UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Specialistica in Ingegneria Informatica

Tesi di Laurea Magistrale in Ingegneria Informatica

Modern approach for monitoring cryogenic systems for particle accelerators

Anno Accademico 2018/2019

relatore

Ch.mo prof Arpaia Pasquale

Co-relatori
PhD. Pezzetti Marco
Ch.mo prof Tramontana Porfirio

Candidato
Domenico Francesco De Angelis
matr. M63000697

[Dedica]

Contents

Modern approach for monitoring cryogenic systems for particle accelerators I
Contents III
Introduction 5
Chapter 1: CERN 8
1.1 CRG – Cryogenic Department 9
Chapter 2: Continuous Integration in TE-CRG 11
2.1 PLC 11
2.2 UNICOS 13
2.2 Continuous Integration in TE-CRG context 15
ST Language and FBD language 18
Chapter 3: Related Work 21
Chapter 4: Re-enginering PLC code 27
4.1 Translator FBD2ST 30
4.1.1 Parsing and Analysis 31
4.1.2 Generation of Intermediate Code 39
4.1.3 Optimization 41
4.1.4 Optimization with JSON rules 44
4.1.5 Translation from Intermediate Code to ST 50
Chapter 5: Testing 52
4.1 UnitTest ParserST for third level optimization 54
4.2 Unit Test of intermediate code generator for optimizer 58
4.3 Unit Test Analysis phase 59
4.3 Unit Test Intermediate code generator 62
4.5 System test Project 64
Conclusion 69
Future development 72
References 73

Introduction

Each software system, after the release need modification. Software systems evolve throughout their entire life. This is a consequence of some changes made in the systems, due to the needs of those who use it or the environment in which the systems operate.

The cryogenic group, at CERN, exploit PLCs complex system in order to monitor and manage the *cryoplant* systems.
The **p**rogrammable **l**ogic **c**ontroller (PLC) is an industrial state-solid computer, with customized operating system for better performance in specific monitoring and management context. Those PLCs can be programmed with different language types: graphical and textual. All specifications, on PLC environments, are defined on IEC61131-3 standard and better presented on second chapter (2.1).

The developers use the UNICOS framework in order to reduce the code to write in the PLC software projects. UNICOS is a CERN-made framework to develop industrial control applications, with layered architecture. The layers, in this framework, are two: Supervision and Control. The control layer is specified on PLC software. In these software projects its possible to write custom programs to resolve specific problems.

At CRG there exists a lot of projects that are PLC based with component programmed in FBD language, a graphical language with the main purpose to define combinatorial expression. The problem of this legacy code components is that they use an old UNICOS framework version, defined in chapter 2, and these are also expensive to maintain. This requires a re-engineering of legacy code.

The solution presented in this thesis is a system, named FBD2ST, for the migration of legacy code in PLC context, from FBD language to ST language, a textual language with pascal-like syntax.

The solution proposed is a toolchain based on pipe-&-filter architecture. It has an initial input in the FBD source code stored in the xml file. Each block reads an abstraction of source. The output chain is the translated code in ST language.

This system permits the optimization of intermediate code, in order to improve the generation of code. The optimization is structured on different layers. The first layer is implemented internally as code. The second and third layer are implemented by exploit strategy design pattern. This pattern simplifies the global implementation of optimizer, because the implementation of single optimizer subcomponent may be third parties. The format file used from the optimizer is JSON format. It is structured in two sections: rule and action. The first section defines the pattern of the intermediate code. The second section defines the action to apply at this intermediate code set matched with the pattern, defined before.

The abstraction, between each block, is represented from directed acyclic graph (DAG). In a stage, this abstraction in converted in intermediate code set. The intermediate code is based on quadruples: <*result, operator, op1, op2*>.

The result can be a variable name. The operator specifies the operation type of intermediate code. The op1 and op2 are the operands of intermediate code.

At the end, the output represents the ST source code saved in a textual file. The translate generation due one-by-one, with correspondent intermediate code.

The last step of this thesis is test of some system's components in order to validate them. The validations are based on unit test for each component and system test for whole system. The tests is based on requirements.

Now in this thesis is explain how this toolchain is implemented following the steps write before.

The next chapters explain this solution. Briefly here is a resume of each chapter's contents:

- **Chapter 1: CERN**, is a little description of centre of research and the description of TE-CRG work.
- **Chapter 2: Continuous Integration TE-CRG**, is a little description of continuous integration system used by Cryogenic group and the explanation of problem.
- **Chapter 3: Related Work**, is a little resume similar projects to this thesis.
- **Chapter 4: Reengineering PLC**, is the description of the solution and how it is implemented.
- **Chapter 5: Testing**, is the description of how some system component are tested using the system test software.
- **Conclusion**: the thesis conclusion.
- **Future development**: is a little resume of improvement or testing that can be done in the future.

Chapter 1: CERN

Figure 1 - CERN Logo

CERN (*Conseil européen pour la recherche nucléaire*) born after world war II, due to the european need for research in particle physics. The first person who had this idea was Louis de Broglie, french nobel, during the European Conference about Culture in 1949 and he proposed to establish a European scientific laboratory.

For this purpose, in 1952 eleven countries decided to build a concilium called *European Council for Nuclear Research,* from wich came the CERN arconim, along the French-Swiss Border near Geneva. Two years later, in 1954, this concilium became the European Organization for Nuclear Research, but it kept the acronym CERN. Established by 12 European countries, it grew into its actual form which includes 23 member states from different continents.

CERN can boast scientists from some 600 institutes and universities around the world that use CERN facilities every year. CERN has ten times more engineers and technicians than physicists. Indeed, given the specificity and the diversity of experiences, engineers are essential to CERN's activities. It employs some 2,500 physicists, engineers, technicians and workers from all fields, and around 12,000 visiting scientists come to CERN to conduct research, more than 120 nationalities and 70 countries are represented there.

CERN's foundation has been using 7 main accelerators. The **L**arge **H**adron **C**ollider (LHC) has been running since 2008 and it should work until its upgrade. The High Luminosity Large Hadron Collider will be completed around 2025. The structure of LHC is a ring with a 27 km circumference placed at a medium depth of 100 m between France and Switzerland, and is planned to be kept for its upgrade. LHC accelerate hadrons, composed subatomic particles to let them collide in complex machines which work as detectors. There, the collisions are observed and its data is stored and elaborated. Each detector is based on different technologies and they have different goals.

The role of CERN in the scientific field is not only that of research in particle physics, but also that of developing new technologies. An important support for CERN technology was the creation of the world wide web, a main service in order to navigate and use multimedia content.

1.1 CRG – Cryogenic Department

Cryogenic group (CRG) is responsible of the design, construction, operation & maintenance of cryogenic systems for accelerators and detectors. It is the group that has to ensure the temperature required in all CERN's complex, and especially extremely low temperatures.

CERN's installations can work thanks to ***LHe*** (*liquid* ***he****lium*) that can only be attained at a temperatures below **1.9K**. It can be notified that helium isn't the only cryogenic fluid used at CERN, an important amount of nitrogen is also used. Its purpose will be elaborated on in the presentation of the cryoplant, just below.

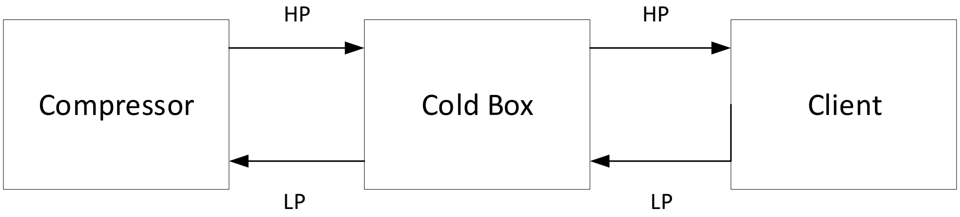


Figura 2 Cryoplant

The cryoplant, is dividable in three parts:

- *Compressor*: it receives as an input the low-pressure helium sent back by the coldbox and then it sends it back at a high pressure. To do so, it has two levels of compression.

- *Coldbox*: it may be described as the main component of the cryoplant. Indeed, the final objective of the cryoplant is to liquefy helium and so it has to have it at a temperature below 4.5K (systems are cooled at 1.9K) and the coldbox is the place where helium attains that temperature and is liquefied.
- *Client*: it represents the destination of liquefied helium. We can meet different types of clients such as experimentations or Dewar.

Every cryogenic plant is composed by a big number of elements. A typical PLC application can have about more or less of 200 input/output signals. In these applications, typical input signals are temperature, pressure and flow measures, while output signals are the setpoints for control valves or heaters.

The current applications involve a Siemens or a Schneider PLC, which is supervised through a SCADA system developed with WinCC OA.

Chapter 2: Continuous Integration in TE-CRG

The development of process control systems for the cryogenic infrastructure at CERN is based on an automatic software generation approach. The overall complexity of the systems, their frequent evolution as well as the extensive use of databases, repositories, commercial engineering software and CERN frameworks have led to further efforts towards improving the existing automation-based software production methodology.

At CERN, the cryogenic systems are an integral part of the infrastructure of the most important accelerators and experimental facilities and require complex industrial process control systems for operation. Their large scale and evolving requirements, functional and environmental, are big challenges for developers, who must produce without mistake, robust and safe control system software.

From the beginning, the large scale of the control system forced the use of automatic code production in development processes. The existing CERN code generation tools were adapted to cover the requirements of the control system, which became fully operative for the first time in 2008.

2.1 PLC

All the *cryoplant*, at CERN cryogenic system is monitored and managed with **PLC** (***P****rogrammable* ***L****ogic* ***C****ontroller*).

The PLC is a ruggedized industrial computer used for automation environment and is needed in any activity that requires high reliability control and ease of programming and process fault diagnosis.

This kind of computer can be programmed with the language defined in IEC 61131-3. The IEC 61131-3 is a standard that deals with the basic software architecture and programming languages of the control program within the PLC.

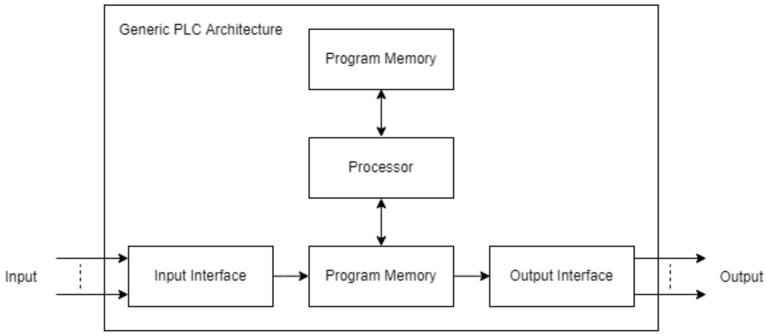


Figure 3 PLC Architecure

The software model is described by "*configuration*", it defines the hardware structure and logic of the PLC system and contains resources, which are able to execute programs. These resources are controlled by tasks, which invoke the execution of software blocks that make up the PLC project. These software blocks are referred to as Program Organization Units. In fact, this notion of configuration is just a fancy way to refer to the entire PLC software. On a lower level, a configuration can contain one or more resources. A resource can be seen as providing a processing environment for executing the program contained within the resource. A program is composed by definition sets of software elements called *functions* and *function blocks* and their application within a sequence of statements. These are the basic building blocks of a program, able to exchange data.

A *function* can be thought of as a subprogram that has no internal memory, so it returns a value rather than an output. This means that any time a function is invoked, if the same inputs are used, the same value will be returned.

***F**unction **b**locks* (FBs) are segments of reusable code that have internal memory and can return different outputs even when the same inputs are used. In other words, the results of a FB are conditional on the previous output of the FB or the current state of the process or action. This is because a FB also contains a data structure, in which values can be stored also after it's execution has finished. These stored values can influence the FB's result, which account for the possibility of differing outputs when executed with the same set of inputs.

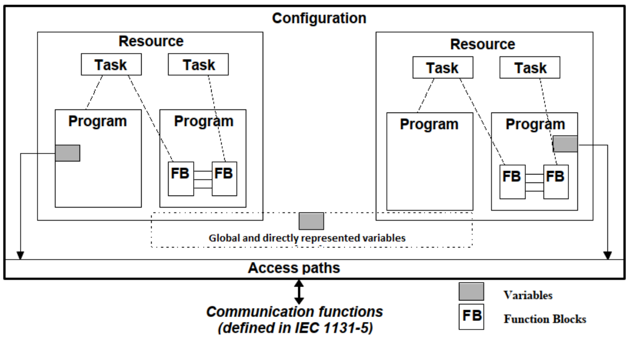


Figure 4 Programmable Controller System model

Generally, *Programs* can be called by *tasks*. *Function blocks* can be called by *programs* or by other *function blocks*. *Functions* can be invoked by *any POU type*.

2.2 UNICOS

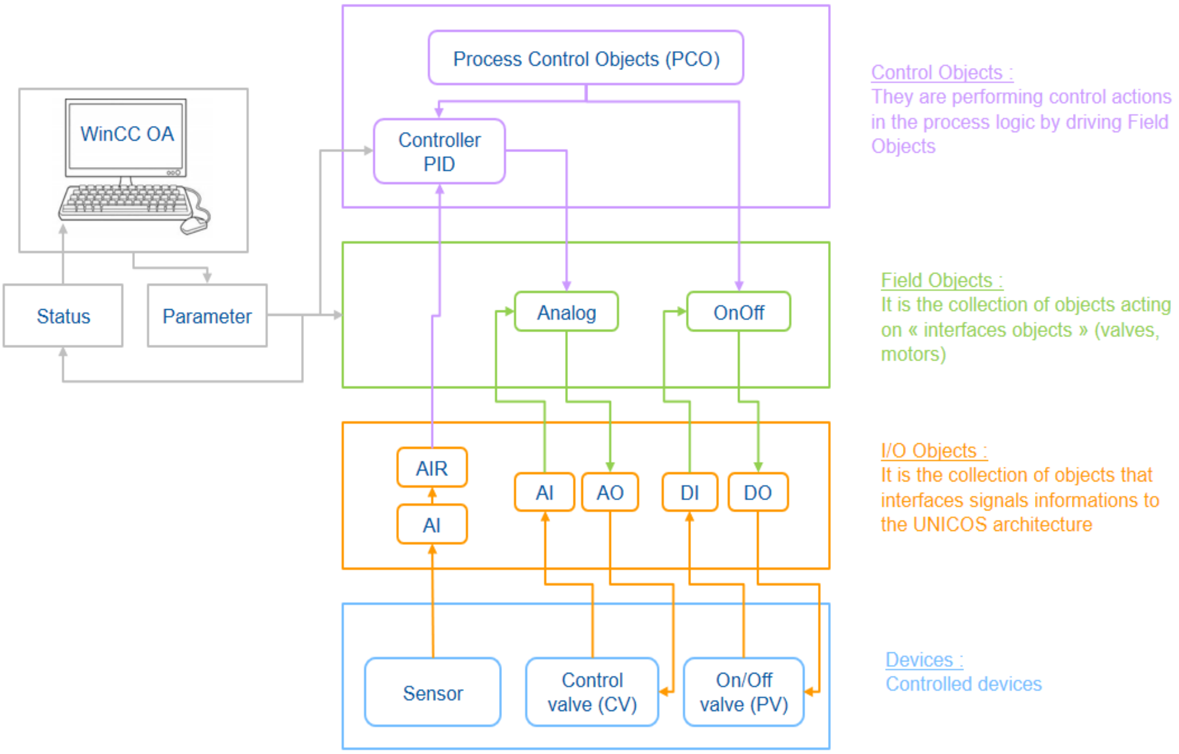

Process Control Objects (PCO)
Controller
PID
WinCC OA
Status
Parameter
Analog
OnOff
AIR
AI
AI
AO
DI
DO
Sensor
Control
valve (CV)
On/Off
valve (PV)
Control Objects :
They are performing control actions in the process logic by driving Field Objects
Field Objects :
It is the collection of objects acting on « interfaces objects » (valves, motors)
I/O Objects :
It is the collection of objects that interfaces signals informations to the UNICOS architecture
Devices :
Controlled devices

Figure 5 - Software model

UNICOS (**UN**ified **I**ndustrial **Con**trol **S**ystem)[1] deals with both control and supervision and provides a standardized methodology to work with monitoring applications. This framework with its Continuous Process Control package (UNICOS CPC, UCPC) is a standard for building programmable logic controller (PLC) based applications at CERN and other laboratories.

UNICOS offers several advantages, such as:

- Emphasizing good practices for both design and operation of the continuous process control application;
- Reduce the cost of automating continuous processes;
- Optimize life-cycle engineering efforts (e.g. using automatic code generation tools).

This is a framework built by CERN in order to automaticaly generate the code for industrial control process. The idea of this framework is generalized with UNICOS object standard in each PLC code, so it's cross-platform.

This framework provides a library of standard device types (objects), a methodology and a set of tools to design and implement industrial control applications. The standard device types implement various kind of patterns in order to generalize the usage of sensors and actuators inside the system. In this way, it is possible to reduce also the code written request in order to implement a PLC code to control a system.

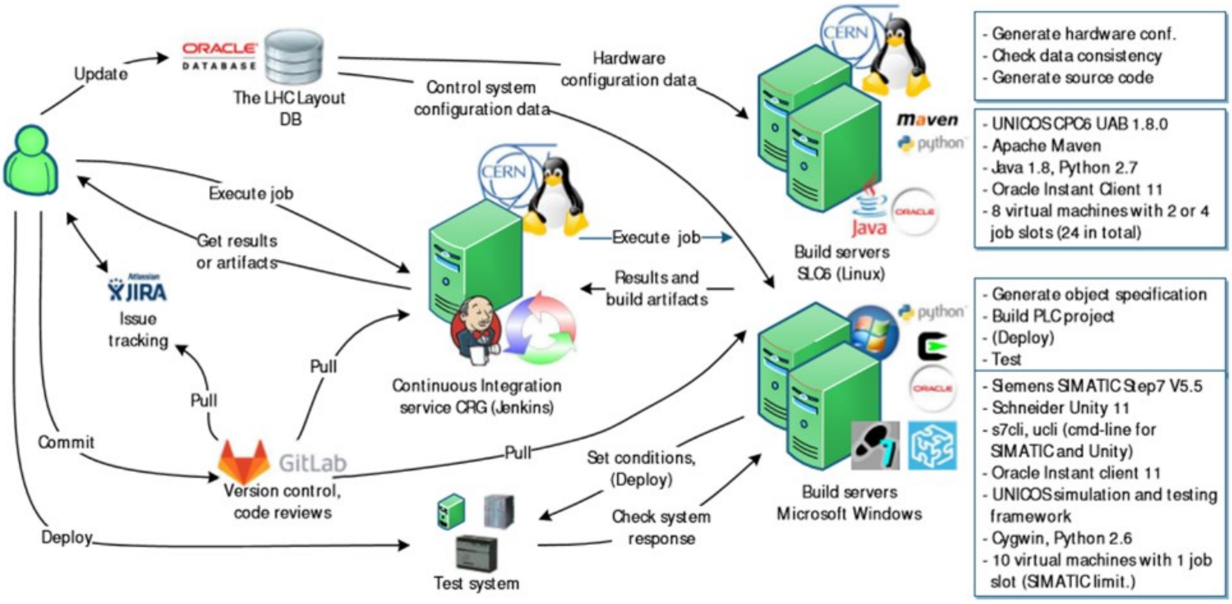
The objects, implemented in this framework, are instantiated in the PLCs and each of them communicate with a proxy instantiated in the supervision layer. The result of each command or process changes and is sent from the PLC object to its proxy to be elaborated. [3]

The PLC objects implement the process behaviour and the internal logic receives:

- Information from the process;
- Commands from operators;
- Commands from the control logic;
- Configuration parameters set during the programming phase and accessible for modification by a program specialist.

This framework supports the developments of control applications based on different models of SIEMENS and SCHNEIDER PLCs and SCADA applications based on WinCC OA.

2.2 Continuous Integration in TE-CRG context


Update
ORACLE DATABASE
The LHC Layout DB
Control system configuration data
Hardware configuration data
CERN
maven
python
Java
ORACLE
Build servers SLC6 (Linux)
- Generate hardware conf.
- Check data consistency
- Generate source code
- UNICOS CPC6 UAB 1.8.0
- Apache Maven
- Java 1.8, Python 2.7
- Oracle Instant Client 11
- 8 virtual machines with 2 or 4 job slots (24 in total)
Execute job
Get results or artifacts
JIRA
Issue tracking
Execute job
Results and build artifacts
Continuous Integration service CRG (Jenkins)
Pull
Pull
Commit
GitLab
Version control, code reviews
Pull
Set conditions, (Deploy)
Check system response
Deploy
Test system
python
ORACLE
Build servers Microsoft Windows
- Generate object specification
- Build PLC project
- (Deploy)
- Test
- Siemens SIMATIC Step7 V5.5
- Schneider Unity 11
- s7cli, ucli (cmd-line for SIMATIC and Unity)
- Oracle Instant client 11
- UNICOS simulation and testing framework
- Cygwin, Python 2.6
- 10 virtual machines with 1 job slot (SIMATIC limit.)

The updated architecture of the CI system.

Figure 6 Continuous Integration TE-CRG-CE

The infrastructure for Continuous Integration (CI) in Cryogenic group is performed by a set of components linked together. The CI service was built using Jenkins, chosen for its flexibility concerning various types of jobs on multiple platforms and wide possibilities to integrate with miscellaneous services. Other services used in this infrastructure are:

- Jenkins – jobs orchestrator;
- Git – versioning control system;
- Maven – automation building;
- JIRA – bug tracker;
- Database – with Oracle DBMS;
- CERN Virtualization Infrastructure – for build jobs execution.

In the Git repositories project and model relative at cryogenic systems and projects to manage the data are stored.

JIRA is a property issues tracker and mainly used for bug tracking of projects mentioned earlier. To manage multiple jobs internally Jenkins has been installing the plugin *multi-jobs*, in this way it is possible to execute more jobs at the same time. The jobs implemented serve to generate new PLC code, validate data inside the model on excel file with data stored in database or compiled the PLC code.

At time of Long Shutdown 1, in order to execute the operation mentioned before used 16 virtual machines with Microsoft Windows® and Linux systems with all the required software, such as Siemens SIMATIC Step7, Java® 1.7, Apache Maven, UNICOS Application Builder (UAB), Oracle Instant Client® 11, Python 2.x.

The UNICOS Application Builder (UAB), is the core software tool for the UCPC development process for generation of PLC code. It is an application with a typical graphical user interface and there is the possibility to use it with a simple command-line interface, which is very well suited for small and medium size projects.
The UAB application uses a reengineered excel model file, in order to generate a base of code for specific PLC systems.

In order to compile the PLC project, it was decided to create interfaces that would allow to automate the compilation through the dynamic compiler's libraries (*dll files*), so as not to have to use the graphical interface. This reengineering of PLC compilers were done with both Schneider and Siemens.

The tests that can be performed are of a different nature such as hardware unit test.

The service is configured in a way that it reflects the project's and team's workflow. It has significant impact on the software's development process and the result. It also meant that entering the Run 2 of the LHC with a development environment, allowing to address any change requests in more efficient and secure ways.

As mentioned earlier, the measurement and management of *cryoplant* is done by PLCs complex system. In this context, the deployment of this object is not directly applied by the user but is based on UAB generation code and the user's one.
A PLC, qualified for safety applications, is a good choice in order to save a lot of time in verification and validation efforts as compared to development of embedded control systems.
The five programming languages defined in the **IEC-61131-3** [7] norm is the industry standard for programming PLCs. The most popular of those languages is FDB, which gives to the user the opportunity to build a software solution using some graphical tools. However, the textual language **S**tructured **T**ext (ST) gives more flexibility to the programmer during the operation of writing the user-defined **P**rogram **O**rganization **U**nits (POU) requiring complex logics.

The peculiarity of FDB, is preferred by those who have not obtained yet any computer studies. In this way, the number of projects with module writing in FDB is incremented exponentially.

This approach makes a problem in terms of verification of version sources, because it requires the opening of IDE with both versions, new version and old committed version, and at the end compare manually the FDB source. This kind of solution is not practical when you have many sources to control.
Another problem, with this legacy code is the expansive cost of maintenance due to the use of the old UNICOS framework version.

The solution, in order to provide an evolutive and adaptive reengineering code, is to develop a custom toolchain to translate the code from FBD language to ST language.

ST Language and FBD language

As mentioned earlier, these two languages are standard and implemented from industry companies like Schneider and Siemens, in order to implement reliable programs with less complexity.

The ST language is a textual language with syntax Pascal-like. This language follows the block structured, procedural paradigms. For this reason, it has the iterative and selective construct in order to manage the instruction flow. The next table represents an example of selective construct available in ST language.

```
IF [condition] THEN
  <statement>;
ELSIF [condition] THEN
  <statement>;
ELSE
  <statement>;
END_IF;
```

```
CASE [numeric expression] OF
  result1: <statement>;
  ...
  resultN: <statemtent>;
ELSE
  <statement>;
END_CASE;
```

The next table represents the syntax of iterative construct available.

```
FOR count := initial_value TO final_value BY increment DO
  <statement>;
END_FOR;
```

```
WHILE [condition] DO
  <statement>;
END_WHILE;
```

```
REPEAT
  <statement>;
UNTIL [condition]
END_REPEAT;
```

In accordance with standard, this language has math, logical and comparison operators.

The FBD is a graphic language with the goal to describe the relationship between the input variable to output variable, using a net of function blocks, each block is linked with others by connection lines. This relationship input/output is akin to the electronical and block diagrams design of analogic and digital circuits.

The function blocks used in FBD are characterized by name instance, name function, input and output pin. The pin of a block represents the parameter input or the output returned by the block. In this figure we have an example of block in FBD.

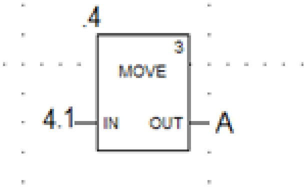


Figure 2 - Example of function block in FBD

All *connection lines* can have only one source and can be an output of other blocks. The connection line is unidirectional left to right and the type is influenced by the input block as linked.

In Schneider plc *implicit feedbacks* can be programmed. For this, the involved variables must be declared as feedback variables and the rules mentioned below must be observed when programming implicit feed-backs. Those can be used to realize a storing behaviour of the safety logic. The storing behaviour of the safety logic which results from programming an implicit feedback may lead to a complex timing of the entire application.

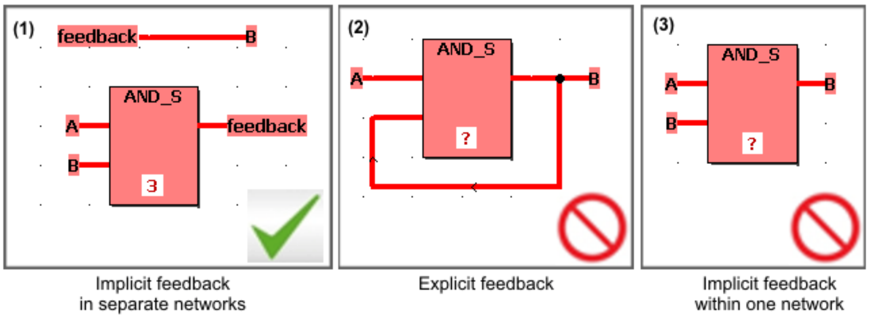


Figure 3 - rules for programming implicit feedbacks in FBD

In order to implement a feedback connection between blocks, namely an implicit loop must be declared a feedback variable. This type of variable must be declared in local scope and must be indicate in a specific form, selecting the option "*Feedback*".

In FBD and LD language, the function blocks have a special feature not used in the other languages defined in **IEC 61131-3**: here the functions possess an additional parameter in input and output, called enable. These two variables are Boolean, and the respective name is EN (input) and ENO (output). The meaning of the EN variable is to specify whether the block can be used or not. Instead, the meaning of the ENO variable is to report the input value to the EN variable.[6] In this way, it is possible to implement the equivalent of an if-statement inside the FBD. This graphic language does not support directly the iterative and selection construct. The EN and ENO block pins is a way to avoid the problem of selection construct.

This figure showes the example of a block with EN and ENO ports.

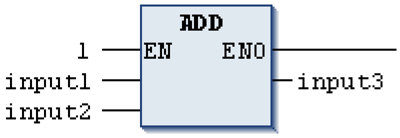


Figure 4 - ADD block with EN and ENO ports

The implementation in Schneider's programs of FBD sections have a grid behind them. This grid is built by grid unit, namely it is the smallest possible space between two objects in an FBD section.[6]

The FBD programming language is not cell oriented and the execution of code is generated followed the link between the blocks, such as the electrotonic circuit.

In FBD it is also possible to use the function blocks made from ST or other languages available in standard, like boxes.

A common feature between the two languages is that program run in iterative loop. In this way all instruction written inside a program FBD or ST can look like an infinite loop.

Another important thing is that POUs may not be recursive either directly or indirectly.

The conversion is only possible in this situation:

$$FBD \rightarrow ST \qquad reducted\ ST \rightarrow FBD$$

Chapter 3: Related Work

Chikofsky and Cross, wrote how "software re-engineering" is a process of development done to improve the maintainability of the system [8].
The main objective of re-engineering is the description of a cost-effective option for the system evolution, describing the activities involved in the software maintenance process, data re-engineering and to explain the problems of data re-engineering.

Re-engineering is mostly used in the context where a legacy system is involved[9]. Software systems are evolving at a high rate due to research, so more technologies are improving, more difficulties are arising for legacy software to operate on a new computing platform. Re-engineering is a set of activities that are carried out to re-structure a legacy system to a new system with better functionalities, whilst conforming to the hardware and software quality constraint.

Generally, the re-engineering of systems is due when a system's changes are confined to one subsystem, which needs to be re-engineered or the hardware/software support becomes obsolete or the tools to support restructuring are readily available.

The changing subsystem problem is resolved using different strategies such as wrapping legacy code, where the legacy code is incorporated into a wrapper compatible with the new system or rewriting the legacy code. This means extrapolating the logic from the legacy code and rewriting it through a language compatible with the new system.

In this context, as mentioned earlier, the nature of our problem is to migrate to new version of UNICOS framework used on PLC systems. In order to execute this migration, the idea is rewriting the legacy code from FBD language to ST language.

Though it is easy for someone with PLC programming knowledge to convert from legacy code in FBD to ST language. The objective is to build a system to convert source code automatically, in order to reduce the time of a manual conversion and to be sure that the translated code matches with the legacy code result.

In order to implement a project like this, we need parsing for the target legacy language code and after generating an abstraction representation of it, we should convert this abstraction in the target output language. Practically, we have described the job of a compiler.

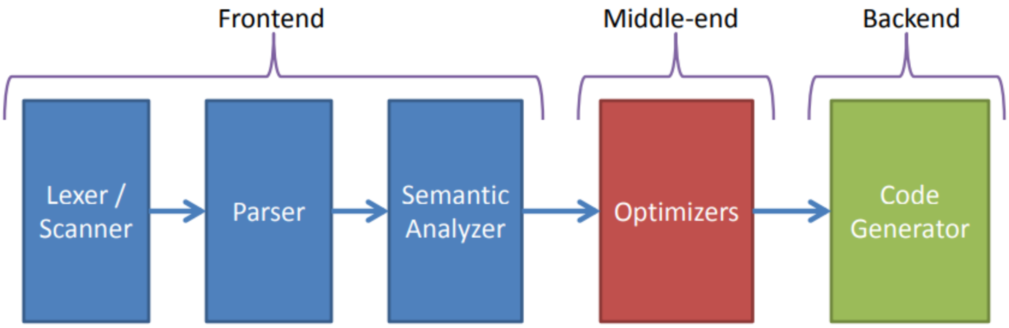


Figure 5 - Compiler Pipeline

The first step is a *scanning* or *lexer analyzation*. This is a process of converting a set of characters into a sequence of tokens. The second step is *parsing*, it analyses the set of tokens in input in order to recognize the grammar rule and return a syntax tree. The *semantic analysis* phase analyses the parse tree for context-sensitive information and controls if the rule of grammar is respected. This phase returns in output an annotated parse tree. Typically, *parser* and *semantic analyser* are merged in once phase.

The optimization step is to minimize or maximize some attributes of output program. At the end, the code generator translates the intermediate code into specific target code.

Currently there are various software tools for creating the frontend, described above. The most import parser generator are Flex/Bisson, Lex/Yacc and ANTLR.

Before 1975 writing a compiler was a very time-consuming process. Then *M. E. Lesk, E. Schmidt* published work on *lex* and *Stephen C. Johnson* published papers on *yacc*. These utilities greatly simplify compiler writing, because these two tools were provided to write a parser for a specific grammar. Yacc and Lex was born for Unix Systems.

Yacc reads the grammar descriptions in input textual file and generates a syntax analyser, that includes function *yyparse*, this tool returns two files: a header and a source file. The Lex tool reads the pattern descriptions in input textual file, includes header file returned by yacc, and generates a lexical analyser, that includes function *yylex*.

At the end, the lexer and parser are compiled and linked together to create executable files. From the main function we call yyparse to run the compiler. Function yyparse automatically calls yylex to obtain each token. The figure shows how it works. [4]

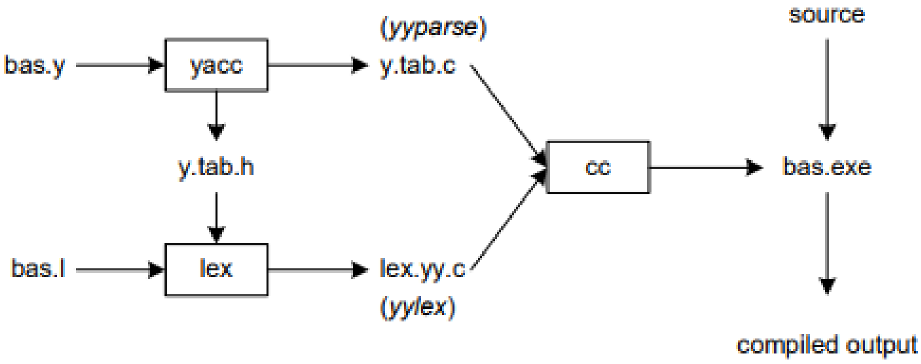


Figure 6 - Compiler structure with Lex\Yacc

Is also possible to continue the chain after realizing the parser and using the syntax tree such as intermediate code, after that applying some technique of optimization and at the end converting the intermediate code tree in the target code.

The usage of Yacc/Lex is useful when the target language is a **Regular language** or **Syntactical language.** The regular language that can be expressed with a regular expression, a deterministic, non-deterministic finite automata or state machine. The syntactical language has specific grammar and rules surrounding what makes proper syntax.

After some years, Robert Corbett in 1985 wrote the Bison a free software available under the GNU General Public License. A year later, Bison was made *Yacc-compatible* by Richard Stallman.

Bisson is a general-purpose parser generator, of GNU project, that converts an annotated context-free grammar into a LALR parser. The Bisson uses a textual file input, where is describes the target context-free grammar language and output return a function written in different languages such as C, C++ and Java.

This function returned by Bisson, recognizes the correct target language instances.

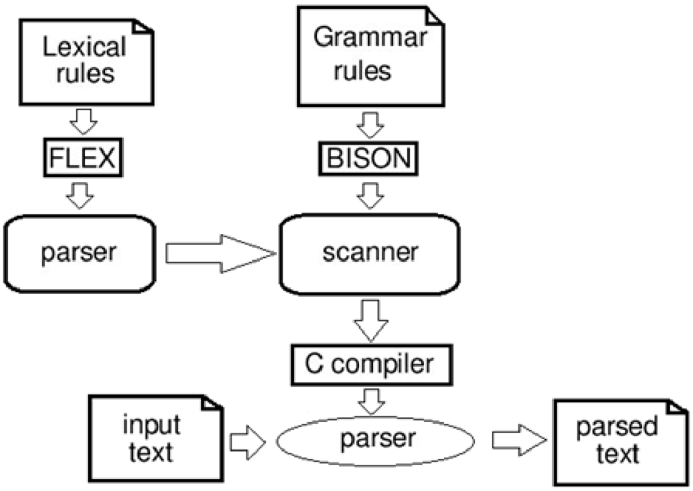


Figure 7 - Structure of parsing based on BISON

Such as Lex, also Flex generates a function for parsing so that the string follows the grammar rule. Bison reads the files generated from Flex and the grammar rules. Bison generates a syntax analyser.

There are some differences between Lex and Flex. Generally, the difference is the input file for the rules. The main difference is in the area of input lookahead. In Lex, you can provide your own input code and modify the character stream, instead in Flex you can not do that.

As mentioned earlier, Yacc and Bison are closely compatible, though Bison has some more features than Yacc such as location tracking, rich syntax error messages in the generated parsers.

An alternative for these traditional tools is ANTLR.

ANTLR or Another Tool For Language Recognition is a general-purpose parser generator. A tool that helps you to create parsers for reading, processing, executing, or translating structured text or binary files. The parser is the process of analyzing a string of symbols, conforming to the rules of a formal grammar, i.e. following a meta-syntax like Backus Normal Form (BNF). Generally, a parser takes the code and transforms it in an organized structure, such as an Abstract Syntax Tree (AST). The AST is a tree used to describe the abstract syntactic structure of source code written in target language.

ANTLR is recursive descent parser Generator (ALL(*) parser), this means it is a top-down parser built with a set of mutually recursive procedures, or not, where each procedure implements one of the non-terminals of grammar. A feature of ANTLR is that once you define your grammar you can ask ANTLR to generate multiple parsers in different languages.[20]

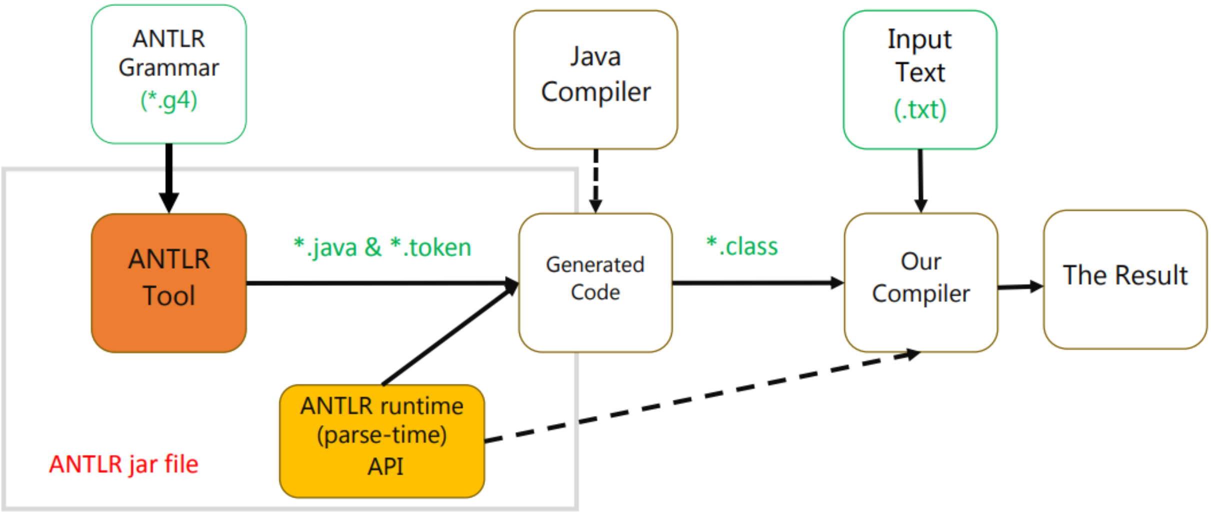


Figure 8 - Structure of ANTLR architecture

ANTLR takes as input a grammtical term that specifies a language and generates an output source code for a recognizer for that language. Those files and source codes generated by ANTLR must be linked with ANTLR API in order to have a compiled library. This library generates checks whether the grammar of the input file is correct and generates a syntax tree.

The main difference between those solutions mentioned earlier are the algorithms to parse the input file. The first two solutions are based on LALR parser and the ANTLR is based on LL parser.

The LL or Left-to-right parser is a top-down parser for a subset of context-free languages. It considers the input symbols from the left to the right and attempts to construct a leftmost derivation. This operation is done for each symbol in order to expand out the leftmost non-terminal until it arrives at the target string. This parser has two operation types:[19]

- **Predict**: Based on the leftmost nonterminal and some number of lookahead tokens, choose which production ought to be applied to get closer to the input string.
- **Match**: Match the leftmost guessed terminal symbol with the leftmost unconsumed symbol of input.

The LR parser is a left-to-right, rightmost derivation, meaning that it reads the input text from the left to right and produces a rightmost derivation. This parser is deterministic, meaning it produces a single correct parsing in which you have linear time, in worst-case performance of $O(n^3)$ time.

The parser continuously picks a substring of the input and attempts to reverse it back to a nonterminal.[19]

The loop of LR parser is formed by the choice of different actions which are:

- **Shift**: push the scan next input token to a buffer for consideration.
- **Reduce**: Appling grammar rule in order to reduce a set of terminals and non-terminals in this buffer to some non-terminal by reversing a production.
- **Done**: End of parsing.
- **Error**: Report a syntax Error. The parser ends.

In this chapter we presented some toolchains for generation of parsers. At the end all parsers generated by these tools give in output a parse tree. The nature of FBD source file are more similar a direct acyclic graph, we’ll talk about it later.

Chapter 4: Re-enginering PLC code

The solution presented in this thesis is inspired by projects mentioned earlier. In this case, we preferred not to use one of the projects mentioned earlier above due to the nature of the input text file. Other motivation is because we want to give the possibility to add new optimization rules through text files in a hastily.
At the end, we want to give in output we want give also same information about the initial source code.
The input file represents the FBD language source code, it is saved in XML format by Control Expert Schneider. In order to read this file, we use a specific parser for xml file. Generally, it is possible to do it using a (Document Type Definition) DTD file and an analysis system for XML structures. The DTD file defines that all legal structure is possible to define inside an XML file format.

In order to implement this system, the idea is to use a library that can parse xml files and then manipulate the newly extrapolated data to perform the optimization and code generation phases.

The methodology used in this project is V-model, that may be considered as an extension of the waterfall model. In this development methodology each phase of the development life cycle is associated with a phase of testing, in order to validate the specific phase and get back its worst case.

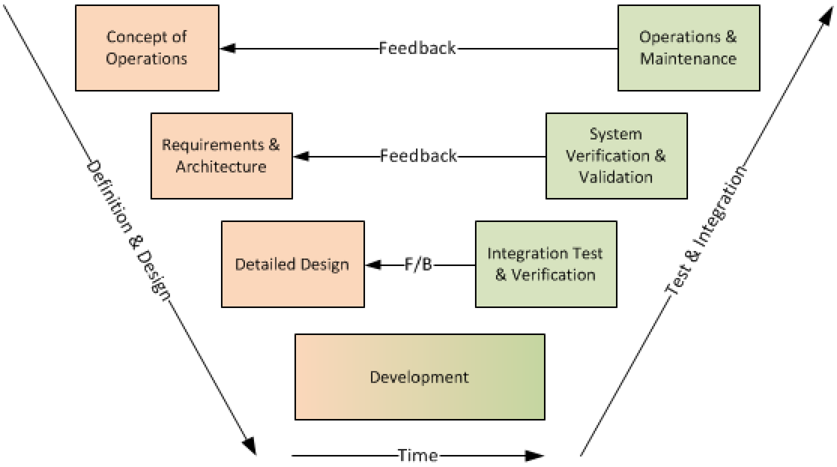


Figure 9 - V-Model

This deployment method is chosen in order to test the code and in the case of problem on a specific layer of development to get back on development phase of components.

The functional requirement describes service or function offered by system. It describes the interactions between the system and it's environment regardless of its implementation. It can be:

- **complete**: must indicate all the services requested by users.
- **consistent**: requirements must not have contradictory definitions.

The non-functional requirement describes constraints on the services offered by the system, and on the development process itself. It describes aspects of the system that are not directly related to the behaviour (functionality) of the system.

In other words, a functional requirement describes what a software system should do, while non-functional requirements place constraints on how the system will operate.
The Domain Requirement refers to system features that derive from general application domain features.

Now we must define the project requirement. The list of functional requirements is:

- the system shall recognize the correctness of the input xml file
- the system shall interpret the math blocks of FBD source language
- the system shall interpret the assignation blocks of FBD source language
- the system shall interpret the comparison blocks of FBD source language
- the system shall interpret the instance blocks of FBD source language

- the system shall interpret the Boolean blocks of FBD source language
- the system shall manage the use of EN/ENO input/output
- the system shall interpret the net of blocks
- the system shall optimize the intermediate code
- the system shall read parameter for optimization with ST language format
- the system shall read input textual file in order to describe the optimization
- the system shall generate the temporal variable names following the formula: LV_ + name_prj + name_type + <index>
- the system shall limit the size of temporal variable names to 30 characters.

The list of non-functional requirements is:

- the system shall write in python
- the user should be able to write optimization rule in JSON format file
- the system shall implement a restricted parser for ST language
- the system shall save the code generated in textual file
- the system should be able to read the input textual data exported from Schneider Control Expert

The architecture chosen for this project is the pipe-&-filter, as also suggested by the nature of the context. In which a given input data stream must follow various stages of analysis and processing, until it becomes a new output data stream. In this architecture type each block transforms the input stream, while the output is inputted to another block. The processes may work in competition, in order to maintain the stream inside the pipeline.

4.1 Translator FBD2ST

The translation type, used in this project, is straightforward because ST language has the same construct of the FDB Language. From a language's point of view, ST language looks like an encapsulation of FDB language. The steps followed for this translation is similar for a compiler and showed in followed figure.

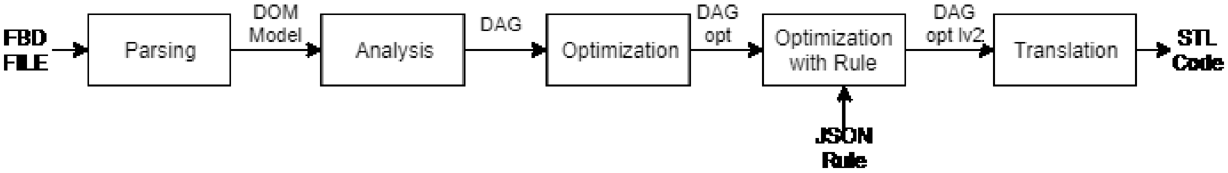


Figura 1 Translation Phase

The first step, parsing of ".xbd" file with xml format, it consists in observing if there are particular tags*: program, FDBBlock, linkFB* and extract this particular information.

In the next paragraph, we talk about steps in detail, but shortly we have a parsing of XML file after with a DOM model in output. This model is input for the analysis block which has the task to extrapolate the data in order to build a Direct Acyclic Graph (DAG).

This DAG graph is also converted into intermediate code set.

This intermediate code set is used to start the optimization phase, that provides its simplification using *De Morgan theorem* and other optimization technique.

In the last optimization step, the user has the possibility to define some optimization rules using a JSON file, which contains a description of a specific pattern to apply a specific rule.

At the end we have a translation phase, where the optimized intermediate code is converted in textual format.

4.1.1 Parsing and Analysis

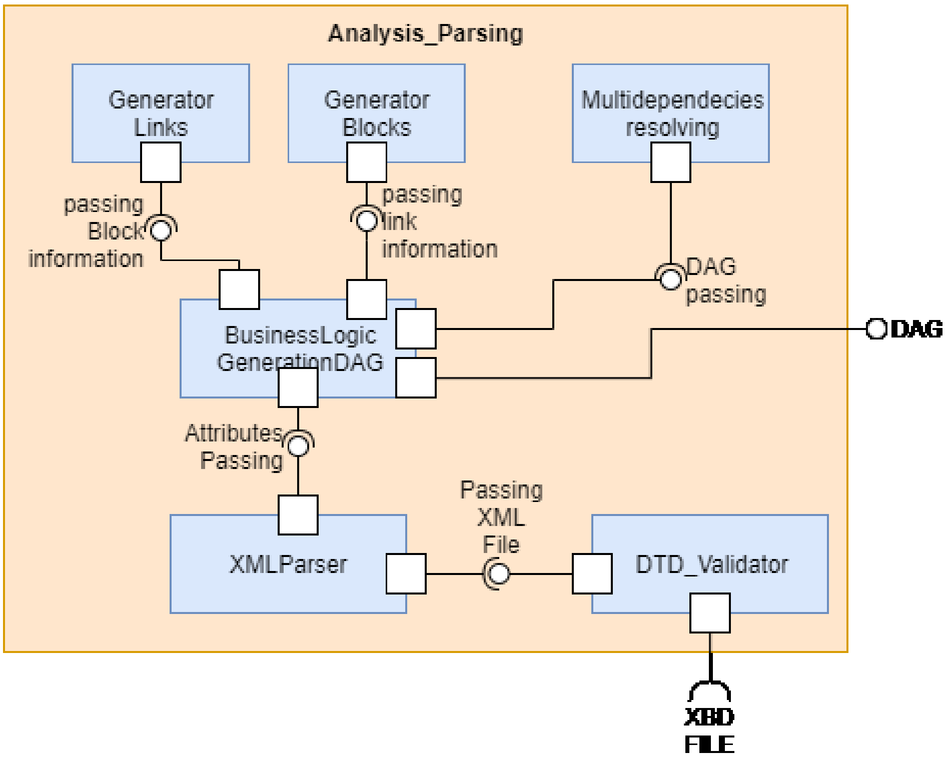


Figure 10 - Component diagram of first steps

The parsing and the analysis of input source file is provided by Analysis_Parser component. This component describes the validation and the parsing of input source code file and the generation of **d**irected **a**cyclic **g**raph (DAG) representing the syntax of source code, we'll talk about it later. In this chapter it is explained how this component works and how it's implemented.

The exportation data offered by custom interface of Unity© application, built in TE-CRG group, return a compressed folder containing a group of files with '.xbd' extension, in a xml format. The same operation can be performed manually by Control Expert© application, with the appropriate command. These data pieces are the inputs of the FBD2ST system.

The parsing of xml file is done by *minidom* standard python library. Mini DOM is a minimal implementation of the *Document Object Model* interface.

The **Document Object Model** (**DOM**) [9] is the data representation of the objects that contain the structure and content of a document on the web. The representation due by nodes and objects. One important property of DOM structure models is *structural isomorphism*:

if any two Document Object Model implementations are used to create a representation of the same document, they will create the same structure model, with precisely the same objects and relationships.

As earlier mentioned, the ‘.xbd’ file is described in xml format, structured in three main sections:

- *Section with declaration of function block*, the order of blocks depended form user how put the block inside the diagram.
- *Section with linking blocks*
- *Section of variable* used inside the code, global and not.

The syntax of language resulting (ST) from translation can be described using a **B**ackus **n**ormal **f**orm (BNF) notation, as specified below:

```
<expression> ::= <op> <assign> <op> | <op> <operator> <op> | '(' <expression> ')'
          | <const-string> '(' <parameter> ')' | <expression>  ';'
<parameter> ::= <parameter-in> | <parameter-out> | <parameter> ',' <parameter>
<parameter-out> ::= <const-string>  '=>' <op>
<parameter-in> ::= <const-string>  ':='  <op>
<op> ::= <const-value> | <pin>
<pin> ::= <const-string> | <pin> '.' <const-string> | <pin> '.' <const-integer>
<assign> ::= ':='
<operator> ::= 'AND' | 'OR' | 'NOT' | 'XOR' | '<' | '>' | '>=' | '<=' | '=' | '<>' | etc...
<if-statement> ::= 'IF'  <expression>  'THEN'  <expression>  'END_IF;'
<comment> ::= ‘(*’ <everything> ‘*)’
```

Where the meaning of:

- *const-string* - indicate the string value
- *const-integer* - indicate the integer number value and
- *const-value* – indicate all the constants value representable in ST language (string, number, time).

The file exported by *Control Expert© application* give us a file in xml format where are present three type of information, as mentioned earlier.

This specific file is structured as follow:

- *FBDExchangeFile,* is the root node of our xml file.
- *fileHeader, contentHeader, program, dataBlock* is the child of root node

- *fileHeader, contentHeader* contains generic information about project and Control Expert.
- *Program,* contains nodes in order to define the FBD graphs
- *DataBlock,* is optional node and give us a specific of instance made in this FBD.

The "*program*" node contains two kind of nodes: *FFBBlock* and *linkFB.* The first contains information about specific block used in diagram, with information about input/output signal and graphical position in the diagram. The second node, contain information about the link between two nodes, defined by *FFBBlock.*

The node FFBlock has the following attributes:

- *additionalPinNumber*, defined whether block use more of 2 input.
- *enEnO*, defined if is used the EN and ENO pin of block. This pin defines block activation.
- *Height* and *width* are graphical information in the diagram.
- *instanceName*, is name of specific instance of this block.
- *typeName*, is function name defined by the block.

This node contains two nodes: *objPosition* and *descriptionFFB*.

The first node defines where is posed the block in the graphical diagram, the second node define what is the input and output of block. The *descriptionFFB* node have an attribute "*execAfter*" and contain two kind of nodes: *inputVariable* and *outputVariable*.

The attribute "*execAfter*" defines after which block must be executed. The nodes input and output variables have the following attributes:

- *effectiveParameter*, is optional and is defined if him is linked with a signal or a constant value, in other case is defined with a link.
- *formalParameter*, is the pin's name.
- *invertedPin*, is a boolean value and define if the signal in input or output is inverted or not.

Each nodes *linkFB* is compose from *linkSource*, *linkDestination*. These two nodes have the followed attribute and a child objPosition.

- *parentObjectName*, the instance name of object linked
- pinName, the pin name of object linked

The node *objPosition* specify only the information of position in graphical context.

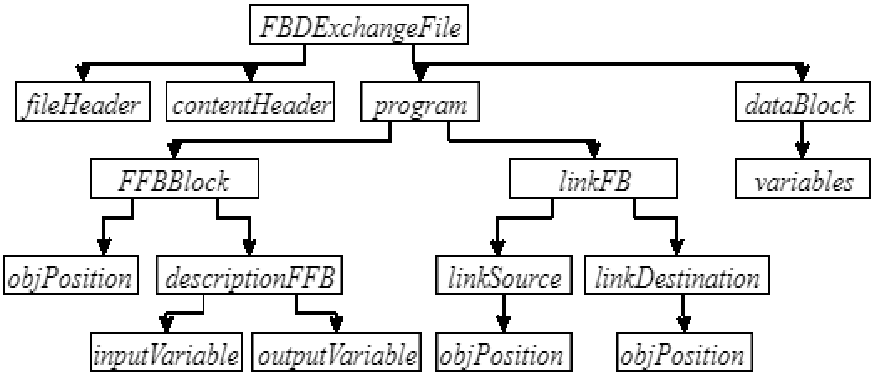


Figure 11 FBD xml structure file generated by Control Expert

The figure above contains a slim resume of xml structure described by the Control Expert program.
With an xml file, we're able to avoid the development of a parser, but we must extrapolate the date from this file, in order to have a high abstraction of FBD code.

The management of xml file is performed from the class *fbdFileReader*. This class reads the input file control to see if the file is structured in good way, in order to pass all XML nodes at phase of analysis. In order to understand if the file xml in the results follow the specification mentioned earlier we use a **d**ocument **t**ype **d**efinition (DTD) file, it is a set of mark-up declarations that define a document type. This DTD file is present in Control Expert© application folder.

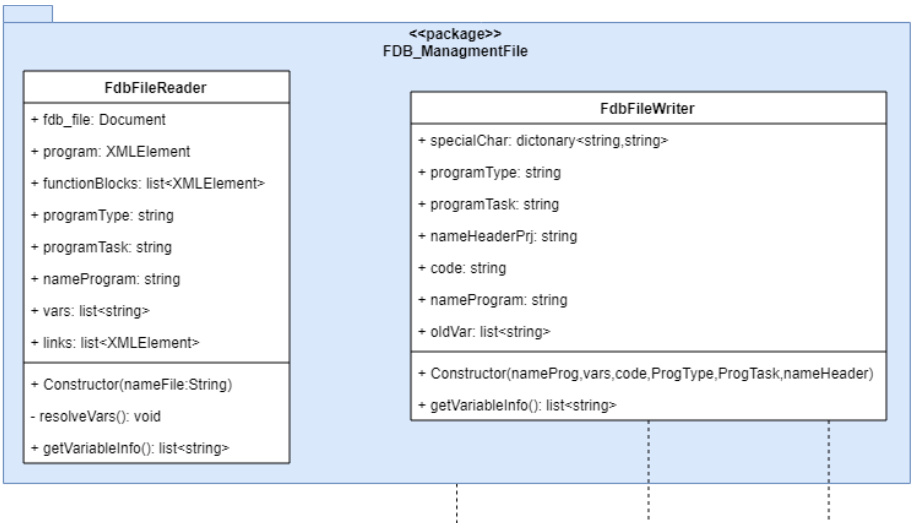


Figura 2 Class Diagram - FDB and ST File Managment

The classes used in order to store all information described in initial model are:

- *BlockFDB*, represents the instance of block in the initial program in functional diagram block.

- *PinFDB*, represents the pin in input or output of each block and this is an interface to communicate with other block, variable or constant
- *LinkFDB*, represents the link between two blocks defined in FDB.

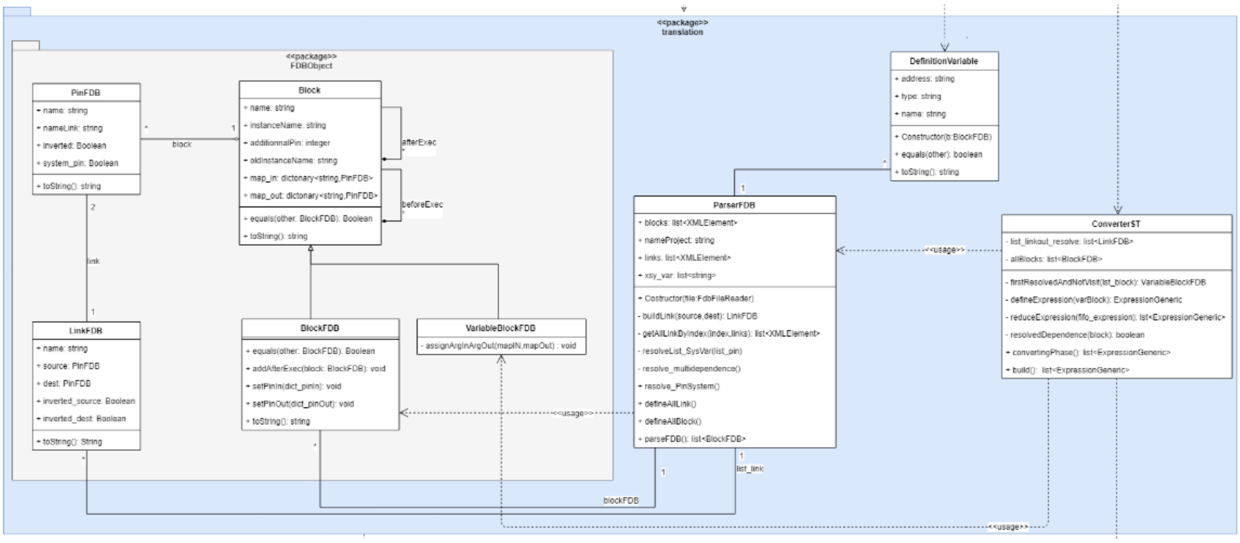

<<package>>
translation
<<package>>
FDBObject
PinFDB
+ name: string
+ nameLink: string
+ inverted: Boolean
+ system_pin: Boolean
+ toString(): string
block
*
1
Block
+ name: string
+ instanceName: string
+ additionnalPin: integer
+ oldInstanceName: string
+ map_in: dictonary<string,PinFDB>
+ map_out: dictonary<string,PinFDB>
+ equals(other: BlockFDB): Boolean
+ toString(): string
afterExec
beforeExec
2
link
1
LinkFDB
+ name: string
+ source: PinFDB
+ dest: PinFDB
+ inverted_source: Boolean
+ inverted_dest: Boolean
+ toString(): String
BlockFDB
+ equals(other: BlockFDB): Boolean
+ addAfterExec(block: BlockFDB): void
+ setPinIn(dict_pinIn): void
+ setPinOut(dict_pinOut): void
+ toString(): string
VariableBlockFDB
- assignArgInArgOut(mapIN,mapOut) : void
<<usage>>
DefinitionVariable
+ address: string
+ type: string
+ name: string
+ Constructor(b:BlockFDB)
+ equals(other): boolean
+ toString(): string
1
ParserFDB
+ blocks: list<XMLElement>
+ nameProject: string
+ links: list<XMLElement>
+ xsy_var: list<string>
+ Costructor(file:FdbFileReader)
- buildLink(source,dest): LinkFDB
- getAllLinkByIndex(index,links): list<XMLElement>
- resolveList_SysVar(list_pin)
- resolve_multidependence()
+ resolve_PinSystem()
+ defineAllLink()
+ defineAllBlock()
+ parseFDB(): list<BlockFDB>
<<usage>>
ConverterST
- list_linkout_resolve: list<LinkFDB>
- allBlocks: list<BlockFDB>
- firstResolvedAndNotVisit(lst_block): VariableBlockFDB
- defineExpression(varBlock): ExpressionGeneric
- reduceExpression(fifo_expression): lst<ExpressionGeneric>
- resolvedDependence(block): boolean
+ convertingPhase(): list<ExpressionGeneric>
+ build(): list<ExpressionGeneric>
1
1
list_link
blockFDB
<<usage>>

Figura 3 Class Diagram - Translation Bussiness package

The data extrapolation is based on the algorithm described in the activity diagram shown in the figure above, where:

- the first phase is declaring all blocks in our program with all pins;
- the second phase is declaring all links described always in xml file;
- the third phase is resolving the blocks multi-dependencies.

In this phase the multi-dependencies are also solved with the consequent local variable declaration, and the result of this phase is a graph with a representation of FDB program.

As mentioned earlier, the data structure build in this phase is a **d**irected **a**cyclic **g**raph (hereafter DAG). Generally, the nodes in a syntax tree represent constructs in the source program; the children of a node represent the meaningful components of a construct. A DAG for an expression identifies the common subexpressions (subexpressions that occur more than once) of the expression. The difference is that a node N in a DAG has more than one parent if N represents a common subexpression. In an **A**bstract **S**yntax **T**ree (AST), the tree for the common subexpression would be replicated as many times as the subexpression appears in the original expression. Thus, a DAG not only represents expressions more sufficiently, it gives the compiler important clues regarding the generation of efficient code to evaluate the expressions.

The DAG is stored in an array of instance of specific class called BlockFDB and represents the graph nodes. In each node is defined the name of the object class, name of instance and specified input and output nodes, in order to describe the expression graph.

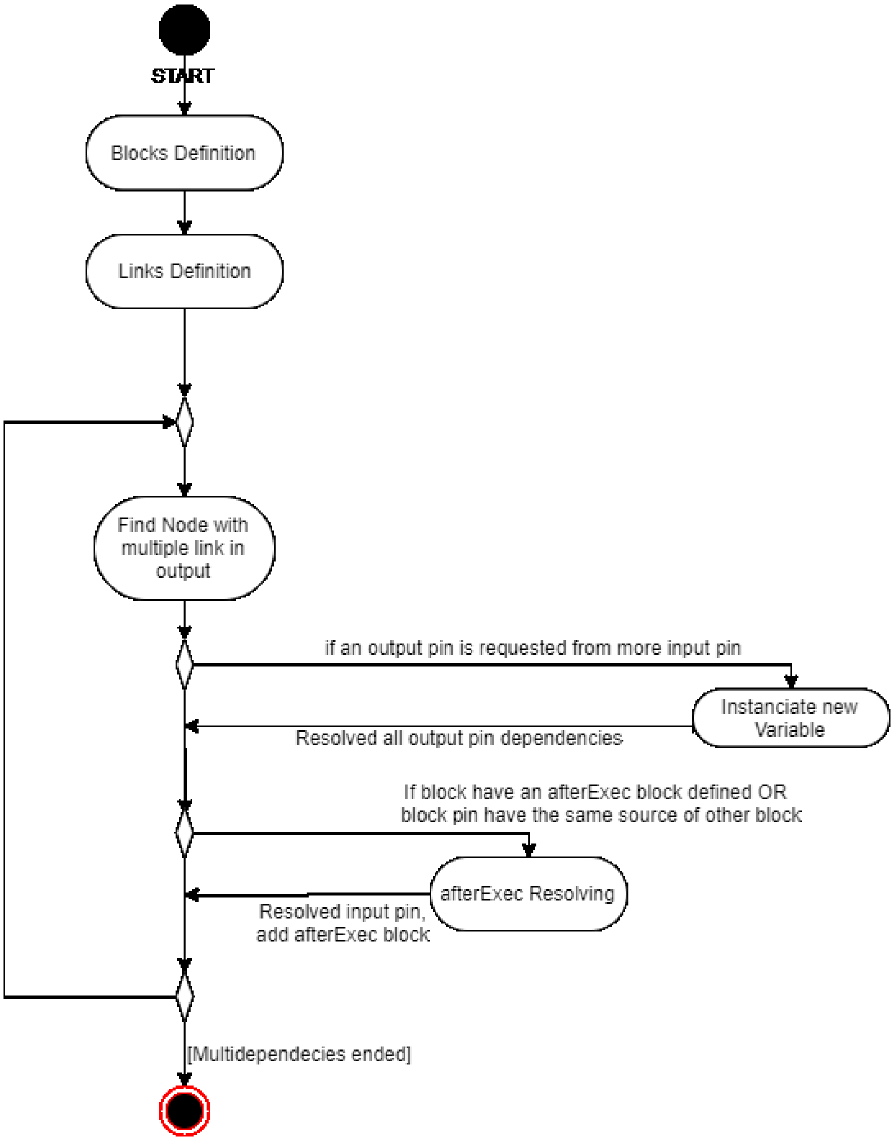

START
Blocks Definition
Links Definition
Find Node with multiple link in output
if an output pin is requested from more input pin
Instanciate new Variable
Resolved all output pin dependencies
If block have an afterExec block defined OR block pin have the same source of other block
afterExec Resolving
Resolved input pin, add afterExec block
[Multidependecies ended]

Figure 12 Activity diagram - Build the DAG of code

The first step is generation of all blocks defined in xml file. For each instance of FFBblock node instantiates an object of *BlockFDB class*, after this it also instantiates the objects for defining the pin with the *PinFDB* class.

In this first step, some links between the variables, constants or another block can be already defined. In this case we can do two different things.

If the pin is linked at variable or constant value, then store this information inside the *PinFDB* instance, in the other case we need a generation of link. Inside the instance pin is stored

information about pointer of parent block, the pin name such as a string, and a temporal name such as string used in two different ways dependent of the pin type.
If the pin is used in order to link the block with another block, the temporal name is generated with the following formula: <*block parent name*> *'.'* <*pin name*>. If the pin is used in input or output at a variable or constant (in input), this information is stored in a Boolean variable and the temporal name represents the effective value linked at pin, a constant value in case of value or a name of variable in case of variables linked.

The second step, after all blocks are defined, we need to link all blocks with proper input and output. In order to do that, for each instance of *linkFB* it instantiates an object of *LinkFDB*, linked with specific pin input and specific pin output. In order to recognize the link, in phase of optimization or final generation code, it brings a temporal name generated using the following formula: <*name of destination block*> *'.'* <*name of destination pin*>

The third step is a little optimization of DAG in order to improve the generation code. This step consists to search pin multi-dependencies between two blocks in order to avoid it, in the next phase. The meaning of pin multi-dependencies term is the connection of more than two blocks to the same output pin.
This is a property of DAG and it is an advantage for graph structure, but in case of generation this kind of thing is translated such as rewriting the same expression more times respective of one.
Two cases are possible:

- The output pin with multi-dependencies is an output of an object, and this pin can be called alone with a dot notation.
- The output pin of block corresponds to a math expression and in this case corresponds to a calculus.

In the first case, it is an advantagage to have multi-dependencies, because the block is instantiated like an object and at the end is used such as an object with dot notation.
In the second case, in order to not re-write the same expression more times it is better to substitute the multi-dependencies with usage of a variable. The assignation of name at local variable is built in the following way:

$$\mathrm{LV\_} + [nameProject] + [nameTypeInstance] + \langle index \rangle$$

The index is an integer value and it indicates the number of occurrences with the same name. The name cannot contain more than 30 characters, so if there are multiple letters, some letters of the name will be deleted. In order to maintain the connection between the blocks the action performed has to add the block, with output temporal variable, in "*list after execution*" of all block with same input temporal variable, we'll talk about it later.
This part, this is the only optimization done before translating the DAG into intermediate code.

The fourth step for each block in the project connects all blocks with others, respective of the "*after execution*" definition. In FBD language it is possible to define an execution flow in two ways:

- With linking, the flow is design with the link between the blocks.
- With "after execution" definition, where the flow is described in specific parameter.

In order to describe this behaviour, the class *BlockFDB* have a precedence list, in this list is stored all precedencies. The blocks are double linked in order to know the precedence and antecedence of blocks.

4.1.2 Generation of Intermediate Code

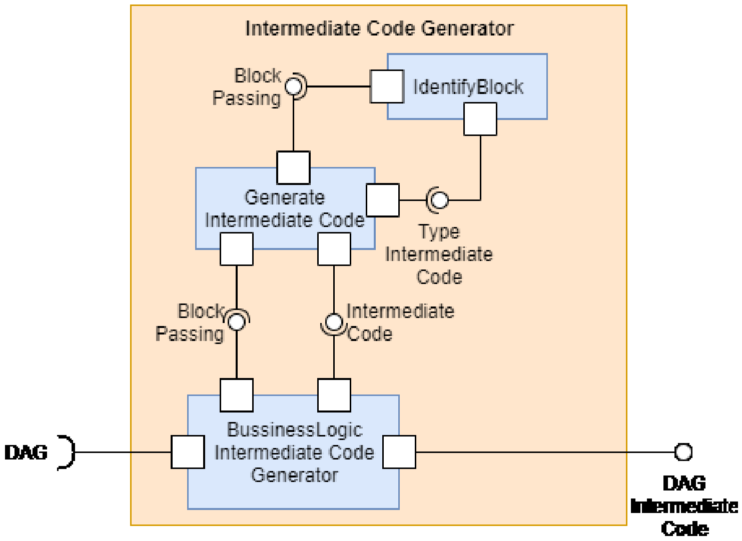


Figure 13 - intermediate code generator component

After generated the DAG with class mentioned earlier, the third step of translation is optimization of DAG. In order to do this, we are considering the graph obtained as a set of expressions that make up the final program. The graph initially consists of *BlockFDB* instances and will then be converted into an intermediate version between DAG and final code. The intermediate code is based on this hierarchy defined as follows in the figure.

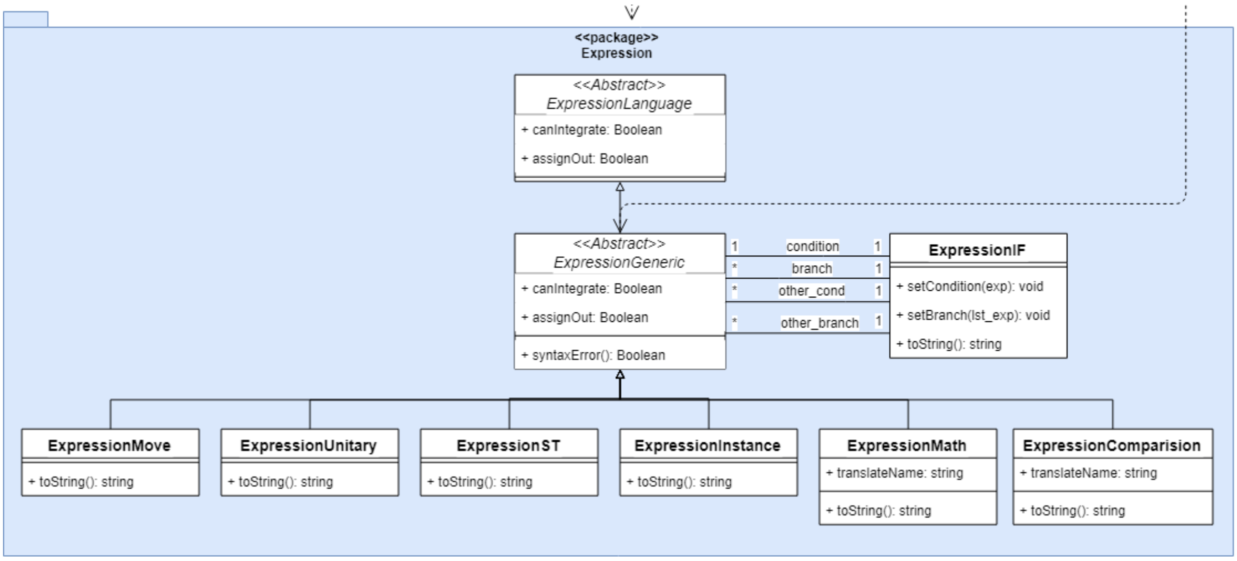


Figure 14 Class Diagram - Expression hierarchy

In order to do this conversion, from DAG with *BlockFDB* instance to intermediate code set with *Expression* class, the type of block must be recognized and translated.

The classes defined in the hierarchy are as follows:

- *ExpressionMove* describes an assignation instruction and is used in case of MOVE blocks.
- *ExpressionUnitary* describes the assignation of a negate value and is used in case of NOT blocks.
- *ExpressionMath* describes the basic mathematical blocks, which are: addition, subtraction, division, multiplication, modulus and exponential. Generally, these kinds of blocks can assign directly the value in a variable, for that this expression can be translated also like an assignation at variable.
- ExpressionInstance describes the passage of the input variables and an instance of an FB. These input values can be: variables, constants or other expressions.
- *ExpressionComparision* describes the use of comparison operators, i.e. the blocks EQ, GT, etc…
- *ExpressionST* describes the use of Boolean operator: AND, OR, XOR.
- *ExpressionIF* describes the specific case of if-statement.

The conversion of the initial DAG to that based on the hierarchy of expressions is carried out with a correspondence from one-by-one. Indeed, the expression class still contains the information relating to the initial block.
The conversion, from initial DAG to intermediate code, generates a new DAG with fewer nodes with more than one parent node number. This new DAG, for the rest of the thesis will be called as intermediate code set.

The intermediate code is based on quadruples: <*result, operator, op1, op2*>.
The result can be a variable name. The operator specifies the operation type of intermediate code. The op1 and op2 are the operands of intermediate code.

The generation of intermediate code set is structured in following phases:

- Identification block types, in order to convert with specific intermediate code.
- Instantiation of intermediate code and push it in to FIFO structure of code.

The intermediate code set is stored in an array.

4.1.3 Optimization

The optimization is managed in three different levels:

- First level - defines two optimization types: "*dead code elimination*" and "*common sub-expression elimination*".
- Second level - realizes the optimization of the intermedia code using "*peephole optimization*".
- Third level - realizes the optimization using the rules defined by the user.

In the *first level*, the *intermediate code set* is optimized with the technique of "*dead code elimination*" and "*common sub-expression elimination*".
The *dead code elimination* is an optimization that removes the code that doesn't affect the program results.
The *common sub-expression elimination* is an optimization that searches for instances of identical expressions, and analyses whether it is worthwhile replacing them with a single variable holding the computed value. It was described earlier, when we talked about new temporal variables. This phase is performed before the translation into intermediate code.
In order to apply the *dead code elimination*, the strategy was to observe if the output value of an expression was assigned to a variable.

In this case the expression is removed because it is useless and does not produce any result that will be used in the future, except in the case that the expression is an *ExpressionInstance class* instance, because this represents an FB with its internal behaviours unpredictable in advice.

The *second level* is *peephole optimization* based performed on a small set of compiler-generated instructions, which involves changing the small set of instructions to an equivalent set that has better performance. In this level, the common techniques used are logic port transformation, algebraic simplifications with usage of "*De Morgan Theorem*" and boolean property.

The *third level* is the last level of optimization and is based on the rules written from user based on *Pattern Search,* that is, given a defined scheme, if this is recognized then optimization can be applied.

The optimization is oriented towards the block and the network of connections that create the specific graph. Thus, defining possible schemes that can be recognized and optimized.

In order to simplify the management of optimization class, the second and third optimization levels use the *strategy design pattern*. This design pattern falls into the "*behavioural design pattern*" category. It defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it. In the following figure you will see how the design pattern is used in order to define the class of optimization.[18] This pattern simplifies the global implementation of optimizer, because the implementation of single optimizer subcomponent may be third parties.

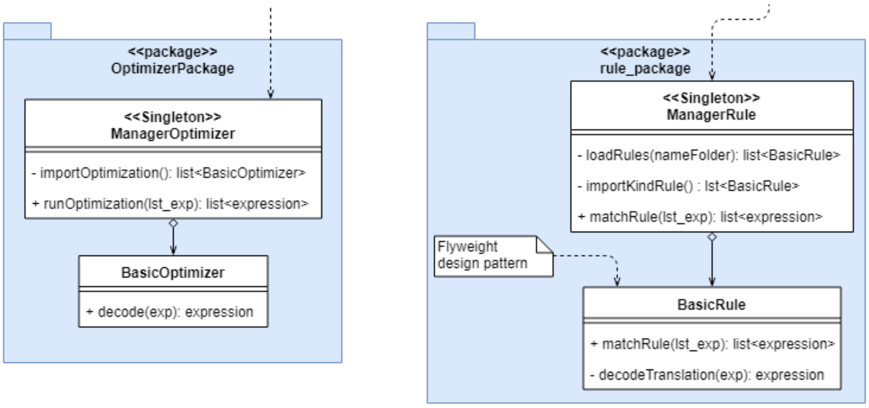

<<package>>
OptimizerPackage
<<Singleton>>
ManagerOptimizer
- importOptimization(): list<BasicOptimizer>
+ runOptimization(lst_exp): list<expression>
BasicOptimizer
+ decode(exp): expression
<<package>>
rule_package
<<Singleton>>
ManagerRule
- loadRules(nameFolder): list<BasicRule>
- importKindRule() : lst<BasicRule>
+ matchRule(lst_exp): list<expression>
Flyweight
design pattern
BasicRule
+ matchRule(lst_exp): list<expression>
- decodeTranslation(exp): expression

Figure 15 Class Diagram of management of Optimization and Translation Rule class

The main motivation in order to split the second and third layers of optimization is to reduce the number of classes to be written by the user.

As mentioned earlier, the second level implements a peephole optimization, which is done by examining a sliding window of targets and replacing instructions sequences within the peephole by a shorter or faster sequence, whenever possible. The characteristic of a peephole is that the intermediate code shouldn't be contiguous, although some implementations do require it. This optimization class is subdivided in other optimization techniques:

- Redundant-instruction eliminations
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms

The *Redundant-instruction eliminations* can be used by given it a set of instructions, there is a smaller one that has the same result, this optimization type is applied on LOAD and STORE instruction in a generic compiler.

The *Flow-of-control optimizations* used in order to simplify all the jump generated in an intermediate code set.

The *usage of machine idioms* can be used if the target machine may have hardware instructions to implement certain specific operations efficiently.

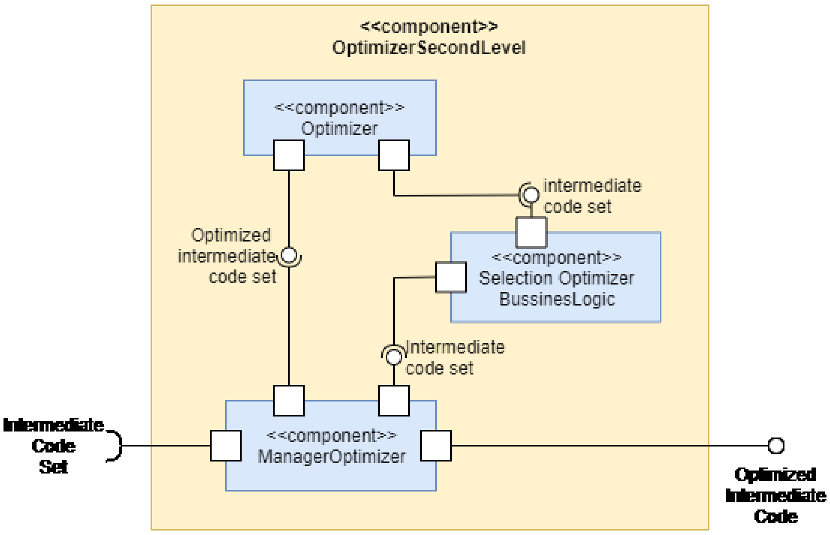


Figure 16 - Component Diagram of OptimizerSecondLevel

The second level is characterized by "low-level" optimization directly on intermediate code. Those classes are defined by inheritance from *"BasicOptimizer"*, which work directly on classes of Expression that represent the intermediate code set. The function of these classes is to give an input expression of intermediate code and obtain an optimized one, with a smaller or equal size to the input one. In this optimization level, the defined technique is "*logic transformation"* in order to reduce the number of logic ports used in generic expression and "*constant folding"* technique.

The component diagram, in figure, show how it is structured. It is based on three subcomponents:

- *ManagerOptmizer* component represents the scan of intermediate code
- *SelectionOptmizer BussineLogic* component represents the method provided to select the right intermediate code subsequence, in order to optimize it.
- *Optmizer* component represents the generic component used to simplify the intermediate code set.

The optimization of the arithmetic and logic expression are evaluated if the constants are acquired in input and in order to replace the whole expression with its result. The arithmetic and/or logic expression should be evaluated the same way at compile time as they are at run time.

An example of an optimization class is the *NotOptimizer*, which, as the name implies, deals with optimization of the *NOT* logical port. The algorithm defined by this optimizer is showed in figure.

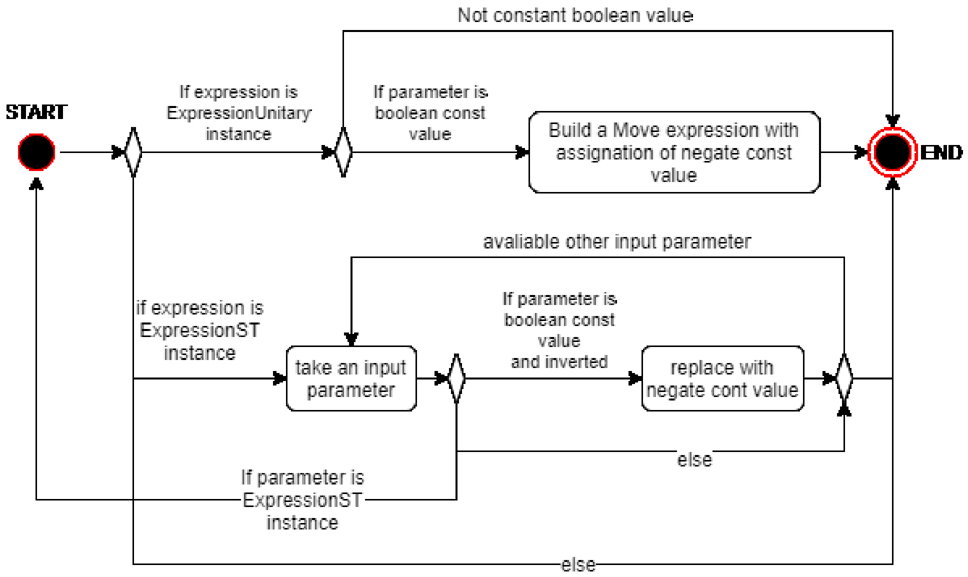

Not constant boolean value
START
If expression is ExpressionUnitary instance
If parameter is boolean const value
Build a Move expression with assignation of negate const value
END
avaliable other input parameter
if expression is ExpressionST instance
take an input parameter
If parameter is boolean const value and inverted
replace with negate cont value
else
If parameter is ExpressionST instance
else

Figure 17 Not Optimization port - Activity diagram

If the expression passed at *NotOptimizer* is an *ExpressionUnitary* instance and the input parameter is a constant boolean value, then replace the not logic port expression with a move expression where the negated value is directly assigned.

In another case, the expression passed is an *ExpressionST* instance, all input parameters are checked. If the input parameter is a constant boolean value and the input is inverted then the input is replaced with negate a boolean value, if the input parameter is an *ExpressionST* instance then start an optimization of this expression.

Once performed, these techniques defined in this chapter, if they have been applied to at least one set of intermediate instructions, are then rerun to see if there are other plausible optimizations to be performed.

4.1.4 Optimization with JSON rules

The optimization type applied in this level is the replacement of intermediate code sets with other intermediate code sets, in order to implement techniques like

The idea of improving the optimization of code with the use of prewritten rules in JSON format, where it is structured in two main sections: RULE, ACTION. The rule in this way is not written directly inside the code but written through external files, in order to simply the writing of this file.

In *RULE* section defines the name function of blocks to optimize relative input parameters and relative output parameters.

In *ACTION* section defines how to translate the group of blocks matched with the rule defined earlier, with specific actions. This section can be seen as the passage of useful parameters to the class that defines the optimization action itself.

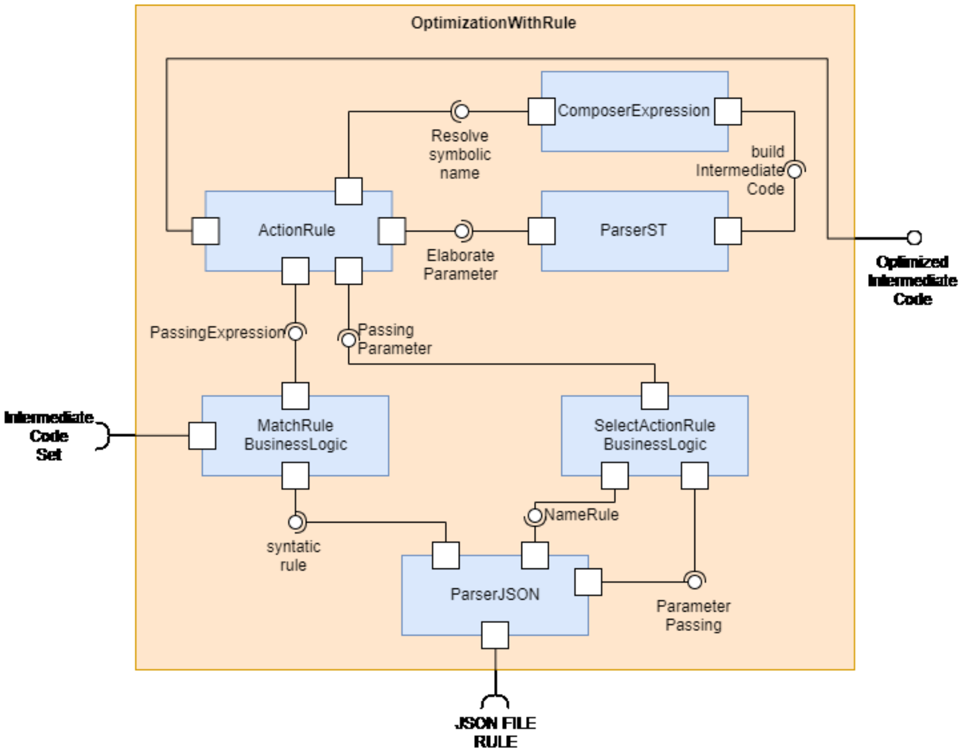


Figure 18 - Component Diagram OptimizationWithRule Component

This component diagram explains briefly, how this level of optimization with rule is structured, the components used are:

- *ParserJSON* component implemented by python library.
- *ActionRule* component implemented by BasicRule hierarchy.
- *MatchRule* component represents the algorithm provided to find the intermediate code set that is respective of the rule written in JSON file.
- *SelectionActionRule* component represents the method provided to select the right ActionRule component and pass the parameter reads by JSON file.
- *ParserST* component implemented by ParserST class, represents a parser for limited ST language.

- *ComposerExpression* component implemented by ComponentExpression class, represents the generator of intermediate code, from syntax tree of limited ST language.

The class ParserST shall parse only move, math, boolean and comparison expression, with the utilization, the output of parser is a syntax tree based on ***R**everse **P**olish **n**otation,* we'll discuss it later, and the instance object Token. The class ComposerExpression picked input the syntax tree result from ParserST return intermediate code.

The optimization action is described in class derived from *BasicRule* class. In this class type, the actions performed are: a recognition of the input blocks pattern, replace the name in parameter passed from json file, and generate a new intermediate code to replace with the intermediate code matched.

The first step, as mentioned earlier, is movement of the blocks pattern defined in RULE section. In order to perform this step, each expression in intermediate code is analysed in order to search the same type of parameter and the same type of block defined in code.
The parameters that will be passed in the *RULE* section therefore symbolically define the names of the functions or objects that can be used in ST or FBD language, with the related attributes.

In the second step we should replace the name parameter with the optimizer actor. The *name resolution* is the resolution of the tokens within program expressions to the intended program components.
In this way, it is possible to generate new intermediate code with correct input and output parameter.

In the third step, new intermediate code is generated following the speciation defined in optimizer and in parameter passed in JSON file. The parameter sometimes can also be simple *boolean/math/assign* expression. In order to implement this last system, it uses the *Shunting-yard* algorithm to covert the expression parameter written from *infix mode* to *postfix mode*.
The *Shunting Yard Algorithm* is an *operator precedence parser*, used to turn the infix mathematic expression to "***R**everse **P**olish **n**otation*" (RPN), which can be calculated easily, invented by *Edsger Dijkstra*. Its name comes from the use of a stack to rearrange the operators and operands into the correct order for evaluation, which is rather reminiscent of a railway siding.
This algorithm is based on *shift-reduce parser*. Generally, the shift-reduce parser works on two operation:

- **Shift**, this step advances in the input stream by one symbol.
- **Reduce**, this step uses the grammar in order to join all parsed trees.

As in RPN, we scan the formula from left to right, processing each operand and operator in order. The complexity request to build an RPN with this algorithm is *O(N)* and the complexity to interpret the expression is *O(N)*.

The parserST must respect the requirements:

- Parser shall recognize the assigned expression;
- Parser shall recognize the comparison expression;
- Parser shall recognize the math expression;
- Parser shall recognize the boolean expression;
- Parser shall permit the sub-expression inside expression, with usage of more operators;
- Parser shall use the parenthesis in order to implement a sub-expression precedence.

The parser based on Shunting yard Algorithm needs a stack for operations, a queue of the output and an array of tokens.

```
While (tokens stack is not empty):
      token = pop from stack
      If (token is number instance) add token to queue
      If (token is operator)
           While (token operator on the top of the stack with greater precedence):
                   Pop operators from the stack onto the output queue
           Push the current operator onto the stack
      If (token is '(') push '(' onto the stack
      If (token is ')')
           While (there's not a left bracket at the top of the stack):
                    Pop operators from the stack onto the output queue.
           Pop the left bracket from the stack and discard it
While (operators on the stack):
      pop them to the queue
```

The tokens are represented by Token instance class. This class has two attributes: value, type. The value represents the textual value of token. The type represents the token type. The token type is implemented with an enumerator structured by five elements: constantValue, operand, pinName, assignOp, ExpressionNamingResolved.

The operator precedence used in this algorithm is defined in IEC61131-3 standard, in the following table is a resume of them.

No.	OPERATION	Symbol	Precedence
1	Parenthesis	(expression)	HIGHEST
2	Function evaluation	identifier (argument list)	
3	Exponentiation	**	
4	Negation, Complement	-, NOT	
5	Multiply, Divide, Module	*, /, MOD	
6	Add, Subtract	+, -	
7	Comparison	< , > , <= , >=	
8	Equality, Inequality	=, <>	
9	Boolean AND	AND, &	
10	Boolean Exclusive OR, Boolean OR	XOR, OR	LOWER

The function evaluation, as mentioned earlier, is not implemented in this version of parsing and generation of intermediate code, because the idea is to replace statements representing function calls with simpler statements.

The generation of intermediate code defined in followed state machine, in figure.

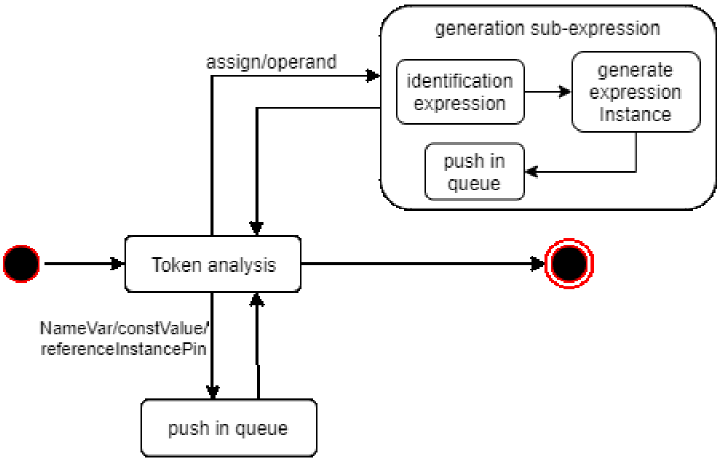
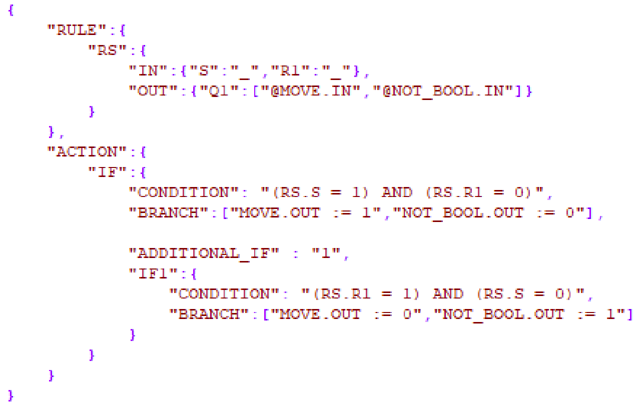

generation sub-expression
identification expression
generate expression Instance
push in queue
assign/operand
Token analysis
NameVar/constValue/ referenceInstancePin
push in queue

Figure 19 Parser Expression restricted ST - state machine

The ST expression parsed from this state machine is restricted at operation of assignment and math operation (numerical and boolean), because the goal is transforming part of FB instance in a simple set of expressions.

An example of class describing an optimization action is *RuleIF*. The goal of this transformation is taking a set of intermediate code and replacing it with a set of if-statements with relative branches.

An example of *RuleIF* class use is the json rule called "*RuleRS.json*" showed in the figure below.



```
{
    "RULE":{
        "RS":{
            "IN":{"S":"_","R1":"_"},
            "OUT":{"Q1":["@MOVE.IN","@NOT_BOOL.IN"]}
        }
    },
    "ACTION":{
        "IF":{
            "CONDITION": "(RS.S = 1) AND (RS.R1 = 0)",
            "BRANCH":["MOVE.OUT := 1","NOT_BOOL.OUT := 0"],

            "ADDITIONAL_IF" : "1",
            "IF1":{
                "CONDITION": "(RS.R1 = 1) AND (RS.S = 0)",
                "BRANCH":["MOVE.OUT := 0","NOT_BOOL.OUT := 1"]
            }
        }
    }
}
```

Figure 20 JSON instance of rule

The goal of ruleRS is converting from instance with something in input and output used from a move and a not logic port, to two if-statements with relative branches. Specifically, what we want is to translate, with this rule, is an instance like this:

```
RS_INSTANCE(S := VAR_A, R1 := VAR_B);
VAR_OUT_MOVE := RS_INSTANCE.Q;
VAR_OUT_NOT := NOT(RS_INSTANCE.Q);
```

Into

```
IF (VAR_A = 1 AND VAR_B = 0) THEN
   VAR_OUT_MOVE := 1;
   VAR_OUT_NOT := 0;
END_IF;
IF (VAR_A = 0 AND VAR_B = 1) THEN
   VAR_OUT_MOVE := 0;
   VAR_OUT_NOT := 1;
END_IF;
```

Once performed, these techniques defined in this chapter, if applied to at least one set of intermediate instructions, are then rerun to see if there are other plausible optimizations to be performed.

4.1.5 Translation from Intermediate Code to ST

The last phase converts the intermediate code into ST language. Starting from the intermediate code set, the translation of the code now takes place from a one-by-one correspondence. In order to implement this translation, each class of hierarchy Expression has redefined the method of casting in string (in this specific case, in python is *"__str__"*) in order to simplify the translation of the intermediate code, transforming all the translation action into a "*toString*" method call.

In the following list is a little resume of rules in order to convert Intermediate code into ST language. These translation rules defined here can also be found in the *Control Expert Scheneider's* reference manual. The manual shows how to write a specific function in all available languages.

In the scenario of *ExpressionMove* it can be translated in two ways:

- If is a sub-expression: *'(' <input parameter> ')'*
- *<output parameter> := ' (' <input parameter> ');'*

In the scenario of *ExpressionUnitary* it can be translated in two ways:

- If is a sub-expression: *'NOT(' <input parameter> ')'*
- *<output parameter> := 'NOT(' <input parameter> ');'*

In the scenario of *ExpressionInstance* it can be translated in the following ways:

- *<nameInstance> '(' <nameParameterInput1> := <valueParameterInput1> ',' ...*
  <nameParameterInputN> := <valueParameterInputN>);
- *<nameInstance> '(' <nameParameterInput1> := <valueParameterInput1> ',' ...*
  <nameParameterInputN> := <valueParameterInputN>','
  <nameParameterOutput1> => <valueParameterOutput1> ',' ...
  <nameParameterOutputN>=> <valueParameterOutputN>);

In this scenario, explanatory comments are also written regarding the object initially defined in the initial code. The information in comment are isoforms between the initial block in FBD and the result in ST language.

In the scenario of *ExpressionComparision it* can be translated in four ways:

- *'(' <parameter1> <textualOperator> <parameter2> ')'*
- *<output parameter> := <parameter 1> <textualOperator> <parameter 2> ';'*
- *'(' <parameter1><textualOperator><parameter2> 'AND'…'AND' <parameter N-1> <textualOperator> <parameter N> ')'*
- *<output parameter> := <parameter1> <textualOperator> <parameter2> 'AND' … 'AND' <parameter N-1> <textualOperator> <parameter N> ';'*

Where the *<textualOperator>* can be: <, >, <>, =, >=, <=.

In the scenario of *ExpressionMath it* can be translated in two ways:

- *'(' <input parameter1> <textMathOp> <input parameter2> <textMathOp> … <inputMathOpN> ')'*
- *<output variable> ':=' <input parameter1> <textMathOp> <input parameter2> <textMathOp> … <inputMathOpN> ';'*

Where the *<textMathOp>* can be: + addition, - subtraction, / division, * multiplication, ** exponential, MOD module.

In the scenario of *ExpressionST* it can be translated like the *ExpressionMath*, but the *<textMathOp>* token take other values: AND, OR, XOR.

When all intermediate code is converted into ST language, the code is saved in two textual files. The first file contains the source code generated by translation. The second file contains the name and type of variables used inside the program.

Chapter 5: Testing

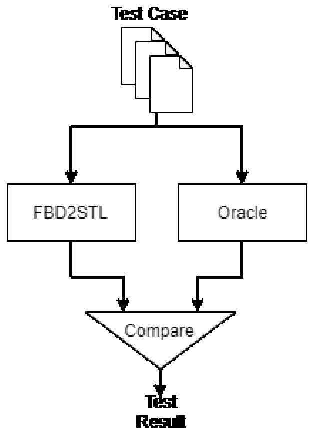


Figure 21 - Test Case compared with oracle

The success of the execution of a test case should be evaluated in an objective way. In order to design a test case, the Oracle shall know exactly what is the expected output of the system under the test in response to the test case. An oracle test is a mechanism, different from the program in exam that can be used to check the correctness of the output of the program for the test cases. [14]

Oracle test runs on an oracle program and it must be developed. Like the project, an oracle is built from the requirement specification and it is subject to interpretation by the developer and may contain faults. A faulty oracle can be trouble, because it may return false positives or false negatives.[12] For this, it does not have to be very complex so that the system can be considered reliable. The Oracle knows the expected behaviour of the software system for each test case.
The Comparator can compare the oracle's expected behaviour and the tested software behaviour and evaluate if a failure has occurred.

In 1960 in the *Software Engineering Techniques*, Dijkstra said "Testing shows the presence, not the absence of bugs". This means the correctness of a program is a problem undecidable.
So, a testing process must allow you to purchase trusted software, showing that it is ready for operational use. [15]
In an ideal world, a failure of the test implies correctness of the program and contains all combinations of the input data to the program. These test types are not always practical because the combinations of the input data can be a lot.
Exhaustive testing consists of the execution of all the possible behaviours of a software system, if exhaustive testing does not show any failure, the program is correct.[15]
In order to describe the test case specification, we used a table. The tables are structured in followed way:

- identification numbers
- a precondition that is a hypothesis that must be verified before the test is executed
- input values
- Expected output values (Oracle)
- Postconditions, that is a Hypothesis that must be verified after the test is executed

In order to test the single component of this project we use unit tests and at finally a system test.
A unit is the smallest testable part of any software. It usually has one or a few inputs and usually a single output.
All test mentioned are in blackbox mode.
The unit test is performed by ***unittest*** (Unit testing framework) defined in python. The unit testing framework supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework.[11] In order to test the reduced ST parsed, the following class was written following the *unittest* model, previously mentioned.

4.1 UnitTest ParserST for third level optimization

In third level, as mentioned earlier, the json file can contain some parameters written with ST language syntax in order to understand the parameters, we use a parser and a generator of intermediate code.
In order to test the part of parsing and the generation of intermediate code a unit test has been unitized.

This test case specification is used in order to test the component ParserST implemented inside the class ParserST.

ID	Pre condition	Input	Expected Output	Post condition
1		"A := 14"	[Token ("A", TokenType.PinName), Token ("14", TokenType.constantValue), Token (":=", TokenType.assignOp)]	
2		*"MOVE.OUT:=21 - MOVE.IN* 33/(1+2)*"	[Token ("MOVE.OUT", TypeToken.pinName), Token ("21", TypeToken.constantValue), Token ("MOVE.IN", TypeToken.pinName), Token ("33", TypeToken.constantValue), Token ("*", TypeToken.operand), Token ("1", TypeToken.constantValue) Token ("2", TypeToken.constantValue), Token ("+", TypeToken.operand), Token ("/", TypeToken.operand), Token ("-", TypeToken.operand), Token (":=",TypeToken.assignOp)]	
3		*"A:=((b+c)*(SEL.IN-SEL.OUT))/5"*	[Token ("A", TypeToken.pinName), Token ("b", TypeToken.pinName), Token ("c", TypeToken.pinName), Token ("+", TypeToken.operand), Token ("SEL.IN", TypeToken.pinName),	

			Token ("SEL.OUT", TypeToken.pinName), Token ("-", TypeToken.operand), Token ("*", TypeToken.operand), Token ("5", TypeToken.constantValue), Token ("/", TypeToken.operand), Token (":=", TypeToken.assignOp)]	
4		*“RS.S = 0 AND RS.R1 = 1”*	[Token ("RS.S", TypeToken.pinName), Token ("0", TypeToken.constantValue), Token ("=", TypeToken.operand), Token ("RS.R1", TypeToken.pinName), Token ("1", TypeToken.constantValue), Token ("=", TypeToken.operand), Token ("AND", TypeToken.operand)]	
5		*“NOT(RS.S)=0 OR RS.S=1”*	[Token ("RS.S", TypeToken.pinName), Token ("NOT", TypeToken.operand), Token ("0", TypeToken.constantValue), Token ("=", TypeToken.operand), Token ("RS.S", TypeToken.pinName), Token ("1", TypeToken.constantValue), Token ("=", TypeToken.operand), Token ("OR", TypeToken.operand)]	
6		*“(A XOR (B OR C) AND D) = 0”*	[Token ("A", TypeToken.pinName), Token ("B", TypeToken.pinName), Token ("C", TypeToken.pinName), Token ("OR", TypeToken.operand), Token ("XOR", TypeToken.operand), Token ("D", TypeToken.pinName), Token ("AND", TypeToken.operand), Token ("0", TypeToken.constantValue), Token ("=", TypeToken.operand)]	
7		*“MOVE.OUT:=(MOVE.IN+15)MOD 126”*	[Token ("MOVE.OUT", TypeToken.pinName), Token ("MOVE.IN", TypeToken.pinName),	

			Token ("15", TypeToken.constantValue), Token ("+", TypeToken.operand), Token ("126", TypeToken.constantValue), Token ("MOD", TypeToken.operand), Token (":=",TypeToken.assignOp)]	
8		*“A := NOT(B AND C) OR D”*	[Token ("A", TypeToken.pinName), Token ("B", TypeToken.pinName), Token ("C", TypeToken.pinName), Token ("AND", TypeToken.operand), Token ("NOT", TypeToken.operand), Token ("D", TypeToken.pinName), Token ("OR", TypeToken.operand), Token (":=", TypeToken.assignOp)]	

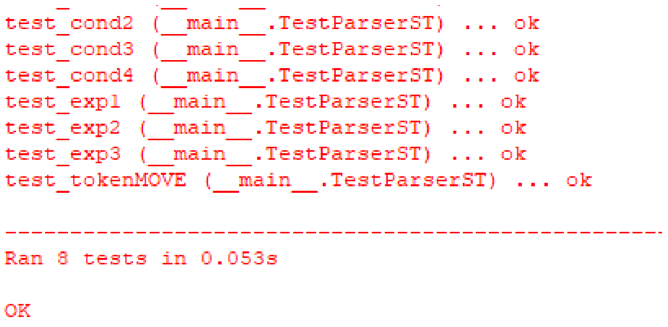
As mentioned earlier, the unit test is performed with unittest framework of python, in the following text is an extract of this.

```python
import ParserST
import unittest
from collections import deque
from Token import Token
from Token import TypeToken

class TestParserST(unittest.TestCase):
    def test_tokenMOVE(self):
        ...
    def test_exp1(self):
        ...
    def test_exp2(self):
        ...
    def test_exp3(self):
        ...
    def test_cond1(self):
        ...
    def test_cond2(self):
        move_deque = ParserST.parseExpression("NOT(RS.S) = 0 OR RS.S = 1");
        deque_test = deque();
        deque_test.append(Token("RS.S",TypeToken.pinName));
        deque_test.append(Token("NOT",TypeToken.operand));
        deque_test.append(Token("0",TypeToken.constantValue));
        deque_test.append(Token("=",TypeToken.operand));
        deque_test.append(Token("RS.S",TypeToken.pinName));
        deque_test.append(Token("1",TypeToken.constantValue));
        deque_test.append(Token("=",TypeToken.operand));
        deque_test.append(Token("OR",TypeToken.operand));
        self.assertEqual(move_deque, deque_test);
    def test_cond3(self):
        ...
    def test_cond4(self):
        ...
    def test_exp3(self):
        ...

if __name__ == '__main__':
    unittest.main();
```

The result of these tests is positive and therefore without errors. In the following figures are the result of test.



```
test_cond2 (__main__.TestParserST) ... ok
test_cond3 (__main__.TestParserST) ... ok
test_cond4 (__main__.TestParserST) ... ok
test_exp1 (__main__.TestParserST) ... ok
test_exp2 (__main__.TestParserST) ... ok
test_exp3 (__main__.TestParserST) ... ok
test_tokenMOVE (__main__.TestParserST) ... ok

----------------------------------------------------------------------
Ran 8 tests in 0.053s

OK
```

Figure 22 - Result test on Parser ST

4.2 Unit Test of intermediate code generator for optimizer

In order to test the ComposerExpression component of third level optimization component, we built another unit test.

ID	Pre condition	Input	Expected Output	Post condition
1		A := 14	ExpressionMove(“A”, “14”)	

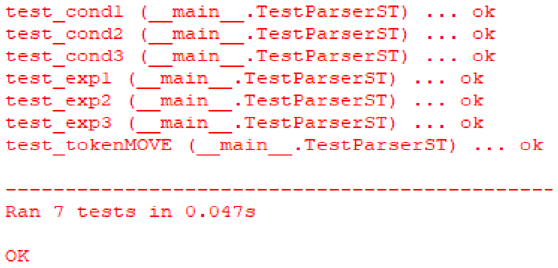
2		A := (5 - B) * C	T = ExpressionMath(“-”, “5”, “B”)
			T2 = ExpressionMath(“*”, T, “C”)
			ExpressionMove(“A”, T2)	

3		NOT(A) = B	T = ExpressionUnitary(“NOT”, “A”)
			ExpressionComparision(“=”, T, “B”)	

4		A AND B = C OR D >= 10.5	T = ExpressionComparision(“=”, “B”, “C”)
			T2 = ExpressionST(“AND”, “A”, T)
			T3 = ExpressionComparision(“>=”,“D”, “10.5”)
			ExpressionST(“OR”, T2, T3)	

5		(A XOR B) = (B AND C)	T = ExpressionST(“XOR”, “A”, “B”)
			T2 = ExpressionST(“AND”, “B”, “C”)
			ExpressionComparision(“=”, T, T2)	

The result of these tests is positive and therefore without errors. In the following figures are the result of test.



```
test_cond1 (__main__.TestParserST) ... ok
test_cond2 (__main__.TestParserST) ... ok
test_cond3 (__main__.TestParserST) ... ok
test_exp1 (__main__.TestParserST) ... ok
test_exp2 (__main__.TestParserST) ... ok
test_exp3 (__main__.TestParserST) ... ok
test_tokenMOVE (__main__.TestParserST) ... ok

----------------------------------------------
Ran 7 tests in 0.047s

OK
```

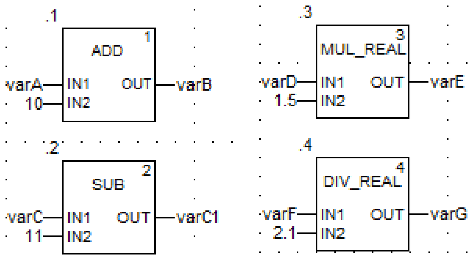
Figure 23 - Result Unit Test generation intermediate code

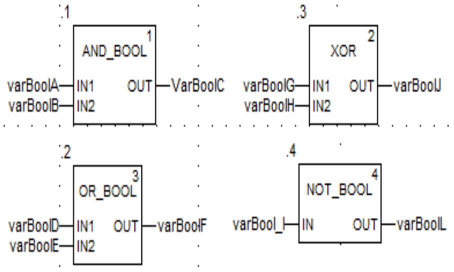
4.3 Unit Test Analysis phase

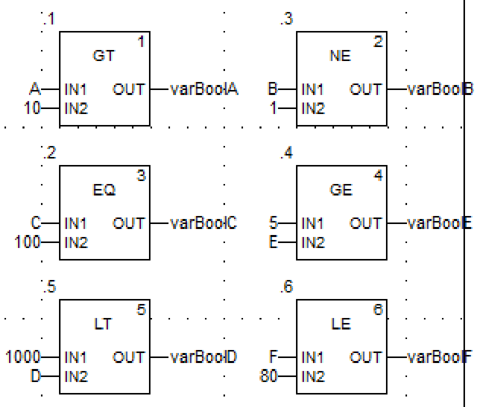
In order to test this other component of the project, such in precedence, we used a unit test. In this case in order to test the component for analysis we used a set of “.xbd” file, that represent FBD program. In the next list, is a description of the input and output test.

The Expected Output, except the first output, represent the array of output DAG

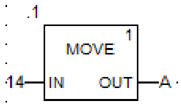
ID	Pre condition	Input	Expected Output	Post condition
1		Xml file with attributes not correspondent at DTD file	“File not good”	

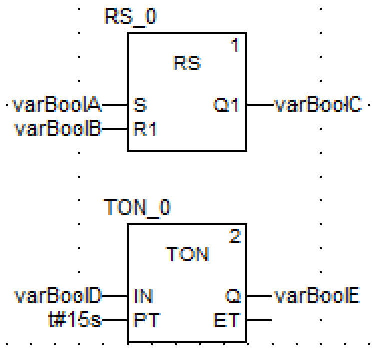
2		File input XML describe this FBD:  .1
		ADD
		1
		varA
		IN1
		OUT
		varB
		10
		IN2
		.2
		SUB
		2
		varC
		IN1
		OUT
		varC1
		11
		IN2
		.3
		MUL_REAL
		3
		varD
		IN1
		OUT
		varE
		1.5
		IN2
		.4
		DIV_REAL
		4
		varF
		IN1
		OUT
		varG
		2.1
		IN2
			T1 = BlockFDB(“ADD”, “.1”)
			T1.map_in({“IN1”:“varA”, “IN2” : “10”})
			T1.map_out({“OUT”: “varB”})
			T2 = BlockFDB(“SUB”, “.2”)
			T2.map_in({“IN1”:“varC”, “IN2”: “11”})
			T2.map_out({“OUT”: “varC1”})
			T3 = BlockFDB(“MUL”, “.3”)
			T3.map_in({“IN1”:“varD”, “IN2”: “1.5”})
			T3.map_out({“OUT”: “varE”})
			T4 = BlockFDB(“DIV”, “.4”)
			T4.map_in({“IN1”:“varF”, “IN2”: “2.1”})
			T4.map_out({“OUT”: “varG”})	

3		File input XML describe this FBD:  .1
		AND_BOOL
		1
		varBoolA
		varBoolB
		IN1
		IN2
		OUT
		VarBoolC
		.3
		XOR
		2
		varBoolG
		varBoolH
		IN1
		IN2
		OUT
		varBoolJ
		.2
		OR_BOOL
		3
		varBoolD
		varBoolE
		IN1
		IN2
		OUT
		varBoolF
		.4
		NOT_BOOL
		4
		varBool_I
		IN
		OUT
		varBoolL
			T1 = BlockFDB(“AND”, “.1”)
			T1.map_in({“IN1”:“varBoolA”, “IN2” : “varBoolB”})
			T1.map_out({“OUT”: “varBoolC”})
			T2 = BlockFDB(“OR”, “.2”)
			T2.map_in({“IN1”:“varBoolD”, “IN2” : “varBoolE”})
			T2.map_out({“OUT”: “varBoolF”})
			T3 = BlockFDB(“XOR”, “.3”)
			T3.map_in({“IN1”:“varBoolG”, “IN2” : “varBoolH”})
			T3.map_out({“OUT”: “varBoolJ”})
			T4 = BlockFDB(“NOT”, “.1”)
			T4.map_in({“IN”: “varBool_I”})
			T4.map_out({“OUT”: “varBoolL”})	

4		File input XML describe this FBD:  .1
		GT
		1
		A
		10
		IN1
		IN2
		OUT
		varBoolA
		.3
		NE
		2
		B
		1
		IN1
		IN2
		OUT
		varBoolB
		.2
		EQ
		3
		C
		100
		IN1
		IN2
		OUT
		varBoolC
		.4
		GE
		4
		5
		E
		IN1
		IN2
		OUT
		varBoolE
		.5
		LT
		5
		1000
		D
		IN1
		IN2
		OUT
		varBoolD
		.6
		LE
		6
		F
		80
		IN1
		IN2
		OUT
		varBoolF
			T1 = BlockFDB(“GT”, “.1”)
			T1.map_in({ “IN1” : “A”, “IN2” :“10”})
			T1.map_out({“OUT”: “varBoolA”})
			T2 = BlockFDB(“EQ”, “.2”)
			T2.map_in({“IN1”:“C”, “IN2” : “100”})
			T2.map_out({“OUT”: “varBoolC”})
			T3 = BlockFDB(“NE”, “.3”)
			T3.map_in({“IN1”:“B”, “IN2” : “1”})	

			T3.map_out({“OUT”: “varBoolB”})
			T4 = BlockFDB(“GE”, “.4”)
			T4.map_in({ “IN1” : “1000”, “IN2”: “D”})
			T4.map_out({“OUT”: “varBoolD”})
			T5 = BlockFDB(“GE”, “.5”)
			T5.map_in({“IN1”: “5”, “IN2”: “E”})
			T5.map_out({“OUT”: “varBoolE”})
			T6 = BlockFDB(“LE”, “.6”)
			T6.map_in({“IN1”: “F”, “IN2”: “80”})
			T6.map_out({“OUT”: “varBoolF”})	

		File input XML describe this FBD:
		
			T = BlockFDB(“MOVE”, “.1”)
			T.map_in({“IN”: “14”})
			T.map_out({“OUT”: “A”})	

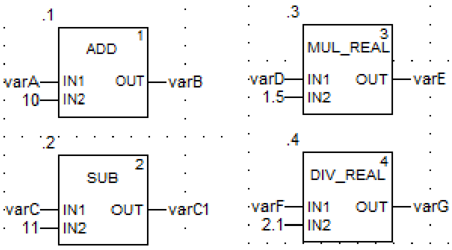
		File input XML describe this FBD:
		 RS_0
		RS
		1
		varBoolA
		S
		varBoolB
		R1
		Q1
		varBoolC
		TON_0
		TON
		2
		varBoolD
		IN
		t#15s
		PT
		Q
		varBoolE
		ET
		The name program is “CODE”
			T = BlockFDB(“RS”, “LV_CODE_RS”)
			T.map_in({“S”: “varBoolA”, “R1”: “varBoolB”})
			T.map_out({“Q1”: “varBoolC”})
			T = BlockFDB(“RS”, “LV_CODE_TON”)
			T.map_in({“IN”: “varBoolD”, “R1”: “varBoolB”})
			T.map_out({“Q”: “varBoolE”})	

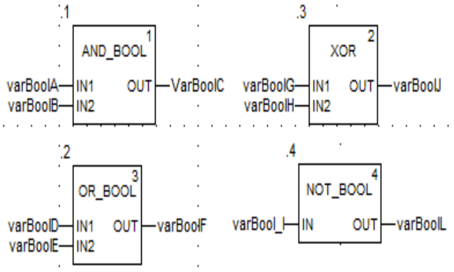
4.3 Unit Test Intermediate code generator

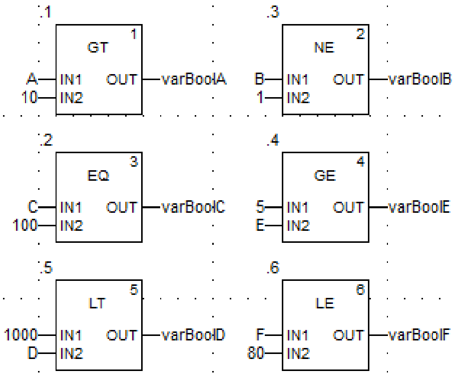
In order to test this other component of project, such in precedence, is used a unit test. In this case in order to test the component for analysis is used a set of ".xbd" file, that represent FBD program. In the next list, is described the input and output of test.

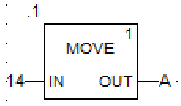
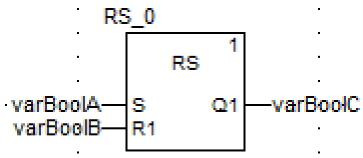
The Expected Output, except the first output, represent the array of intermediate code set. In order to implement this test is supposed the correctness of Analysis phase.

ID	Pre condition	Input	Expected Output	Post condition

1		File input XML describe this FBD:
		
		 .1
		ADD
		1
		varA
		10
		IN1
		IN2
		OUT
		varB
		.2
		SUB
		2
		varC
		11
		IN1
		IN2
		OUT
		varC1
		.3
		MUL_REAL
		3
		varD
		1.5
		IN1
		IN2
		OUT
		varE
		.4
		DIV_REAL
		4
		varF
		2.1
		IN1
		IN2
		OUT
		varG
			ExpressionMath("ADD", "varB", "varA", "10")
			ExpressionMath("SUB", "varC1", "varC", "11")
			ExpressionMath("MUL", "varE", "varD", "1.5")
			ExpressionMath("DIV", "varG", "varF", "2.1")	

3		File input XML describe this FBD:
		
		 .1
		AND_BOOL
		1
		varBoolA
		varBoolB
		IN1
		IN2
		OUT
		VarBoolC
		.3
		XOR
		2
		varBoolG
		varBoolH
		IN1
		IN2
		OUT
		varBoolJ
		.2
		OR_BOOL
		3
		varBoolD
		varBoolE
		IN1
		IN2
		OUT
		varBoolF
		.4
		NOT_BOOL
		4
		varBool_I
		IN
		OUT
		varBoolL
			ExpressionST("AND", "varBoolC","varBoolA", "varBoolB")
			ExpressionST("OR", "varBoolF", "varBoolD", "varBoolE")
			ExpressionST("XOR", "varBoolG", "varBoolH", "varBoolJ")
			ExpressionUnitary("NOT", "varBoolL", "varBool_I")	

4		File input XML describe this FBD:
		
		 .1 GT 1 A IN1 10 IN2 OUT varBoolA
		.3 NE 2 B IN1 1 IN2 OUT varBoolB
		.2 EQ 3 C IN1 100 IN2 OUT varBoolC
		.4 GE 4 5 IN1 E IN2 OUT varBoolE
		.5 LT 5 1000 IN1 D IN2 OUT varBoolD
		.6 LE 6 F IN1 80 IN2 OUT varBoolF
			ExpressionComparison(“GT”, “varBoolA”, “A”, “10”)
			ExpressionComparison(“EQ”, “varBoolC”, “C”, “100”)
			ExpressionComparison(“NE”, “varBoolB”, “B”, “1”)
			ExpressionComparison(“GE”, “varBoolE”, “5”, “E”)
			ExpressionComparison(“LT”, “varBoolD”, “1000”, “D”)
			ExpressionComparison(“LE”, “varBoolF”, “F”, “80”)	

		File input XML describe this FBD:  	ExpressionMove(“MOVE”, “A”, “14”)	
		File input XML describe this FBD:  The name program is “CODE”	ExpressionInstance(“RS”, “LV_CODE_RS”, { “S : “varBoolA”, “R1”: “varBoolB”}, {“Q1”: “varBoolC”})	

4.5 System test Project

In order to test the entire project, we must use a system test, oracle testing based.
In this case, the program in order to compare the FBD2ST system result and Oracle result, has been deployed following the diagram in figure.

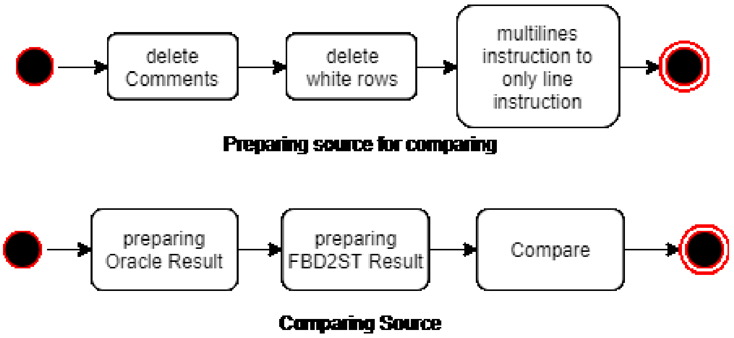


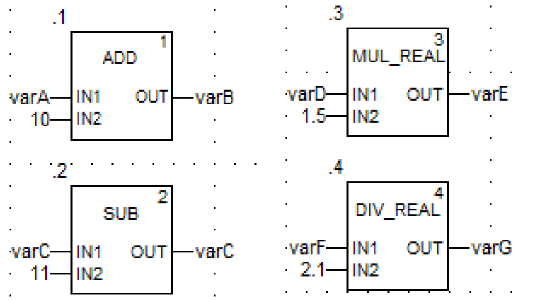
Figure 24 - Oracle Test System

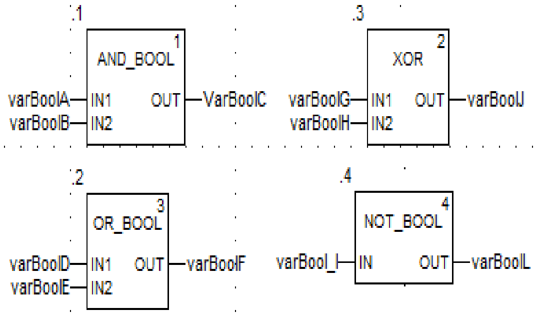
The first chain used in order to prepare the text file output from oracle and FBD2ST system to be compared, in the second chain is the real comparison. The first chain is structured in the following action:

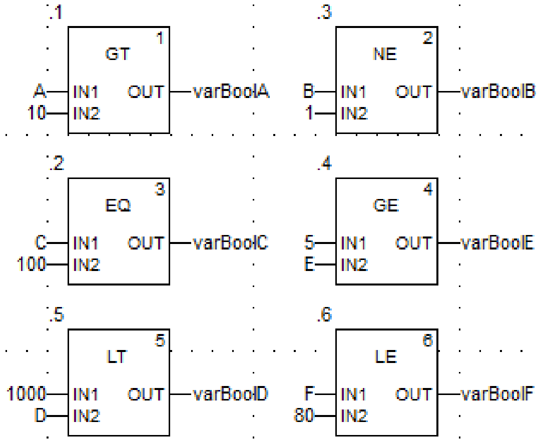
- delete comments, in this step the source is brought like an array of string, and all comments described inside the code are deleted, the only part remaining is the code.
- delete white rows, in this step delete all white rows and delete the blank space inside the other rows.
- in third step, put all the lines that make up an instruction on one line.

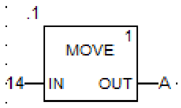
This token, generated by source code generate from FBD2ST and Oracle source code result, is compared one by one in order to understand if the result is equal or not. In the next table we present the test case specification based on requirements of system.

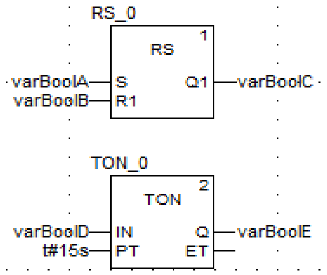
ID	Pre condition	Input	Expected Output	Post condition
1		Xml file with attributes not correspondent at DTD file	“File not good”	

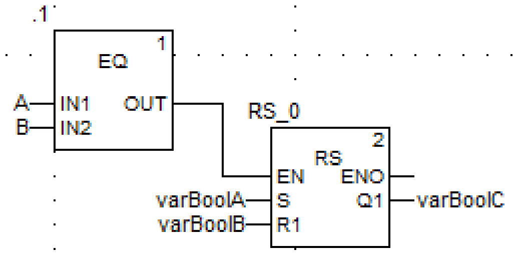
2		File input XML describe this FBD:
		 .1
		ADD
		1
		varA
		10
		IN1
		IN2
		OUT
		varB
		.2
		SUB
		2
		varC
		11
		IN1
		IN2
		OUT
		varC
		.3
		MUL_REAL
		3
		varD
		1.5
		IN1
		IN2
		OUT
		varE
		.4
		DIV_REAL
		4
		varF
		2.1
		IN1
		IN2
		OUT
		varG
			varB := varA + 10;
			varC := varC - 11;
			varE := varD * 1.5;
			varG := varF / 2.1;	

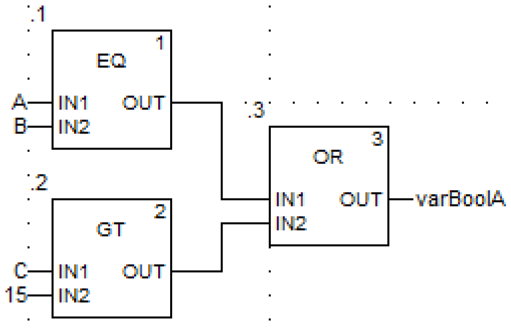
3		File input XML describe this FBD:
		 .1
		AND_BOOL
		1
		varBoolA
		varBoolB
		IN1
		IN2
		OUT
		VarBoolC
		.3
		XOR
		2
		varBoolG
		varBoolH
		IN1
		IN2
		OUT
		varBoolJ
		.2
		OR_BOOL
		3
		varBoolD
		varBoolE
		IN1
		IN2
		OUT
		varBoolF
		.4
		NOT_BOOL
		4
		varBool_I
		IN
		OUT
		varBoolL
			VarBoolC := varBoolA
			AND varBoolB;
			varBoolF := varBoolD
			OR varBoolE;
			varBoolJ := varBoolG
			XOR varBoolH;
			varBoolL :=
			NOT(varBool_I);	

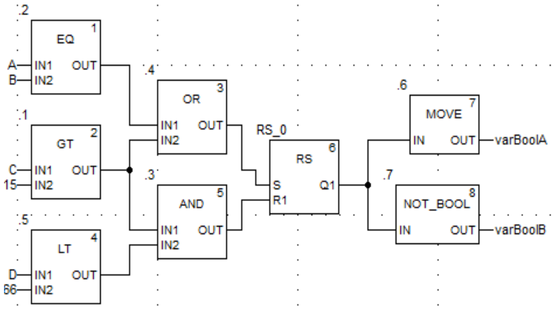
4		File input XML describe this FBD:
		 .1
		GT
		1
		A
		10
		IN1
		IN2
		OUT
		varBoolA
		.3
		NE
		2
		B
		1
		IN1
		IN2
		OUT
		varBoolB
		.2
		EQ
		3
		C
		100
		IN1
		IN2
		OUT
		varBoolC
		.4
		GE
		4
		5
		E
		IN1
		IN2
		OUT
		varBoolE
		.5
		LT
		5
		1000
		D
		IN1
		IN2
		OUT
		varBoolD
		.6
		LE
		6
		F
		80
		IN1
		IN2
		OUT
		varBoolF
			varBoolA := A > 10;
			varBoolC := C = 100;
			varBoolB := B <> 1;
			varBoolE := 5 >= E;
			varBoolD := 1000 < D;
			varBoolF := F <= 80;	

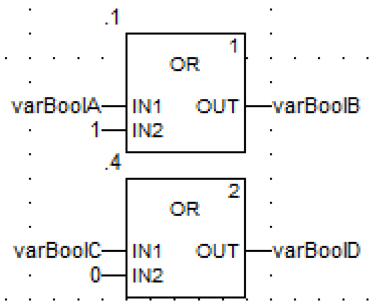
5		File input XML describe this FBD: 	A := 14;	

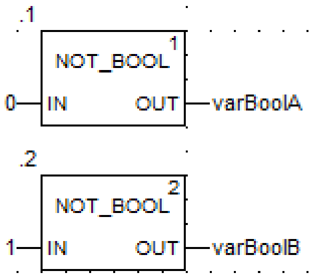
6		File input XML describe this FBD:
		
		 RS_0
		RS
		1
		varBoolA
		S
		Q1
		varBoolC
		varBoolB
		R1
		TON_0
		2
		TON
		varBoolD
		IN
		Q
		varBoolE
		t#15s
		PT
		ET
		The name program is “CODE”
			varBoolC := LV_CODE_RS(
			S:=varBoolA, R1:=varBoolB);
			varBoolE := LV_CODE_TON(
			IN := varBoolD, PT := t#15s);	

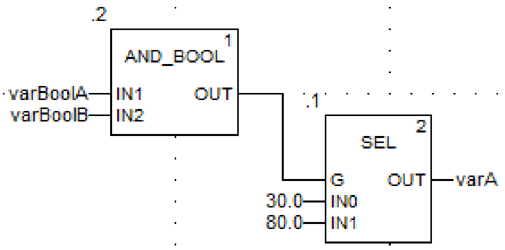
7		File input XML describe this FBD:
		
		 .1
		EQ
		1
		A
		IN1
		OUT
		B
		IN2
		RS_0
		2
		RS
		EN
		ENO
		varBoolA
		S
		Q1
		varBoolC
		varBoolB
		R1
		The name program is “CODE”
			IF (A = B) THEN
			varBoolC := LV_CODE_RS(
			S := varBoolA,
			R1 := varBoolB);
			END_IF	

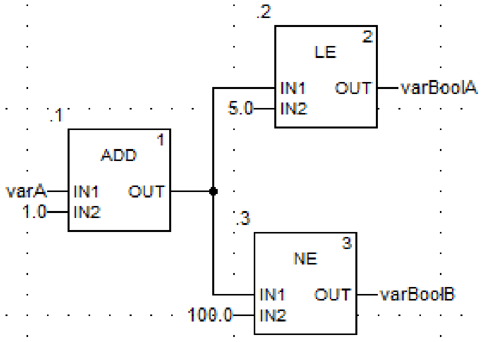
8		File input XML describe this FBD:
		
		 .1
		EQ
		1
		A
		IN1
		OUT
		B
		IN2
		.3
		3
		OR
		IN1
		OUT
		varBoolA
		IN2
		.2
		GT
		2
		C
		IN1
		OUT
		15
		IN2
		The name program is “CODE”
			varBoolA := (A = B) OR
			(C > 15);	

9		File input XML describe this FBD:
		
		 EQ
		A
		B
		IN1
		IN2
		OUT
		GT
		C
		15
		LT
		D
		66
		OR
		AND
		RS_0
		RS
		S
		R1
		Q1
		MOVE
		IN
		varBoolA
		NOT_BOOL
		varBoolB
		The name program is "CODE"
			LV_CODE_GT := C > 15;
			LV_CODE_RS(S := ((A = B)
			OR LV_CODE_GT),
			R1 := (LV_CODE_GT
			AND (D < 66)));
			varBoolA := LV_CODE_RS.Q1;
			varBoolB := NOT (LV_CODE_RS.Q1);	

10		File input XML describe this FBD:
		
		 .1
		OR
		1
		varBoolA
		IN1
		OUT
		varBoolB
		1
		IN2
		.4
		OR
		2
		varBoolC
		IN1
		OUT
		varBoolD
		0
		IN2
			varBoolD := varBoolC;
			varBoolB := 1;	

11		File input XML describe this FBD:
		
		 .1
		NOT_BOOL
		1
		0
		IN
		OUT
		varBoolA
		.2
		NOT_BOOL
		2
		1
		IN
		OUT
		varBoolB
			varBoolA := 1;
			varBoolB := 0;	

12		File input XML describe this FBD:
		
		 .2
		AND_BOOL
		1
		varBoolA
		varBoolB
		IN1
		IN2
		OUT
		.1
		SEL
		2
		G
		IN0
		IN1
		OUT
		varA
		30.0
		80.0
		“*ruleSEL.json*” for optimization:
		{"RULE":{ "SEL":{
		"IN":{"G":"_","IN0":"_","IN1":"_"},
		"OUT":{"OUT":"^"} } },
		"ACTION" : { "IF":{
		"CONDITION": "SEL.G = 0",
		"BRANCH": ["SEL.OUT := SEL.IN0"],
		"LAST_BRANCH":["SEL.OUT:=SEL.IN1"]
		} }
		}
			IF ((varBoolA AND varBoolB) = 0) THEN
			varA := 30.0;
			ELSE
			varA := 80.0;
			END_IF;	

13		File input XML describe this FBD:
		
		 .2
		LE
		2
		IN1
		IN2
		OUT
		varBoolA
		5.0
		.1
		ADD
		1
		varA
		1.0
		IN1
		IN2
		OUT
		.3
		NE
		3
		IN1
		IN2
		OUT
		varBoolB
		100.0
		The name program is “QPRN_CODE_NAME_VERY_LONG”
			LV_PRN_CODE_NAME_VERY_LONG_ADD := varA + 1.0;
			varBoolA := LV_PRN_CODE_NAME_VERY_LONG_ADD <= 5.0 ;
			varBoolB := LV_PRN_CODE_NAME_VERY_LONG_ADD <> 100.0 ;	

All the tests were successful. The case test 10, 11, 12 were successful only in optimize output.

Conclusion

As mentioned earlier, the goal of the thesis is to present a migration code solution in PLC context, from the FBD to ST language. The idea that was described in earlier chapters was to translate the legacy code in FBD language to ST language. This also will allow to simplify the re-engineering of PLC programs in order to save time from re-writing codes.

The architecture permitting the end user to translate from a program written in FBD language, to a program written in ST language.

The output ST program can be released in two way:

- optimized
- non-optimized

The difference between these two source codes outputs is the systems application of rule for optimization.

The architecture is built with the purpose to read external optimization rules, in order to transform specific patterns recognized in code with other code defined in external rule.

An example of execution can be briefly represented below.

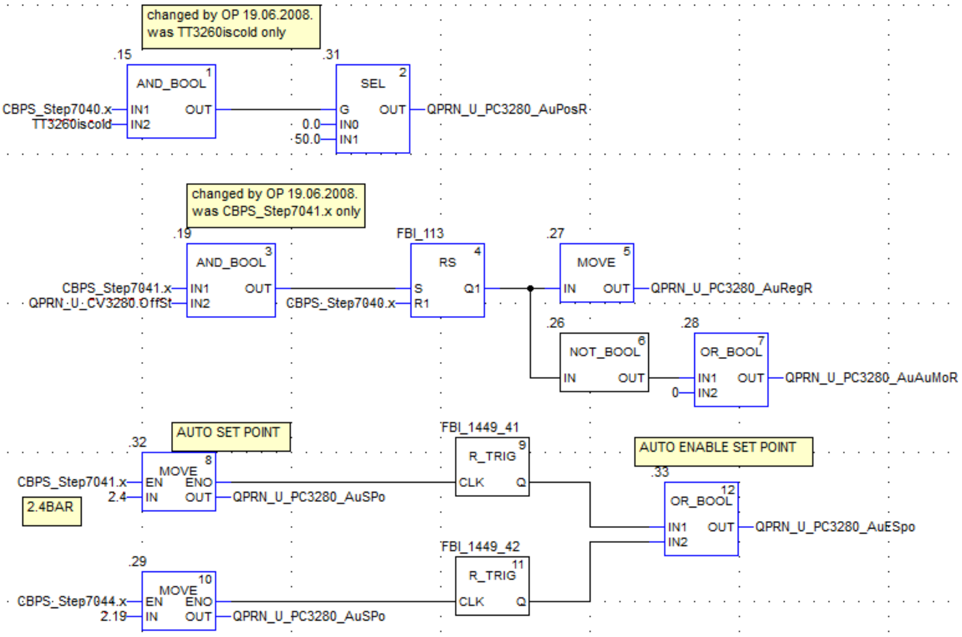


Figure 25 - a sample of FBD program in project to migrate

This is the representation of xml input file at FDB2ST system. The results of this program are two, such as mentioned earlier. The first non-optimized:

```
IF CBPS_Step7044.x THEN
        QPRN_U_PC3280_AuSPo := 2.19;
END_IF;

(* LV_QPRN_U_CBPS_PC3280_R_TRIG is R_TRIG - FBI_1449_42 *)
LV_QPRN_U_CBPS_PC3280_R_TRIG(CLK := CBPS_Step7044.x);

IF CBPS_Step7041.x THEN
        QPRN_U_PC3280_AuSPo := 2.4;
END_IF;

(* LV_QPRN_U_CBPS_PC3280_R_TRIG1 is R_TRIG - FBI_1449_41 *)
LV_QPRN_U_CBPS_PC3280_R_TRIG1(CLK := CBPS_Step7041.x);

QPRN_U_PC3280_AuESpo := LV_QPRN_U_CBPS_PC3280_R_TRIG1.Q
        OR LV_QPRN_U_CBPS_PC3280_R_TRIG.Q;
(* .31 is SEL - .31 *)
QPRN_U_PC3280_AuPosR := SEL(G := (CBPS_Step7040.x AND TT3260iscold),
        IN0 := 0.0, IN1 := 50.0);

(* LV_QPRN_U_CBPS_PC3280_RS is RS - FBI_113 *)
LV_QPRN_U_CBPS_PC3280_RS(S := (CBPS_Step7041.x AND QPRN_U_CV3280.OffSt),
        R1 := CBPS_Step7040.x);
QPRN_U_PC3280_AuRegR := LV_QPRN_U_CBPS_PC3280_RS.Q1;
QPRN_U_PC3280_AuAuMoR := NOT(LV_QPRN_U_CBPS_PC3280_RS.Q1) OR 0;
```

The optimized code is:

```
IF CBPS_Step7044.x THEN
        QPRN_U_PC3280_AuSPo := 2.19;
END_IF;
(* LV_QPRN_U_CBPS_PC3280_R_TRIG is R_TRIG - FBI_1449_42 *)
LV_QPRN_U_CBPS_PC3280_R_TRIG(CLK := CBPS_Step7044.x);

IF CBPS_Step7041.x THEN
        QPRN_U_PC3280_AuSPo := 2.4;
END_IF;

(* LV_QPRN_U_CBPS_PC3280_R_TRIG1 is R_TRIG - FBI_1449_41 *)
LV_QPRN_U_CBPS_PC3280_R_TRIG1(CLK := CBPS_Step7041.x);

QPRN_U_PC3280_AuESpo := LV_QPRN_U_CBPS_PC3280_R_TRIG1.Q
        OR LV_QPRN_U_CBPS_PC3280_R_TRIG.Q;
(*      RuleIF applied on:      .31
        Functional Block:       SEL*)
IF ((CBPS_Step7040.x AND TT3260iscold) = 0 ) THEN
        QPRN_U_PC3280_AuPosR := 0.0;
ELSE
        QPRN_U_PC3280_AuPosR := 50.0;
END_IF;
(* LV_QPRN_U_CBPS_PC3280_RS is RS - FBI_113 *)
LV_QPRN_U_CBPS_PC3280_RS(S := (CBPS_Step7041.x AND QPRN_U_CV3280.OffSt),
        R1 := CBPS_Step7040.x);

QPRN_U_PC3280_AuRegR := LV_QPRN_U_CBPS_PC3280_RS.Q1;
QPRN_U_PC3280_AuAuMoR := NOT(LV_QPRN_U_CBPS_PC3280_RS.Q1);
```

Since this is the first actual version of the project, a test case has not been created to observe the behaviour of the ST code resulting by system and pre-existing FBD code.

Future development

In future this system may be improved with other tecnique of optimization not implemented yet and can be used in other context like a component, in order to save the time to translate the legacy code.

The project can be imported inside the continuous integration test of cryogenic system at CERN, in order to automate the legacy code conversion.

To improve the test system, in future the code generated from FBD2ST must be executed in order to compare the behaviour with the real system. This may be done, in future, by applying the PROCOS[3] standard for simulation, in order to execute the entire code on PLC linked with a pc where is simulated the real system with a *EcosimPro* model.

References

[1] Unicos, http://unicos.web.cern.ch/

[2] C. Fludera, T. Wolak, A. Drozd, M. Dudek, F. Frassinelli, M. Pezzetti, A. Tovar-Gonzaleza, M. Zapolskib, Improved software production for the LHC tunnel cryogenics control system, ICEC 25–ICMC 2014

[3] Ph. Gayet, R. Barillére, UNICOS a framework to build industrial-like controls systems principles methodology, ICALEPCS05, Geneva, Switzerland, 2005.

[3] https://www.capsl.udel.edu//courses/cpeg421/2012/slides/Tutorial-Flex_Bison.pdf

[4] https://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf

[5] https://product-help.schneider-electric.com/Machine%20Expert/V1.1/en/mwt/topics/instantiation.htm

[6] *EcoStruxure™ Control Expert Program Languages and Structure Reference Manual*, https://download.schneider-electric.com/files?p_enDocType=User+guide&p_File_Name=35006144_K01_000_22.pdf&p_Doc_Ref=35006144K01000

[7] International Standard IEC 61131-3, Second Edition, Programmable Controllers-Part 3: Programming Languages. International Electro technical Commission, Geneva, 2003.

[8] Chikofsky, E. and Cross, J., 1990. Reverse Engineering and Design Recovery: A Taxonomy. IEEE Software, 7(1):13-18.

[9] Asit Kumar Gahalaut et. al., REVERSE ENGINEERING: AN ESSENCE FOR SOFTWARE RE-ENGINEERING AND PROGRAM ANALYSIS, International Journal of Engineering Science and and Technology, Vol. 2(06), 2010, 2296-2303

[10] https://www.w3.org/TR/WD-DOM/introduction.html

[11] https://docs.python.org/3/library/unittest.html

[12] Gregory Gay, 2016, Test Oracles, CSCE 747 - Spring 2016

[13] P. Gayet, B. Bradu, PROCOS: A REAL TIME PROCESS SIMULATOR COUPLED TO

THE CONTROL SYSTEM, ICALEPCS2009, volume, pagine, 2009

[14] E. Dijkstra, ALGOL-60 Translation, 1961, http://www.cs.utexas.edu/~EWD/MCReps/MR35.PDF

[15] P. Tramontana, Teoria fondamenti testing, Ingegneria del Software II, 2018

[16] devo mettere il libro dei compilatori

[17] P. Tramontana, Manutenzione e Reverse Engineering, Ingegneria del Software II, 2018

[18] A. Fasolino, Ingengneria del Software, Design Pattern, 2018

[19] Dick Grune, Ceriel J.H. Jacobs, Parsing Techniques - Second Edition, Ellis Horwood, Chichester, England, 1990

[20] Morteza Zakeri, An Introduction to ANTLR, 2016

[21]

[22]