementation of these constructs is very similar to the already explained for ST and does not need additional detail, as it mainly consists of expression evaluation and branching.

This stage also generates the logic for action control. This requires computing first the result of the internal standard Function Blocks' output value, which uses the internal state flags stored in the action association structure itself. Then, the logic explained in the action control from the standard is performed with the yielded values and the action qualifiers flags (also stored in the same structure). The resulting boolean value indicates whether the action associated with a particular step should be performed.

The actual calculations for action control for the supported qualifiers are:

```
    S_FF      := (S_FF OR S) AND NOT R;
    P_TRIG   := P AND NOT P_TRIG;
    P0_TRIG  := NOT P0 AND NOT P0_TRIG;
    P1_TRIG  := P1 AND NOT P1_TRIG;
    ACTION.Q := NOT R AND (N OR S_FF OR P_TRIG
                           OR P0_TRIG OR P1_TRIG);
```

S_FF, P_TRIG, P0_TRIG, and P1_TRIG are the variables which represent the internal state; their value must be kept across invocations and is then stored in the action association structure instance; to achieve this, at the end of a cycle, their values will be updated as follows:

```
    P_TRIG  := P;
    P0_TRIG := NOT P0;
    P1_TRIG := P1;
```

For the actual code generation, the algorithm is described in pseudo-code (as illustrated in Figure 6.4):

```
for every action association:
    if associated step is active:
        compute internal standard Function Block outputs;
        if action control logic == true:
            generate code for action;
```

Usage of the LLVM API to generate IR for this algorithm is again very similar to the already explained: creating a conditional branching instruction (where the condition is the previous step's state flag); creation of *AND*, *OR*, and *NOT* operations to evaluate ACTION.Q and using it as a conditional branching expression to a block with the action instructions.

An action can be a variable of type BOOL or a list of statements. In the first case, the action code generation is only a call to the CreateStore function to that specific variable; the value to be stored is the one yielded by the action control logic, either *true* or *false*. In the second case, IR code generation for the list of ST statements is done as already explained in Section 6.3.

The third pass still generates code within the __BODY function and its purpose is to update the internal state of every variable instance. Internal state flags store the previous cycle variables' values, so at the end of each cycle, these are updated to reflect the current value; this way, in the next cycle, those flags will have the previous value, completing SFC logic code generation.

The generated functions, __INIT and __BODY should then be called by the runtime. The __INIT function should be called once and before the corresponding __BODY call. Multiple calls are possible as it only initialises the internal structures used by the SFC execution and it might be useful to reset or restart an SFC. After initialisation, the __BODY should be called repeatedly, for each PLC cycle. A simple SFC example program and its corresponding generated LLVM IR can be found in Annex C (the standard Functions and Function Blocks declarations have been omitted for brevity). LLVM IR is very easy to read and understand despite being low-level and assembly-like.

# Chapter 7

# Evaluation and Testing

## 7.1 Completeness and Standards Compliance

The new code generator supports only the ST and textual SFC languages. The compiler front-end also understands IL instructions, but the code generator currently ignores IL constructs. The standard also defines SFC transitions to be possibly written in IL and actions in any other IEC 61131-3 language, but current SFC implementation supports only ST expressions and statements.

From the Common Elements, support for all types of POU is implemented, that is Programs, Function Blocks, and Functions. All elementary date types are also implemented; derived (user-defined) data types such as `STRUCT`, `ENUM`, and `ARRAY` are yet to be implemented. Some basic `STRUCT` support was added due to being necessary to implement some parts of SFC, but it is very limited and not generally working for other purposes.

ST-specific constructs are fully implemented: function invocation, logical and arithmetic operations; assignment, selection (`IF` and `CASE`), Function and Function Control (`RETURN`), and iteration (`FOR`, `WHILE`, `REPEAT`, and `EXIT`) statements are also implemented.

In SFC, timing qualifiers and steps' `.T` indicators are not implemented. Timing properties require *runtime* support. Since MatIEC does not target any specific runtime, underlying operating system, hardware, *etc.*, it is not possible to handle timing constructs. For a PLC running an operating system, most of the time functions for getting current time, setting timers and other timing-related operations will be available. Not targeting a specific operating system disallows the use of any such API. Most hardware platforms or systems also present such capabilities, but none is assumed for MatIEC and it is thus impossible to implement such constructs without officially supporting a specific system or set of systems. MatIEC's job is to translate IEC 61131-3 programs to binary, executable, code of some architecture and this is successfully achieved. Maybe having an indirection layer providing an API for necessary constructs such as timing operations might be useful, to be called by MatIEC generated code when such constructs are needed. Then, adding MatIEC support for a particular system is only a matter of implementing the API functions for the specific system, but this is outside the current scope of MatIEC.

SFC steps and actions association internal structures are currently only declared in the generated LLVM IR. For the moment, they need to actually be instantiated as global variables from anywhere in the final linked executable. Future work on this new target might also define these global variables in the moment of declaration.

SFC actions have an implicit field, `.Q` to indicate whether it is active and `.A` to enable action execution for network-based action definitions (LD, FBD). These are also not yet implemented.

To allow debugging of an IEC 61131-3 program in a traditional debugger, adding *debug information* to the generated binary is required. LLVM allows adding debug information in a similar API to the IR generation. This is partially implemented; some debug information is already present but not everything. Unfortunately, the currently generated debug information is not enough to be used inside a debugger, so debugging IEC 61131-3 programs is not yet possible.

Despite all that is yet to be implemented, IEC 61131-3 support is already very significant. All ST-specific constructs are implemented and most of the SFC elements as well. Within the most commonly used constructs, the only one yet missing is derived data types support.

The other graphical languages (LD and FBD) are usually possible to convert to a textual language (ST) by IEC 61131-3 editors (*e.g.* Beremiz), therefore some success using these might be possible, although not tested.

## 7.2 Performance Comparison

The new target is already usable and mostly complete for the most common constructs, but it must undergo testing for correctness and performance. Compared to the older MatIEC target, the performance of the new one is expected to be at least slightly faster; translating to ANSI C and compiling that code with a C compiler may prevent some optimisations. Programmatically generated code is not as idiomatic and makes the job of a compiler more difficult to recognize common constructs and optimisation opportunities.

For this purpose, a few tests were constructed, both for ST and SFC. For both languages, an empty program was made. This measures a cycle's overhead or the time a CPU will "waste" executing the (necessary) runtime code for a PLC cycle instead of the intended program.

Since MatIEC does not generate a runtime, a very simple C implementation will call the generated code and measure the elapsed time. Measuring the time a program takes to run is tricky; usually, the program will be running on top of an operating system rather than on *bare-metal*, that is, with no underlying operating system, directly on the hardware. This introduces a problem since an operating system runs several processes concurrently, constantly running and pausing a process without it ever knowing about it. This will affect both targets, obviously, but we cannot assume it will have an equal impact on both; the scheduler may not give an equal processor time for both programs, it is free to decide which process it wants to run.

This means we cannot simply measure the elapsed time from the beginning to the end of such a benchmark, as it might be heavily influenced by the scheduler and operating system decisions; we need a way to measure the *processor time* given to a specific process.

Some hardware registers with instruction counters are also available, but they suffer the same problems as calculating elapsed time in software, as the counters are incremented independently of which process is running.

To address this problem, the C standard library header *time.h* provides useful mechanisms. Being standard C functions, they should be available for practically every system and this abstracts away the underlying system limitations. This header provides a function, `clock`, which does what we need (from the Linux *man-page*):

> "The clock() function returns an approximation of processor time used by the program."

We can use the return value to *estimate* the processor time used by the process. Note that the implementation of this function on the Windows operating system is not correct and should not be relied upon, but on Linux/macOS/Unix-based operating systems this should be correct.

The `clock` function return value is of the type `clock_t` and represents the used CPU time. To convert this value to seconds, the macro `CLOCKS_PER_SEC` can be used: dividing a `clock_t` variable by `CLOCKS_PER_SEC` will result in the equivalent time in seconds, a floating-point value with *double* precision, which allows time measurements' precision of microseconds[1].

To compare the performance of the two targets, two different systems were chosen: the *x86_64* and *ARM* architectures, two of the most common architectures; the x86_64 architecture is the most common for industrial PCs and personal computers (for soft PLCs) and the ARM architecture is very common for PLC CPUs.

The x86_64 device is a personal computer running the Fedora 31 operating system, with Linux kernel 5.6.15; with an Intel Core i7-4710HQ CPU. The ARM device is an ARMv6 Raspberry Pi board, model B+, running *raspbian buster lite* with Linux 4.19.97; with a Broadcom BCM2835 SoC and an ARM1176JZF-S CPU.

### 7.2.1 Structured Text

For the ST language, the interesting things to test are: arithmetic and logic expressions, Function and Function Block invocation, flow control (*if/else* and *case* constructs), and loops (*for, while,* and *repeat*). To ensure more accurate timing measurements, for each test, the in-testing construct was executed 1,000 times.

The test for arithmetic and logic expressions has several different expressions repeated until 1,000 expressions are present. Some examples of the expressions, with different data types and operations are:

[1]The actual clock resolution is dependent on the operating system; on POSIX-compliant systems, this is mandated to be 1 $\mu$s, as is the case of Linux

```
out := ra + rb + rc;
rb := ra - 44.0;
tt := T#1s_0.00001ms;
tt := tt / (INT#2);
tod_v := TOD#0:1:0;
date_v := DATE#1-1-1;
ok := (out >= ra) & (rb + rc < 4.0);
```

Testing Function and Function Block invocation is done by calling a Function or Function Block repeatedly, with different argument values.

A simple example of one of the created Function Blocks, whose invocation will trigger the invocation of another Function Block:

```
FUNCTION_BLOCK foo_fb
  VAR_INPUT ra : REAL := 88.4; END_VAR
  VAR_OUTPUT rb : REAL; END_VAR
  VAR_TEMP Rt: REAL; END_VAR
  VAR RETAIN
    bar_v :bar;
    RC: REAL := 66.0;
  END_VAR

  bar_v(ra := 99.0, rb => rb);
  rb := ra;

END_FUNCTION_BLOCK
```

Function and Function Block invocation is then repeated until about 1,000 in total are done. The calculated values in these are not important as the focus is on the call itself.

Flow control testing is similar, about 1,000 selection statements. Example `IF/ELSE` and `CASE` statements are:

```
IF (ra = 1.0) THEN
  RA := 11.0;
ELSIF (ra = 2.0) THEN
  RA := 21.0;
ELSIF (ra = 3.0) THEN
  RA := 31.0;
ELSE
  Ra := 0.0;
END_IF;
```

```
CASE ra+rb OF
  10: test_case :=  0;
  11: test_case := 10;
  12: test_case := 20;
  13, 14, 15, 16, 17, 18, 19: test_case := 30;
  20..30: test_case := 40;
  100..110, 120..130, 200: test_case := 50;
  ELSE test_case := 99;
END_CASE;
```

Testing loops is also similar to the previous tests, about 1,000 loop constructs (`FOR`, `WHILE`, and `REPEAT`). Some examples are:

```
FOR int_v :=  -66 TO   -1 BY 2 DO
  real_v := real_v + 1.0;
  IF ((real_v / 10.0) >= 100.0)
    THEN EXIT;
  END_IF;
END_FOR;

REPEAT
  int_v := int_v + 1;
  real_v := real_v + 1.0;
  IF ((real_v / 10.0) >= 100.0)
    THEN EXIT;
  END_IF;
UNTIL  int_v = 33
END_REPEAT;

WHILE int_v >-33
DO
  int_v := int_v - 1;
  real_v := real_v + 1.0;
  IF ((real_v / 10.0) >= 100.0)
    THEN EXIT;
  END_IF;
END_WHILE;
```

Each test was run 10,000 times and the execution times for all the runs were averaged to obtain a comparable mean execution time for each target.

The results of these benchmarks for the ST language, with different optimisation levels (-*O0* to -*O3*), for both compilers and architectures, can be seen in Table 7.1.

Table 7.1: ST x86_64 execution times

| **Test** | **Optimisation Level** | ***iec2c* Execution Time ($\mu s$)** | ***iec2ll* Execution Time ($\mu s$)** |
|---|---|---|---|
| empty | *-O0* | 0.6 | 0.7 |
| empty | *-O1* | 0.5 | 0.5 |
| empty | *-O2* | 0.5 | 0.5 |
| empty | *-O3* | 0.5 | 0.5 |
| expressions | *-O0* | 10.7 | 2.0 |
| expressions | *-O1* | 2.5 | 0.8 |
| expressions | *-O2* | 2.3 | 0.7 |
| expressions | *-O3* | 2.3 | 0.6 |
| functions | *-O0* | 16.7 | 3.8 |
| functions | *-O1* | 6.7 | 2.6 |
| functions | *-O2* | 15.2 | 2.8 |
| functions | *-O3* | 11.9 | 2.8 |
| if/case | *-O0* | 7.3 | 1.1 |
| if/case | *-O1* | 0.7 | 1.0 |
| if/case | *-O2* | 0.6 | 1.0 |
| if/case | *-O3* | 0.6 | 1.0 |
| loops | *-O0* | 52.0 | 46.9 |
| loops | *-O1* | 48.1 | 33.8 |
| loops | *-O2* | 47.3 | 33.7 |
| loops | *-O3* | 47.6 | 33.7 |

For the ARM architecture, the equivalent execution times can be found in Table 7.2.

Upon a quick analysis of the execution results for ST, the difference in execution times is much bigger than initially expected. The empty cycle times are approximately the same, which makes sense; all others are substantially faster using the LLVM-based target. The only exception to this is the *loops* test for the ARM architecture, which is significantly slower. Further analysis should be made to understand the cause for this discrepancy; maybe using the *clang* compiler (LLVM-based C compiler) with the *iec2c* target instead of GCC to ensure it is not an LLVM ARM back-end limitation (which is unlikely, but possible).

It is also worth noting that both targets in some tests, with higher optimisation levels, will remove computations altogether as they do not have side-effects; they are close to an empty cycle execution time. Specifically, for the *expressions* and *if/case* tests, *-O2* and onwards execution times for both architectures are very close to an empty program; this makes higher optimisation tests less useful as they do not represent the actual execution of the generated code.

To better visualise the comparison results, a graphical representation can be seen in Figures 7.1 and 7.2, pertaining only to the tests with optimisations *disabled* (*-O0*), as they are the most interesting to inspect.

Charts comparing the effect of the level of optimisation can be found in Figures 7.3 and 7.4 in a linear scale and in Figures 7.5 and 7.6 in a logarithmic scale. The logarithmic scale is useful

Table 7.2: ST ARM execution times

| **Test** | **Optimisation Level** | ***iec2c* Execution Time ($\mu s$)** | ***iec2ll* Execution Time ($\mu s$)** |
|---|---|---:|---:|
| empty | *-O0* | 3.7 | 3.6 |
| empty | *-O1* | 3.6 | 3.4 |
| empty | *-O2* | 3.1 | 3.4 |
| empty | *-O3* | 3.5 | 3.5 |
| expressions | *-O0* | 392.6 | 60.9 |
| expressions | *-O1* | 92.0 | 11.8 |
| expressions | *-O2* | 74.2 | 12.3 |
| expressions | *-O3* | 70.2 | 12.0 |
| functions | *-O0* | 684.3 | 132.1 |
| functions | *-O1* | 250.2 | 77.3 |
| functions | *-O2* | 231.7 | 75.8 |
| functions | *-O3* | 237.4 | 86.0 |
| if/case | *-O0* | 838.2 | 60.3 |
| if/case | *-O1* | 6.0 | 60.6 |
| if/case | *-O2* | 6.0 | 60.4 |
| if/case | *-O3* | 4.3 | 61.0 |
| loops | *-O0* | 2122.4 | 5040.7 |
| loops | *-O1* | 1220.7 | 4595.5 |
| loops | *-O2* | 1134.2 | 4554.2 |
| loops | *-O3* | 4.4 | 4557.2 |

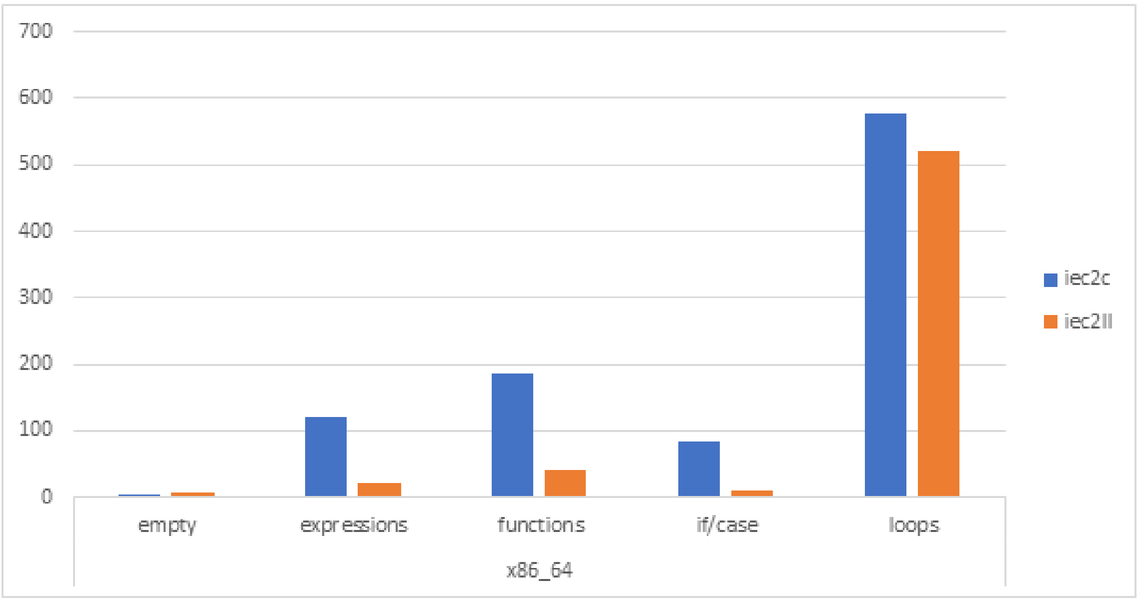


Figure 7.1: ST x86_64 execution times (-*O0*)

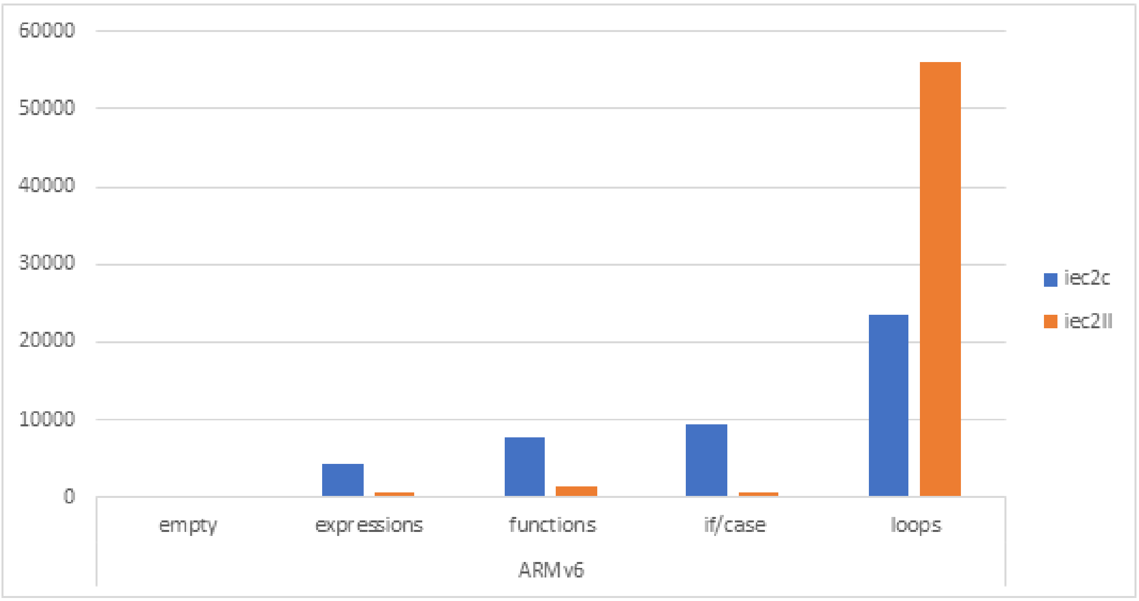


Figure 7.2: ST ARMv6 execution times (-*O0*)

when the values presented span a wide range of values, as is the case in some graphs. It also makes it easier to understand the general trend of execution times affected by the different optimisation levels. Since a logarithmic scale, by nature, softens differences between values, the linear scale is still useful to have a clearer notion of these differences.

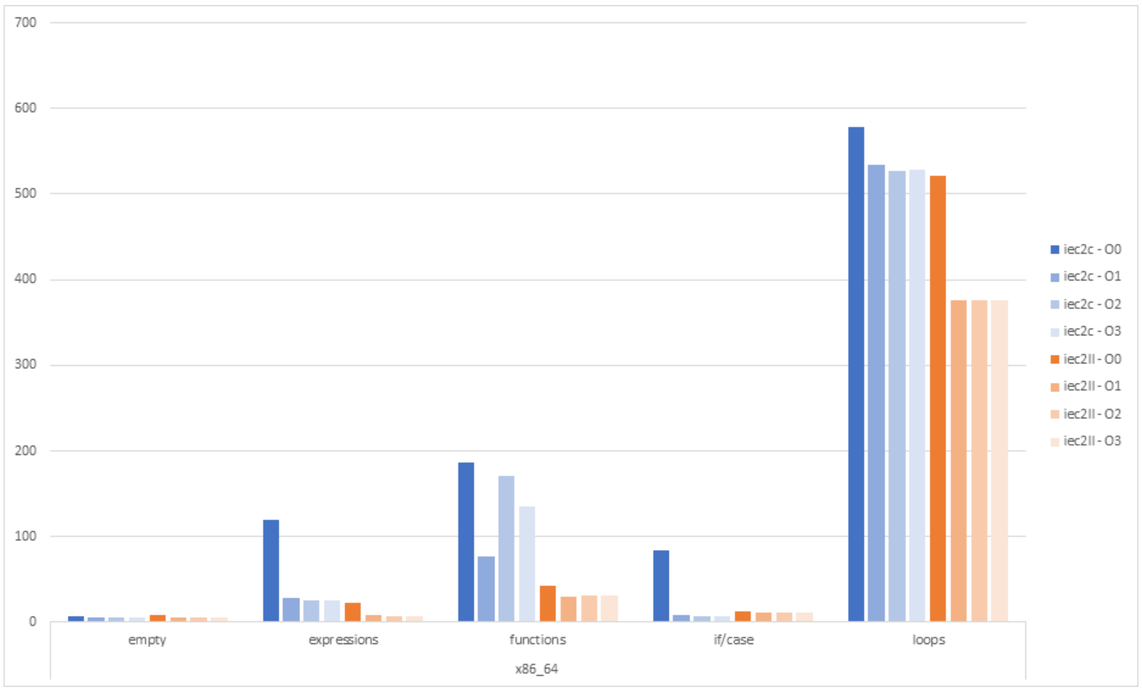


Figure 7.3: ST x86_64 execution times

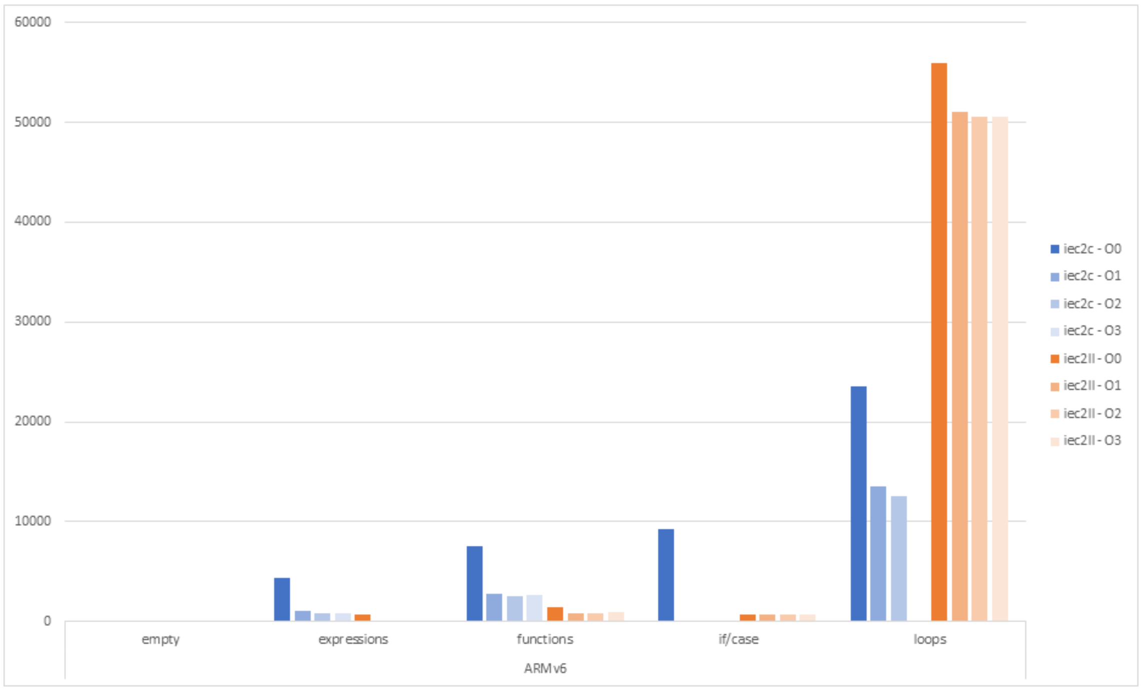


Figure 7.4: ST ARMv6 execution times

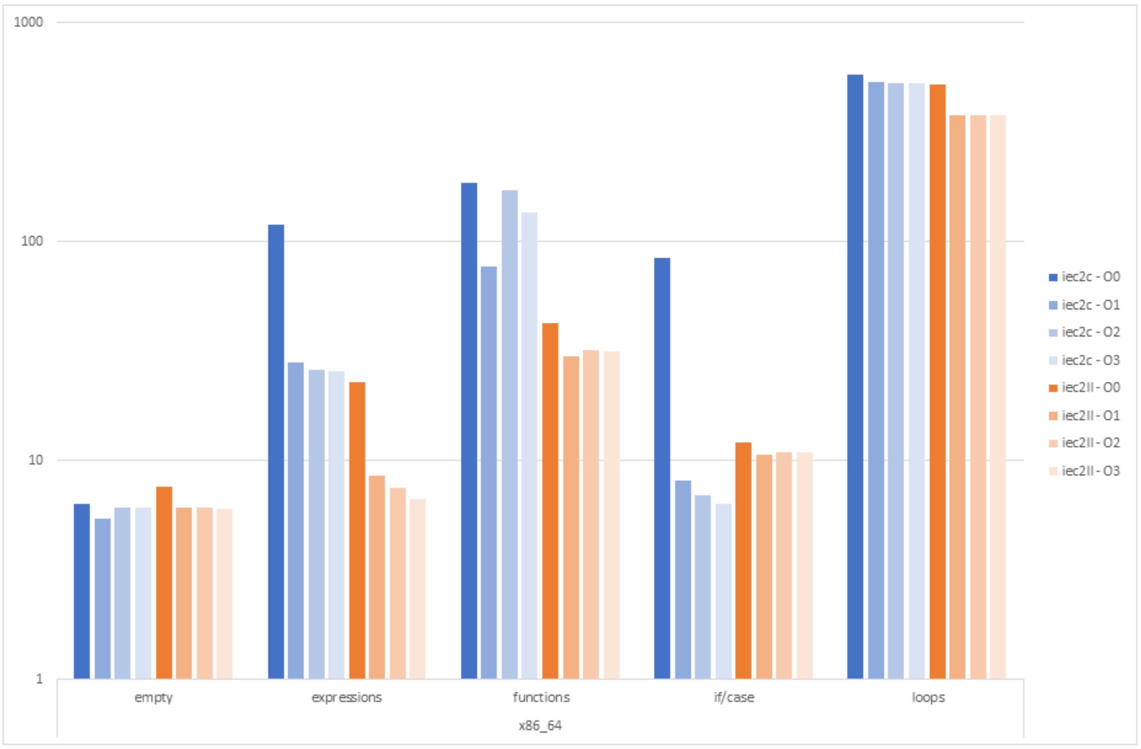


Figure 7.5: ST x86_64 execution times (logarithmic scale)

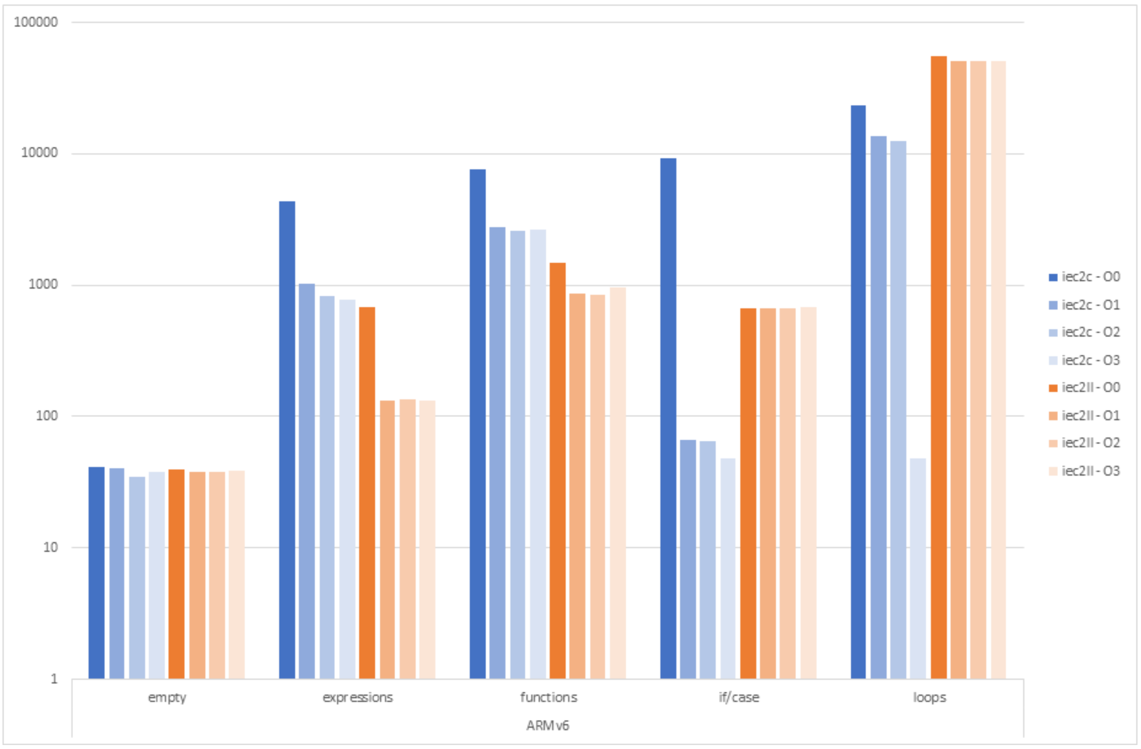


Figure 7.6: ST ARMv6 execution times (logarithmic scale)

### 7.2.2 Sequential Function Chart

In SFC programs, it is a bit harder to isolate specific constructs to test their performance, such as step evolution, transitions and actions. To do this, five different tests were defined. The first is an empty program to test the overhead a SFC execution runtimes represents. It should be slightly slower than an empty ST program but still very close.

The second test pretends to profile step evolution. This is an SFC with 1,000 steps, but with no actions. Transitions are constants, `TRUE`, which can be evaluated at compile-time.

Then, transition evaluation speed is performed, with a very similar test but where transitions are not *static* but dependent on the previous step. As an example, the transition from *step51* to *step52* shall be `step51.X`. This should not increase execution time by a significant factor but is still interesting to evaluate.

The fourth test is for action execution evaluation. The same 1,000 steps will have an action associated to it, with the `N` qualifier, a boolean variable. To ensure transition evaluation impact is minimal, the transitions are as in the second step—a compile-time constant, `TRUE`.

The fifth and final test will join every construct tested so far; it is similar to the previous test, but with dynamic transitions dependent on the previous step, like the third test.

Summarised, the tests for SFC are:

**empty sfc:** Empty SFC cycle

**steps:** SFC with 1,000 steps, no actions, *static* transitions (`TRUE`)

**transitions:** SFC with 1,000 steps, no actions, variable transitions (*previous_step.X*)

**actions:** SFC with 1,000 steps, each with a `N`-type boolean action, *static* transitions (`TRUE`)

**transitions actions:** SFC with 1,000 steps, each with a `N`-type boolean action, variable transitions (*previous_step.X*)

These tests are run by a simple C runtime, which calls the generated `__INIT` function once and then repeatedly calling the function `__BODY`.

The results of these benchmarks, with different optimisation levels (*-O0* to *-O3*), for both compilers and architectures, can be seen in Table 7.3.

For the ARM architecture, the equivalent execution times are in Table 7.4.

Again, it is clear that higher optimisation levels sometimes make the compilers remove most of the computations. This is less common as with ST constructs, which is also predictable since the SFC logic performs many more operations behind the scenes compared to ST statements execution, which makes optimisation opportunities harder for a compiler to recognize.

Inspecting the execution times, an empty SFC cycle is verified to be similar to an empty cycle, as expected, for both architectures. The execution of SFC tests with the *iec2ll* target is very significantly faster than with the older target for all tests.

Plotting these results into graphics helps visualise the difference magnitude; the execution times with optimisation disabled for both targets can be seen in Figures 7.7 and 7.8. The plotted results once again pertain only to tests done with optimisations *disabled* (*-O0*).

Table 7.3: SFC x86_64 execution times

| **Test** | **Optimisation Level** | ***iec2c*** **Execution Time** ($\mu s$) | ***iec2ll*** **Execution Time** ($\mu s$) |
|---|---|---|---|
| empty sfc | *-O0* | 0.5 | 0.6 |
| empty sfc | *-O1* | 0.6 | 0.5 |
| empty sfc | *-O2* | 0.5 | 0.6 |
| empty sfc | *-O3* | 0.5 | 0.6 |
| sfc steps | *-O0* | 15.9 | 1.3 |
| sfc steps | *-O1* | 4.8 | 0.8 |
| sfc steps | *-O2* | 12.9 | 0.8 |
| sfc steps | *-O3* | 12.1 | 0.8 |
| sfc transitions | *-O0* | 17.7 | 1.6 |
| sfc transitions | *-O1* | 4.6 | 1.5 |
| sfc transitions | *-O2* | 12.9 | 1.5 |
| sfc transitions | *-O3* | 11.7 | 1.7 |
| sfc actions | *-O0* | 38.1 | 11.9 |
| sfc actions | *-O1* | 6.2 | 3.3 |
| sfc actions | *-O2* | 18.1 | 3.5 |
| sfc actions | *-O3* | 17.8 | 3.6 |
| sfc transitions actions | *-O0* | 40.1 | 12.8 |
| sfc transitions actions | *-O1* | 5.9 | 4.3 |
| sfc transitions actions | *-O2* | 19.0 | 4.2 |
| sfc transitions actions | *-O3* | 18.9 | 4.4 |

Charts comparing the effect of the level of optimisation can be found in Figures 7.9 and 7.10 in a linear scale and in Figures 7.11 and 7.12 in a logarithmic scale.

Table 7.4: SFC ARMv6 execution times

| **Test** | **Optimisation Level** | ***iec2c* Execution Time ($\mu s$)** | ***iec2ll* Execution Time ($\mu s$)** |
|---|---|---:|---:|
| empty sfc | *-O0* | 4.6 | 3.2 |
| empty sfc | *-O1* | 3.4 | 3.5 |
| empty sfc | *-O2* | 3.9 | 3.5 |
| empty sfc | *-O3* | 3.3 | 3.5 |
| sfc steps | *-O0* | 1393.6 | 120.3 |
| sfc steps | *-O1* | 1097.8 | 112.3 |
| sfc steps | *-O2* | 984.9 | 100.9 |
| sfc steps | *-O3* | 908.8 | 16.7 |
| sfc transitions | *-O0* | 1417.8 | 251.6 |
| sfc transitions | *-O1* | 1121.3 | 195.4 |
| sfc transitions | *-O2* | 914.5 | 200.2 |
| sfc transitions | *-O3* | 971.1 | 176.2 |
| sfc actions | *-O0* | 2492.3 | 1450.3 |
| sfc actions | *-O1* | 1455.6 | 875.1 |
| sfc actions | *-O2* | 1208.6 | 911.7 |
| sfc actions | *-O3* | 1226.8 | 683.5 |
| sfc transitions actions | *-O0* | 2476.2 | 1539.3 |
| sfc transitions actions | *-O1* | 1447.3 | 948.9 |
| sfc transitions actions | *-O2* | 1196.2 | 960.3 |
| sfc transitions actions | *-O3* | 1205.3 | 955.1 |

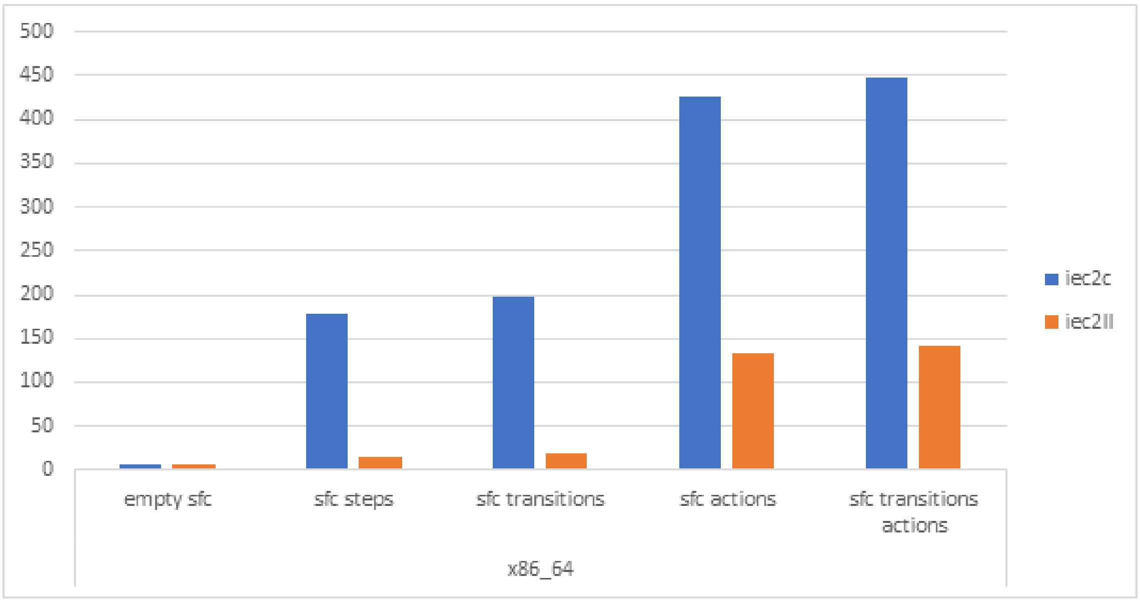


Figure 7.7: SFC x86_64 execution times (-*O0*)

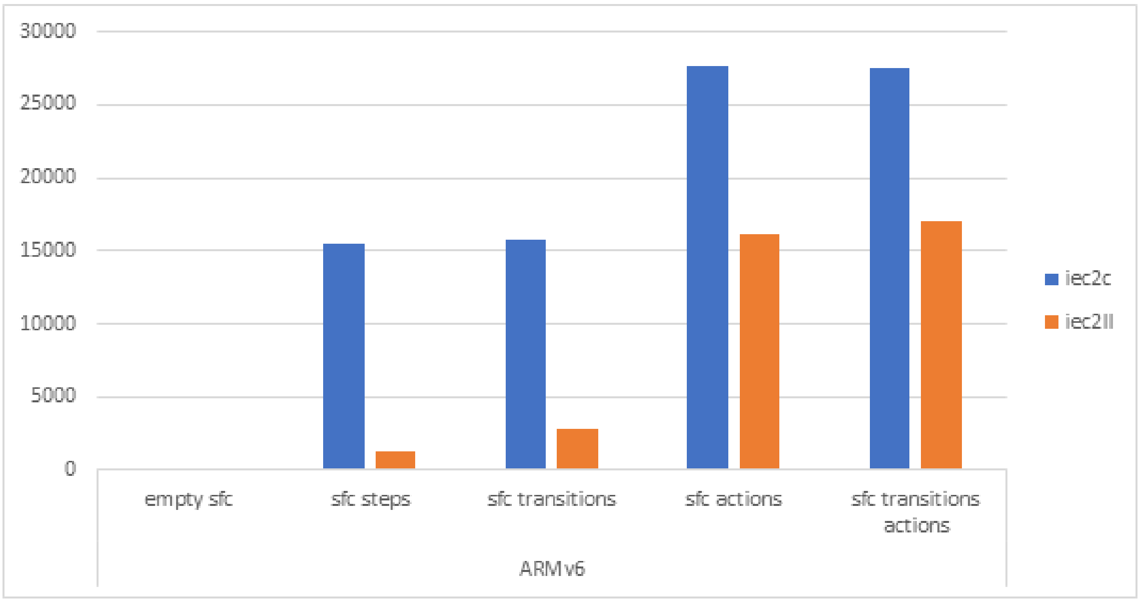


Figure 7.8: SFC ARMv6 execution times (-*O0*)

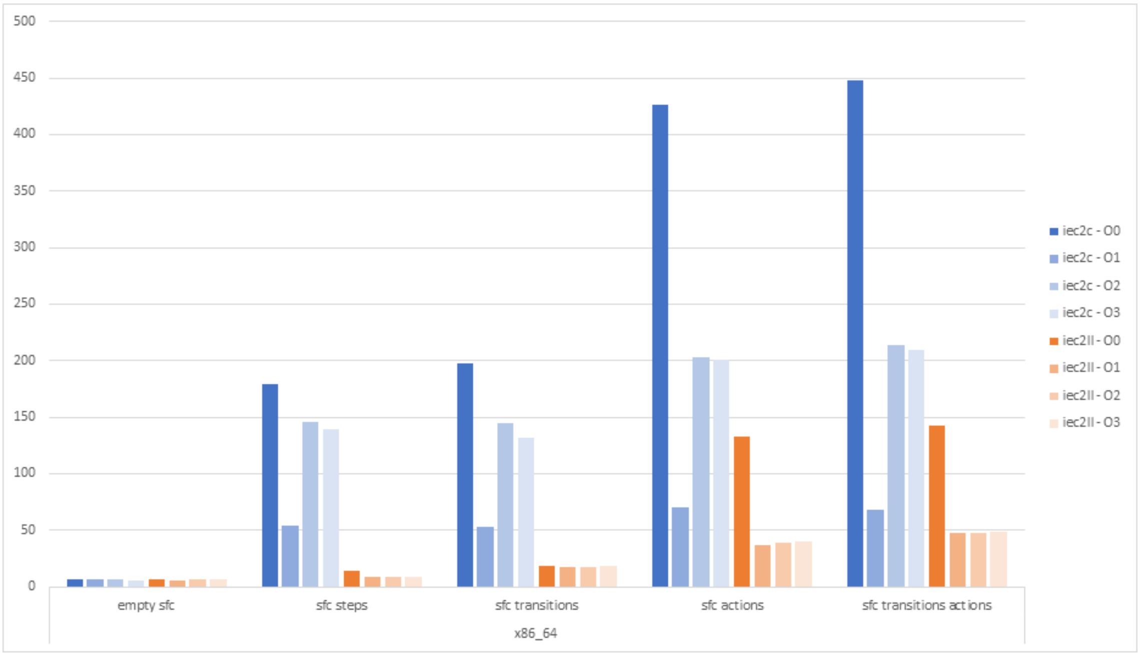


Figure 7.9: SFC x86_64 execution times

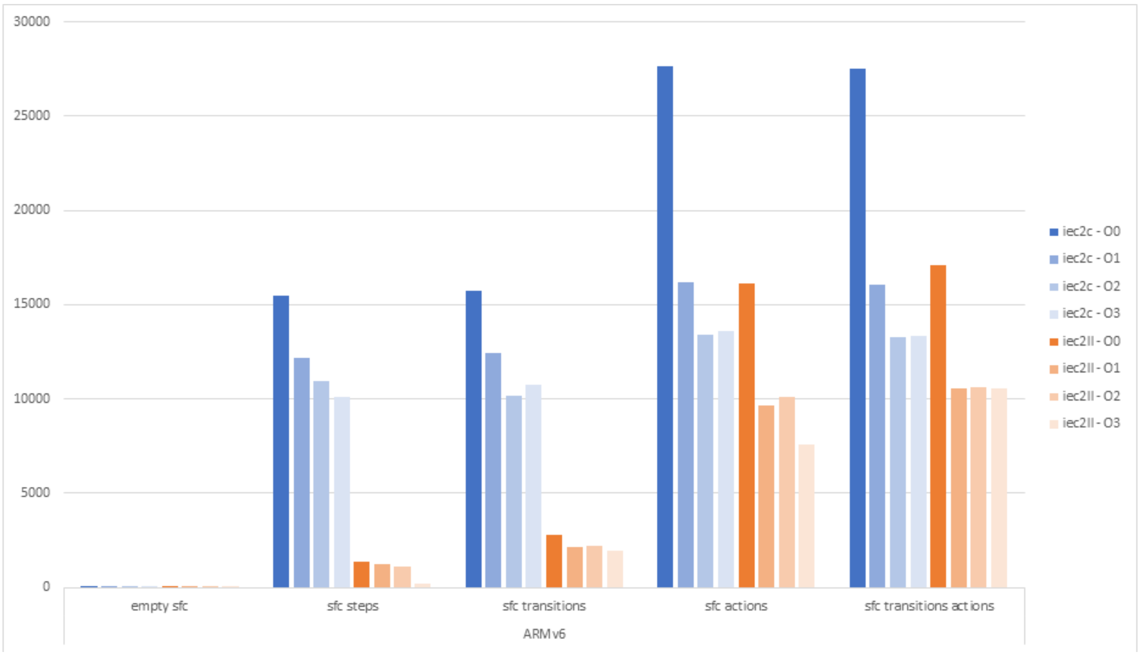


Figure 7.10: SFC ARMv6 execution times

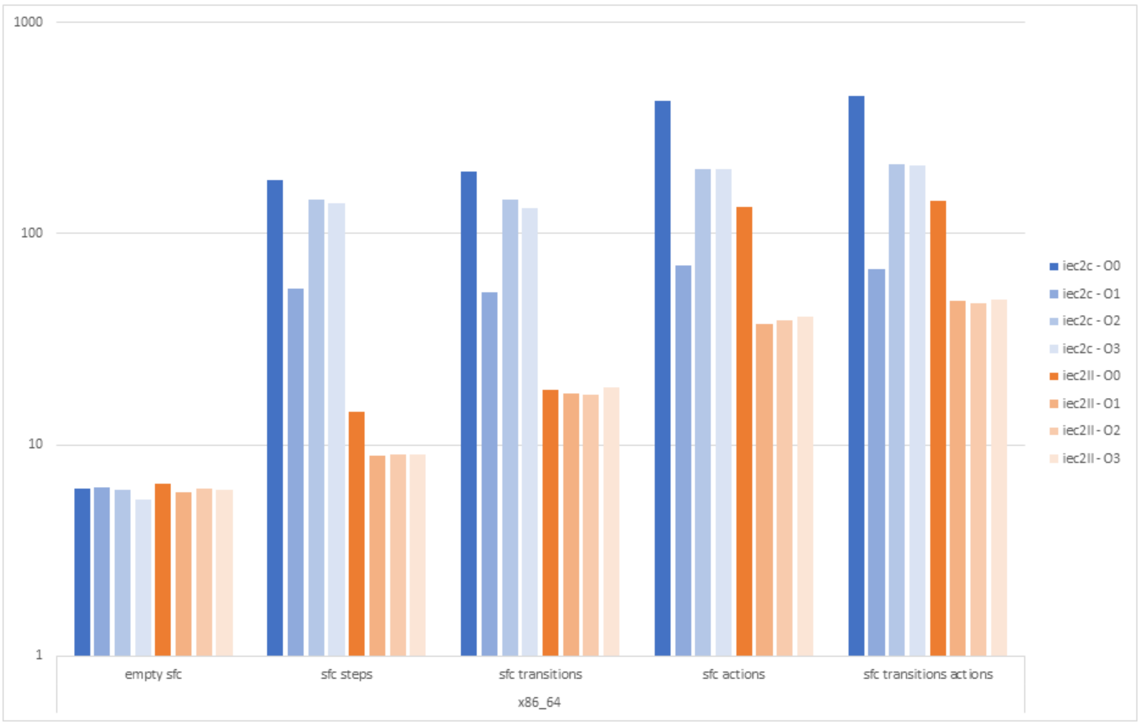


Figure 7.11: SFC x86_64 execution times (logarithmic scale)

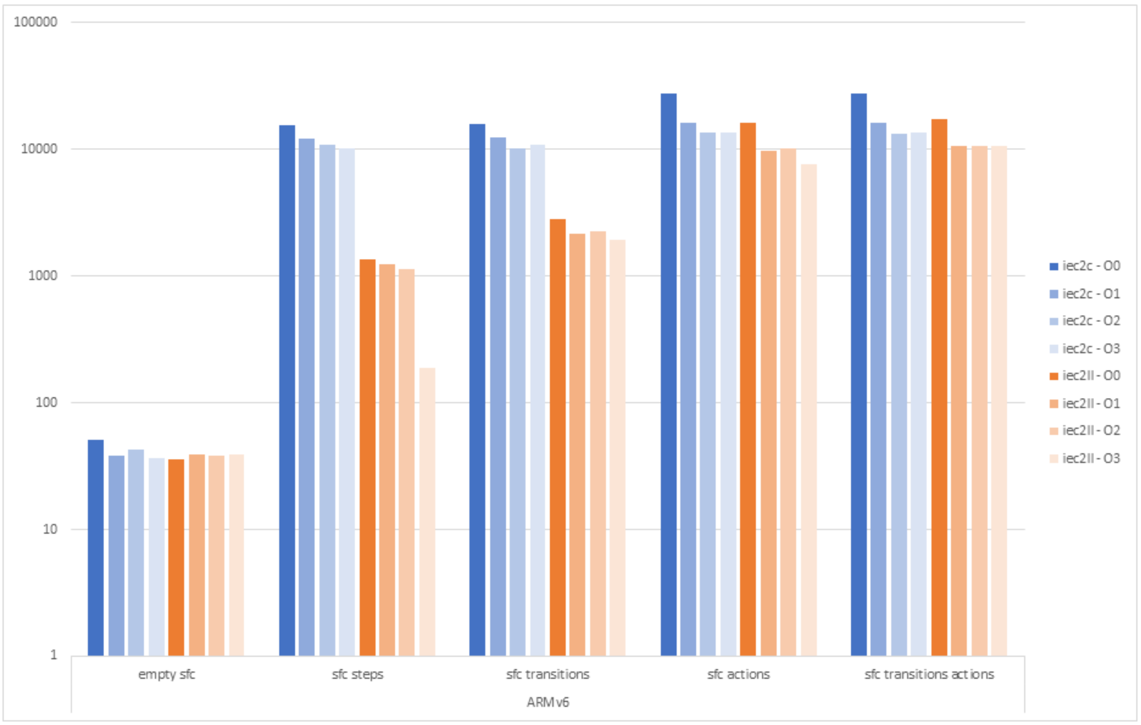


Figure 7.12: SFC ARMv6 execution times (logarithmic scale)

### 7.2.3 Codesys

The obtained results are promising, but we still need to understand how the new target behaves compared to a production compiler. For this purpose, a platform-independent development environment is best as it is not tied to a particular vendor or system; also, runtime support for at least one of the systems already used for performance testing is a must so we can make an adequate comparison.

The Codesys programming environment was the chosen controller development system. It is fairly common and used so a more realistic comparison can be made and since it is IEC 61131-3-compliant (with a few extensions), we can run the same tests with little to no modification. Codesys has a runtime for the Raspberry Pi board and the latest version (v3.5.16.0) was used.

One problem is that the Codesys environment has a massive cycle overhead because of its connection to the development environment itself (150–170 $\mu$s compared to 3–4 $\mu$s for *iec2c* and *iec2ll*). It has a lot of useful functionality such as live debugging and controlling program execution parameters and flow, but this makes the comparison unfair, as the simple C implemented runtime we used earlier would be several orders of magnitude faster than the Codesys runtime, as they do not come close in functionality and cannot be compared. To address this problem, for both languages, the empty cycle time was taken and averaged for 100,000 runs; then, each test was also run 100,000 times, the execution times averaged and the mean empty cycle time subtracted. This balances the comparison between a MatIEC target and Codesys because the overhead of each one is subtracted from its test results.

The Codesys environment does not specify how IEC 61131-3 programs are compiled (or interpreted, but that is unlikely) or any other detail of that matter. Compilation flags and optimisation levels are not visible to the user; for this reason, comparisons with the MatIEC targets were only done with optimisations *disabled*.

The results (with optimisations *disabled* for MatIEC targets) can be found in Table 7.5[2]. To help visualisation, the same results are plotted in Figures 7.13 and 7.14.

Comparison of ST tests execution times show that the *iec2c* MatIEC target is similar to Codesys, with Codesys being faster only in the *expressions* test; in which Codesys execution time is very similar to the *iec2ll* target. In all but the *loops* test, the new MatIEC target is faster than the other compilers, sometimes by a very significant margin. As is the case of the previous analysis with only the other MatIEC target, in the *loops* benchmark, the new target is much slower; either due to a poor loop implementation (which should also affect the x86_64 architecture) or a limitation of LLVM's ARM back-end.

Regarding SFC testing, when no actions are present (*steps* and *transitions* tests), *iec2ll* is much faster than Codesys and *iec2c*, which are similar. With actions present in the SFC, Codesys and *iec2ll* are very similar and *iec2c* is a bit slower.

[2]The results for MatIEC targets are not the same as previously presented; in these, the empty cycle mean time is substracted from the results and as such, the empty ST and SFC tests' results are omitted as they would be equal to 0

Table 7.5: MatIEC and Codesys execution times comparison

| **Test** | ***iec2c*** **Execution Time ($\mu s$)** | ***iec2ll*** **Execution Time ($\mu s$)** | **Codesys Execution Time ($\mu s$)** |
|---|---|---|---|
| expressions | 388.9 | 57.3 | 217 |
| functions | 680.6 | 128.5 | 748 |
| if/case | 834.5 | 56.7 | 375 |
| loops | 2118.7 | 5037.1 | 935 |
| sfc steps | 1389.0 | 117.1 | 1270 |
| sfc transitions | 1413.2 | 248.4 | 1217 |
| sfc actions | 2487.7 | 1447.1 | 1454 |
| sfc transitions actions | 2471.6 | 1536.1 | 1416 |

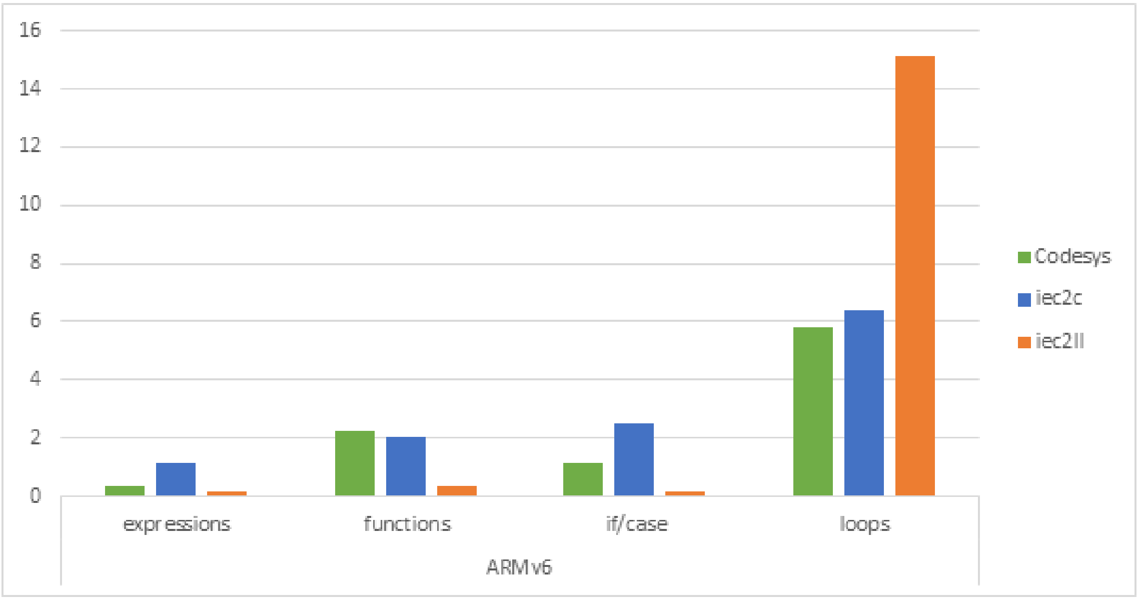


Figure 7.13: ST Codesys and MatIEC (-*O0*) execution times comparison

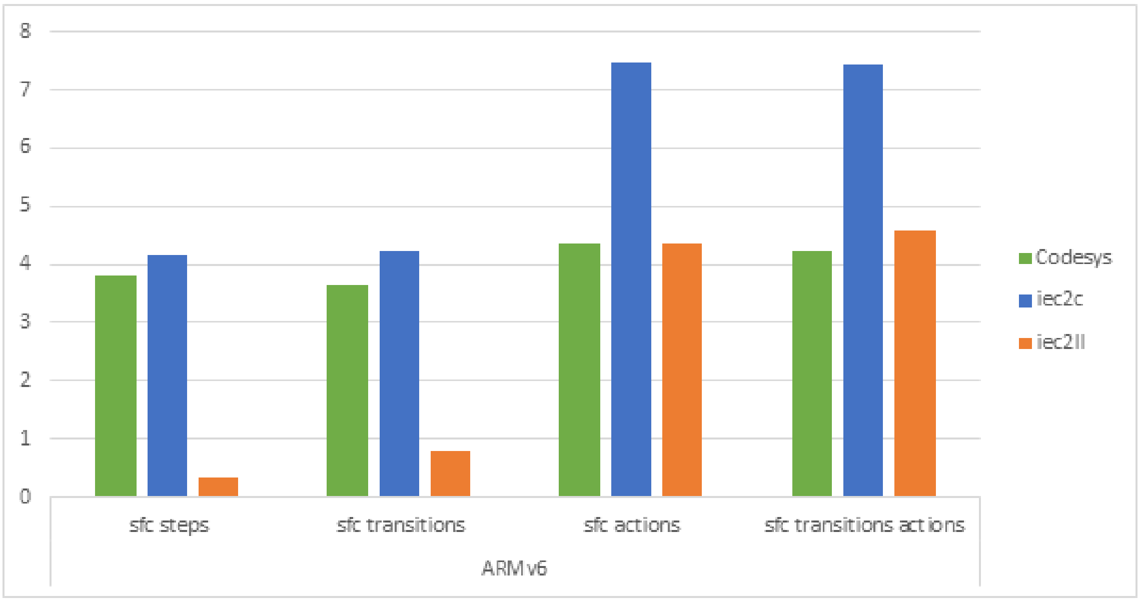


Figure 7.14: SFC Codesys and MatIEC (*-O0*) execution times comparison

These are very promising results. Being on par with a production compiler, which is mature and is used in real applications, is not expected since the *iec2ll* target has a lot of room for improvement and this is only an initial implementation and a proof of concept. Many things can and should be refined, IR code generation for loop constructs as an example, and are yet to be implemented, such as derived data types. Nonetheless, these are impressive results for such an unfinished and unoptimised code generator.

# Chapter 8

# Conclusions and Future Work

The implemented LLVM-based MatIEC target supports the ST and SFC languages and is in a working state. Some of the IEC 61131-3 Common Elements are not yet supported, as has been already explained, but the majority of the constructs are. This initial implementation is able to compile most of the commonly used ST and SFC constructs, meaning the primary goal of this dissertation was achieved.

Extensive testing for correctness to ensure the implementation is correct and compliant to the IEC standard was not done; the correctness testing was superficial and while implementation errors were not detected, they might (and most likely) exist, especially for edge cases. This is especially true for SFC programs, as their structure and logic impose greater obstacles for correctness testing.

Nonetheless, the summary correctness testing for both ST and SFC has not revealed implementation errors in the final version of this target, for the implemented constructs. There are many opportunities for improvement and optimisation in this initial implementation, but, as a proof of concept, this is working as intended.

The development of this project led to the submission of a Work-in-Progress paper for the IEEE's 25th Emerging Technologies and Factory Automation (ETFA) 2020 international conference detailing the performance comparison of the new MatIEC target against both the older target and Codesys.

## 8.1 Results and Conclusions

The results of performance comparison, both with the current MatIEC target, *iec2c*, and with a widely adopted proprietary environment Codesys are very promising for such an initial implementation. The *iec2ll* target proved to be on par with, and even out-performing in many situations, the other compilers. This is especially true for the *iec2c* target and provides the most interesting comparison: they both share the same compiler front-end, AST structure, implementation architecture, but the end results are consistently better by a large margin with the new target.

In the ARM architecture, as seen, loops implementation is more than twice as slow as with *iec2c*. This test was performed several times to ensure it was not a one-time anomaly but a consistent result. Different optimisation levels have little impact, but it is consistent with their impact on the same test with the x86_64 architecture.

A few possibilities arise: either it is an implementation with very poor performance, a problem with LLVM's ARM back-end, or an optimisation issue. The first possibility should similarly affect the x86_64 architecture, although maybe not in the same proportion as with ARM since this system is much faster, but at least the same tendency should be noticeable on both architectures. Since this is not the case, this option is not very likely the cause of this problem.

The second hypothesis is a poorer LLVM's ARM back-end and architecture-specific code generation compared to the GCC compiler. This is also not very likely, as ARM is a very widely used CPU architecture, but repeating the tests using an LLVM-based C compiler for the *iec2c* target should remove the doubt.

The third option, and the most likely in the author's opinion, is a problem with optimisation. The ARM architecture varies widely between devices as it is very flexible and extensible. Multiple vendors exist producing their own different chip versions, with multiple revisions and micro-architectures. Since each revision may have slightly different ISA extensions (on top of a common ARM ISA), a compiler back-end must know exactly which CPU revision it is targeting. If a generic ARM target is indicated, the compiler is still able to generate executable and correct code, but it cannot issue more performant, microarchitecture-specific, instructions and having to restrict itself to the common instructions, which most of the time are slower than the more specific ones.

The performance tests were cross-compiled in the x86_64 machine to the ARM architecture for both targets, passing the target triplet (or triple) as an argument for the compiler, since they would take a lot of time to compile in such a slow device. Most likely, this is where the performance difference originates; maybe the LLVM IR to object code compiler/assembler, *llc*, is more conservative in issued instruction choices when cross-compiling and not having full knowledge of processor capabilities.

Not giving much importance to that specific test due to the explained reasons, the performance of the new target is very good, consistently matching or outperforming other IEC 61131-3 compilers. This is impressive for such an unoptimised and initial implementation and only begins to show the advantages and possibilities of direct code generation.

## 8.2 Future Work

As already explained throughout this document, a few concepts were not implemented in this initial version of the MatIEC target. This is part of future work to achieve a complete and standard-compliant IEC 61131-3 compiler.

The Common Elements derived data types, the non-opaque interface implementation for SFC POUs, and improving the `CASE` selection statement performance by using a jump table for numeric cases and the current one for complex data types are examples of future work that were already mentioned and explained;

Since the previous target also supports IL instructions, it is also desirable to add support for them in this new target;
Verifying that conversion of the graphical languages (FBD and LD) to textual languages such as ST works as intended is also necessary to announce support for them;
For the sake of standards compliance and interoperability with other development environments, it is desirable to add support for this IEC 61131-10 XML format, by writing a compiler back-end that generates this format instead of machine code. It is also possible to add a compiler front-end for the IEC 61131-10 XML to allow the compilation of programs written in any development environment. This also provides a great compiler testing possibility: compiling an XML source to an XML target will ease the identification of compiler bugs—without optimisations, the source and target file should be equivalent and almost the same.

All these improvements to the compiler will make it more mature and since they are useful for developers, their implementation might help MatIEC adoption in other systems.

More extensive implementation correctness and performance testing is also left as future work. It is of significant importance that a compiler implementation of a language is deemed correct if it is to be used in a serious environment.

# Appendix A

# SFC Action Control

```
                                                         +---+
   +----------------------------------------------------O| & |---Q
   |                                            +-----+  |   |
N--|--------------------------------------------| >=1 |--|   |
   |                      S_FF                  |     |  +---+
R--+                     +----+                 |     |
   |                     | RS |                 |     |
S--|---------------------|S Q1|-----------------|     |
   +---------------------|R1  |                 |     |
   |                     +----+  +---+          |     |
L--|---------+-------------------| & |----------|     |
   |         |      L_TMR    +--O|   |          |     |
   |         |     +-----+   |   +---+          |     |
   |         |     | TON |   |                  |     |
   |         +-----|IN  Q|---+      D_TMR       |     |
   |  +------------|PT   |         +-----+      |     |
   |  |            +-----+         | TON |      |     |
D--|--|----------------------------|IN  Q|------|     |
   |  +----------------------------|PT   |      |     |
   |  |             P_TRIG         +-----+      |     |
   |  |           +--------+                    |     |
   |  |           | R_TRIG |                    |     |
P--|--|-----------|CLK    Q|--------------------|     |
   |  |    SD_FF  +--------+   SD_TMR           |     |
   |  |   +----+               +-----+          |     |
   |  |   | RS |               | TON |          |     |
SD-|--|---|S Q1|---------------|IN  Q|----------|     |
   +--|---|R1  |  +------------|PT   |          |     |
   |  |   +----+  |   DS_TMR   +-----+  DS_FF   |     |
   |  +-----------+   +-----+          +----+   |     |
   |  |               | TON |          | RS |   |     |
DS-|--|---------------|IN  Q|----------|S Q1|---|     |
   |  +---------------|PT   |      +---|R1  |   |     |
   |  |               +-----+      |   +----+   |     |
   +--|----------------------------+            |     |
   |  |        SL_FF                            |     |
   |  |       +----+                            |     |
   |  |       | RS |                     +---+  |     |
SL-|--|-------|S Q1|--+------------------| & |--|     |
   +--|-------|R1  |  |     SL_TMR   +--O|   |  +-----+
      |       +----+  |    +-----+   |   +---+
      |               |    | TON |   |
      |               +----|IN  Q|---+                    +-----+
T-----+--------------------|PT   |      +--------+        | >=1 |
                           +-----+      | F_TRIG |    Q---|     |---A
               +--------+           Q---|CLK    Q|--------|     |
               | R_TRIG |               +--------+        |     |
P1-------------|CLK    Q|---------------------------------|     |
               +--------+     +--------+                  |     |
                              | F_TRIG |                  |     |
P0----------------------------|CLK    Q|------------------|     |
                              +--------+                  +-----+
```

Figure A.1: ACTION_CONTROL function block body with "final scan" logic

```
                                                          +---+
   +-----------------------------------------------------O| & |---Q
   |                                             +-----+  |   |
N--|---------------------------------------------| >=1 |--|   |
   |                       S_FF                  |     |  +---+
R--+                      +----+                 |     |
   |                      | RS |                 |     |
S--|----------------------|S Q1|-----------------|     |
   +----------------------|R1  |                 |     |
   |                      +----+  +---+          |     |
L--|---------+--------------------| & |----------|     |
   |         |       L_TMR    +--O|   |          |     |
   |         |      +-----+   |   +---+          |     |
   |         |      | TON |   |                  |     |
   |         +------|IN  Q|---+      D_TMR       |     |
   |  +-------------|PT   |         +-----+      |     |
   |  |             +-----+         | TON |      |     |
D--|--|-----------------------------|IN  Q|------|     |
   |  +-----------------------------|PT   |      |     |
   |  |              P_TRIG         +-----+      |     |
   |  |            +--------+                    |     |
   |  |            | R_TRIG |                    |     |
P--|--|------------|CLK    Q|--------------------|     |
   |  |    SD_FF   +--------+   SD_TMR           |     |
   |  |   +----+                +-----+          |     |
   |  |   | RS |                | TON |          |     |
SD-|--|---|S Q1|----------------|IN  Q|----------|     |
   +--|---|R1  |   +------------|PT   |          |     |
   |  |   +----+   |   DS_TMR   +-----+  DS_FF   |     |
   |  +------------+   +-----+          +----+   |     |
   |  |                | TON |          | RS |   |     |
DS-|--|----------------|IN  Q|----------|S Q1|---|     |
   |  +----------------|PT   |      +---|R1  |   |     |
   |  |                +-----+      |   +----+   |     |
   +--|-----------------------------+            |     |
   |  |         SL_FF                            |     |
   |  |        +----+                            |     |
   |  |        | RS |                     +---+  |     |
SL-|--|--------|S Q1|--+------------------| & |--|     |
   +--|--------|R1  |  |     SL_TMR   +--O|   |  |     |
      |        +----+  |    +-----+   |   +---+  |     |
      |                |    | TON |   |          |     |
      |                +----|IN  Q|---+          |     |
T-----+---------------------|PT   |              |     |
          +--------+        +-----+              |     |
          | R_TRIG |                             |     |
P1--------|CLK    Q|-----------------------------|     |
          +--------+     +--------+              |     |
                         | F_TRIG |              |     |
P0-----------------------|CLK    Q|--------------|     |
                         +--------+              +-----+
```

Figure A.2: ACTION_CONTROL function block body without "final scan" logic

# Appendix B

# LLVM API Example Usage

```
#include "llvm/Analysis/Verifier.h"
#include "llvm/Assembly/PrintModulePass.h"
#include "llvm/CallingConv.h"
#include "llvm/Function.h"
#include "llvm/Module.h"
#include "llvm/PassManager.h"
#include "llvm/Support/IRBuilder.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

Module* makeLLVMModule() {
    // Module Construction
    Module* mod = new Module("test", getGlobalContext());

    Constant* c =
      mod->getOrInsertFunction("mul_add",
                                /*ret type*/ IntegerType::get(32),
                                /*args*/ IntegerType::get(32),
                                IntegerType::get(32),
                                IntegerType::get(32),
                                /*varargs terminated with null*/ NULL);

    Function* mul_add = cast<Function>(c);
    mul_add->setCallingConv(CallingConv::C);

    Function::arg_iterator args = mul_add->arg_begin();
    Value* x = args++;
    x->setName("x");
    Value* y = args++;
    y->setName("y");
    Value* z = args++;
    z->setName("z");

    BasicBlock* block =
      BasicBlock::Create(getGlobalContext(), "entry", mul_add);
    IRBuilder<> builder(block);

    Value* tmp = builder.CreateBinOp(Instruction::Mul, x, y, "tmp");
    Value* tmp2 = builder.CreateBinOp(Instruction::Add, tmp, z, "tmp2");

    builder.CreateRet(tmp2);

    return mod;
}

int main(int argc, char** argv) {
    Module* Mod = makeLLVMModule();

    verifyModule(*Mod, PrintMessageAction);

    PassManager PM;
    PM.add(createPrintModulePass(&outs()));
    PM.run(*Mod);

    delete Mod;
    return 0;
}
```

Appendix C

# SFC LLVM IR Generation Example

```
PROGRAM SFC

    VAR
        x : BOOL;
    END_VAR



    INITIAL_STEP START: END_STEP
    STEP END: x(N); END_STEP

    TRANSITION FROM START TO END := START.X; END_TRANSITION


END_PROGRAM
```

Listing C.1: SFC example program

## Generated LLVM IR for the SFC example program

```
; ModuleID = 'module'
source_filename = "module"

%sfc__step = type { i1, i64 }
%sfc__action = type { i1, i1, i1, i1, i1, i1, i1, i1, i1, i1, i1, i1,
    i1, i1, i1 }
%SFC = type { i1 }

@STEP__START = external global %sfc__step
@STEP__END = external global %sfc__step
@ACTION__END__x = external global %sfc__action
define void @SFC__BODY(%SFC* %pou_state_var__) {
init__:
  %x = getelementptr %SFC, %SFC* %pou_state_var__, i32 0, i32 0
  br label %main__

main__:                                ; preds = %init__
  %0 = load i1, i1* getelementptr inbounds (%sfc__step, %sfc__step*
      @STEP__START, i32 0, i32 0)
  br i1 %0, label %trans, label %end_trans

ret__:                                 ; preds = %end_action
  ret void

trans:                                 ; preds = %main__
  store i1 false, i1* getelementptr inbounds (%sfc__step, %sfc__step*
      @STEP__START, i32 0, i32 0)
  store i1 true, i1* getelementptr inbounds (%sfc__step, %sfc__step*
      @STEP__END, i32 0, i32 0)
  store i64 0, i64* getelementptr inbounds (%sfc__step, %sfc__step*
      @STEP__END, i32 0, i32 1)
  br label %end_trans

end_trans:                             ; preds = %trans, %main__
  %1 = load i1, i1* getelementptr inbounds (%sfc__step, %sfc__step*
      @STEP__END, i32 0, i32 0)
  br i1 %1, label %action, label %end_action

action:                                ; preds = %end_trans
  %2 = load i1, i1* getelementptr inbounds (%sfc__action, %sfc__action*
       @ACTION__END__x, i32 0, i32 0)
  %3 = load i1, i1* getelementptr inbounds (%sfc__action, %sfc__action*
       @ACTION__END__x, i32 0, i32 1)
  %4 = load i1, i1* getelementptr inbounds (%sfc__action, %sfc__action*
      @ACTION__END__x, i32 0, i32 2)
  %5 = load i1, i1* getelementptr inbounds (%sfc__action, %sfc__action*
      @ACTION__END__x, i32 0, i32 3)
  %6 = load i1, i1* getelementptr inbounds (%sfc__action, %sfc__action*
      @ACTION__END__x, i32 0, i32 4)
  %7 = load i1, i1* getelementptr inbounds (%sfc__action, %sfc__action*
      @ACTION__END__x, i32 0, i32 5)
  %8 = load i1, i1* getelementptr inbounds (%sfc__action, %sfc__action*
      @ACTION__END__x, i32 0, i32 6)
  %9 = load i1, i1* getelementptr inbounds (%sfc__action, %sfc__action*
      @ACTION__END__x, i32 0, i32 7)
  %10 = load i1, i1* getelementptr inbounds (%sfc__action, %sfc__action
     * @ACTION__END__x, i32 0, i32 8)
  %11 = load i1, i1* getelementptr inbounds (%sfc__action, %sfc__action
     * @ACTION__END__x, i32 0, i32 9)
  %12 = load i1, i1* getelementptr inbounds (%sfc__action, %sfc__action
     * @ACTION__END__x, i32 0, i32 10)
  %13 = load i1, i1* getelementptr inbounds (%sfc__action, %sfc__action
     * @ACTION__END__x, i32 0, i32 11)
  %14 = load i1, i1* getelementptr inbounds (%sfc__action, %sfc__action
     * @ACTION__END__x, i32 0, i32 12)
  %15 = load i1, i1* getelementptr inbounds (%sfc__action, %sfc__action
     * @ACTION__END__x, i32 0, i32 13)
  %16 = load i1, i1* getelementptr inbounds (%sfc__action, %sfc__action
     * @ACTION__END__x, i32 0, i32 14)
  %17 = or i1 %4, %13
  %18 = xor i1 %3, true
  %19 = and i1 %18, %17
  %20 = xor i1 %14, true
  %21 = and i1 %20, %7
  %22 = xor i1 %11, true
  %23 = xor i1 %15, true
  %24 = and i1 %23, %22
  %25 = xor i1 %16, true
  %26 = and i1 %25, %12
  %27 = xor i1 %3, true
  %28 = or i1 %2, %19
  %29 = or i1 %28, %21
  %30 = or i1 %29, %24
  %31 = or i1 %30, %26
  %32 = and i1 %31, %27
  br i1 %32, label %actionb, label %end_action

actionb:                                  ; preds = %action
```

```
  store i1 true, i1* %x
  br label %end_action

end_action:                              ; preds = %actionb, %action, %
    end_trans
  %33 = load i1, i1* getelementptr inbounds (%sfc__action, %sfc__action
      * @ACTION__END__x, i32 0, i32 0)
  %34 = load i1, i1* getelementptr inbounds (%sfc__action, %sfc__action
      * @ACTION__END__x, i32 0, i32 1)
  %35 = load i1, i1* getelementptr inbounds (%sfc__action, %sfc__action
      * @ACTION__END__x, i32 0, i32 2)
  %36 = load i1, i1* getelementptr inbounds (%sfc__action, %sfc__action
      * @ACTION__END__x, i32 0, i32 3)
  %37 = load i1, i1* getelementptr inbounds (%sfc__action, %sfc__action
      * @ACTION__END__x, i32 0, i32 4)
  %38 = load i1, i1* getelementptr inbounds (%sfc__action, %sfc__action
      * @ACTION__END__x, i32 0, i32 5)
  %39 = load i1, i1* getelementptr inbounds (%sfc__action, %sfc__action
      * @ACTION__END__x, i32 0, i32 6)
  %40 = load i1, i1* getelementptr inbounds (%sfc__action, %sfc__action
      * @ACTION__END__x, i32 0, i32 7)
  %41 = load i1, i1* getelementptr inbounds (%sfc__action, %sfc__action
      * @ACTION__END__x, i32 0, i32 8)
  %42 = load i1, i1* getelementptr inbounds (%sfc__action, %sfc__action
      * @ACTION__END__x, i32 0, i32 9)
  %43 = load i1, i1* getelementptr inbounds (%sfc__action, %sfc__action
      * @ACTION__END__x, i32 0, i32 10)
  store i1 %38, i1* getelementptr inbounds (%sfc__action, %sfc__action*
       @ACTION__END__x, i32 0, i32 12)
  %44 = xor i1 %42, true
  store i1 %44, i1* getelementptr inbounds (%sfc__action, %sfc__action*
       @ACTION__END__x, i32 0, i32 13)
  store i1 %43, i1* getelementptr inbounds (%sfc__action, %sfc__action*
       @ACTION__END__x, i32 0, i32 14)
  br label %ret__
}

define void @SFC__INIT() {
main__:
  store i1 true, i1* getelementptr inbounds (%sfc__step, %sfc__step*
      @STEP__START, i32 0, i32 0)
  store i1 false, i1* getelementptr inbounds (%sfc__step, %sfc__step*
      @STEP__END, i32 0, i32 0)
  store i1 false, i1* getelementptr inbounds (%sfc__action, %
      sfc__action* @ACTION__END__x, i32 0, i32 0)
  store i1 false, i1* getelementptr inbounds (%sfc__action, %
     sfc__action* @ACTION__END__x, i32 0, i32 1)
  store i1 false, i1* getelementptr inbounds (%sfc__action, %
     sfc__action* @ACTION__END__x, i32 0, i32 2)
  store i1 false, i1* getelementptr inbounds (%sfc__action, %
     sfc__action* @ACTION__END__x, i32 0, i32 3)
  store i1 false, i1* getelementptr inbounds (%sfc__action, %
     sfc__action* @ACTION__END__x, i32 0, i32 4)
  store i1 false, i1* getelementptr inbounds (%sfc__action, %
     sfc__action* @ACTION__END__x, i32 0, i32 5)
  store i1 false, i1* getelementptr inbounds (%sfc__action, %
     sfc__action* @ACTION__END__x, i32 0, i32 6)
  store i1 false, i1* getelementptr inbounds (%sfc__action, %
     sfc__action* @ACTION__END__x, i32 0, i32 7)
  store i1 false, i1* getelementptr inbounds (%sfc__action, %
     sfc__action* @ACTION__END__x, i32 0, i32 8)
  store i1 false, i1* getelementptr inbounds (%sfc__action, %
     sfc__action* @ACTION__END__x, i32 0, i32 9)
  store i1 false, i1* getelementptr inbounds (%sfc__action, %
     sfc__action* @ACTION__END__x, i32 0, i32 10)
  store i1 false, i1* getelementptr inbounds (%sfc__action, %
     sfc__action* @ACTION__END__x, i32 0, i32 11)
  store i1 false, i1* getelementptr inbounds (%sfc__action, %
     sfc__action* @ACTION__END__x, i32 0, i32 12)
  store i1 true, i1* getelementptr inbounds (%sfc__action, %sfc__action
     * @ACTION__END__x, i32 0, i32 13)
  store i1 false, i1* getelementptr inbounds (%sfc__action, %
     sfc__action* @ACTION__END__x, i32 0, i32 14)
  store i1 true, i1* getelementptr inbounds (%sfc__action, %sfc__action
     * @ACTION__END__x, i32 0, i32 0)
  br label %ret__

ret__:                                            ; preds = %main__
  ret void
}

!llvm.dbg.cu = !{!0}

!0 = distinct !DICompileUnit(language: DW_LANG_C_plus_plus, file: !1,
    producer: "matiec iec2ll", isOptimized: false, runtimeVersion: 1,
    emissionKind: FullDebug, enums: !2)
!1 = !DIFile(filename: "standard_functions.txt", directory: "")
!2 = !{}
```

Listing C.2: SFC example program generated LLVM IR

# References

[1] International Electrotechnical Commission. IEC 61131-3:2013 International Standard. Technical report, IEC, 2013.

[2] Eli Bendersky. Abstract vs. Concrete Syntax Trees. https://eli.thegreenplace.net/2009/02/16/abstract-vs-concrete-syntax-trees, 2009.

[3] Chris Lattner. LLVM. In *The Architecture of Open Source Applications*, chapter 11. lulu.com, 2012.

[4] International Electrotechnical Commission. IEC - International Electrotechnical Commission Website. https://www.iec.ch/.

[5] International Electrotechnical Commission. IEC 61131-1:2003 International Standard. https://webstore.iec.ch/publication/4550.

[6] International Electrotechnical Commission. IEC TR 61131-8:2017 International Standard. https://webstore.iec.ch/publication/33021.

[7] International Electrotechnical Commission. IEC 61131-10:2019 International Standard. https://webstore.iec.ch/publication/33034.

[8] Mario de Sousa. On analyzing the semantics of IEC 61131-3 ST and IL applications. 7:559–571, 2013.

[9] Bruno Goncalves Silva and Mario De Sousa. Internal inconsistencies in the third edition of the IEC 61131-3 international standard. In *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, volume 2016-Novem. Institute of Electrical and Electronics Engineers Inc., nov 2016.

[10] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, and Alfred V. Aho. *Compilers : principles, techniques, & tools*. Pearson Education, Inc, 2 edition, 2006.

[11] Ray Toal. LMU CMSI 488 Compiler Architecture. Technical report, Loyola Marymount University.

[12] LLVM Developer Group. The LLVM Compiler Infrastructure Project Website. https://llvm.org/.

[13] Chris Lattner and Vikram Adve. The LLVM compiler framework and infrastructure tutorial. In *Lecture Notes in Computer Science*, volume 3602, pages 15–16, 2005.

[14] LLVM Developer Group. Reference — LLVM 10 documentation. https://llvm.org/docs/Reference.html.

[15] LLVM Developer Group. LLVM: A First Function. http://releases.llvm.org/2.6/docs/tutorial/JITTutorial1.html.

[16] Edouard Tisserant, Laurent Bessard, and Mário De Sousa. An open source IEC 61131-3 integrated development environment. In *IEEE International Conference on Industrial Informatics (INDIN)*, volume 1, pages 183–187, 2007.

[17] M. De Sousa and A. Carvalho. An IEC 61131-3 compiler for the MatPLC. In *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, volume 1, pages 485–490. Institute of Electrical and Electronics Engineers Inc., 2003.

[18] Bruno Gonçalves Silva. Compiler Front-end for the IEC 61131-3 v3 Languages, feb 2016.

[19] Dariusz Rzoñca, Jan Sadolewski, and Bartosz Trybus. Prototype environment for controller programming in the IEC 61131-3 ST language. *Computer Science and Information Systems*, 4(2):131–146, 2007.

[20] Bartosz Trybus. Development and Implementation of IEC 61131-3 Virtual Machine. *Theoretical and Applied Informatics*, 23(1):21–35, jul 2011.

[21] Zhou Chunjie and Chen Hui. Development of a PLC virtual machine orienting IEC 61131-3 standard. In *2009 International Conference on Measuring Technology and Mechatronics Automation, ICMTMA 2009*, volume 3, pages 374–379, 2009.

[22] Minghui Zhang, Yanxia Lu, and Tianjiao Xia. The design and implementation of virtual machine system in embedded softplc system. In *Proceedings - 2013 International Conference on Computer Sciences and Applications, CSA 2013*, pages 775–778. IEEE Computer Society, 2013.

[23] Salvatore Cavalieri, Giuseppe Puglisi, Marco Stefano Scroppo, and Luca Galvagno. Moving IEC 61131-3 applications to a computing framework based on CLR Virtual Machine. In *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, volume 2016-Novem, pages 1–8. Institute of Electrical and Electronics Engineers Inc., nov 2016.

[24] Salvatore Cavalieri, Marco Stefano Scroppo, and Luca Galvagno. A framework based on CLR virtual machine to deploy IEC 61131-3 programs. In *IEEE International Conference on Industrial Informatics (INDIN)*, pages 126–131. Institute of Electrical and Electronics Engineers Inc., jul 2016.

[25] M. Chmiel, R. Czerwinski, and P. Smolarek. IEC 61131-3-based PLC implemented by means of FPGA. In *IFAC-PapersOnLine*, volume 28, pages 374–379. Elsevier, jun 2015.

[26] M. Chmiel, J. Kulisz, R. Czerwinski, A. Krzyzyk, M. Rosol, and P. Smolarek. An IEC 61131-3-based PLC implemented by means of an FPGA. *Microprocessors and Microsystems*, 44:28–37, jul 2016.