M 2015

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

CONVERSOR ENTRE LINGUAGENS IEC61131-3

RAFAEL FERNANDO MONTEIRO GOMES

DISSERTAÇÃO DE MESTRADO APRESENTADA
À FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO EM
MESTRADO INTEGRADO EM ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES

U.PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

MIEEC - MESTRADO INTEGRADO EM ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES 2014/2015

A Dissertação intitulada

"Conversor Entre Linguagens IEC61131-3"

foi aprovada em provas realizadas em 20-02-2015

o júri

Presidente Professor Doutor Armando Jorge Miranda de Sousa
Professor Auxiliar do Departamento de Engenharia Eletrotécnica e de Computadores
da Faculdade de Engenharia da Universidade do Porto

Professor Doutor Jaime Francisco Cruz Fonseca
Professor Associado Departamento Eletrónica Industrial da Universidade do Minho

Professor Doutor Mário Jorge Rodrigues de Sousa
Professor Auxiliar do Departamento de Engenharia Eletrotécnica e de Computadores
da Faculdade de Engenharia da Universidade do Porto

O autor declara que a presente dissertação (ou relatório de projeto) é da sua exclusiva autoria e foi escrita sem qualquer apoio externo não explicitamente autorizado. Os resultados, ideias, parágrafos, ou outros extratos tomados de ou inspirados em trabalhos de outros autores, e demais referências bibliográficas usadas, são corretamente citados.

Autor - Rafael Fernando Monteiro Gomes

Faculdade de Engenharia da Universidade do Porto

Faculdade de Engenharia da Universidade do Porto

Conversor entre Linguagens IEC61131-3

Rafael Fernando Monteiro Gomes

Dissertação realizada no âmbito do
Mestrado Integrado em Engenharia Eletrotécnica e de Computadores
Major Automação

Orientador: Mário Jorge Rodrigues De Sousa (Professor)

27 de Fevereiro de 2015

Resumo

Este documento apresenta um Projeto desenvolvido que deverá funcionar em conjunto com o Projeto matiec. O matiec trata-se de um Projeto que pretende fornecer, de forma gratuita, um compilador para as linguagens de programação definidas na Norma IEC 61131-3. Estas linguagens são, essencialmente, utilizadas para a programação de PLCs (Programmable Logic Controllers).

Destas linguagens, a norma define duas delas como textuais: O IL (Instruction List) e o ST (Structured Text). A norma define também três linguagens gráficas, sendo elas: O FBD (*Function Block Diagram*), o LD (*Ladder Diagram*) e o SFC (*Sequential Function Chart*). Das cinco linguagens acima mencionadas, a norma define representações textuais para as linguagens IL, ST e SFC. São estas linguagens as suportadas pelo Projeto matiec, desde que as mesmas estejam escritas na sua representação textual. Assim, surge a necessidade do desenvolvimento do Projeto aqui descrito. Este Projeto terá a função de permitir ao matiec a compatibilidade com todas as linguagens de programação presentes na Norma IEC 61131-3. Esta função será implementada através da realização de conversões de programas nas duas linguagens gráficas ainda não suportadas pelo matiec (FBD e LD) numa linguagem textual (ST). Estas linguagens encontram-se descritas num documento XML num formato próprio (TC6-XML) que está de acordo com a Norma IEC 61131-3. Ora, como o matiec apenas aceita a representação textual das linguagens ST, IL ou SFC, é necessária a criação de uma ferramenta de conversão que consiga receber o ficheiro XML com o devido programa ST (convertido ou não de um programa LD ou FBD), IL ou SFC e convertê-lo na sua representação textual de acordo com a Norma IEC 61131-3. Assim o Projeto aqui descrito tem como base a criação de duas ferramentas. A primeira ferramenta trata-se de um Conversor XML responsável por converter programas LD e FBD em programas ST estruturados (TC6-XML) segundo a norma. A segunda ferramenta consiste num Conversor TXT que é responsável por converter programas ST, IL ou SFC (estruturados num documento XML segundo a Norma IEC 61131-3) na sua representação textual de acordo com a Norma IEC 61131-3.

Abstract

This document presents a developed Project that should work together with the matiec Project. The matiec is a Project that aims to provide, freely, a compiler for the programming languages defined in IEC 61131-3. These languages are essentially used for programming PLCs (Programmable Logic Controllers).

Of these languages, the standard defines them as two textual: IL (Instruction List) and ST (Structured Text). The standard also defines three graphical languages, namely: The FBD (Function Block Diagram), LD (Ladder Diagram) and SFC (Sequential Function Chart). Of the above five languages, the Standard defines textual representations for IL languages, ST and SFC. These are languages supported by matiec Project, provided that they are written in its textual representation. Thus arises the need of the here described Project development. This Project will serve to enable the matiec compatibility with all programming languages present in IEC 61131-3. This function is implemented by performing conversions of programs in both graphical languages not yet supported by matiec (FBD and LD) in a textual language (ST). These languages are described in an XML document in a proprietary format (TC6-XML) which is in accordance with IEC 61131-3. As the matiec accepts only the textual representation of languages ST, IL or SFC, the creation of a conversion tool is needed that can receive the XML file with due ST program (converted or not from a LD or FBD program), or IL or SFC programs and convert it in its textual representation in accordance with IEC 61131-3. Thus the Project described here is based on the creation of two tools. The first tool it is an XML converter responsible for converting FBD and LD programs to ST programs structured (XML-TC6) according to standard. The second tool is a TXT Converter that is responsible for converting ST, IL or SFC programs (structured in a XML document in accordance with IEC 61131-3) in its textual representation in accordance with IEC 61131-3.

Agradecimentos

Quero aproveitar este espaço para exprimir a minha gratidão aos meus pais, que tudo fizeram para que pudesse chegar onde cheguei e à minha irmã, que apesar de não me parar de chatear, também merece uma menção neste texto.

Deixar, igualmente, uma palavra de profundo reconhecimento e de carinho à minha namorada. Quero-te agradecer por tudo, Joana, por estares sempre ao meu lado, nos piores e nos melhores momentos, por me compreenderes como ninguém e por me fazeres o homem que sou hoje.

Agradeço a toda a minha família que tanto estimo e que me ensinaram valores cruciais na minha vida académica.

Quero prestar agradecimento ao meu Orientador Professor Mário Jorge Rodrigues de Sousa pela supervisão prestada durante todas as fases do Projeto, pelas reuniões de aconselhamento e verificação do trabalho e como guia na execução do trabalho da forma mais correta.

Deixo também uma palavra de apreço a todos os meus amigos de infância que, ainda hoje, mantenho contacto e que tanta falta fazem no meu dia-a-dia. Vocês são os melhores.

Não quero deixar de fazer referência aos meus novos amigos. Aqueles que tive desde a minha entrada na FEUP. Estiveram sempre disposto a ajudar, a falar, a partilhar conhecimentos mas acima de tudo, dispostos a rir e a brincar. Vocês são, igualmente, os melhores.

A todas as outras pessoas que passaram pela minha vida e deixaram marca de alguma forma um obrigado.

Um Muito Obrigado a todos!

Rafael Fernando Monteiro Gomes

"No tengas miedo de la perfección - que nunca lo alcanzarás."

Salvador Dalí

Índice

Lista de figuras

Lista de tabelas

Abreviaturas e Símbolos

API	*Application Programmers Interface*
CPU	*Central Processing Unit*
DOM	*Document Object Model*
FB	*Function Block*
FBD	*Function Block Diagram*
GUI	*Graphical User Interface*
HTML	*Hyper Text Markup Language*
IEC	*International Electro technical Commission*
IL	*Instruction list*
ISO	*International Organization for Standardization*
JDOM	*Java Distributed Object Model*
LD	*Ladder*
POU	*Program Organization Unit*
PLC	*Programmable logic controller*
SAX	*Simple API for XML*
SFC	*Sequential function chart*
SGML	*Standard Generalized Markup Language*
ST	*Structured Text*
TXT	*Text*
W3C	*World Wide Web Consortium*
XML	*eXtensible Markup Language*
XSL	*eXtensible Stylesheet Language*

Capítulo 1

Introdução

1.1 - Motivação

Hoje em dia, a maioria dos setores industriais utilizam os Controladores Lógicos Programáveis (PLCs) para a execução dos seus processos. Este dispositivo eletrónico foi introduzido pela primeira vez nos anos 60 para automatizar e controlar diferentes sistemas.

A evolução dos equipamentos de informática e tecnologias de comunicação e a sua posterior integração na indústria tem incentivado, ao longo das últimas décadas, os engenheiros eletrotécnicos e de computadores a se especializarem, cada vez mais, no setor da automação. Essa especialização, no entanto, não foi um processo trivial, uma vez que eles mesmos tiveram que se familiarizar com ambientes de programação muito específicos e desconhecidos. Assim sendo, foi necessária a padronização dos mesmos. Organizações como a IEC (*International Electrotechnical Commission*) ou a ISO (*International Organization for Standardization*) passaram a considerar este aspeto no início dos anos 90 criando a Norma IEC 61131-3 [8]. Trata-se de uma norma que propõe uma interface comum para PLCs, com cinco linguagens de programação, três das quais têm uma interface gráfica:

- Diagrama Sequencial de Funções (SFC): uma linguagem de programação gráfica que define uma máquina de estados, baseada em Grafcets (pode também ser expressa na sua representação textual).
- Diagrama de Blocos de Funções (FBD): uma linguagem de programação gráfica baseada em blocos com funções que podem ser definidos pelo utilizador;
- Diagrama *Ladder* (LD): uma linguagem de programação gráfica, semelhante à metodologia de circuitos eletrónicos;

Enquanto existem outras duas linguagens baseadas em texto simples:

- Lista de Instruções (IL): uma linguagem de programação textual, um pouco como a linguagem assembly;
- Texto Estruturado (ST): uma linguagem de programação textual e que, em termos de sintaxe, é semelhante ao Pascal;

Das cinco linguagens acima mencionadas, a norma define representações textuais para as linguagens IL, ST e SFC. Estas linguagens são as linguagens suportadas pelo Projeto matiec que pretende fornecer, de forma gratuita, um compilador para as linguagens de programação definidas na Norma IEC 61131-3.

Assim é necessário compatibilizar todas as linguagens de programação presentes na norma para que o Projeto matiec seja uma ferramenta de automação compatível com todas as linguagens. No fundo, é necessário colocar o Projeto matiec a suportar todas as linguagens gráficas (LD e FBD), visto que o mesmo já suporta as linguagens que têm representação textual (ST, IL e SFC). Para isso será necessário recorrer a conversões de linguagem que converterão as linguagens LD e FBD numa outra linguagem já compatível com o matiec (ST). Contudo, esta conversão, não pode ser realizada de forma isolada (devido ao formato do documento de saída – o formato TC6-XML de acordo com a Norma IEC 61131-3), assim é necessária uma outra conversão para a representação textual do programa. Esta conversão será então responsável por receber programas em ST, IL e SFC, no formato XML (TC6-XML), e convertê-las na sua representação textual num documento TXT. Assim, desta forma, podem ser processadas pelo Projeto matiec.

1.2 - Objetivos

Tendo em vista o foco principal presente neste Projeto (colocar o matiec compatível com todas as linguagens gráficas descritas na Norma IEC 61131-3) é necessário que o mesmo responda a certos objetivos. Estes objetivos podem ser descriminados na possibilidade de converter programas LD e/ou FBD em formato XML em programas ST no mesmo formato e na conversão para a sua representação textual segundo a Norma IEC 61131-3.

Assim, pretende-se alcançar o objetivo final de produção de código C pela parte do matiec, mas desta feita para qualquer tipo de linguagem, tornando o Projeto matiec compatível com qualquer linguagem presente na Norma IEC 61131-3.

1.3 - Estrutura do Documento

Este documento será dividido em 5 capítulos.

No Capítulo 2 intitulado de Revisão Bibliográfica são apresentadas as cinco linguagens de automação, a linguagem XML (mais especificamente o formato TC6-XML), soluções estudadas e existentes no mercado para o processamento de documentos XML, técnicas e uma breve referência à história de alguns termos e equipamentos relacionados com o tema.

No Capítulo 3 são apresentados os algoritmos utilizados na criação dos conversores. Esta apresentação é intercalada com a descrição da implementação de cada um dos algoritmos.

No Capítulo 4 são apresentados Resultados. Neste capítulo são apresentados resultados dos testes realizados, assim como a sua validação.

No Capítulo final são apresentadas conclusões e ideias para Trabalhos Futuros. Apresentam-se ideias que não foram executadas e propostas de resolução de alguns problemas encontrados ou soluções para melhorar certas partes do trabalho bem como conclusões de um modo geral e em alto nível.

Em anexos estão presentes imagens referentes ao diagrama de classes dos dois conversores criados para executar os algoritmos, bem como fluxogramas representativos de certas funções utilizadas nos algoritmos. Contém também um documento XML e um documento TXT que são resultados dos dois conversores criados.

Capítulo 2

Revisão Bibliográfica

Neste capítulo serão abordadas técnicas e temáticas essenciais à realização e desenvolvimento do Projeto. Serão, primeiramente, apresentadas as cinco linguagens de programação de autómatos, ST, LD, FBD, IL e SFC revendo, através de uma pequena introdução, a Norma IEC 61131-3. Estas linguagens serão analisadas nas suas principais funções e estruturação e sempre que possível será feita a "ponte" entre as mesmas, de forma a compreender as especificidades de cada uma.

Será igualmente apresentada a linguagem XML que será o elemento central deste Projeto, evidenciando as regras e estruturação da mesma e utilizando sempre que possível, exemplos ilustrativos. Aqui serão também referenciadas e estudadas as especificidades da estrutura TC6-XML utilizada na representação dos programas de automação.

Outra temática abordada serão as ferramentas de processamento da linguagem XML, ou seja, o estudo de algumas tecnologias que poderão ser utilizadas no desenvolvimento deste Projeto. Esta análise será baseada em alguns parâmetros comparativos entre os quais, a carga de processamento, o tamanho, entre outros.

Por fim será apresentado o *software* Beremiz e a sua interface de criação de projetos.

2.1 - Norma IEC 61131-3

A Norma IEC 61131-3 é uma norma internacional que tem como objetivo padronizar as linguagens de programação de Controladores Lógicos Programáveis na área da automação industrial. Foi desenvolvida para dar resposta a pressões da indústria para uma maior compatibilidade entre os PLCs e a sua programação. A norma define cinco linguagens, sendo duas delas gráficas, *Ladder Diagram* (LD) e *FunctionBlock Diagram* (FBD), duas textuais, *Instruction List* (IL) e *Structured Text* (ST), e uma quinta, *Sequential Function Chart* (SFC -

"GRAFCET") muito utilizada em programação sequencial de eventos através da definição de etapas e condições de transição entre elas [9].

Através da implementação da Norma IEC 61131-3 existe uma uniformização das linguagens de automação, criando um modelo único para o *software* (independente do fabricante), permitindo a reutilização de software já desenvolvido e o uso de funções e blocos de funções já uniformizados. A norma permite ainda a possibilidade de utilizar várias linguagens no mesmo programa, define uma série de dados com tipos definidos e ainda suporta estruturas de dados complexas tais como *arrays*, estruturas, enumerações, entre outros.

Para além destes aspetos referidos, a norma, define ainda:

- Tipos de dados (*data types*);
- Linguagens de programação;
- Unidades de Organização de Programas (POUs, sendo que neste ponto pode ser adicionada a parte das declarações de configurações dos projetos);

Nos tipos de dados são definidos dados do tipo numérico (inteiros e reais que permitem efetuar operações aritméticas), dados do tipo booleano (BOOL, BYTE, entre outros, que não permitem operações aritméticas com este tipo de dados, mas permitem operações lógicas bit a bit), dados do tipo tempo e data (TIME, DATE, entre outros) e dados do tipo *string* (STRING e WSTRING). Na Tabela 2.1 podem ser observados alguns destes tipos de dados. Contudo, a Norma IEC 61131-3, também permite a definição de novos tipos de dados derivados.

Tabela 2.1 – Exemplos de alguns tipos de dados de acordo com a Norma IEC 61131-3

Tipo	***Data Type***	**Designação**
INT	*Integers*	Inteiro (*Integer*)
DINT	*Integers*	Duplo Inteiro (*Double Integer*)
REAL	*Reals*	Real (*Real*)
LREAL	*Reals*	Real Longo (*Long Real*)
BOOL	*Bit Oriented*	Valor de Lógica Booleana (*Boolean Logic Value*)
BYTE	*Bit Oriented*	Sequência de 8 bits (*Sequence of 8 bits*)
WORD	*Bit Oriented*	Sequência de 16 bits (*Sequence of 16 bits*)
STRING	*String*	Sequência de caracteres (*Sequence of characters*)
TOD	*Date and Time*	Tempo do Dia (*Time of Day*)
TIME	*Date and Time*	Intervalo de Tempo (*Time Interval*)

Quanto às Unidades de Organização de Programas, a norma, permite a organização do código em blocos que possam ser reutilizáveis. Esses tipos de blocos podem ser programas (PROGRAM), funções (FUNCTION) ou blocos de funções (FUNCTION BLOCK). Os programas são a estrutura de mais alto nível (são quase como o equivalente ao main() num programa de C/C++) e podem ser escritos em qualquer das linguagens definidas na Norma IEC 61131-3. As funções permitem agrupar código que se pretenda repetir ao longo de um programa, são muito semelhantes às funções de C e PASCAL e podem ser escritas em LD, IL, FBD e ST. Os blocos de funções permitem, igualmente, o agrupar do código que se pretenda repetir, mas com uma diferença em relação às funções. A principal diferença entre as funções e blocos de funções é que as funções produzem sempre o mesmo resultado (valor da função), quando chamadas com os mesmos parâmetros de entrada, ou seja, elas não têm "memória" pois não têm variáveis estáticas. Os blocos de funções têm o seu próprio registo de dados e podem, portanto, "lembrar-se" das informações de *status* (instanciação). Contudo, existem blocos de funções e funções uniformizadas pela norma (por exemplo funções de temporização) [10, 11]. Na Figura 2.1 encontra-se representada a relação dos POUs aqui abordados para uma melhor compreensão dos conceitos.

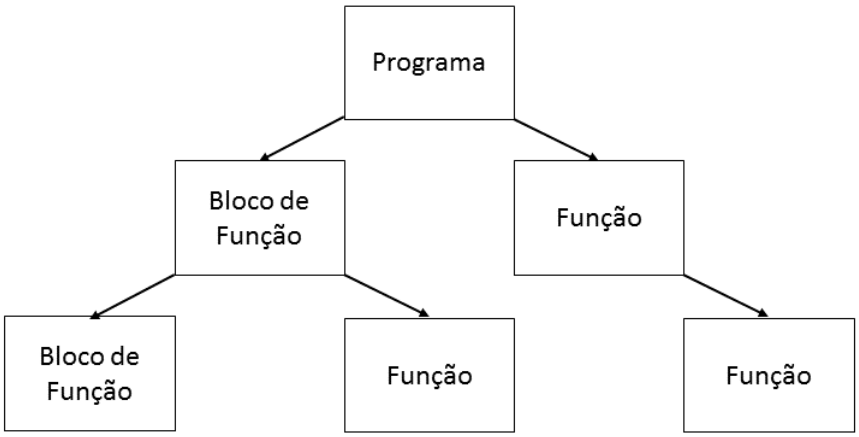


Figura 2.1 — Relação entre POUs definidos na Norma IEC 61131-3

Tem-se ainda as declarações das configurações dos programas num autómato que é definida numa estrutura chamada "*CONFIGURATION*". Estas agregam toda a informação necessária para definir o estado exato desejado do *software* para fazer *download* para o PLC. Estas informações incluem a definição de variáveis, instâncias de programas e o planeamento da execução do programa (não inclui configurações de *hardware* especificas). Para além disto, cada PLC pode ser composto por várias unidades de processamento (CPUs), os quais são conhecidos como recursos (*resources*) na Norma IEC 61131-3. É aqui que se define a execução dos vários programas. Os programas diferem na prioridade ou tipo de execução (periódica, cíclica ou por

interrupção) e a cada programa está associada uma tarefa (*task*). Assim, é necessário configurar que instâncias do programa vão ser executadas por cada tarefa. [10].

2.2 - Linguagem Structured Text

Para a realização deste Projeto esta linguagem tem uma especial atenção devido a ser a linguagem escolhida como a linguagem de "destino" da conversão das linguagens de LD e FBD, ou seja, as linguagens LD e FBD serão convertidas na linguagem ST.

Neste tópico pretende-se abordar a linguagem ST evidenciando as suas características. A linguagem ST assemelha-se a uma linguagem de programação de alto nível (por exemplo Pascal ou C) e tem vindo, cada vez mais, a ser utilizada na indústria de automção. É uma linguagem que permite a construção de estruturas complexas (ciclos *for*, *while*, entre outros que serão abordados mais à frente neste mesmo subcapítulo) e a utilização de expressões complexas. Assim, as funções de cálculo, e os dados podem ser implementados muito mais facilmente nesta linguagem do que em LD ou IL.

A inserção de comentários torna-se mais fácil na linguagem ST tornando-a numa linguagem mais flexível, ou seja, torna a tarefa de estruturar um programa complexo em algo mais simples e de fácil execução [5, 12].

É necessário ter em atenção certas particularidades deste tipo de linguagem. Um programa ST é organizado em declarações, sendo que cada declaração (*statement*) consiste num dos seguintes elementos:

- Rótulo (*label*) é usado para fazer referência a uma declaração numa entidade do programa;
- Comentário (*comments*) que tem a função de facilitar a interpretação do programa ST e encontra-se entre "(*" e "*)";
- Instruções (*Instructions*) que podem existir várias numa declaração em ST. Cada instrução tem de terminar com o caracter ";";

Na Figura 2.2 pode ser observado um exemplo de um programa ST e a rotulação do mesmo nos elementos abordados.

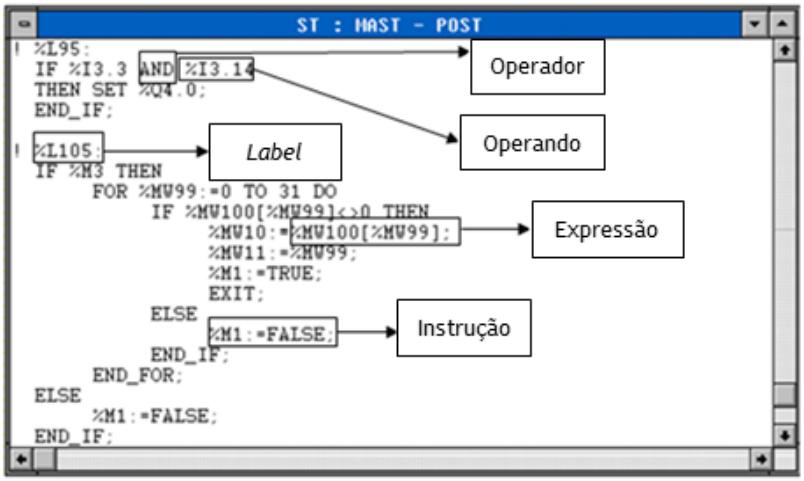


Figura 2.2 — Exemplo de um programa ST e divisão do mesmo pelos seus elementos

Pode-se afirmar que um programa ST é uma sequência de declarações de atribuição (*assignment statement*), invocações de FBs (FB *invocation*) e estruturas de controlo (*control flow statements* e *iteration statements*) [11].

Nas declarações de atribuição tem-se a seguinte estrutura:

- variável := expressão (a variável e a expressão têm de ser do mesmo tipo de dados – a expressão pode conter operadores);

Para invocações de FBs tem-se:

- instânciaFB(parâmetro_entrada := expressão, parâmetro_saída => variável) - esta é a forma formal de invocação;
- instânciaFB(expressão, variável) - esta é a forma informal de invocação;

De notar que a sintaxe de invocação de funções é a mesma que a sintaxe de invocação de FB's.

Quanto à estrutura de controlo, existem 4 estruturas:

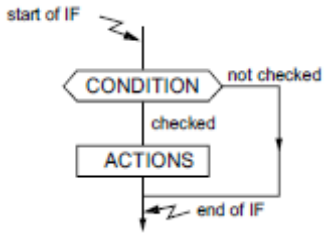
- A ação condicional IF;
- As ações iterativas condicionais WHILE e REPEAT;
- A ação repetitiva FOR;

Cada estrutura de controlo deve estar aglomerada entre palavras-chave e começam e acabam na mesma declaração. Contudo é possível agrupar estruturas de controlo umas dentro de outras sem qualquer restrição quanto ao seu tipo.

2.2.1 -A ação condicional IF…END_IF;

Neste caso a instrução executa uma ação se a condição for verificada/verdadeira.

Tabela 2.2 — Sintaxe e modo de execução do ciclo IF…END_IF [12]

Sintaxe	Operação
IF condition **THEN** actions ; **END_IF;**	

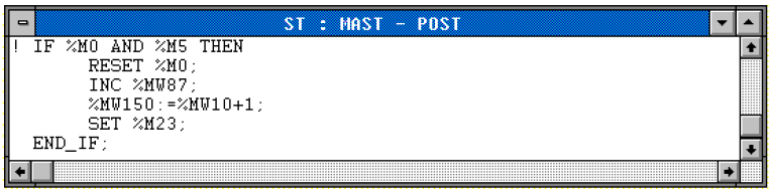
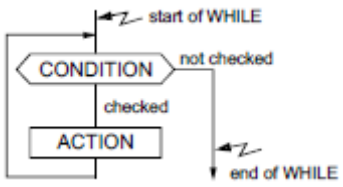


Figura 2.3 — Exemplo da utilização IF…END_IF [12]

2.2.2 -A ação iterativa condicional WHILE…END_WHILE;

A instrução repete uma ação enquanto a condição for verificada.

Tabela 2.3 — Sintaxe e modo de execução do ciclo WHILE…END_WHILE [12]

Sintaxe	Operação
WHILE condition **DO** action; **END_WHILE;**	

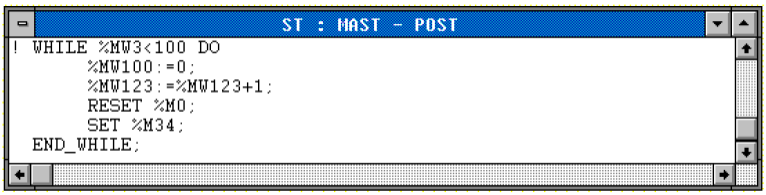
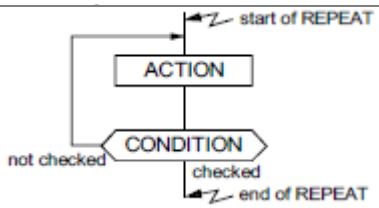


Figura 2.4 — Exemplo da utilização WHILE…END_WHILE [12]

2.2.3 -A ação iterativa condicional REPEAT…END_REPEAT;

A instrução repete uma ação até a condição ser verificada.

Tabela 2.4 — Sintaxe e modo de execução do ciclo REPEAT…END_REPEAT [12]

Sintaxe	Operação
REPEAT action; **UNTIL** condition **END_REPEAT;**	

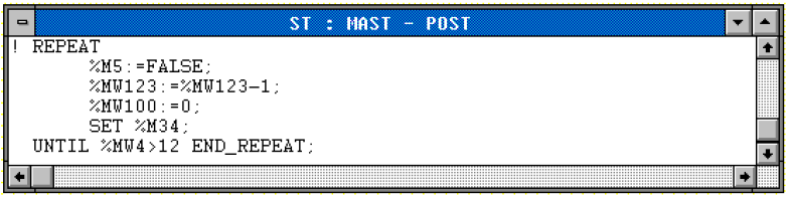


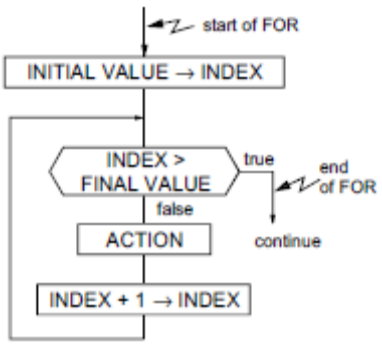
Figura 2.5 — Exemplo da utilização REPEAT…END_REPEAT [12]

2.2.4 -A ação repetitiva FOR…END_FOR;

A instrução executa uma operação um certo número de vezes, aumentando um indicie numa unidade em cada *loop*. Esta declaração pode conter a expressão [11]:

- "BY" expressão, resultando assim na seguinte estrutura:
  - "FOR" variável := expressão "TO" expressão "BY" expressão "DO"
    lista_declarações
    "END_FOR";

Tabela 2.5 — Sintaxe e modo de execução do ciclo FOR…END_FOR [12]

Sintaxe	Operação
FOR index := initial value **TO** final value **DO** action; **END_FOR;**	

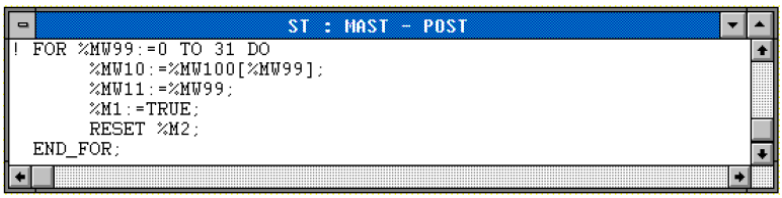


Figura 2.6 — Exemplo da utilização FOR…END_FOR [12]

2.3 - Linguagem Ladder

A linguagem *Ladder* (LD) foi a primeira a ser desenvolvida para a programação de PLCs e nos dias que correm ainda é a mais utilizada pelos técnicos, estando presente em praticamente todos esses controladores. Algumas vantagens desta linguagem são:

- Linguagem gráfica, baseada em símbolos;
- Simbologia semelhante a um circuito elétrico;
- Pouca variação de fabricante para fabricante;
- A simplicidade de interpretação e desenvolvimento;

Apesar destas vantagens, esta linguagem apresenta hoje em dia algumas limitações:

- Dificuldade em construir estruturas complexas;
- Dificuldade em construir sequências/ciclos complexos;

A linguagem LD é baseada no princípio de contatos elétricos. Cada um dos componentes pode possuir um número infinito de contatos que são limitados pela capacidade de memória do controlador programável [13].

O nome *Ladder* surgiu devido à estrutura da linguagem ser semelhante a uma escada (*ladder*), na qual duas barras verticais paralelas são interligadas por uma lógica de controlo, formando os degraus (*rungs*) da escada. Portanto, a cada lógica de controlo existente no programa de aplicação, dá-se o nome de *rung*, que é composta por colunas e linhas [14].

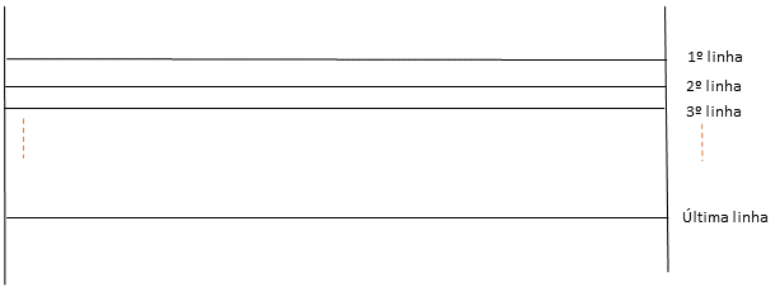


Figura 2.7 — Estrutura de um programa *Ladder* (simplificado) [13]

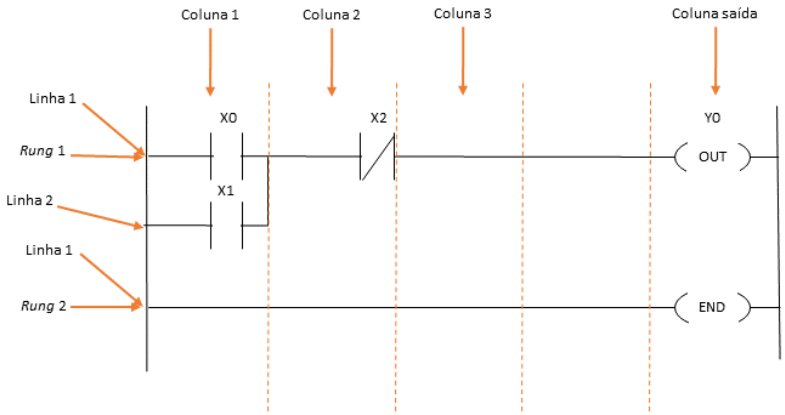


Figura 2.8 — Estrutura de um programa *Ladder* (completo)

Na elaboração de um programa em LD, são adotadas algumas convenções:

- As linhas verticais do diagrama representam as linhas de energia, entre as quais os circuitos são ligados;
- Cada degrau na escada define uma operação no processo de controlo;
- Um diagrama em escada (como é um programa em LD) é lido da esquerda para a direita e de cima para baixo. A Figura 2.9 mostra a forma de leitura de um programa neste tipo de linguagem. O degrau superior é lido da esquerda para a direita até ser atingida a linha de alimentação do lado direito (*right power rail*). Em seguida, passa-se à extrema-esquerda do segundo degrau e volta-se a ler da esquerda para a direita e assim sucessivamente;

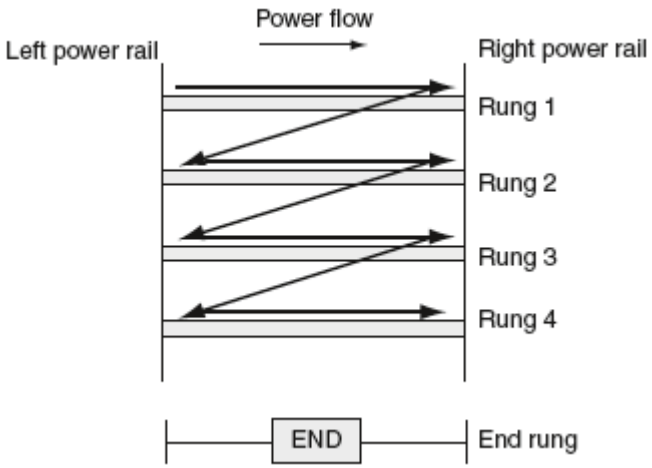


Figura 2.9 — Modo de leitura de um programa LD [14]

Como foi referido em cima, esta é uma linguagem gráfica, de baixo nível, análoga à construção de um circuito elétrico com relés. A Tabela 2.6 mostra os principais símbolos utilizados nesta linguagem e a sua correspondência com os elementos utilizados no desenho de um circuito elétrico.

Tabela 2.6 — Principais símbolos da programação em *Ladder*

Tipo	Símbolo	Circuito Elétrico
Contacto aberto		
Contacto fechado		
Saída		

Torna-se essencial, para a realização do trabalho, o paralelismo entre a linguagem LD e ST. Assim sendo as principais funções utilizadas na programação de autómatos serão comparadas nas duas linguagens.

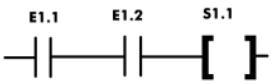
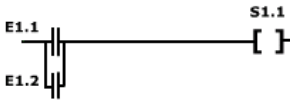
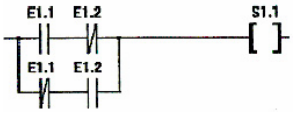
Tabela 2.7 — Operadores das instruções de Load

Ladder (LD)	Texto Estruturado (ST)	Modo de operação
	E1	Contacto aberto: contacto efetuado (resultado 1) enquanto o bit de controlo está a 1.
	NOT(E6)	Contacto fechado: contacto efetuado (resultado 1) enquanto o bit de controlo está a 0.
	R_TRIG1(CLK := E4); … := R_TRIG1.Q;	Contacto no flanco ascendente: contacto efetuado durante um ciclo quando se deteta um flanco ascendente no bit de controlo.
	F_TRIG2(CLK := E3); … := F_TRIG2.Q;	Contacto no flanco descendente: contacto efetuado durante um ciclo quando se deteta um flanco descendente no bit de controlo.

Tabela 2.8 — Operadores das instruções de *Store*

Ladder (LD)	Texto Estruturado (ST)	Modo de operação
	S1 := …	O resultado da função lógica ativa a bobina (*coil*) respetiva.
	S1 := NOT(…)	O resultado negado da função lógica ativa a bobina associada.
	IF…THEN S1 := TRUE; (*set*) END_IF;	O resultado da função lógica é armazenado no relé associado (*sets the latch*).
	IF…THEN S1 := FALSE; (*reset*) END_IF;	O resultado da função lógica é armazenado no relé associado (*resets the latch*)

Tabela 2.9 — Algumas funções e a sua representação em LD

Função	Representação (LD)	Declaração (*statement*) (ST)
Função E (AND)		S1.1 := E1.1 AND E1.2;
Função OU (OR)		S1.1 := E1.1 OR E1.2;
Função OU EXCLUSIVO (XOR)		S1.1 := XOR (E1.1, E1.2);

2.4 - Linguagem Function Block Diagram

O *Function Block Diagram* (FBD) é uma linguagem gráfica que permite ao utilizador programar elementos (por exemplo, blocos – estes blocos podem ser funções e/ou instâncias de blocos de funções), de tal forma que esses mesmos elementos encontram-se ligados entre si como que se de um circuito elétrico se trata-se [5]. Ou seja, é uma linguagem gráfica, de alto nível, baseada no conceito de fluxo de sinal (sinal esse, que pode ser de qualquer *data type*), o qual é "transmitido" através da ligação dos elementos do FBD. É uma linguagem, que nesta perspetiva, incorpora conceitos de programação orientada a objetos.

Uma das desvantagens deste tipo de linguagem é a sua difícil usabilidade quando a correção do programa depende da sequência de como os blocos são executados. Na Figura 2.10 encontra-se um exemplo deste tipo de programas.

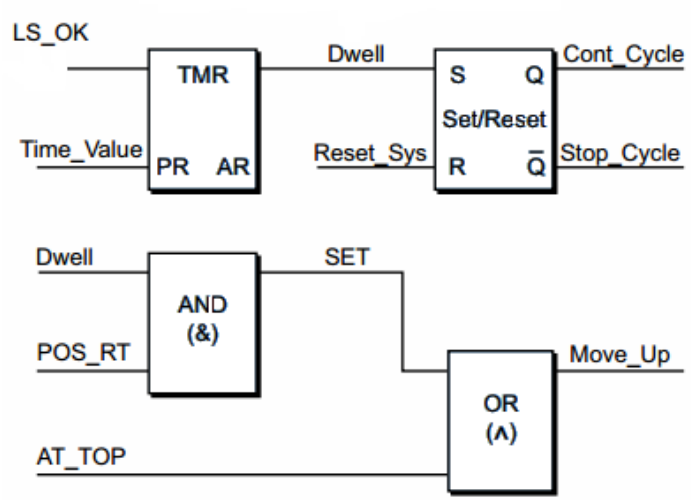


Figura 2.10 — Exemplo de um programa na linguagem FBD [5]

A linguagem FBD utiliza blocos de funções e/ou funções padronizadas pela Norma IEC 61131-3 e especificadas pelo fornecedor, tal como portas lógicas, contadores, temporizadores. Para além destes blocos, a linguagem FBD pode "usufruir" da implementação de blocos criados pelo utilizador assim como a Norma IEC 61131-3 permite (Subcapítulo 2.1). Existe, contudo, algumas regras de escrita nesta linguagem:

- Sinais podem divergir de uma saída para várias entradas;
- Ligações de entradas e saídas têm de ser do mesmo tipo;
- Instâncias de blocos de funções têm o seu nome acima do bloco;
- Longas linhas de ligação podem ser substituídas por conectores (*connectors*);

2.5 - Linguagem Sequential Function Charts

Sequential Function Charts, ou SFC, é uma linguagem gráfica e de alto nível que fornece uma representação esquemática de sequências de controlo de um programa. Basicamente, o SFC corresponde à implementação do Grafcet, isto porque é usada para definir máquinas de estados. Tem como principal fundamento a descrição das sequências de operações e interações entre processos paralelos, sequenciais e concorrentes. Esta linguagem não é propriamente uma linguagem de programação, mas antes uma linguagem de estruturação do programa. Tendo esta ideia em mente é facilmente percetível o porquê de ser necessário recorrer a uma das outras linguagens presentes na norma para definir ações e transições [11, 15].

Um programa SFC contém 4 elementos:

- Etapas (*steps*): representam o estado atual do sistema e cada etapa tem um identificador único. Pode existir mais que uma etapa ativa num determinado momento de execução do programa;
- Transições (*transitions*): representam as condições que definem a evolução do SFC;
- Ações (*actions*): representam as ações tomadas pelo SFC. Encontram-se associadas a uma etapa, assim sendo, dependem da ativação da etapa a que estão ligadas e do seu qualificador (*qualifier*). Estas ações podem ser escritas em IL, LD, ST, FBD ou SFC;
- Ligações (*directed links*): definem a sequência de ativação das etapas aquando de uma transição ativa. É sempre lida de cima para baixo (a não ser que seja especificado o contrário). A sua entrada é na parte superior da etapa e a sua saída na parte inferior da etapa;

De notar que existem algumas restrição aquando da escrita deste tipo de linguagem. Uma etapa tem de ser seguida de uma transição, não podem haver duas transições ligadas entre si diretamente e entre duas transições tem de haver uma etapa. Sequências "OR" têm sempre mais que uma transição e as sequências "AND" só têm uma transição assim como pode ser visto na Figura 2.11.

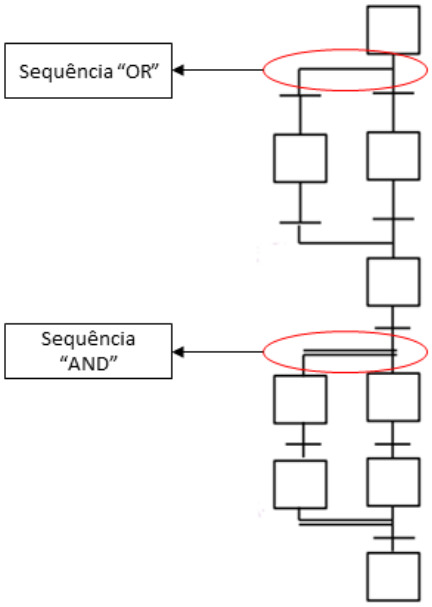


Figura 2.11 — Exemplo de um "esqueleto" de um programa SFC com as sequências "*AND*" e "*OR*"

Como exemplo, encontra-se na Figura 2.12 um excerto, de um programa SFC, devidamente legendado com a exemplificação dos elementos aqui abordados.

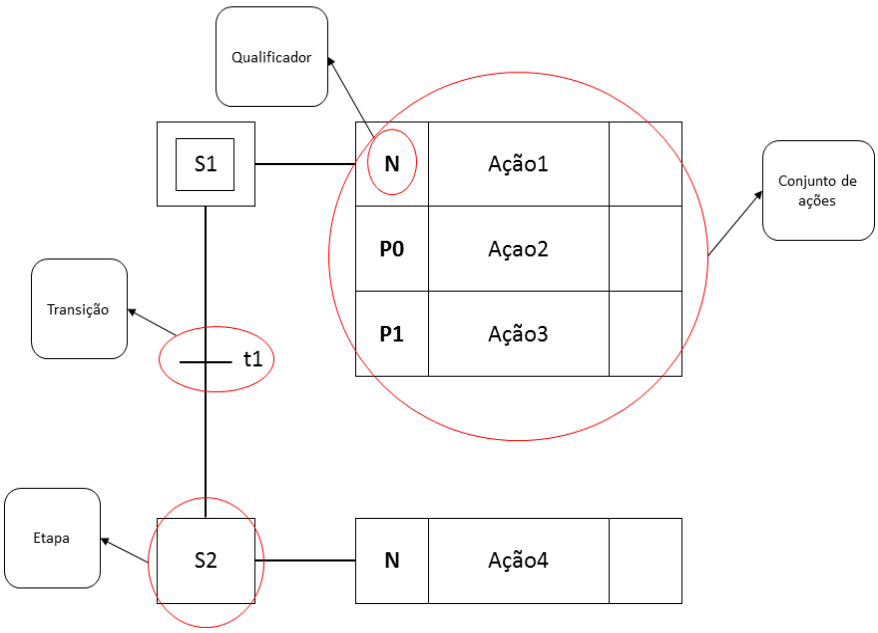


Figura 2.12 — Exemplo de um programa SFC

Com mais pormenor pode ser visto na Figura 2.13 os elementos presentes num bloco de ações. E na Tabela 2.10 podem ser consultados os diferentes tipos de qualificadores existentes.

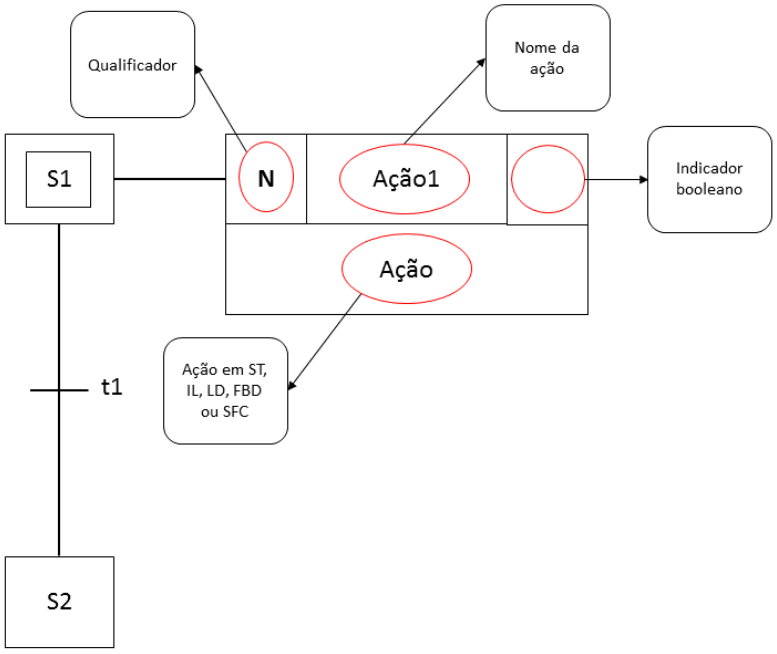


Figura 2.13 – Elementos constituintes de um bloco de ações num programa SFC

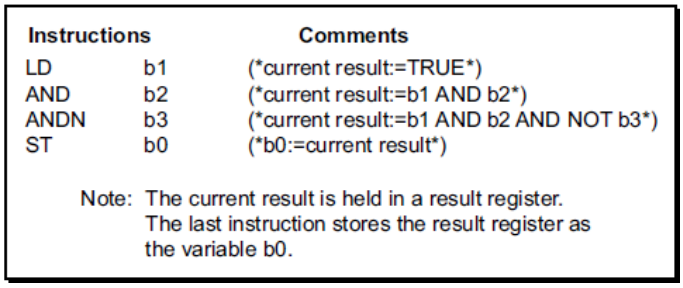
Tabela 2.10 – Conjunto de qualificadores das ações definidos pela Norma IEC 61131-3 [11]

Ações-Qualificadores definidos	
Qualificador	**Descrição**
N	*Non-Stored*
P1	*Pulse*, subida
P0	*Pulse*, descida
P	*Pulse*, <=> to P1
S	*Set* ou *Stored*
R	*Reset*
L	*Limited*
D	*Delayed*
SD	*Stored and Time Delayed*
DS	*Delayed and Stored*
SL	*Stored and Time Limited*

2.6 - Linguagem Instruction List

A Lista de instruções (IL) é uma linguagem de baixo nível semelhante à linguagem máquina ou *assembley* usada em microprocessadores (Figura 2.14). Este tipo de linguagem é útil para pequenas aplicações, bem como aplicações que requerem otimização da velocidade do programa ou uma rotina específica no programa. É uma linguagem um pouco enigmática e como tal, manteve-se na norma por razões históricas. Uma das principais desvantagens deste tipo de linguagem é o facto de ser difícil construir estruturas complexas e/ou sequências/ciclos complexos. Uma lista de instruções é composta por uma sequência de instruções. Cada instrução deve começar numa nova linha e irá conter um operador com modificadores opcionais e, se necessário, um ou mais operandos separados por vírgulas. Contudo, linhas vazias podem ser inseridas entre as instruções.

A instrução pode ser precedida por um rótulo de identificação seguido por dois pontos ":". Um comentário, se presente, deve ser o último elemento de uma linha.



Instructions		Comments
LD	b1	(*current result:=TRUE*)
AND	b2	(*current result:=b1 AND b2*)
ANDN	b3	(*current result:=b1 AND b2 AND NOT b3*)
ST	b0	(*b0:=current result*)

Note: The current result is held in a result register. The last instruction stores the result register as the variable b0.

Figura 2.14 — Exemplo de um programa escrito em IL [5]

Tabela 2.11 — Exemplo de um programa escrito em IL com os seus elementos descriminados

Label	Operador	Operando	Comentário
START:	LD	%IX1	(* PRIMIR BOTÃO *)
	ANDN	%MX5	(* NÃO ATIVADO *)
	ST	%QX2	(* LIGAR MOTOR *)

2.7 - eXtensible Markup Language

O XML (*eXtensible Markup Language*) é uma linguagem de marcação textual concebida para ser capaz de armazenar, transportar e trocar dados. Apresenta uma característica bastante relevante, visto que permite aos seus utilizadores a definição de etiquetas (*tags*) personalizadas, sendo que a única condição é que estas novas *tags* sigam as regras especificadas pela linguagem XML.

Por forma a apresentar os benefícios do XML, serão mencionados os seus antecessores e as suas características:

- SGML (*Standard Generalized Markup Language*);
- HTML (*Hyper Text Markup Language*);

O SGML é a língua-mãe de todas as linguagens de marcação. É um padrão internacional para a descrição de documentos e como tal, consegue separar o conteúdo da apresentação do formato. Quanto ao HTML é mais usado para capturar e publicar conteúdos em *Websites*, dando mais ênfase ao modo de apresentação da informação. Embora o SGML seja muito poderoso, é muito complexo, especialmente para os usos quotidianos de navegação da internet. Quanto ao HTML, este é composto por um conjunto pré-definido de *tags* e é usado principalmente para definir como o conteúdo deve ser apresentado. Em 1996, o *World Wide Web Consortium* (W3C) começou a definir uma nova linguagem de marcação com o poder do SGML e com a simplicidade de construção do HTML. Em 1998, o W3C aprovou a versão 1.0 da especificação XML, assim o XML é um subconjunto do SGML que consegue manter o poder do mesmo, mas através da simplicidade do HTML [16].

```
<?xml version="1.0"?>
<carros_VW>
        <vw>
                <modelo>Passat</modelo>
                <cor>Preto</cor>
                <ano>1998</ano>
        </vw>
        <vw>
                <modelo>Golf</modelo>
                <cor>Cinza</cor>
                <ano>2002</ano>
        </vw>
        <vw>
                <modelo>Polo</modelo>
                <cor>Vermelho</cor>
                <ano>2000</ano>
        </vw>
</carros_VW>
```

Figura 2.15 — Exemplo de um documento em XML

2.7.1 -Características

Devido às suas origens, o XML contém *tags*, atributos e valores, tal como o HTML. Apesar destas similaridades com o HTML, o XML não tem a mesma função. Isto é, ao contrário de servir como uma linguagem para exibir informações (característica principal do HTML), o XML é uma linguagem utilizada para armazenar e transportar informações como referido em cima. Outra vantagem do XML é que ele é facilmente adaptado, ou seja, o uso do XML permite "desenhar" linguagens de marcação personalizada para que, se possa usar essas linguagens para armazenar qualquer informação. Assim, pode-se obter uma linguagem de marcação personalizada que conterá *tags* que realmente descrevem os dados que elas contêm. O XML também pode ser usado para compartilhar dados entre sistemas e organizações diferentes. A razão para isto é que um documento XML é simplesmente um ficheiro de texto, e nada mais. É bem estruturado, fácil de entender, fácil de analisar, fácil de manipular e é considerado "legível". Finalmente, o XML é uma especificação que está livre para quem quiser utilizar. Este padrão aberto permitiu que as pequenas e grandes organizações pudessem usar o XML como um meio de partilha de informações [17].

Em suma, pode-se afirmar que as principais características positivas do XML são:

- Permite anotação de texto;
- Apresenta texto, dados e conteúdo para aplicações num documento estruturado;
- Facilita a integração de diversas aplicações;
- Em adição a estes benefícios, o XML é fácil de:
    - Ler (todo o texto);
    - Analisar e validar;
    - Procurar conteúdo;
    - Produzir;

2.7.2 -Regras e Estrutura

Para se obter um documento XML bem formado é necessário que o mesmo obedeça a um conjunto de regras, cujas mais importantes serão enumeradas a seguir:

- É obrigatório existir um elemento raíz (*root element*);
- É necessário que todos os elementos tenham *closing tags*;
- O encapsulamento dos elementos tem de ser adequado;
- XML é *case-sensitive*;
- Os valores dos atributos têm de estar entre aspas.

Como referido anteriormente, o XML partilha algumas semelhanças com o HTML entre as quais os mesmos blocos de construção, ou seja, *tags* que definem elementos, os valores desses

elementos e os atributos. Um elemento XML é a unidade mais básica do seu documento, sendo que este pode conter texto, atributos e outros elementos.

Um elemento tem uma *tag* de abertura com um nome escrito entre menor que ("<") e maior que (">"). O nome dado a esse elemento deve ser o mais representativo que possível, do seu conteúdo para uma leitura mais fácil. Um elemento é geralmente concluído com um *tag* de fecho, que terá o mesmo nome precedido por uma barra ("/"), fechado pelo sinal ">". Estes mesmos elementos podem ter atributos que estão contidos dentro da *tag* de abertura de um elemento, têm valores delimitados que descrevem ainda mais o propósito e conteúdo do elemento [17].

As informações que estão num atributo são, geralmente, consideradas metadados (*metadata*-isto é, a informação sobre os dados do elemento, em vez dos dados em si). Um elemento pode ter quantos atributos forem desejados, desde que cada um tenha um nome exclusivo. Na Figura 2.16 e na Figura 2.17 encontra-se exemplificado estes termos abordados.

Figura 2.16 — Exemplo de um excerto de um documento XML

O elemento "Nissan", mostrado na Figura 2.16, contém quatro outros elementos:

- "modelo";
- "cor";
- "ano";
- "peso";

De notar que este elemento não tem nenhum texto próprio. Os elementos "modelo", "cor", "ano" e "peso" contêm valores. O elemento "peso" é o único elemento que tem um atributo.

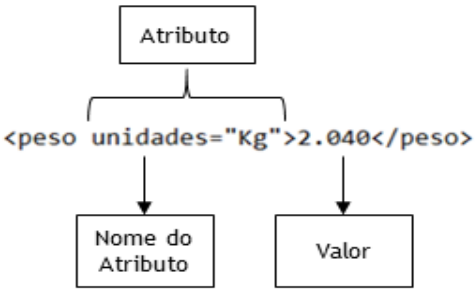


Figura 2.17 – Código XML do elemento "peso"

O elemento "peso" tem um atributo associado, referente às unidades, que serão apresentadas na unidade quilograma (KG).

2.8 - TC6-XML

A compreensão do formato TC6-XML (publicado pela PLCOpen) é crucial para o desenvolvimento deste Projeto, pois os documentos XML de entrada (escritos numa qualquer linguagem definida na Norma IEC 61131-3) que serão sujeitos à conversão estão guardados segundo esta estrutura que se encontra em concordância com a norma IEC 61131-3.

O formato TC6-XML fornece um esquema básico de interoperabilidade entre projetos, elaborados por diferentes ambientes, mantendo o *software* independente do *hardware* subjacente. Alguns ambientes de programação industrial (por exemplo o Beremiz e o Multiprog) já usam este esquema na construção dos seus projetos. Este esquema XML para além de guardar os dados sobre a estrutura dos Projetos ou a sua representação gráfica guarda, também, a informação extra, como elementos, ids ou dependências. Uma vez que todas estas informações adicionais já são apresentadas no arquivo XML, não há necessidade da existência de uma estrutura de dados auxiliar para manter um registro do mesmo. A Figura 2.18 mostra um fragmento de um ficheiro em TC6-XML correspondente a um bloco FBD. Nesta figura, "*localId*" indica o id do bloco, e cada "*refLocalId*" em cada "*connectionPointIn*" representa o ID dos seus antecessores [8]. Nos subcapítulos seguintes será demonstrado com mais detalhe a estrutura e representação deste tipo de documentos aplicados a programas de automação nas diferentes linguagens presentes na norma.

```
<block localId="1" typeName="OR"
executionOrderId="0" height="60"
width="53">
        <position x="295" y="24"/>
        <inputVariables>
          <variable formalParameter="IN1">
            <connectionPointIn>
              <relPosition x="0" y="30"/>
              <connection refLocalId="2">
              </connection>
            </connectionPointIn>
          </variable>
          <variable formalParameter="IN2">
            <connectionPointIn>
              <relPosition x="0" y="50"/>
              <connection refLocalId="3">
```

Figura 2.18 — Excerto da representação de um bloco funcional de FBD seguindo o esquema TC6-XML

2.8.1 -Elementos gerais

Nesta secção serão apresentados alguns elementos presentes nos documentos XML no formato/estrutura TC6-XML. Estes elementos, apesar de presentes, não serão cruciais para a conversão mas sim para a criação de documentos em TC6-XML. Assim sendo, serão abordados de uma forma generalista não dando grande detalhe sobre os mesmos.

Primeiramente existe o elemento "*project*" que tomará a função de *root element*, ou seja, será o elemento raiz dos documentos que irão dar entrada nos programas de conversão.

Este elemento é constituído por (como pode ser visto na Figura 2.19):

1. O elemento "*fileHeader*";
2. O elemento "*contentHeader*";
3. O elemento "*types*";
4. O elemento "*instances*";

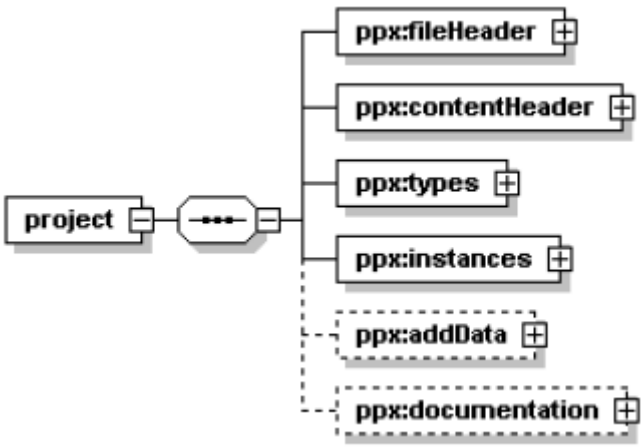


Figura 2.19 — Diagrama do elemento "*project*" [4]

O elemento "*fileHeader*" é usado para fornecer informações sobre a criação do ficheiro de exportação/importação. Os seus atributos obrigatórios são o nome da empresa (*company name*- com URL da empresa opcional), o nome do produto, a versão, informações e também a data e hora da criação do ficheiro. Além disso, o nome da empresa de fabrico e/ou fornecedor do produto pode ser incluído. O formato da data e hora está em conformidade com a especificação do consórcio W3C.

A partir da Tabela 2.12 pode-se observar os atributos do elemento "*fileHeader*", bem como a obrigatoriedade do seu uso ou não.[4]

Tabela 2.12 — Atributos do elemento "*fileHeader*" [4]

Nome	Uso
companyName	Obrigatório
companyURL	Opcional
productName	Obrigatório
productVersion	Obrigatório
productRelease	Opcional
productDateTime	Obrigatório
contentDescription	Opcional

O elemento "*contentHeader*" é usado para fornecer informações gerais a respeito do conteúdo real do ficheiro de exportação/importação. O elemento "*coordinateInfo*", filho do elemento "*contentHeade*r", contém as informações para o mapeamento do sistema de coordenadas.[4]

Tabela 2.13 — Atributos do elemento "*contentHeader*" [4]

Nome	Uso
name	Obrigatório
version	Opcional
modificationDateTime	Opcional
organization	Opcional
author	Opcional
language	Opcional

O elemento "*types*" é o responsável por conter dentro dele os elementos "*dataTypes*" e "pous". O elemento "*dataTypes*" guarda a informação sobre os diferentes *data types* criados num Projeto de automação, enquanto que o elemento "pous" contém a informação sobre os diferentes POUs, ou seja, é onde estará toda a informação relevante para os programas, *function block*s ou funções criadas pelo utilizador. O elemento "*instances*" pode conter um elemento de "*configurations*", que consiste em zero ou mais elementos "*configuration*". Estes guardam informação relativa às configurações criadas para o Projeto criado. Este elemento terá grande importância no desenvolvimento na ferramenta de conversão XML para TXT, pois só nesta ferramenta será preciso analisar o mesmo para a escrita do documento texto [4].

Um elemento bastante importante para ambas as ferramentas de conversão e que já foi mencionado acima, é o elemento "pou". Neste elemento pode-se saber o nome e o tipo de POU que está guardado. Este elemento é o "pai" dos elementos "filhos" que guardam informação sobre as variáveis e sobre o corpo do programa criado ("*interface*" e "*body*").

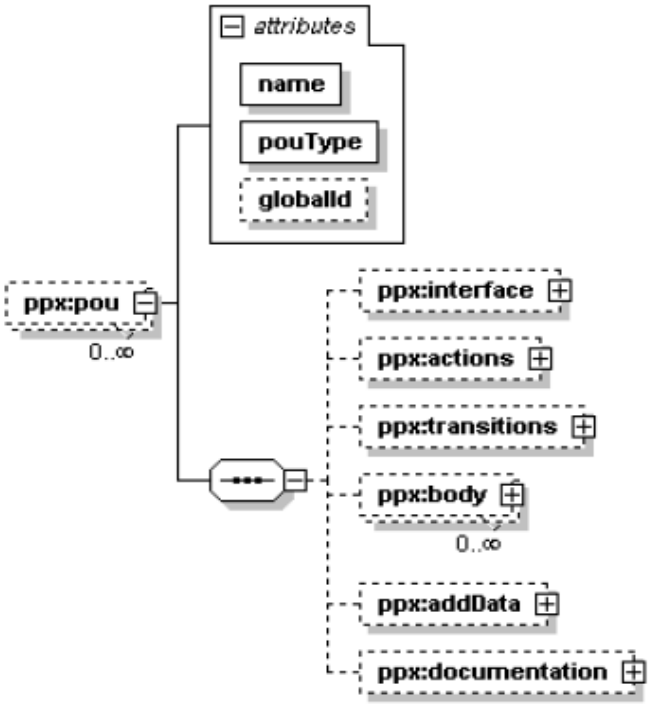


Figura 2.20 — Diagrama do elemento "pou" [4]

Tabela 2.14 — Atributos do elemento "pou" [4]

Nome	Uso
name	Obrigatório
pouType	Obrigatório
globalId	Opcional

A interface de um POU (programas, funções ou blocos de funções) representa uma lista de vários tipos de variáveis:

1. Variáveis locais (representadas pelo elemento "*localVars*");
2. Variáveis temporárias (representadas pelo elemento "*tempVars*");
3. Variáveis de entrada (representadas pelo elemento "*inputVars*");

4. Variáveis de saída (representadas pelo elemento "*outputVars*");
5. Variáveis de entrada/saída (representadas pelo elemento "*inOutVars*");
6. Variáveis externas (representadas pelo elemento "*externalVars*");

Quanto à configuração podem existir variáveis tipos de variáveis:

1. Variáveis de acesso (representadas pelo elemento "*accessVars*");
2. Variáveis globais (representadas pelo elemento "*globalVars*", também utilizada nos *resources* de cada Projeto);

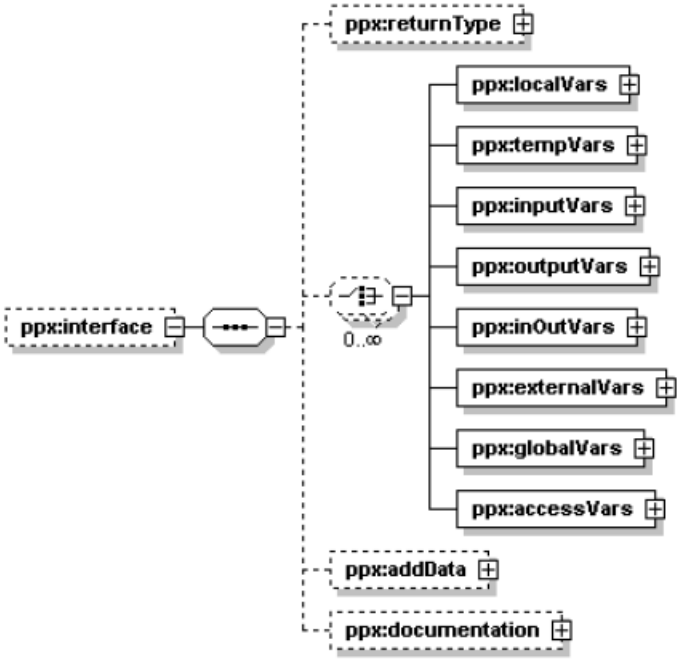


Figura 2.21 — Diagrama do elemento "*interface*" [4]

Qualquer que seja o tipo de variável, ambas seguem a mesma estrutura XML e têm os mesmos atributos [4].

No que diz respeito ao elemento "*body*", é neste que é guardada toda a informação relativa às funções presentes no POU, ou seja, toda a informação relevante para a construção do corpo de cada POU. É neste elemento que se consegue extrair dados sobre o tipo de linguagem de automação (ST, IL, SFC, FBD ou LD – como pode ser visto na Figura 2.22) que está implementado em determinado POU e assim desencadear os métodos mais apropriados para a conversão que se deseja (XML ou TXT).

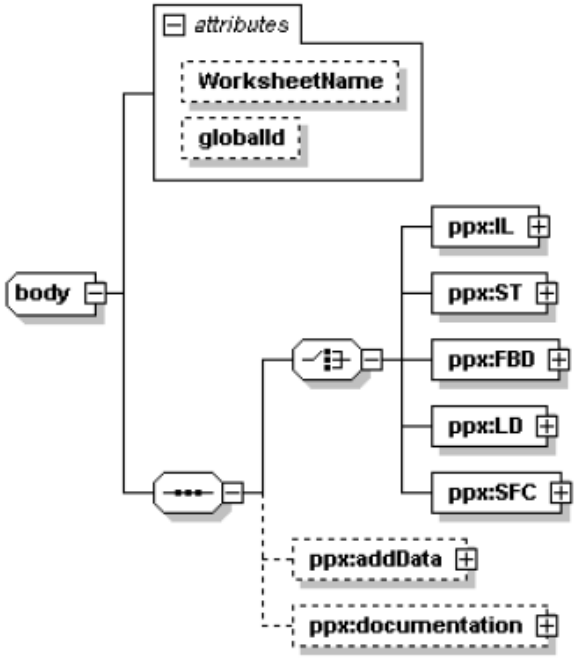


Figura 2.22 — Diagrama do elemento "*body*" [4]

Nas linguagens gráficas existem elementos comuns que serão agora abordados devido à sua crucial compreensão para conseguir desenvolver os dois conversores. Assim, os elementos que serão citados podem ser utilizados em qualquer programa gráfico (LD, FBD e SFC).

Uma visão global desses elementos encontra-se na Figura 2.23.

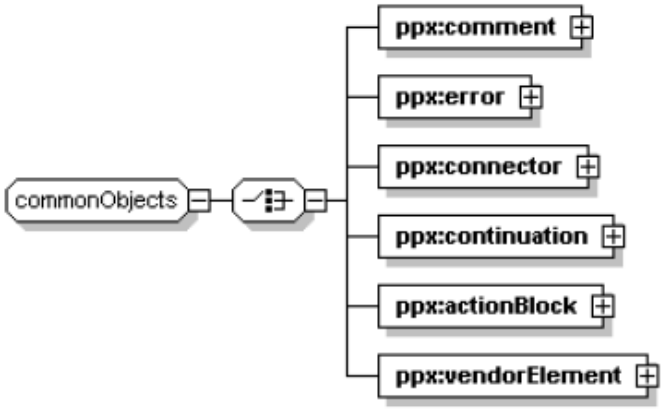


Figura 2.23 — Visão global dos elementos comuns às linguagens gráficas [4]

O elemento "*comment*" é usado para armazenar sequências de textos arbitrários, que não estão associadas a um local gráfico. Estes textos são, por exemplo, apresentados dentro de uma caixa/janela de diálogo dentro da GUI [4].

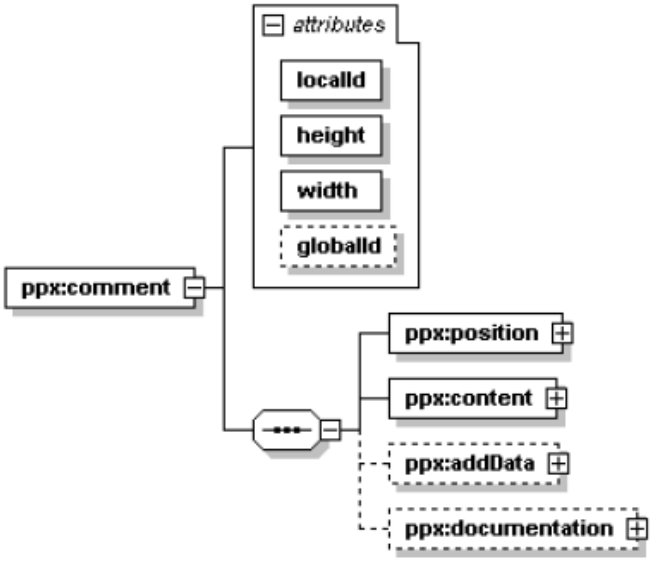


Figura 2.24 — Diagrama do elemento "*comment*" [4]

Assim como no caso do elemento "*comment*", o elemento "*error*", é usado para armazenar sequências de texto dentro de um quadro retangular, que está associado a um local gráfico, que tem uma altura e uma largura definida.

Em contraste com a caixa de comentários, a caixa de erro e o texto dentro dessa caixa são criados automaticamente pelo próprio *software* para indicar falhas durante as operações de conversão destes programas para o formato XML (não confundir com as conversões que se pretendem alcançar com este Projeto) [4].

Sempre que existam pontos de ligação (entrada ou saída) nos elementos presentes nestes tipos de linguagem (contactos, blocos, bobinas, variáveis, entre outros) são criados os elementos "*connectionPointIn*" e "*connectionPointOut*" que representam os seus pontos de entrada e os pontos de saída respetivamente. A sua estrutura pode ser vista na Figura 2.25 e na Figura 2.26.

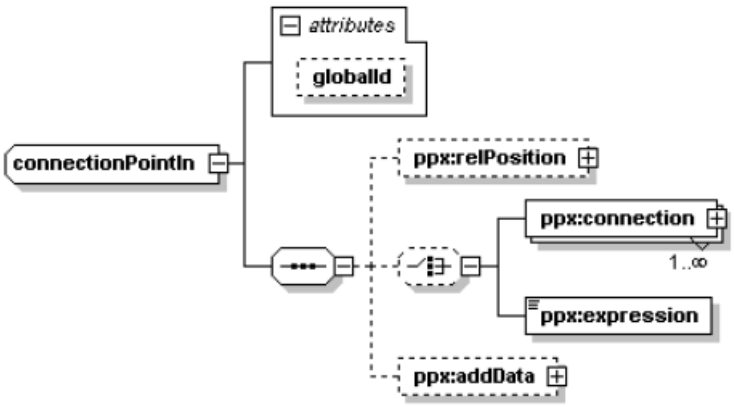


Figura 2.25 — Diagrama do elemento "*connectionPointIn*" [4]

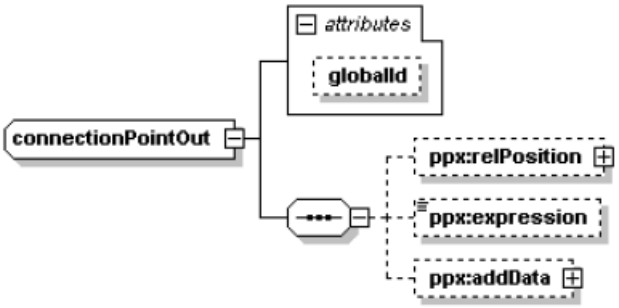


Figura 2.26 — Diagrama do elemento "*connectionPointOut*" [4]

Uma ligação (representada pelo elemento "*connection*" - Figura 2.27) descreve um acoplamento entre um elemento gráfico que "consome" dados (por exemplo, uma variável de entrada) e um outro elemento, que fornece os dados (variável de saída, e, normalmente, está em frente do "elemento pai"). Ela pode conter uma lista de posições que descrevem o percurso ou trajetória da conexão. Se nenhuma informação sobre a posição é fornecida, então a ligação deve ser encaminhada automaticamente. O atributo "*refLocalId*" identifica o elemento que dá origem à conexão (ponto de partida).

Se estiver presente, "*formalParameter*" identifica o elemento que serve como destino à conexão (ponto de chegada).

O atributo "*formalParameter*" ou indica o nome do parâmetro VAR_OUTPUT/VAR_IN_OUT de um bloco POU ou refere-se ao atributo "*formalParameter*" do correspondente "*connectionPointOut*" [4].

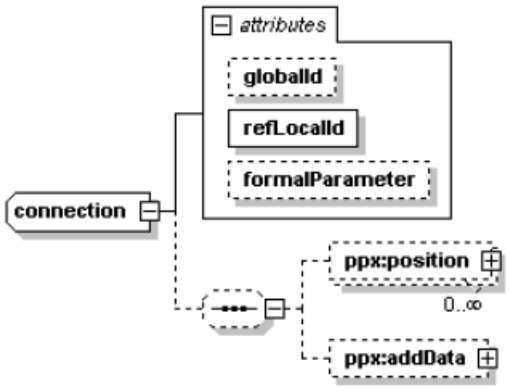


Figura 2.27 — Diagrama do elemento "*connection*" [4]

2.8.2 -TC6-XML (FBD)

Nos programas FBD existem um conjunto de elementos que representam a sua estrutura gráfica, quer seja através de blocos, ligações, variáveis, entre outros [4]. Assim sendo, é de extrema importância a sua compreensão para que seja possível incumbir as funcionalidades desejadas no conversor de linguagens.

Os elementos constituintes do FBD são uma coleção de objetos que podem ser usados em todos os organismos gráficos [4].

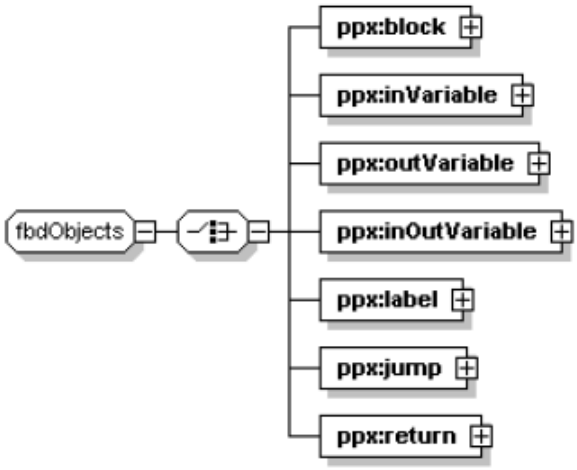


Figura 2.28 — Elementos presentes num programa FBD [4]

Num programa FBD, um bloco é uma representação gráfica de uma operação de uma função ou de um bloco de funções.

O elemento "*block*" é gerado cada vez que é criado um bloco (quer seja função ou bloco de funções) no programa FBD, sendo que o mesmo é constituído por elementos próprios visíveis na Figura 2.29.

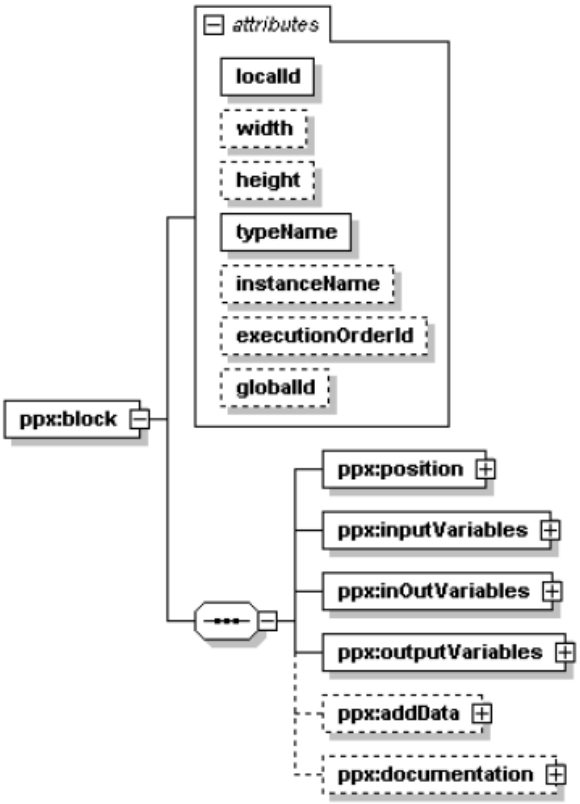


Figura 2.29 — Diagrama do elemento "*block*" [4]

Os três tipos de variáveis (*input variables*, *inOut variables* e *output variables*) podem conter, cada uma, zero ou mais variáveis.

A variável de entrada pode ter um elemento "*connectionPointIn*". A variável de saída pode ter um elemento "*connectionPointOut*". Uma variável de entrada/saída ("*inOutVariable*") pode ter um elemento "*connectionPointIn*" e um elemento "*connectionPointOut*". O elemento "*inVariable*" (Figura 2.30) representa uma variável de entrada. A variável de entrada pode ser negada, mas também pode conter um "*edge modifier*" ou um "*storage modifier*" [4].

De forma análoga o elemento "*outVariable*" representa uma variável de saída e o elemento "*inOutvariable*" representa uma variável que pode ser de saída ou entrada. Ambas podem ser igualmente negadas e podem conter um "edge modifier" ou um "storage modifier".

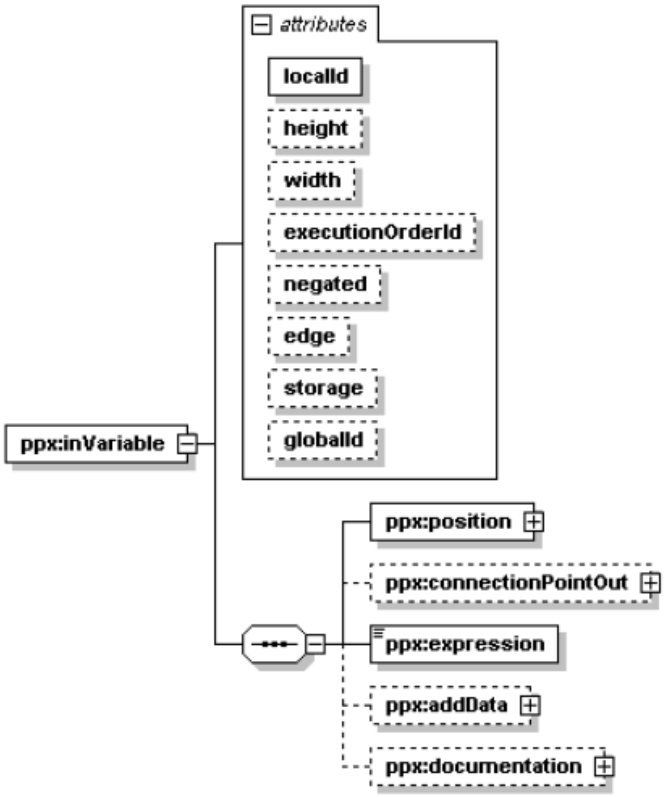


Figura 2.30 — Diagrama do elemento "*inVariable*" [4]

Outros elementos presentes num programa em FBD são as "*labels*" e "*jumps*". Estes elementos são utilizados aquando da implementação de um salto (*jump*) ou de um ponto de chegada de um salto (*label*).

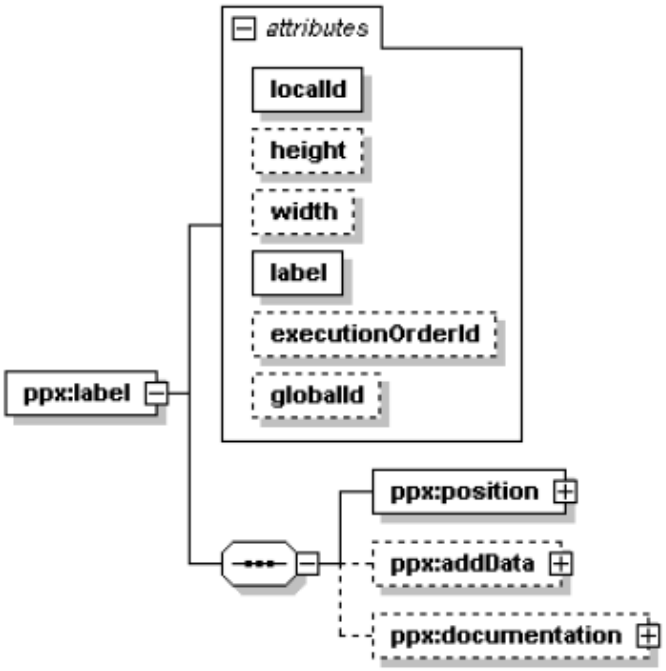


Figura 2.31 — Diagrama do elemento "*label*" [4]

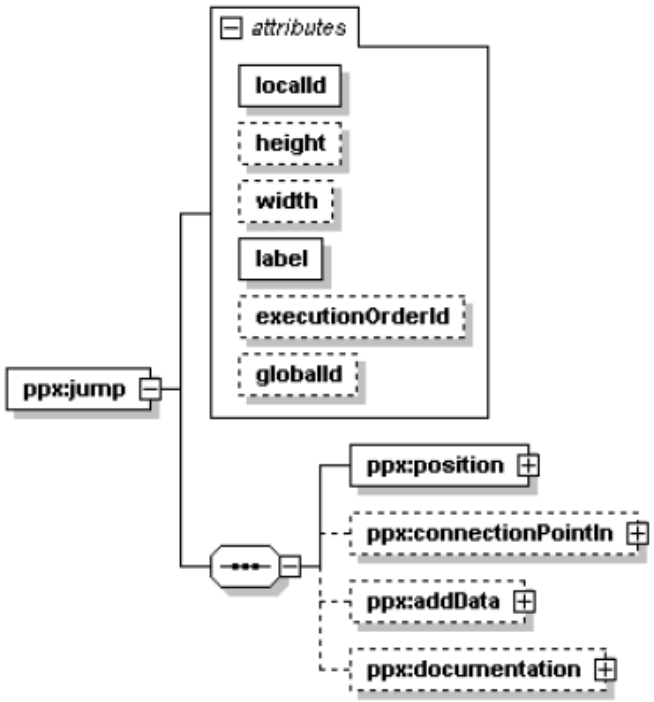


Figura 2.32 — Diagrama do elemento "*jump*" [4]

2.8.3 -TC6-XML (LD)

Num programa *Ladder* podem estar presentes elementos únicos a este tipo de programa, assim como elementos de um programa em FBD, ou seja, são quase como uma extensão dos elementos presentes num FBD que podem ser implementados em LD.

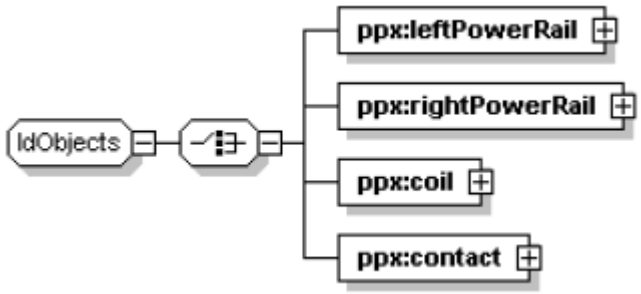


Figura 2.33 — Elementos presentes num programa LD [4]

O elemento "*leftPowerRail*" representa a existência da linha de energia do lado esquerdo, enquanto o elemento "*rightPowerRail*" representa a existência, mas desta feita, da linha de energia do lado direito.

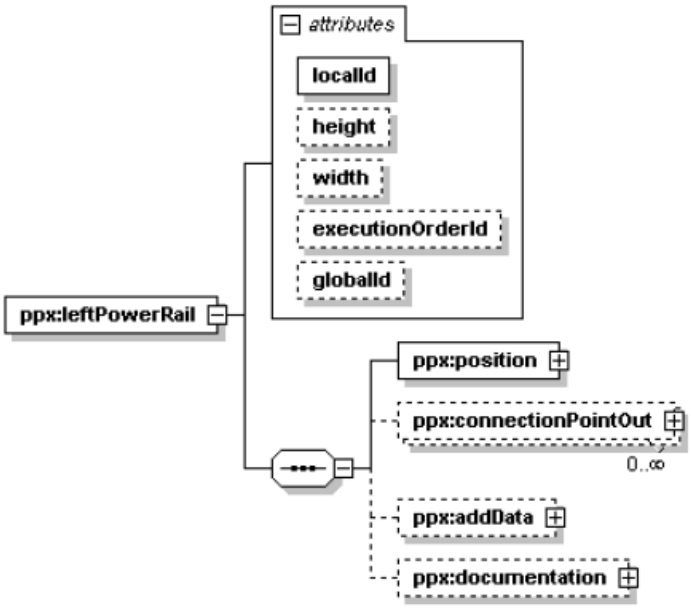


Figura 2.34 — Diagrama do elemento "*leftPowerRail*" [4]

Na Figura 2.34 podem ser observados os elementos que constituem o elemento "*leftPowerRail*", sendo que a única diferença para com o elemento "*rightPowerRail*" é o facto de o elemento "*connectionPointOut*" ser substituído pelo elemento "*connectionPointIn*".

Torna-se de fácil compreensão esta diferença, pois a linha de energia do lado esquerdo só contém pontos de saída e a linha de energia do lado direito só contém pontos de entrada.

As bobinas são representadas através do elemento "*coil*" (Figura 2.35). O mesmo contém um ponto de entrada (elemento "*connectionPointIn*") e um ponto de saída (elemento "*connectionPointOut*") e tem uma variável associada (elemento "*variable*").

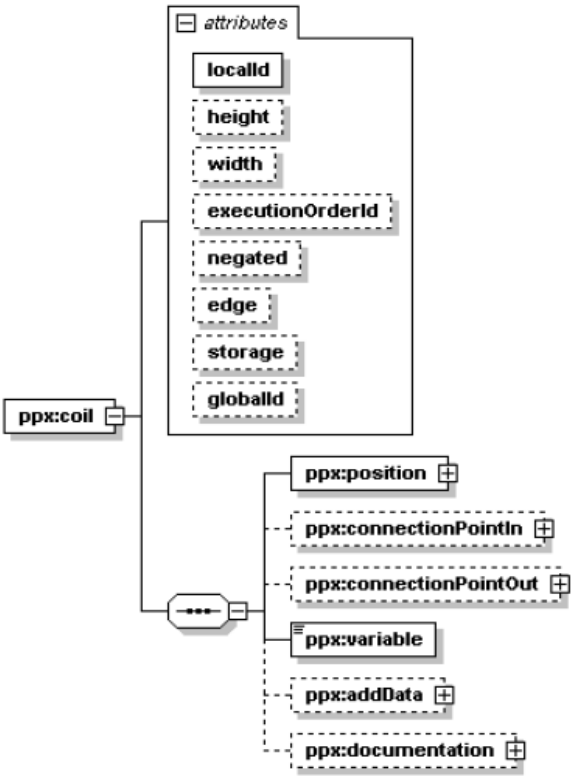


Figura 2.35 — Diagrama do elemento "*coil*" [4]

A cada contacto, presente num programa LD, é criado o elemento "*contact*" (Figura 2.36). De forma análoga ao elemento "*coil*", o mesmo contém um ponto de entrada (elemento "*connectionPointIn*") e um ponto de saída (elemento "*connectionPointOut*") e contém uma variável associada (elemento "*variable*").

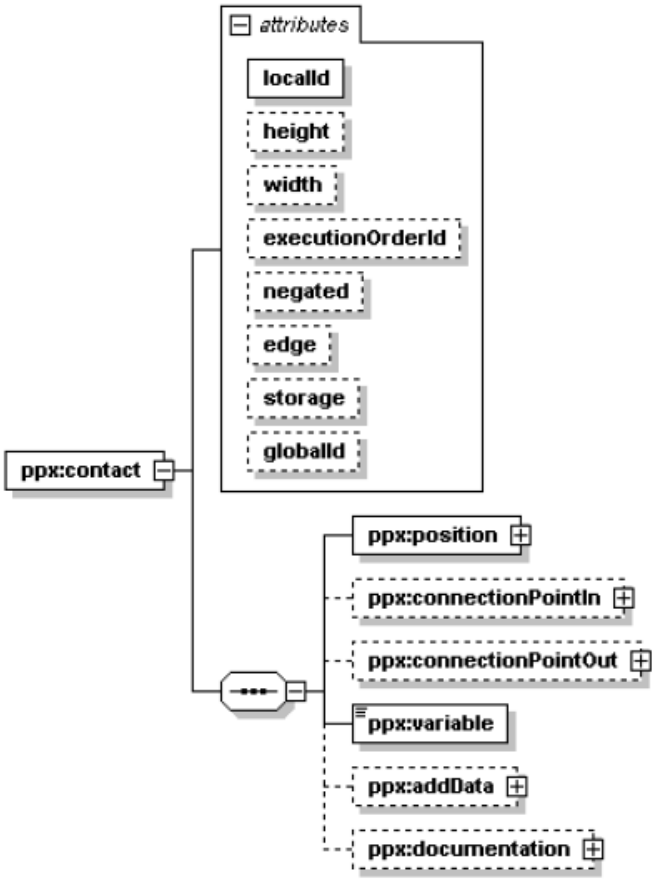


Figura 2.36 — Diagrama do elemento "*contact*" [4]

2.8.4 -TC6-XML (SFC)

Num programa SFC são criados elementos únicos representativos dos elementos presentes nesta linguagem. Alguns desses elementos serão aqui descritos.

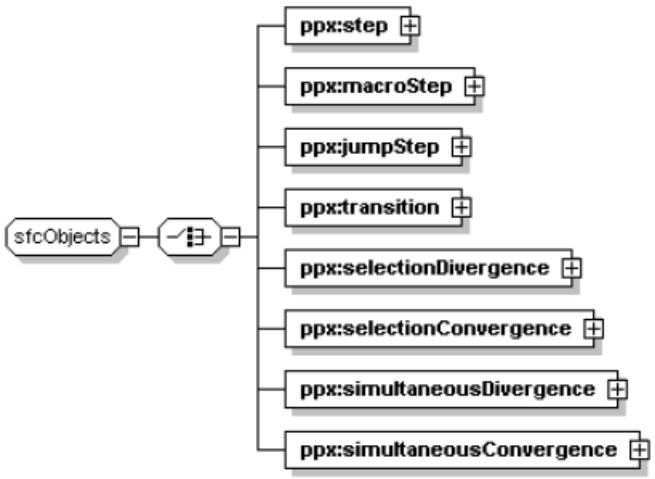


Figura 2.37 — Elementos presentes num programa SFC [4]

O elemento "*step*" (Figura 2.38) representa um único passo ("step") num SFC. Existem ações que estão associadas a uma etapa usando um elemento "*actionBlock*" que contém uma conexão para o elemento "*step*" [4].

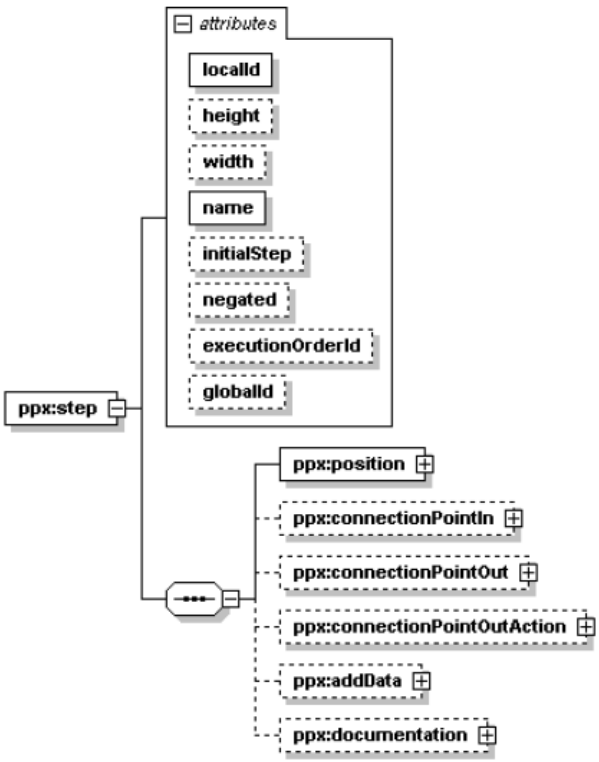


Figura 2.38 — Diagrama do elemento "*step*" [4]

As transições presentes num programa em SFC são representadas pelo elemento "*transition*" (Figura 2.39) que é criado cada vez que exista uma transição.

Estas mesmas transições contêm condições, sendo que as mesmas podem incluir uma referência, uma ligação, ou um código embutido, programado em qualquer uma das cinco linguagens.

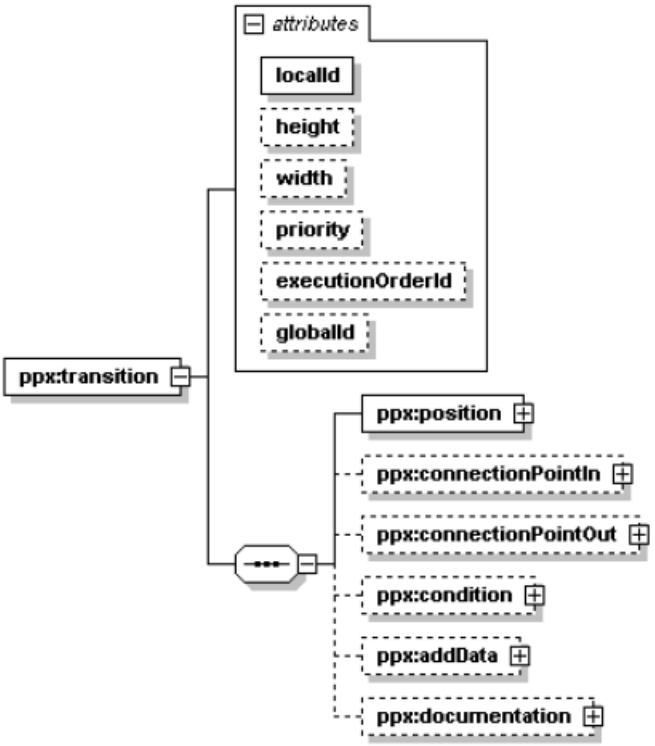


Figura 2.39 — Diagrama do elemento "*transition*" [4]

Na implementação de "ramos" que indiquem a simultaneidade das ações (*Simultaneous Branch*), o elemento "*simultaneousDivergence*" (Figura 2.41) é então criado por forma a representar os mesmos. Aquando do seu fecho é utilizado o elemento "*simultaneousConvergence*" presente na Figura 2.40.

De forma análoga, na implementação de "ramos" que indiquem a seleção das ações (*Selection Branch*) o elemento "*selectionDivergence*" (Figura 2.43) é então criado por forma a representar os mesmos. Aquando do seu fecho é utilizado o elemento "*selectionConvergence*" (Figura 2.42).

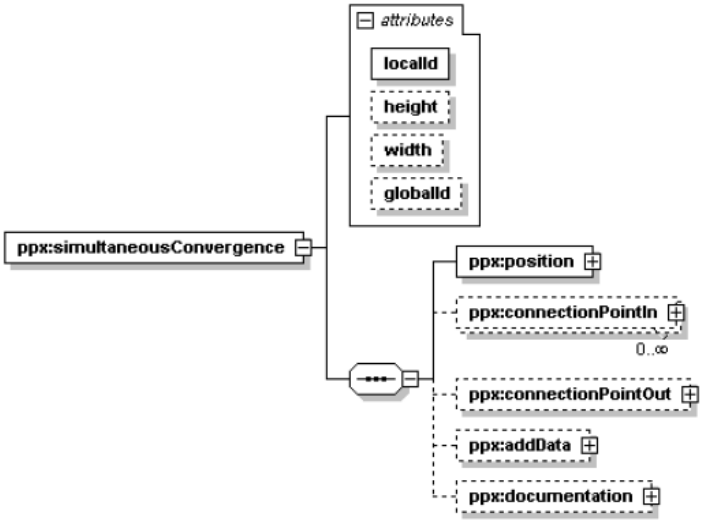


Figura 2.40 – Diagrama do elemento "*simultaneousConvergence*" [4]

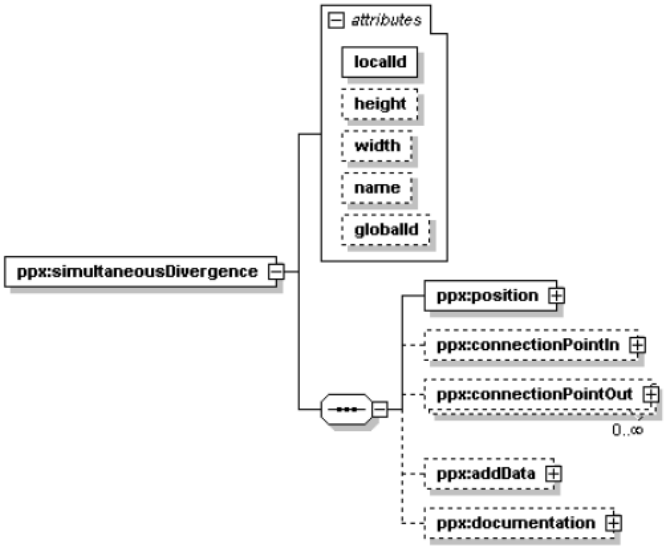


Figura 2.41 – Diagrama do elemento "*simultaneousDivergence*" [4]

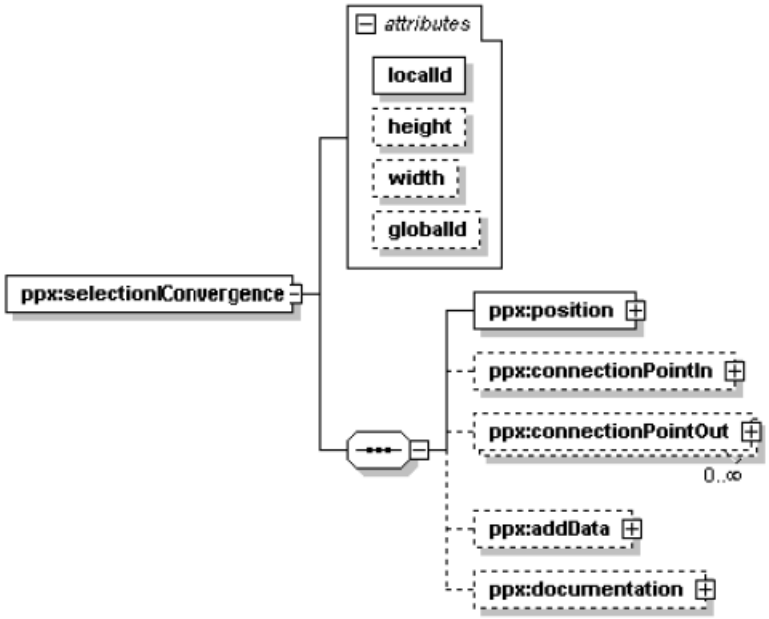


Figura 2.42 – Diagrama do elemento "*selectionConvergence*" [4]

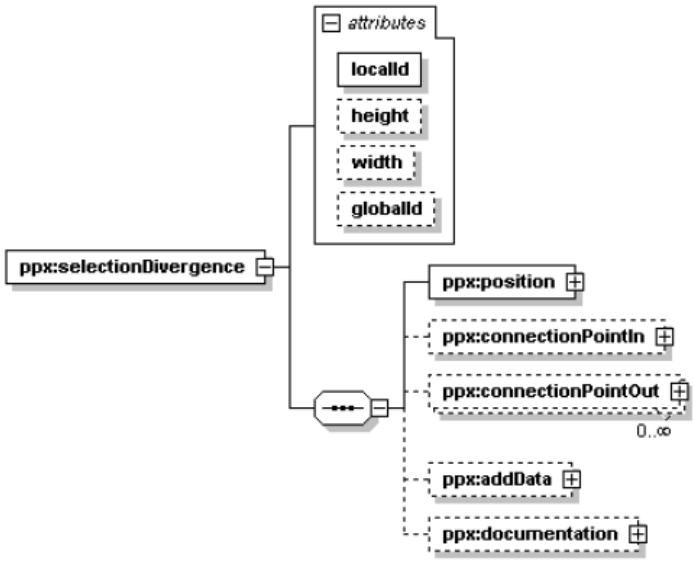


Figura 2.43 – Diagrama do elemento "*selectionDivergence*" [4]

As ações presentes em cada *step* são representadas através do elemento "*connectionPointOutAction*" que representa a ligação de saída de cada *step*. A mesma pode ser vista com maior detalhe na Figura 2.44.

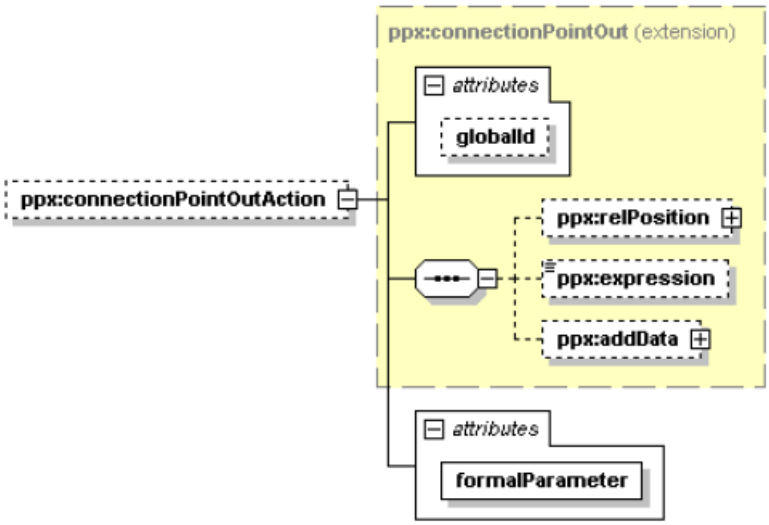


Figura 2.44 — Diagrama do elemento "*connectionPointOutAction*" [4]

2.8.5 -TC6-XML (ST e IL)

De uma forma completamente diferente das linguagens referidas nos Subcapítulos 2.8.2, 2.8.3 e 2.8.4 (FBD, LD e SFC respetivamente), as linguagens textuais não são representadas por elementos que pretendam representar as suas partes gráficas constituintes, isto porque elas simplesmente são inexistentes. Assim, todo o programa escrito neste tipo de linguagem é armazenado num elemento CDATA. O elemento CDATA significa a presença de dados de caracteres (*character data*) em que o texto contido no mesmo não deve ser analisado pelo *parser* de documentos XML. No fundo, todo o texto que estiver contido dentro deste elemento será ignorado pelo interpretador de documento XML.

Estes elementos começam com "<![CDATA[" e terminam com "]]>".

```
<body>
  <ST>
    <xhtml:p><![CDATA[
        functionBlock00(LocalVar0 := LocalVar0, LocalVar1 := LocalVar1);
        LocalVar3 := functionBlock00.LocalVar3 AND LocalVar1;
        LocalVar2 := functionBlock00.LocalVar2;]]>
    </xhtml:p>
  </ST>
</body>
```

Figura 2.45 — Programa ST num documento XML

2.9 - Processamento de documentos XML

Para processar as informações presentes em documentos XML existem diversas linguagens e bibliotecas criadas para esse efeito. Assim, neste subcapítulo, serão analisadas algumas APIs (*Application Programmers Interface*):

- O DOM (*Document Object Model*);
- O SAX (*Simple API for XML*);

Estas APIs são, provavelmente, as abordagens mais populares para o processamento de documentos XML.

O DOM tem como característica a sua abordagem em "árvore". O documento é visto como uma árvore em memória e o acesso aos seus elementos passa por um problema de manipulação de uma estrutura deste tipo. Esta abordagem tem no entanto alguns problemas, como por exemplo a perda de eficiência devido à quantidade de memória consumida por grandes documentos. No caso do SAX a abordagem é "guiada por eventos", assim, o documento é processado sequencialmente, sendo os seus elementos associados a determinados eventos com determinadas consequências. Assim deixa-se, neste caso, de ter sempre acesso ao documento no seu todo. Uma explicação mais detalhada será dada nos subcapítulos subsequentes [18].

2.9.1 -DOM

O DOM é uma ferramenta que permite o acesso dinâmico por parte de programas e scripts, permite a atualização do conteúdo, estrutura e estilo do documento. O DOM encontra-se dividido em três níveis diferentes:

- Core DOM;
- XML DOM;
- HTML DOM;

O Core DOM é um modelo padrão para qualquer documento estruturado, o XML DOM é um modelo padrão para documentos XML e o HTML DOM é um modelo padrão para documentos HTML [1].

Como os documentos que serão sujeitos à conversão fazem uso do XML para salvar os seus dados é necessário apenas abordar o XML DOM. O XML DOM é independente da plataforma e da linguagem, define os objetos e as propriedades de todos os elementos XML e os métodos (interface) para acede-los. Em outras palavras, o XML DOM é um padrão para obter, adicionar, modificar ou remover elementos num documento XML.

De acordo com o DOM tudo o que existe num documento XML (elementos, atributos, comentários e textos) é considerado um nó e esses nós são organizados em forma de árvore. O XML DOM contém funções para percorrer, aceder, inserir e remover nós do documento. O DOM oferece suporte à DTD e toda a API está presente no Java SE 6.0. Alguns *parsers*, incluindo o Xerces, oferecem um método denominado de "*lazy* DOM", que ao ser executado, deixa a maior parte do documento no disco e armazena na memória as partes do documento necessárias ao programa em um determinado momento. Nos exemplos abaixo é exemplificado uma implementação em C++ para construir os elementos presentes num código XML através do *parser* TinyXML [1].

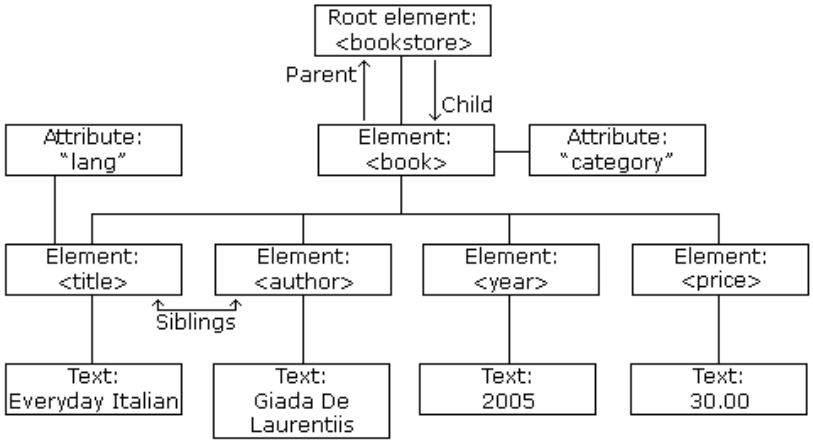


Figura 2.46 — Exemplo da árvore em XML DOM [2]

```
<?xml version="1.0" ?>
<MyApp>
   <!-- Settings for MyApp -->
   <Messages>
      <Welcome>Welcome to
MyApp</Welcome>
      <Farewell>Thank you for using
MyApp</Farewell>
   </Messages>
   <Windows>
      <Window name="MainFrame" x="5"
y="15" w="400" h="250" />
   </Windows>
   <Connection ip="192.168.0.1"
timeout="123.456000" />
</MyApp>
```

Figura 2.47 — Exemplo de um documento XML a ser duplicado pelo *parser* TinyXML [3]

```
void write_app_settings_doc( )
{
    TiXMLDocument doc;
    TiXMLElement* msg;
    TiXMLDeclaration* decl = new TiXMLDeclaration( "1.0", "", "" );
    doc.LinkEndChild( decl );
    TiXMLElement * root = new TiXMLElement( "MyApp" );
    doc.LinkEndChild( root );
    TiXMLComment * comment = new TiXMLComment();
    comment->SetValue(" Settings for MyApp " );
    root->LinkEndChild( comment );
    TiXMLElement * msgs = new TiXMLElement( "Messages" );
    root->LinkEndChild( msgs );
    msg = new TiXMLElement( "Welcome" );
    msg->LinkEndChild( new TiXMLText( "Welcome to MyApp" ));
    msgs->LinkEndChild( msg );

    msg = new TiXMLElement( "Farewell" );
    msg->LinkEndChild( new TiXMLText( "Thank you for using MyApp" ));
    msgs->LinkEndChild( msg );

    TiXMLElement * windows = new TiXMLElement( "Windows" );
    root->LinkEndChild( windows );

    TiXMLElement * window;
    window = new TiXMLElement( "Window" );
    windows->LinkEndChild( window );
    window->SetAttribute("name", "MainFrame");
    window->SetAttribute("x", 5);
    window->SetAttribute("y", 15);
    window->SetAttribute("w", 400);
    window->SetAttribute("h", 250);
```

Figura 2.48 — Código em C++ utilizando o *parser* TinyXML para a construção de um documento XML [3]

2.9.2 -SAX

O SAX (*Simple API for XML*) foi originalmente criado como uma API Java e foi desenvolvida por membros da XML-DEV *mailing-list*. Hoje já existem versões para outras linguagens de programação. É usado para ler grandes documentos XML possuindo uma API baseada em eventos. Com uma API baseada em eventos não há necessidade de armazenar o documento na memória, assim, as notificações (eventos) ocorrem à medida que o documento é analisado. Ou seja, o SAX apresenta cada nó do documento XML em sequência. Então, quando se passar para a fase de processamento de um nó, já se deve ter guardado as informações sobre todos os nós anteriores relevantes. O SAX usa menos memória que o DOM e é uma alternativa adequada para documentos que podem ser processados em sequência, em vez de como um todo [1].

Alguns exemplos de *parsers* que utilizam SAX são: Xerces, JAXP e MSXML 3.0.

A seguir serão apresentados dois códigos, o primeiro referente a um documento XML e o segundo demonstra como o SAX gera os eventos desse mesmo documento.

Dado o documento XML abaixo:

```
<?XML version="1.0"?>
<samples>
<server>UNIX</server>
<monitor>color</monitor
```

Figura 2.49 — Exemplo de um documento XML a ser processado pelo SAX [1]

O SAX gera os seguintes eventos:

```
Start document
Start element (samples)
Characters (white space)
Start element (server)
Characters (UNIX)
End element (server)
Characters (white space)
Start element (monitor)
Characters (color)
End element (monitor)
```

Figura 2.50 — Lista de eventos gerada pelo processamento do SAX [1]

2.9.3 -Comparação

Como podemos perceber pela Tabela 2.15, as APIs baseadas em árvore (DOM) são úteis para um grande número de aplicações, mas elas, normalmente, utilizam grandes recursos do sistema, especialmente se o documento for grande. Além disso, muitas aplicações precisam de construir as suas próprias estruturas de dados em vez de uma árvore genética criada pelo próprio *parser*.

Em ambas as situações, uma API baseada em eventos, proporciona um nível de acesso inferior a um documento XML de uma forma mais simples, ou seja, pode-se analisar documentos muito maiores sem utilizar grandes recursos de memória do sistema e construir estruturas de dados próprias.

Tabela 2.15 – Tabela comparativa entre SAX e DOM [1]

Feature	**SAX**	**DOM**
API TYPE	*Push, streaming*	*In memory tree*
Ease of use	*Medium*	*High*
CPU and Memory Efficiency	*Good*	*Varies*

2.9.4 -JDOM

Tendo em conta o que foi referido no Subcapítulo 2.9 e subpontos, tinha de ser utilizada uma API que conseguisse ser simples e de fácil aplicação e ter todas as funcionalidades pretendidas (leitura, escrita, atualização...).

Assim, chegou-se ao JDom. Trata-se de uma ferramenta baseada em Java, *open source*, de *parsing* de documentos XML que foi projetado especificamente para a plataforma Java para que se possa tirar proveito dos recursos desta linguagem. Esta ferramenta integra o DOM e o SAX, suporta XPath e XSLT. OJDOM foi desenvolvido por Jason Hunter e Brett McLaughlin, a partir de Março de 2000 [7, 19].

```
Element root = new Element("shop");
root.setAttribute("name", "shop for geeks");
root.setAttribute("location", "Tokyo, Japan");
Element item1 = new Element("computer");
item1.setAttribute("name", "iBook");
root.addContent(item1);
```

Figura 2.51 — Excerto de código para criação de elementos e atributos utilizando o JDom [7]

2.10 - Beremiz

Apesar da existência de padrões livres como o IEC 61131, PLCopen e CanOpen, existe uma certa dificuldade em transferir facilmente programas entre fornecedores. Assim, como resposta a este problema, o *software* Beremiz foi desenvolvido. Trata-se de um *software* disponibilizado de forma gratuita para a prática de automação de PLCs que contém alguns aspetos especiais:

- Ambiente de Desenvolvimento Integrado;
- *Software runtime* Incorporado;
- Automação, Controlo e *software* HMI;

O Beremiz é um ambiente de desenvolvimento integrado para automação de máquinas. É livre, encontra-se em conformidade com a Norma IEC 61131 e baseia-se em padrões livres para ser independente do dispositivo de destino [20].

Existem 4 subprojectos dos quais o Beremiz depende:

- PLCOpen *Editor*;
- Compilador MatPLC's IEC;
- CanFestival;
- SVGUI;

É nesta ferramenta que são criados os programas nas linguagens gráficas e textuais (usadas para os testes dos dois conversores), que são armazenados em documentos XML segundo o esquema TC6-XML.

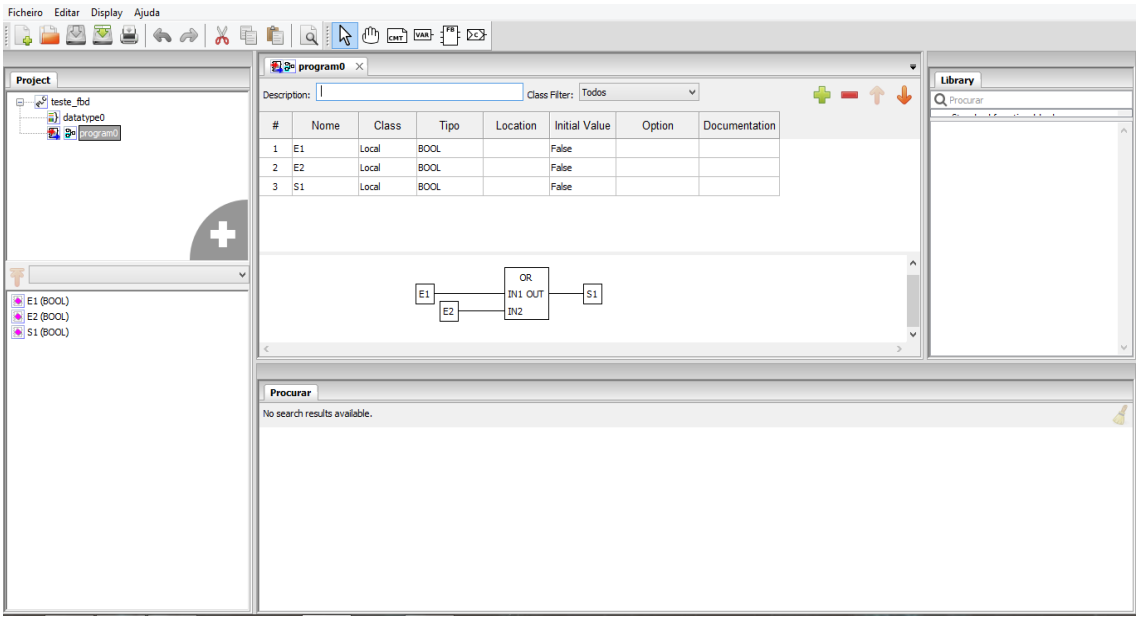


Figura 2.52 — *Software* Beremiz e a criação de um programa simples em FBD

Na Figura 2.52 está presente a interface do utilizador para a criação de programas em linguagens de automação.

No Beremiz é possível criar *data types*, funções, blocos de funções, programas, entre outros, assim como pode ser visto na Figura 2.53 e na Figura 2.54.

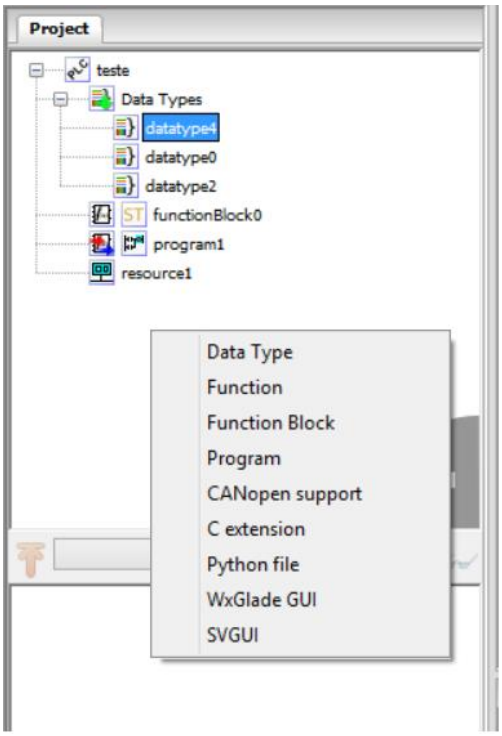


Figura 2.53 — Interface para a criação de POUs

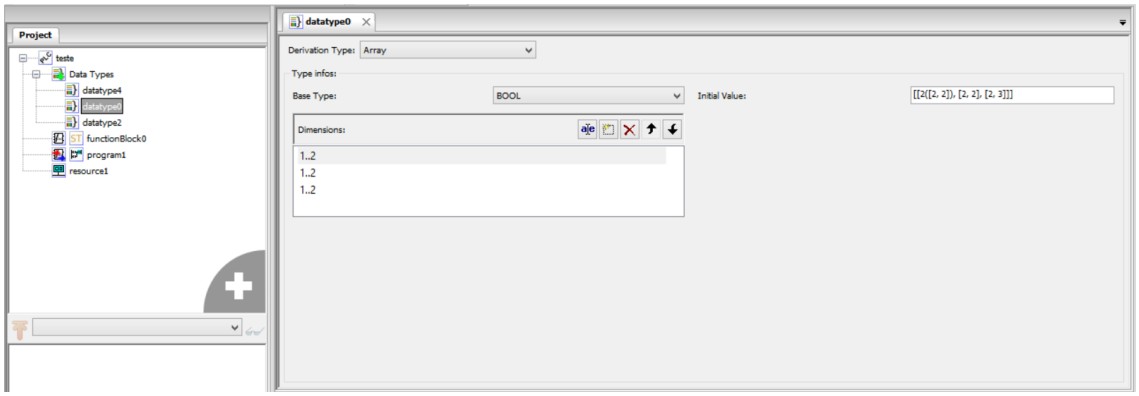


Figura 2.54 — Interface para a criação de *data types*

Capítulo 3

Algoritmos de Conversão

Neste capítulo são apresentadas as propostas de algoritmos utilizados na criação dos dois conversores referidos no Capítulo 1. Assim sendo, primeiramente será apresentado o conceito inicial por detrás dos dois conversores. De seguida, serão escortinados os algoritmos criados e utilizados na criação dos mesmos. Interpoladamente serão abordadas as implementações de cada um dos algoritmos no código fonte dos Projetos.

3.1 - Conceito

As conversões presentes neste Projeto podem ser divididas em dois conversores/ferramentas devido às suas particularidades específicas. Por um lado temos um Conversor XML que tem como entrada e saída ficheiros em formato XML e que estão de acordo com o TC6-XML. Este será o conversor responsável por receber os programas FBD ou LD e converte-los em programas ST. Do outro lado, temos o Conversor TXT responsável por receber os documentos XML (quer sejam programas ST, IL ou SFC) e transformá-los em documentos textuais no formato TXT, representando os programas na sua forma textual de acordo com a Norma IEC 61131-3.

Na Figura 3.1 encontra-se representado o conceito por detrás do Projeto. De notar que o utilizador pode só pretender fazer uma conversão de linguagens e não de formato e que para isso basta parar no fim da primeira fase do Projeto. Se, contudo, pretender obter um documento TXT com a representação textual de um qualquer programa ST, IL ou SFC pode passar à segunda fase do Projeto. Na última fase é exemplificado o caminho a seguir pelos documentos TXT que irão dar entrada no Projeto matiec para que, assim, seja possível gerar o respetivo código em C. De notar que na Figura 3.1 é demonstrado todos os "caminhos" possíveis de acordo com este Projeto, mas o mesmo pode ser executado isoladamente. Assim, os dois

conversores serão projetados de forma independente no desenvolvimento dos seus códigos fonte.

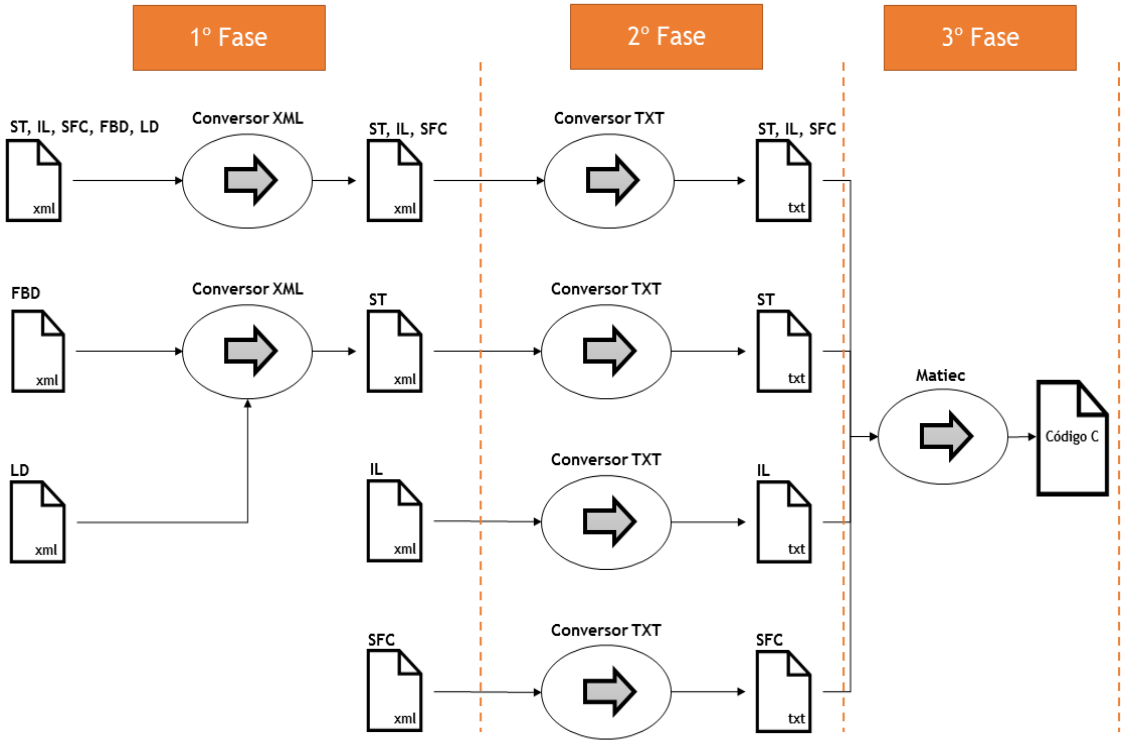


Figura 3.1 — Diagrama representativo do conceito do Projeto

3.2 - Algoritmos do Conversor TXT

No conversor deste tipo pretender-se-á obter um ficheiro de saída com a representação textual, segundo a Norma IEC 61131-3, de um qualquer programa escrito nas linguagens ST, IL e/ou SFC (previamente guardados no esquema TC6-XML). Um exemplo do tipo de texto que terá de ser escrito neste documento pode ser encontrado na Figura 3.2.

```
TYPE
 datatype0 : BOOL;
END_TYPE


PROGRAM program0
 VAR
  E1 : BOOL := False;
  S4 : BOOL := False;
  E3 : BOOL := False;
  E2 : BOOL := False;
  E4 : BOOL := False;
  S1 : BOOL := False;
  functionBlock00 : functionBlock0;
  OR12_OUT : BOOL;
  AND16_OUT : BOOL;
 END_VAR

OR12_OUT := OR(E1, E1);
functionBlock00(LocalVar0 := OR12_OUT, LocalVar1 := E4);
S4 := functionBlock00.LocalVar2;
AND16_OUT := AND(functionBlock00.LocalVar3, E2);
S1 := AND16_OUT;
END_PROGRAM

CONFIGURATION config
 RESOURCE resource1 ON PLC
 VAR_GLOBAL
  LocalVar0 : INT;
  LocalVar1 : INT;
 END_VAR
  TASK cenas(INTERVAL := T#1d0h0m0s0ms, PRIORITY := 0);
  PROGRAM sds WITH cenas: program0;
 END_RESOURCE
END_CONFIGURATION
```

Figura 3.2 — Exemplo do ficheiro TXT de saída do Conversor TXT (linguagem ST)

Através da Figura 3.2, pode-se observar que a representação textual das linguagens SFC, IL ou ST é uma sequência de declarações das variáveis (*data types*), POUs e configurações que podem ser divididas em 3 partes:

1. TYPE…END_TYPE (declaração dos *data types* criados pelo utilizador);
2. PROGRAM…END_PROGRAM e/ou FUNCTION_BLOCK…END_FUNCTION_BLOCK e/ou FUNCTION_BLOCK…END_PROGRAM (declaração dos POUs criados e respetivas variáveis);
3. CONFIGURATION…END_CONFIGURATION (declaração das configurações dos Projeto);

A grande diferença existente entre a conversão dos ficheiros de entrada com programas na linguagem ST e/ou IL e os programas na linguagem SFC é a composição da secção de declaração dos POUs.

As partes reservadas à declaração dos *data types* e das configurações são escritas independentemente do tipo de linguagem, assim sendo é possível escrever as mesmas independentemente do tipo de linguagem de entrada (ST, IL ou SFC). O mesmo já não acontece na parte de declaração dos POUs e das suas variáveis. Nesta parte é necessária a diferenciação do tipo de linguagem de entrada. Na Figura 3.3 encontra-se representado um esquema da estrutura de um POU para melhor compreensão dos conceitos abordados.

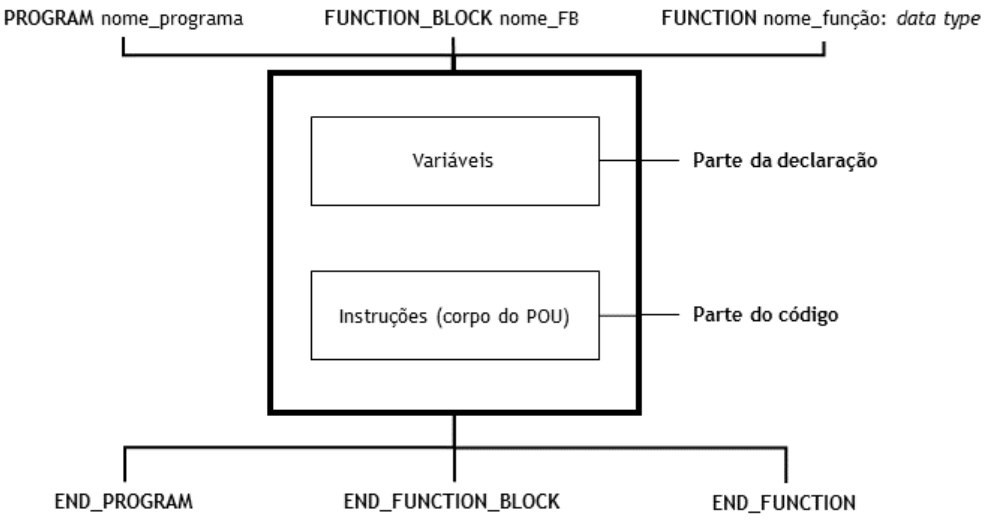


Figura 3.3 — A estrutura comum dos três tipos de POUs

Contudo, existem semelhanças entre o modo como são convertidas as linguagens ST e IL, assim ambas partilham o mesmo algoritmo.

3.2.1 -Programas ST ou IL

O algoritmo proposto para o desenvolvimento desta ferramenta é bastante direto, pois requer apenas a leitura de certos elementos (elemento "ST" ou elemento "IL") do ficheiro de origem (isto para a parte de declaração dos POUs). Esta ferramenta recebe um documento em formato XML que contenha um programa ST ou IL e é então corrido o algoritmo que efetua a criação do documento TXT, a leitura do documento XML e a escrita das respetivas funções no documento TXT. Para uma melhor compreensão do algoritmo, o mesmo encontra-se descrito no Algoritmo 1. De notar que este algoritmo apenas serve para programas cujas linguagens sejam ST ou IL.

Algoritmo 1 Escrita de ficheiro TXT

```
Algoritmo Read&Write_STorIL (ficheiro XML)
  Elements <- ficheiro XML
  Se existir elemento ST ou IL então
    Leia elementos pou
    Enquanto todos os elementos pou não forem lidos faça
      Enquanto todos os elementos datatype não forem lidos faça
        Leia datatype e escreva datatype
      Fim enquanto
      Se pou for do tipo program então
        Escreva "PROGRAM"
      Fim se
      Se pou for do tipo function block então
        Escreva "FUNCTION_BLOCK"
      Fim se
      Se pou for do tipo function então
        Escreva "FUNCTION"
      Fim se
      Enquanto todas as variáveis não forem lidas faça
        Leia Var e escreva Var
      Fim enquanto
      Leia corpo do programa
      Escreva Programa
      Se pou for do tipo program então
        Escreva "END_PROGRAM"
      Fim se
```

Se pou for do tipo function block **então**
Escreva "END_FUNCTION_BLOCK"
Fim se
Se pou for do tipo function **então**
Escreva "END_FUNCTION"
Fim se
Fim enquanto
Enquanto não chegar ao fim do elemento configuration **faça**
Leia elemento configuration e **escreva** configuration
Enquanto todas as variáveis não forem lidas **faça**
Leia Var e escreva Var
Fim enquanto
Fim enquanto
Fim se
Senão
Escreva "Ficheiro não contém um programa válido para conversão."
Fim se
Fim algoritmo

Este algoritmo começa por verificar a existência de elementos ST ou IL, ou seja, procura a presença de POUs nestas linguagens que estejam presentes no documento XML de entrada.

Uma vez verificada a existência de um desses elementos é então iniciada a leitura dos *data types* criados pelo utilizador. Assim pretende-se gerar a parte do documento TXT presente na Figura 3.4.

```
TYPE
 datatype0 : BOOL;
END_TYPE
```

Figura 3.4 – Secção da parte TYPE num documento TXT

Assim que forem escritos todos os *data types* criados pelo utilizador passa-se à seguinte fase que pretende construir a segunda parte de um documento TXT.

Nesta fase é criada a secção representada na Figura 3.5.

```
PROGRAM program0
 VAR
  E1 : BOOL := False;
  S1 : BOOL := False;
 END_VAR

  S1 := E1;
END_PROGRAM
```

Figura 3.5 — Secção da declaração de um POU (programa) num documento TXT

A forma como é gerada é bastante simples, passando por percorrer os elementos XML que contém as informações para a criação das variáveis e escrevendo-as no documento TXT (tendo sempre em atenção o tipo de variável). De seguida é realizada a leitura das instruções escritas para o programa ST ou IL e estas são escritas diretamente no documento final. Na terceira e última parte (Figura 3.6) é criada a configuração de cada Projeto. Nesta parte é necessário ler os elementos da fonte (RESOURCE), das variáveis globais (VAR_GLOBAL) e tarefas (TASK).

```
CONFIGURATION config
 RESOURCE resource1 ON PLC
 VAR_GLOBAL
  LocalVar0 : INT;
 END_VAR
  TASK task1(INTERVAL := T#1d0h0m0s0ms, PRIORITY := 0);
  PROGRAM sds WITH task1: program0;
 END_RESOURCE
END_CONFIGURATION
```

Figura 3.6 — Secção da parte CONFIGURATION num documento TXT

3.2.2 -Programas SFC

O princípio de funcionamento presente na conversão deste tipo de programas é bastante parecido com o funcionamento da conversão mencionado no Subcapítulo 3.2.1. Isto deve-se ao facto de que para a construção dos *data types* e configurações os algoritmos utilizados são iguais.

A diferença surge no corpo do programa, bloco de função e/ou função (POUs) onde estão descritas as funções. Assim, onde anteriormente se encontrava, na secção CDATA, as funções dos POUs em IL e/ou ST, sendo imediata a sua escrita, agora passa a ser inexistente, estando o corpo do programa (SFC) formatado no estilo TC6 – XML como abordado em 2.8.4

Assim sendo, a forma como se escreve o programa no ficheiro texto é diferente da anterior seguindo as regras presentes na Figura 3.7.

```
PRODUCTION RULES:

sequential_function_chart ::= sfc_network {sfc_network}

sfc_network ::= initial_step {step | transition | action}

initial_step ::= 'INITIAL_STEP' step_name ':' {action_association ';'}
'END_STEP'

step ::= 'STEP' step_name ':' {action_association ';'} 'END_STEP'

step_name ::= identifier

action_association ::= action_name '(' [action_qualifier] {',' indicator_name}
')'

action_name ::= identifier

action_qualifier ::= 'N' | 'R' | 'S' | 'P' | timed_qualifier ',' action_time

timed_qualifier ::= 'L' | 'D' | 'SD' | 'DS' | 'SL'

action_time ::= duration | variable_name

indicator_name ::= variable_name

transition ::= 'TRANSITION' [transition_name] ['(' 'PRIORITY' ':=' integer
')']
                'FROM' steps 'TO' steps
                 transition_condition
                'END_TRANSITION'

transition_name ::= identifier

steps ::= step_name | '(' step_name ',' step_name {',' step_name} ')'

transition_condition ::= ':' simple_instruction_list | ':=' expression ';' |
':' (fbd_network | rung)

action ::= 'ACTION' action_name ':' function_block_body 'END_ACTION'
```

Figura 3.7 — Regras para a construção de um programa SFC na sua forma textual [6]

Através destas regras é possível escrever a linguagem SFC num documento texto devidamente estruturado. As regras de interpretação da mesma pode ser vista na Figura 3.8.

```
A.1.3 Production rules
The production rules for textual programmable controller programming languages shall form an extended grammar in which each rule has the form

non_terminal_symbol ::= extended_structure

This rule can be read as:
"A non_terminal_symbol can consist of an extended_structure."

Extended structures can be constructed according to the following rules:
1) The null string, NIL, is an extended structure.
2) A terminal symbol is an extended structure.
3) A non-terminal symbol is an extended structure.
4) If S is an extended structure, then the following expressions are also extended structures:
      (S), meaning S itself.
      {S}, closure, meaning zero or more concatenations of S.
      [S], option, meaning zero or one occurrence of S.
5) If S1 and S2 are extended structures, then the following expressions are extended structures:
         S1 | S2, alternation, meaning a choice of S1 or S2.
         S1 S2, concatenation, meaning S1 followed by S2.
6) Concatenation precedes alternation, that is, S1 | S2 S3 is equivalent to S1
| (S2 S3), and S1 S2 | S3 is equivalent to (S1 S2) | S3.
```

Figura 3.8 — Regras para a interpretação da Figura 3.7 [6]

Através da Figura 3.7 pode-se afirmar que existem 3 elementos cruciais na escrita do documento TXT e são eles as etapas, as ações e as transições. Assim pode-se observar na Figura 3.10, Figura 3.11 e na Figura 3.9 o tipo de texto que é esperado ter no documento final para cada um dos elementos supracitados. De notar também que as transições e ações podem ser escritas nas linguagens ST e/ou IL, bastando para isso, utilizar os mesmos métodos utilizados no Subcapítulo 3.2.1 .

```
TRANSITION FROM Step0 TO Step1
  :=  true;
END_TRANSITION
```

Figura 3.9 — Exemplo de uma representação textual de uma transição num programa SFC

```
STEP Step1 :
  action0(N);
  Action_0_INLINE(N);
END_STEP
```

Figura 3.10 — Exemplo de uma representação textual de uma etapa num programa SFC

```
ACTION action0:
 xx := 0;
END_ACTION
```

Figura 3.11 — Exemplo de uma representação textual de uma ação num programa SFC

3.3 - Implementação do Conversor TXT

A forma como foi implementado este conversor relaciona-se com as 3 partes, mencionadas no Subcapítulo 3.1, que um documento texto desta natureza tem. Assim sendo, existem 3 classes crucias no programa, responsáveis pela estruturação/escrita do documento texto.

Essas classes tratam, cada uma delas, de escrever os *data types* criados pelo utilizador (classe "Datatype_constr"), os programas, funções e/ou blocos de funções que possam existir (classe "Program_constr") e também as configurações presentes para o Projeto (classe "Configuration_constr"). Para além destas classes existem classes criadas para a filtragem do tipo de POU e da linguagem (ST, IL e/ou SFC) empregue (classe "Counting_pou") e criadas para a escrita do tipo de variáveis (classes "Print_type" ou "Print_array_class").

As relações destas classes e das demais constituintes do programa desenvolvido podem ser analisadas nos anexos através do respetivo diagrama de classes.

3.4 - Algoritmos do Conversor XML

O conversor de ficheiros XML apresenta bastantes diferenças entre o conversor descrito no Subcapítulo 3.2. Essas diferenças começam pelos ficheiros de entrada que, desta vez, são documentos XML contendo programas LD e FBD. O documento de saída será igualmente diferente sendo que, neste caso, é necessário criar um documento XML onde os POUs que outrora estivessem nas linguagens LD ou FBD agora se encontrem convertidas para a linguagem ST. Ou seja, todo o documento XML de entrada vai ser replicado no documento XML de saída, mas nas partes em que estejam presentes POUs na linguagem FBD e/ou LD sofrerão a conversão para a linguagem ST com as respetivas funções equivalentes.

3.4.1 -Programas FBD

De modo a ser possível realizar a conversão de um programa FBD para um programa ST foram criados os algoritmos apresentados neste subcapítulo.

A estratégia de desenvolvimento dos algoritmos tem como base alguns aspetos que serão agora mencionados através da análise do programa FBD presente na Figura 3.12.

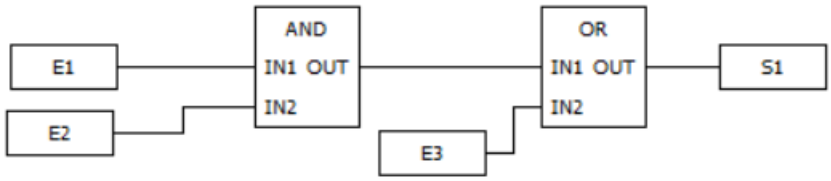


Figura 3.12 — Programa FBD de exemplo para a estratégia do algoritmo

Este programa FBD tem o código ST correspondente presente na Figura 3.13.

```
S1 := E3 OR (E1 AND E2);
```

Figura 3.13 — Programa ST de exemplo da Figura 3.12

Tendo em atenção ao código aqui representado, o programa FBD da Figura 3.12 pode ser escrito de uma outra forma e utilizando as regras mencionadas no Subcapítulo 2.2. Assim sendo, pode-se converter este mesmo programa invocando instâncias dos FBs presentes e passando as variáveis de entrada como parâmetros. Assim, pode-se chegar a um código deste tipo:

```
AND1_OUT := AND(E1, E2);
OR2_OUT := OR(AND1_OUT, E3);
S1 := OR2_OUT;
```

Figura 3.14 — Programa ST alternativo do exemplo da Figura 3.12

Os códigos ST presentes na Figura 3.13 e na Figura 3.14 produzem o mesmo resultado que o programa presente na Figura 3.12. Contudo, os algoritmos aqui presentes para a conversão de programas FBD irão reproduzir o resultado presente na Figura 3.14. Para isto, é então necessário ler as funções empregues em cada FB ou função e criar igualmente uma variável (local) representativa da saída de um bloco (por exemplo a variável "AND1_OUT" representa a saída do bloco "AND" presente no programa da Figura 3.12).

Os algoritmos desenvolvidos têm como base a leitura de elementos/atributos do documento XML e a sua correspondência na linguagem ST. Assim, estes algoritmos apresentam algumas particularidades bastante interessantes para que a conversação seja realizada de modo satisfatório e correto. Em primeiro lugar será necessário processar o documento XML de entrada que contém o programa FBD. Nesta primeira fase é verificada a existência ou não de um documento compatível, ou seja, que contenha um programa, função ou bloco de funções (POUs) na linguagem FBD e é então criada uma cópia do mesmo. Esta cópia do ficheiro de entrada será o ficheiro de destino e de escrita da linguagem ST.

Em seguida, é necessário ler todos os blocos presentes nos programas, funções ou blocos de funções e escrever no documento XML de destino as respetivas variáveis representativas das saídas de cada bloco, como pode ser visto no Algoritmo 2.

De notar que este algoritmo se aplica apenas a blocos não criados pelo utilizador, visto que o próprio programa Beremiz, ao compilar o programa em FBD, cria uma variável própria de representação da saída dos blocos com o tipo adequado, não sendo assim necessária a criação dessa mesma variável.

Algoritmo 2 Criação da variável representativa da saída de cada bloco

```
Algoritmo variable_constr (lista_de_blocos, lista_de_inputs, lista_de_outputs)
Enquanto não chegar ao fim da lista_de_blocos faça
  Element block <- lista_de_blocos
  Se block for criado pelo utilizador e for FB então
  Fim se
  Senão
    Escreva nome_do_block nas variáveis locais
    Enquanto não chegar ao fim da lista_de_inputs faça
      Element input <- lista_de_inputs
      Se input tiver ligado a block então
        Escreva tipo_bloco
      Fim se
    Fim Enquanto
    Enquanto não chegar ao fim da lista_de_outputs e tipo não escrito faça
      Element output <- lista_de_outputs
      Se output tiver ligado a block então
        Escreva tipo_bloco
      Fim se
    Fim Enquanto
    Enquanto não chegar ao fim da lista_de_blocks e tipo não escrito faça
      Element block_compare <- lista_de_blocks
      Se block_compare tiver ligado a block então
        Escreva tipo_bloco
      Fim se
    Fim Enquanto
Fim Enquanto
Fim algoritmo
```

No Algoritmo 2 é necessária a escrita da variável de saída de um qualquer bloco presente no programa em FBD através da instrução ("Escreva nome_do_block nas variáveis locais"). Esta escrita é feita da seguinte forma:

- Escrita do nome do bloco (por exemplo "OR");
- Escrita do ID do bloco em análise (por exemplo "14");
- Escrita da saída do bloco (por exemplo "_OUT");

Tendo em consideração, como exemplo, a presença de um bloco cuja função seja a função "OR", a escrita da sua variável de saída será então descrita pelos 3 passos supracitados, dando origem à variável "OR14_OUT". Estas variáveis criadas pelo conversor são variáveis locais e devidamente guardadas como tal.

O passo seguinte é a criação do programa ST, ou seja, a conversão das funções no programa FBD para as declarações (*statement*) da linguagem ST. Assim, para este efeito, aplica-se o Algoritmo 4. Este algoritmo percorre todos os blocos, através de uma ordem que pode ser

definida pelo utilizador ou não (Algoritmo 3), procurando os *inputs*, *inOutputs* e até mesmo outros blocos que estejam ligados como entrada (blocos que são representados pela sua variável de saída) no bloco em análise.

Por fim, quando todas as entradas do bloco em análise forem lidas e devidamente escritas no documento, passa-se a procurar os *outputs* e *inOutputs* que estejam ligados às saídas do bloco, escrevendo assim o código correspondente de saída na linguagem ST.

Algoritmo 3 Seriação dos blocks

```
Algoritmo sort_block (lista_de_blocos)
  executionOrderId: inteiro
  lista_de_blocos_organizada: ArrayList <inteiro>
  lista_de_blocos_organizada <- 1ºblock
  Enquanto não chegar ao fim da lista_de_blocos faça
    Se lista_de_blocos tiver execution_order_ID faça
      Element block <- lista_de_blocos
      executionOrderId <- block.getAttribute(executionOrderId)
      Enquanto não chegar ao fim da lista_de_blocos_organizada faça
        executionOrderId_compare <- block.getAttribute(executionOrderId)
        Se o executionOrderId do block < executionOrderId_compare então
          Colocar executionOrderId na posição atrás de executionOrderId_compare
        Senão
          Colocar executionOrderId na posição à frente de executionOrderId_compare
        Fim se
      Fim Enquanto
    Senão
      Lista_de_blocos_organizada <- lista_de_blocos
    Fim se
  Fim Enquanto
Fim algoritmo
```

Algoritmo 4 Escrita dos elementos (FBD) para o programa ST

```
Algoritmo function_constr (lista_de_blocos_organizada, lista_de_inputs, lista_de_outputs,
list_de_InOutputs)
variable, entradas: inteiro
Enquanto não chegar ao fim da lista_de_blocos_organizada faça
  Element block <- lista_de_blocos_organizada
  Ler entradas_block
  entradas <- entradas_block
  Escreva nome_do_block
  Enquanto não chegar ao fim da lista_de_inputs faça
    Element input <- lista_de_inputs
    Se input tiver ligado a block então
      Escreva input
      Faça variable= variable+1
    Fim se
  Fim Enquanto
  Enquanto não chegar ao fim da lista_de_blocks faça
    Element block_compare <- lista_de_blocks
    Se block_compare tiver ligado a block então
      Escreva tipo_bloco
      Faça variable= variable+1
    Fim se
  Fim Enquanto
  Enquanto não chegar ao fim da lista_de_ InOutputs faça
    Element InOutputs <- lista_de_ InOutputs
    Se InOutputs tiver ligado a block então
      Escreva InOutputs
      Faça variable= variable+1
    Fim se
  Fim Enquanto
  Se variable == entradas então
    Enquanto não chegar ao fim da lista_de_outputs faça
      Element output <- lista_de_outputs
      Se output tiver ligado a block então
        Escreva output
      Fim se
    Fim Enquanto
    Enquanto não chegar ao fim da lista_de_ InOutputs faça
      Element InOutputs <- lista_de_ InOutputs
      Se InOutputs tiver ligado a block então
        Escreva InOutputs
      Fim se
    Fim Enquanto
  Fim se
      Fim Enquanto
```

Fim algoritmo

Para uma melhor compreensão destes algoritmos encontra-se na Figura 3.15 uma sequência de imagens representativas do funcionamento deste conversor.

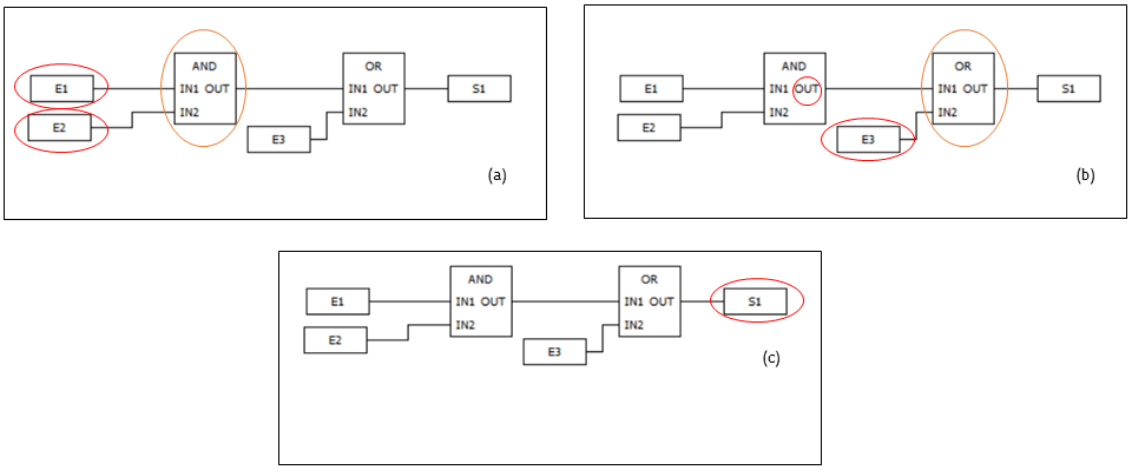


Figura 3.15 — Sequência de leitura/escrita para a conversão de um programa em FBD

(a) Nesta fase o conversor encontra o primeiro bloco (neste caso não se encontra definida a sequência de leitura pelo utilizador, logo o conversor lê o primeiro bloco criado e assim sucessivamente) e escreve a sua variável de representação no corpo do programa ST. Para além disto, encontra as variáveis que estão ligadas aos pontos de entrada do bloco e escreve-as na declaração ST juntamente com a correspondente função (através da leitura do nome do bloco). Resultando na seguinte linha: "AND1_OUT := AND(E1, E2);";

(b) Assim que for concluída a leitura de um bloco, passa-se ao próximo bloco criado. Neste caso as variáveis de entrada são "E3" e a variável representativa da saída do bloco "AND". Assim, temos a seguinte linha: "OR2_OUT := OR(AND1_OUT, E3);";

(c) Como o bloco "OR" tem uma variável de saída associada é necessário criar a atribuição que a mesma tem no respetivo código ST. Resultando na seguinte linha: "S1 := OR2_OUT;";

```
AND1_OUT := AND(E1, E2);
OR2_OUT := OR(AND1_OUT, E3);
S1 := OR2_OUT;
```

Figura 3.16 — Excerto do programa ST do exemplo da Figura 3.15

3.4.2 -Programas LD

O algoritmo desenvolvido para este tipo de linguagem de entrada tem como base, igualmente como no caso anterior, a leitura de elementos/atributos do documento XML e a sua correspondência em código ST. Apesar de seguir o mesmo fundamento, trata-se de uma ferramenta completamente diferente, tornando o algoritmo de conversão igualmente diferente. Na Figura 3.17 encontra-se um exemplo de um simples programa em LD e na Figura 3.18 o respetivo código, em ST, que se pretende alcançar com este conversor.

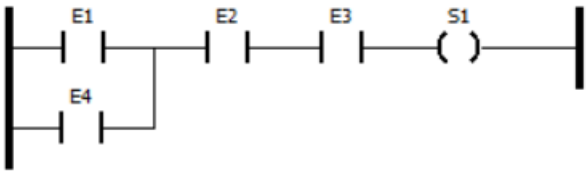


Figura 3.17 — Programa LD de exemplo para a estratégia do algoritmo

```
S1 := E3 AND E2 AND (E1 OR E4);
```

Figura 3.18 — Programa ST de exemplo da Figura 3.17

Nesta ferramenta já não será necessário ter em consideração a ordem dos FBs, visto que a mesma é definida pela sua posição na grelha de contatos (*contacts*) e de bobinas (*coils*). Um outro aspeto a ter em consideração é o facto de previamente ser selecionado o método adequado consoante o programa LD tenha ou não elementos também presentes em programas FBD (Algoritmo 5-esta distinção é feita através da presença ou não de blocos (FB ou funções no programa LD)).

Algoritmo 5 Verificação da presença de elementos FB

```
Algoritmo verify_FB_elements (pou LD)
  elemento: Element
  LDwFB: Booleano
  LDwFB <- false
  Enquanto não ler todos os elementos de LD faça
    Elemento <- elementos_lidos
    Se Elemento for "block" então
      LDwFB <- true
      Executa function_LD
    Fim se
  Fim Enquanto
  Se LDwFB = false então
    Executa function_LD_only
  Fim se
Fim algoritmo
```

Depois de verificado se o programa, a converter, tem ou não elementos FBs é então executado o Algoritmo 6. Este consiste na leitura do programa LD da direita para a esquerda, percorrendo essencialmente, as bobinas e verificando que contactos estão ligados às mesmas. Ora, assim encontra-se a cadeia de todos os contatos ligados entre si e que em última instância estão conectados com a linha de alimentação do lado esquerdo (*left power rail*).

Assim, que for detetado, que um contacto está ligado à linha de alimentação do lado esquerdo passa-se a correr o mesmo algoritmo, mas desta feita, para os contactos que continham mais do que uma ligação de entrada, permitindo assim percorrer o resto das ligações. Ou seja, enquanto são lidos os contactos ligados à bobina, são guardados os contactos que têm mais que uma conexão para que, assim que termine a leitura dos mesmos, se volte atrás (fazendo agora uma leitura da esquerda para a direita) e se aplique o Algoritmo 6 (voltando a uma leitura da direita para a esquerda), mas desta vez a estes contactos com mais que uma ligação e cujos contactos que estão ligados ao mesmo ainda não foram escritos, pois encontram-se noutra linha de ligação.

Algoritmo 6 Escrita do programa LD para ST

```
Algoritmo function_LD_only (lista_de_coils, lista_de_contacts, leftPowerRail)
  lista_de_contactos_multi: ArrayList <Element>
  Enquanto não chegar ao fim da lista_de_coils faça
    Element coil <- lista_de_coils
    Escrever coil
    Enquanto não chegar ao fim da lista_de_contacts faça
      Element contact <- lista_de_contacts
      Se contact estiver ligado a coil então
        Escreva contact
        Se contact tiver mais que uma conexão então
          Adiciona contact a lista_de_contactos_multi
          Executa leitura_de_contacts
        Fim se
      Fim se
      Se contact estiver ligado a leftPowerRail então
        Escreva fim da função
      Fim se
    Fim Enquanto
  Fim Enquanto
Fim algoritmo
```

De notar que no Algoritmo 6 é executada a função leitura_de_contacts, que tem como função realizar um processo em cadeia em que são escritos todos os contactos que estão ligados ao contacto a analisar até ser encontrado um contacto que esteja ligado à linha de alimentação do lado esquerdo. Esta mesma função analisa a necessidade de ser escrito "AND", "AND(" ou "OR" através da presença de mais que uma conexão num mesmo contacto.

Aquando da presença de programas, funções ou blocos de funções com elementos FB o algoritmo a ser executado será bastante parecido com o Algoritmo 4 e o Algoritmo 6, mas com alguns aspetos diferenciativos. Em primeiro lugar, no Algoritmo 4, adiciona-se uma rotina de leitura de contactos e não só de blocos, *inputs* e *inOutputs*, isto porque passa-se a ter FBs cujas entradas podem ser contactos.

Um segundo aspeto é o facto de que no Algoritmo 6 a função irá cessar assim que se encontrar uma linha de alimentação do lado esquerdo, mas também quando um bloco for encontrado ligado a um contacto.

Para uma melhor compreensão dos assuntos aqui abordados, a Figura 3.19 representa a sequência de leituras realizadas pelos algoritmos mencionados através de um simples programa em LD.

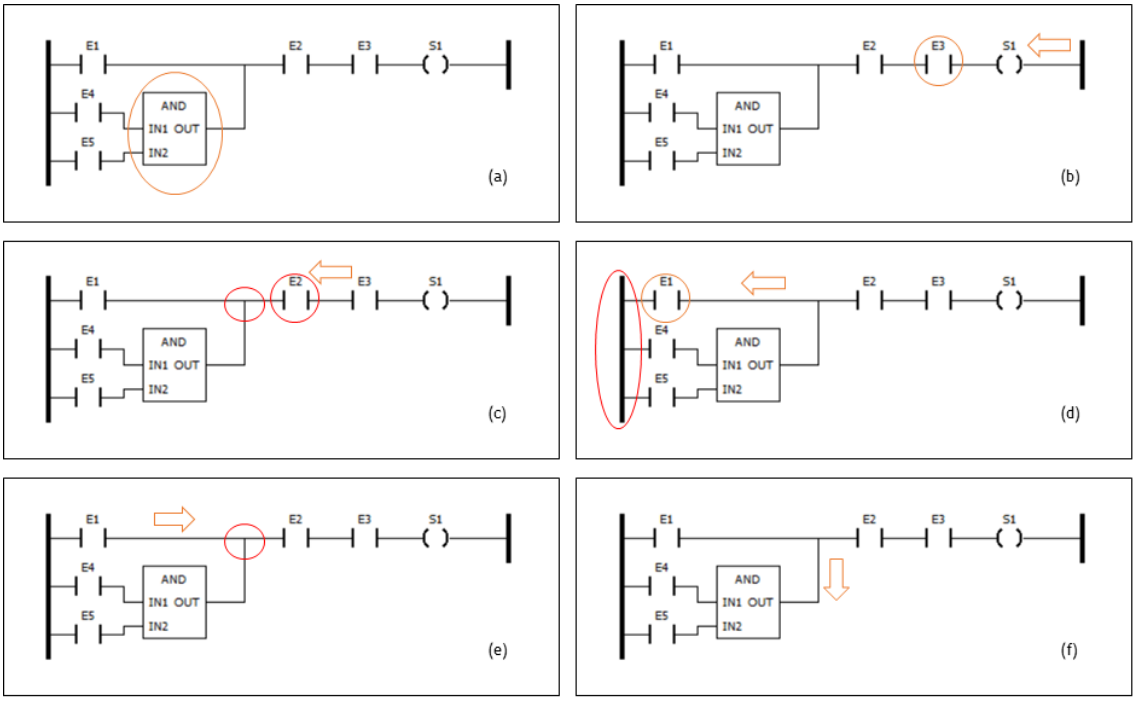


Figura 3.19 — Sequência de leitura/escrita para a conversão de um programa em FBD

(a) Nesta fase o conversor encontra o primeiro bloco (neste caso o conversor lê o primeiro bloco criado e assim sucessivamente) e escreve a sua variável de representação no corpo do programa ST. Para além disto, encontra as variáveis que estão ligadas aos pontos de entrada do bloco e escreve-as na declaração ST

juntamente com a correspondente função (através da leitura do nome do bloco). Resultando na seguinte linha: "AND8_OUT := AND(E4, E5);". Esta fase baseia-se nos algoritmos expressos no Subcapítulo 3.4.1

(b) Assim que for concluída a leitura do bloco, passa-se à leitura da bobina de saída (*coil*). Nesta leitura é escrita a variável associada à bobina e o contacto que se encontra ligado à mesma (como neste caso o contacto "E3" só tem uma ligação de entrada só se escreve a instrução "AND"). Resultando em: "S1 := E3 AND";

(c) Agora é necessário escrever a variável associada ao contacto "E3", que neste caso, é a variável "E2". Este contacto apresenta uma ligeira diferença, sendo que a mesma reside no facto deste contacto ter mais que uma ligação de entrada (duas neste exemplo). Este fator irá provocar uma diferença na escrita da instrução em ST, passando agora a se escrever "AND (" e não "AND". Ficando com a expressão desta forma: "S1 := E3 AND E2 AND (";

(d) Como ainda não foi encontrado nenhum elemento de paragem de leitura dos contactos (linha de alimentação do lado esquerdo ou um bloco) continua-se a leitura dos mesmos. Assim procura-se o elemento ligado à primeira ligação de entrada do contacto "E2". Facilmente se verifica que é o contacto, cuja variável associada é "E1". De notar também que "E1" se encontra diretamente ligado à linha de alimentação do lado esquerdo (*left power rail*) e como tal pode ser interrompida a leitura desta linha de ligação. Resultando em: "S1 := E3 AND E2 AND (E1";

(e) Visto que já se terminou de ler a primeira linha, falta agora, ler a segunda linha de entrada no contacto "E2". Ora esta divergência de ligações encontra-se associada à função "OR". Assim temos: "S1 := E3 AND E2 AND (E1 OR";

(f) Nesta fase é então passada à leitura dos elementos da segunda linha. Neste exemplo, é encontrada a presença de um bloco que faz com que a leitura de elementos presentes num programa LD seja cessada e escrita a variável representativa da saída do bloco. Resultando na declaração final:
"S1 := E3 AND E2 AND (E1 OR AND8_OUT);";

Depois destas sequências de leitura temos o seguinte código ST (Figura 3.20) representativo do programa LD exposto na Figura 3.19.

```
AND8_OUT := AND(E4, E5);
S1 := E3 AND E2 AND (E1 OR AND8_OUT);
```

Figura 3.20 — Excerto do programa ST do exemplo da Figura 3.19

3.5 - Implementação do Conversor XML

A implementação deste conversor teve como alicerce a diferenciação entre os programas na linguagem LD e na linguagem FBD.

Como referido no Subcapítulo 3.4, a ideia principal nesta conversão é a criação de variáveis representativas de cada saída presente em cada bloco (FBs) existente nos programas, funções ou blocos de funções contidos no documento XML (classe "Variable_constr").

Após essa criação são então executados todos os métodos responsáveis pela linguagem FBD e/ou LD, sendo ambas consequência da análise do tipo de linguagem embutida em cada POU presente no documento (classe "Counting_pou").

Caso seja um POU (função, bloco de funções ou programa) escrito na linguagem FBD é necessária a análise à ordem de execução dos blocos de funções (descrita ou não pelo utilizador) e de seguida a escrita da correspondente função na linguagem ST. Esta função é escrita através da análise do nome do bloco em análise. Ou seja, o Conversor XML lê o nome do bloco (atributo "*typeName*") e será este o nome da função a ser escrita em ST no documento de saída (classe "Function_constr").

Se o POU a analisar estiver na linguagem LD, o conversor faz uma distinção entre programas LD com elementos FBD, ou programas LD sem elementos FBD. Esta distinção é realizada através da presença ou ausência de blocos (classe "Chooser").

Assim sendo, é aplicado um algoritmo análogo ao algoritmo de conversão da linguagem FBD, isto caso o programa LD contenha elementos FBD. De seguida são analisadas as entradas e saídas (*contacts* e *coils*) para a escrita das mesmas. Para isso, são executados os algoritmos referidos nos Subcapítulos 3.4.1 e 3.4.2 (classe "Function_constr_LD").

O conversor encontra-se completamente apto a analisar contactos, bobinas, entradas e saídas que contenham qualquer tipo de modificador. Ou seja, é completamente compatível com variáveis normais, negadas, RE, FE, Set e Reset.

As relações destas classes e das demais constituintes do programa desenvolvido podem ser analisadas nos anexos através do respetivo diagrama de classes.

Capítulo 4

Resultados

Neste capítulo são apresentados os resultados dos algoritmos propostos no Capítulo 3 e aplicados a 5 conjuntos de documentos. Este capítulo encontra-se dividido em 3 partes, a primeira parte será referente ao teste do Conversor TXT, segunda referente ao teste do Conversor XML e a terceira parte será referente a alguns casos especiais de teste.

Para apresentação do Conversor TXT irão ser apresentados os resultados referente a três documentos XML cujas linguagens de entrada serão o ST, IL e SFC.

Quanto ao Conversor XML, irão ser apresentados os resultados para dois documentos XML nas linguagens FBD e LD.

4.1 - Conversor TXT

Neste subcapítulo serão apresentados alguns dos programas utilizados para o teste do Conversor TXT e os resultados apresentados pelo mesmo.

As partes do documento texto que são independentes do tipo de linguagem do programa (a declaração dos *data types* e das configurações) serão apresentadas de seguida, sendo que a parte dependente da linguagem em que está escrita (declaração dos POUs) será abordada nos subcapítulos subsequentes.

Na Figura 4.1 pode ser observado um excerto do documento XML referente aos *data types* criados pelo utilizador. Assim, na Figura 4.2, encontram-se esses mesmos *data types* escritos no documento texto de saída através da sua representação textual de acordo com a Norma IEC 61131-3.

```
<types>
  <dataTypes>
    <dataType name="datatype4">
      <baseType>
        <BOOL />
      </baseType>
      <initialValue>
        <simpleValue value="TRUE" />
      </initialValue>
    </dataType>
    <dataType name="datatype2">
      <baseType>
        <array>
          <dimension lower="1" upper="2" />
          <baseType>
            <BOOL />
          </baseType>
        </array>
      </baseType>
      <initialValue>
        <arrayValue>
          <value>
            <simpleValue value="1" />
          </value>
          <value>
            <simpleValue value="3" />
          </value>
        </arrayValue>
      </initialValue>
    </dataType>
  </dataTypes>
```

Figura 4.1 — Excerto do documento XML (entrada) da declaração de *data types*

```
TYPE
 datatype4 : BOOL := TRUE;
 datatype2 : ARRAY [1..2] OF BOOL := [1, 3];
END_TYPE
```

Figura 4.2 — Excerto do documento texto (saída) da declaração de *data types*

De forma análoga encontra-se na Figura 4.3 um excerto do documento XML contendo a parte da configuração do Projeto e na Figura 4.4 encontra-se representada essa mesma parte mas no documento texto.

```
<instances>
  <configurations>
    <configuration name="config">
      <resource name="resource1">
        <task name="tsk" priority="0" interval="T#1d0h0m0s0ms">
          <pouInstance name="sds" typeName="program0"/>
        </task>
        <globalVars>
          <variable name="LocalVar0">
            <type>
              <INT/>
            </type>
          </variable>
          <variable name="LocalVar1">
            <type>
              <derived name="datatype0"/>
            </type>
          </variable>
        </globalVars>
      </resource>
    </configuration>
  </configurations>
</instances>
```

Figura 4.3 — Excerto do documento XML (entrada) da declaração da configuração

```
CONFIGURATION config
  RESOURCE resource1 ON PLC
    VAR_GLOBAL
      LocalVar0 : INT;
      LocalVar1 : datatype0;
    END_VAR
    TASK tsk(INTERVAL := T#1d0h0m0s0ms,PRIORITY := 0);
    PROGRAM sds WITH tsk : program0;
  END_RESOURCE
END_CONFIGURATION
```

Figura 4.4 — Excerto do documento TXT (saída) da declaração da configuração

4.1.1 -Programa ST

Na Figura 4.5 encontra-se representado o programa ST, que se encontra em formato XML (TC6-XML), que será sujeito à conversão por parte do Conversor TXT. Assim sendo, na Figura 4.6 pode ser observado um excerto da representação textual desse mesmo programa gerado pelo conversor (não inclui a declaração dos *data types* e das configurações).

```
        <body>
          <ST>
            <xhtml:p><![CDATA[functionBlock00(LocalVar0 := OR12_OUT, LocalVar1 := E4);
S4 := functionBlock00.LocalVar2;
AND16_OUT := AND(functionBlock00.LocalVar3, E2);
S1 := AND16_OUT;
OR12_OUT := OR(E1, E1 OR E4);
S2 := OR12_OUT;]]></xhtml:p>
          </ST>
        </body>
```

Figura 4.5 — Excerto do documento XML (entrada) do programa ST

```
PROGRAM program0
 VAR
  E1 : BOOL := False;
  S2 : BOOL := False;
  S4 : BOOL := False;
  E3 : BOOL := False;
  E2 : BOOL := False;
  E4 : BOOL := False;
  S1 : BOOL := False;
  functionBlock00 : functionBlock0;
  OR12_OUT : BOOL;
  AND16_OUT : BOOL;
 END_VAR

  functionBlock00(LocalVar0 := OR12_OUT, LocalVar1 := E4);
  S4 := functionBlock00.LocalVar2;
  AND16_OUT := AND(functionBlock00.LocalVar3, E2);
  S1 := AND16_OUT;
  OR12_OUT := OR(E1, E1 OR E4);
  S2 := OR12_OUT;
END_PROGRAM
```

Figura 4.6 — Excerto do documento TXT (saída) do programa ST

4.1.2 -Programa IL

Na Figura 4.7 encontra-se representado o programa IL, em formato XML (TC6-XML), que será sujeito à conversão por parte do Conversor TXT. Assim sendo, na Figura 4.8 pode ser observado um excerto do documento TXT gerado pelo conversor (não inclui a declaração dos *data types* e das configurações).

```
      <body>
        <IL>
          <xhtml:p><![CDATA[ LD b1
AND b2
ANDN b3
ST b0]]></xhtml:p>
```

Figura 4.7 — Excerto do documento XML (entrada) na linguagem IL

```
PROGRAM program1
 VAR
  b1 : BOOL;
  b2 : BOOL;
  b3 : BOOL;
  b0 : BOOL;
 END_VAR

LD b1
AND b2
ANDN b3
ST b0
END_PROGRAM
```

Figura 4.8 — Excerto do documento TXT (saída) do programa IL

4.1.3 -Programa SFC

Porque o SFC é uma linguagem gráfica, será apresentado, na Figura 4.9, o programa correspondente acompanhado de um excerto do documento XML (TC6-XML) do mesmo (Figura 4.10). Na Figura 4.11 pode ser observado um excerto do documento TXT gerado pelo conversor (não inclui a declaração dos *data types* e das configurações).

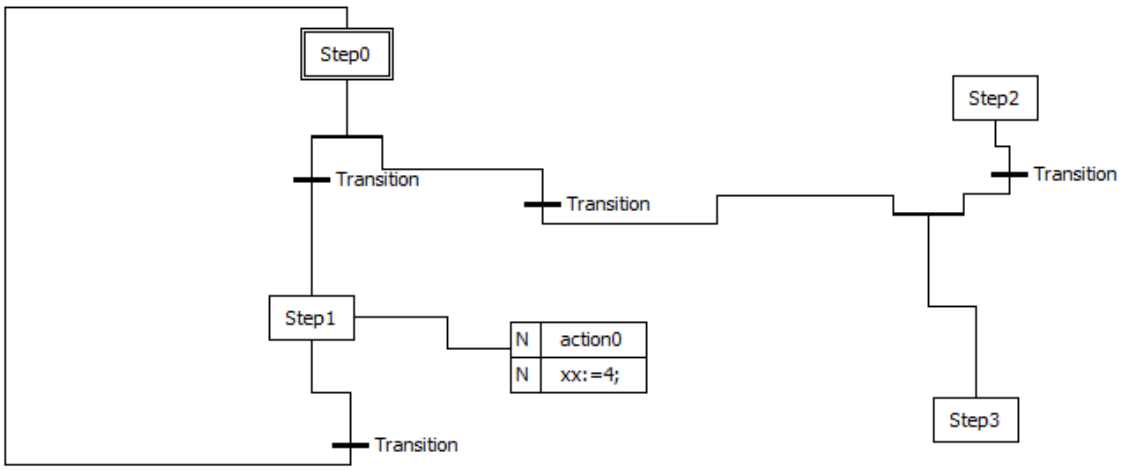


Figura 4.9 — Programa SFC a ser introduzido no Conversor TXT

```
<body>
  <SFC>
    <step localId="1" name="Step0" initialStep="true" height="29" width="52">
      <position x="497" y="24"/>
      <connectionPointIn>
        <relPosition x="26" y="0"/>
        <connection refLocalId="4">
          <position x="523" y="24"/>
          <position x="523" y="12"/>
          <position x="329" y="12"/>
          <position x="329" y="272"/>
          <position x="525" y="272"/>
          <position x="525" y="262"/>
        </connection>
      </connectionPointIn>
      <connectionPointOut formalParameter="">
        <relPosition x="26" y="29"/>
      </connectionPointOut>
    </step>
    <transition localId="2" height="2" width="20">
      <position x="493" y="109"/>
      <connectionPointIn>
        <relPosition x="10" y="0"/>
        <connection refLocalId="7">
          <position x="503" y="109"/>
          <position x="503" y="86"/>
        </connection>
      </connectionPointIn>
      <connectionPointOut>
        <relPosition x="10" y="2"/>
      </connectionPointOut>
      <condition>
        <inline name="">
          <ST>
            <xhtml:p><![CDATA[true]]></xhtml:p>
          </ST>
```

Figura 4.10 — Excerto do documento XML (entrada) na linguagem SFC

```
FUNCTION_BLOCK functionBlock0
 VAR
  xx : INT;
 END_VAR

 INITIAL_STEP Step0 :
 END_STEP

 STEP Step1 :
   action0(N);
   Action_0_INLINE(N);
 END_STEP

 STEP Step2 :
 END_STEP

 STEP Step3 :
 END_STEP

 ACTION Action_0_INLINE :
   xx:=4;
 END_ACTION

 ACTION action0:
  xx := 0;
 END_ACTION

 TRANSITION FROM Step0 TO Step1
   :=  true;
 END_TRANSITION

 TRANSITION FROM Step1 TO Step0
   :=  false;
 END_TRANSITION

 TRANSITION FROM Step0 TO Step3
   :=  false;
 END_TRANSITION

 TRANSITION FROM Step2 TO Step3
   :=  1;
 END_TRANSITION

END_FUNCTION_BLOCK
```

Figura 4.11 — Excerto do documento TXT (saída) do programa SFC

De notar que:

- O nome do *step* é dado pelo atributo "*name*" do elemento "*step*";
- O nome da ação é dado pela composição de "Action_" + o ID deste elemento;
- O nome da transição é dado pelo atributo "*name*" do elemento "*transition*";

Apesar de ser possível a escrita das condições de transição e os blocos de ações nas linguagens LD e FBD, o programa apenas aceita que estas estejam na linguagem ST ou IL.

4.2 - Conversor XML

4.2.1 -Programa FBD

Para o teste deste tipo de linguagem será demonstrado o programa na sua forma tradicional (Figura 4.12) e um excerto do mesmo, mas desta feita em formato XML (TC6-XML), que se encontra presente na Figura 4.13.

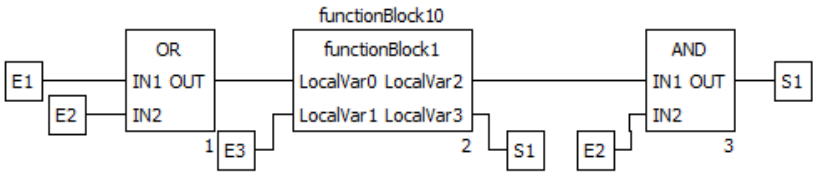


Figura 4.12 — Representação gráfica do programa FBD a ser introduzido no Conversor XML

```
<body>
  <FBD>
    <inVariable localId="13" executionOrderId="0" height="23" width="22" negated="false">
      <position x="191" y="74"/>
      <connectionPointOut>
        <relPosition x="22" y="11"/>
      </connectionPointOut>
      <expression>E1</expression>
    </inVariable>
    <block localId="14" typeName="OR" executionOrderId="1" height="60" width="53">
      <position x="263" y="55"/>
      <inputVariables>
        <variable formalParameter="IN1">
          <connectionPointIn>
            <relPosition x="0" y="30"/>
            <connection refLocalId="13">
              <position x="263" y="85"/>
              <position x="213" y="85"/>
            </connection>
          </connectionPointIn>
        </variable>
        <variable formalParameter="IN2">
          <connectionPointIn>
            <relPosition x="0" y="50"/>
            <connection refLocalId="15">
              <position x="263" y="105"/>
              <position x="239" y="105"/>
            </connection>
          </connectionPointIn>
        </variable>
      </inputVariables>
```

Figura 4.13 — Excerto do documento XML (entrada) na linguagem FBD

```
<body>
  <ST>
    <xhtml:p><![CDATA[
        OR14_OUT := OR(E1, E2);
        functionBlock10(LocalVar0 := OR14_OUT, LocalVar1 := E3);
        S1 := functionBlock10.LocalVar3;
        AND19_OUT := AND(functionBlock10.LocalVar2, E2);
        S1 := AND19_OUT;
]]></xhtml:p>
  </ST>
</body>
```

Figura 4.14 — Excerto do documento XML (saída) do programa FBD em ST

Na Figura 4.14 pode-se observar o excerto do documento XML, produzido pelo Conversor XML, que traduz o programa FBD, presente na Figura 4.12, na linguagem ST.

Como se pode verificar, através das regras descritas no Subcapítulo 2.2, o programa traduzido encontra-se com a sintaxe correta e a produzir o resultado esperado. Para uma dupla verificação correu-se todos os testes no programa Beremiz sendo que, de todos os testes efetuados, foi possível a compilação dos mesmos sem quaisquer erros. De notar a presença de variáveis como por exemplo "AND19_OUT" e "OR14_OUT" que, como referido no Subcapítulo 3.4.1, representam as saídas do bloco com a função "AND" e o bloco com a função "OR" respetivamente.

4.2.2 -Programa LD

O programa LD irá ser apresentado da mesma forma que o programa na linguagem FBD, ou seja, será apresentado o programa LD (Figura 4.15) e a sua representação no documento XML (TC6-XML) na Figura 4.16, através de um pequeno excerto do mesmo.

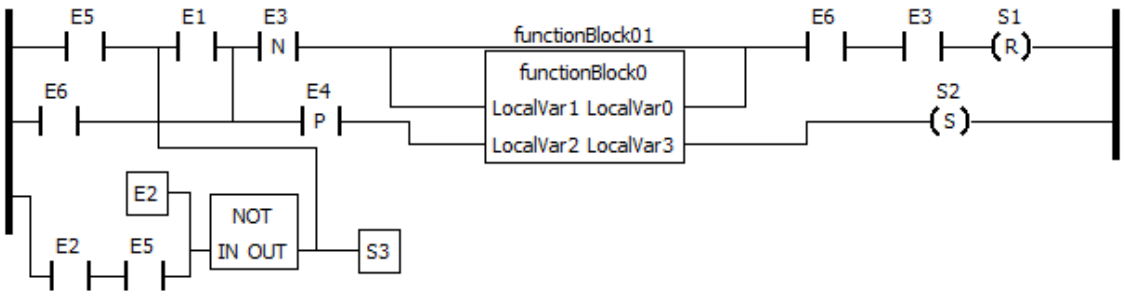


Figura 4.15 — Representação gráfica do programa LD a ser introduzido no Conversor XML

```
<LD>
  <leftPowerRail localId="2" height="120" width="3">
    <position x="24" y="37"/>
    <connectionPointOut formalParameter="">
      <relPosition x="3" y="20"/>
    </connectionPointOut>
    <connectionPointOut formalParameter="">
      <relPosition x="3" y="60"/>
    </connectionPointOut>
    <connectionPointOut formalParameter="">
      <relPosition x="3" y="100"/>
    </connectionPointOut>
  </leftPowerRail>
  <contact localId="3" height="15" width="21" negated="false">
    <position x="116" y="49"/>
    <connectionPointIn>
      <relPosition x="0" y="8"/>
      <connection refLocalId="17">
        <position x="116" y="57"/>
        <position x="77" y="57"/>
      </connection>
      <connection refLocalId="23" formalParameter="OUT">
        <position x="116" y="57"/>
        <position x="106" y="57"/>
        <position x="106" y="111"/>
        <position x="191" y="111"/>
        <position x="191" y="166"/>
        <position x="181" y="166"/>
      </connection>
    </connectionPointIn>
    <connectionPointOut>
      <relPosition x="21" y="8"/>
    </connectionPointOut>
    <variable>E1</variable>
  </contact>
```

Figura 4.16 — Excerto do documento XML (entrada) na linguagem LD

```
    <variable name="NOT23_OUT">
      <type>
        <BOOL />
      </type>
    </variable>
    <variable name="F_TRIG13">
      <type>
        <derived name="F_TRIG" />
      </type>
    </variable>
    <variable name="R_TRIG16">
      <type>
        <derived name="R_TRIG" />
      </type>
    </variable>
  </localVars>
</interface>
<body>
  <ST>
    <xhtml:p><![CDATA[
        F_TRIG13(CLK := E3);
        R_TRIG16(CLK := E4);
        NOT23_OUT := NOT(E5 AND E2 OR E2);
        S3 := NOT23_OUT;
        functionBlock01(LocalVar1 := F_TRIG13.Q  AND (E1 AND (E5 OR NOT23_OUT) OR E6), LocalVar2 := R_TRIG16.Q  AND E6);
        IF E3 AND E6 AND (F_TRIG13.Q  AND (E1 AND (E5 OR NOT23_OUT) OR E6) OR functionBlock01.LocalVar0) THEN
        S1 := FALSE; (*reset*)
        END_IF;
        IF functionBlock01.LocalVar3 THEN
        S2 := TRUE;  (*set*)
        END_IF;
]]></xhtml:p>
  </ST>
</body>
```

Figura 4.17 — Excerto do documento XML (saída) do programa LD em ST

Na Figura 4.17 pode ser observada a conversão efetuada pelo conversor, passando da linguagem LD para a linguagem ST.

De forma análoga ao processo de verificação dos resultados da linguagem FBD, no que diz respeito à linguagem LD, pode-se verificar, através das regras descritas no Subcapítulo 2.2, o programa traduzido encontra-se com a sintaxe correta e a produzir o resultado esperado. Para uma dupla verificação correu-se todos os testes no programa Beremiz sendo que, de todos os testes efetuados, foi possível a compilação dos mesmos sem quaisquer erros.

Pode também ser observável parte da declaração das variáveis onde o conversor introduz três novas variáveis, em que duas delas são responsáveis por representar as variáveis E3 e E4 que se encontram com o modificador FE e RE respetivamente.

4.3 - Casos especiais

Neste subcapítulo serão demonstrados os resultados, obtidos pelo Conversor XML, aquando da presença de programas pouco ortodoxos, ou seja, que introduzem algumas *nuances* especiais para a conversão na linguagem ST.

4.3.1 -"Paralelos" nos Programas LD

Encontra-se, na Figura 4.18, um programa LD que contém umas ligações paralelas e que as mesmas podem causar algum tipo de erro ao Conversor XML.

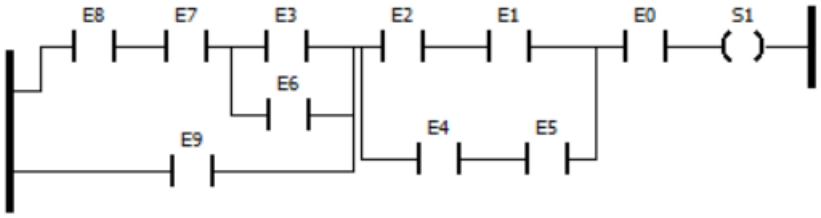


Figura 4.18 — Programa LD com contactos paralelos

Este programa pode ser traduzido na expressão presente na Figura 4.19, sendo que o resultado obtido pelo Conversor XML encontra-se na Figura 4.20.

```
S1 := E0 AND (E1 AND E2 AND ((E3 OR E6) AND E7 AND E8 OR E9) OR E5 AND E4 AND E3 AND E7 AND E8);
```

Figura 4.19 — Código ST do programa presente na Figura 4.18

```
<body>
  <ST>
    <xhtml:p><![CDATA[
        S1 := E0 AND (E1 AND E2 AND (E3 AND E7 AND E8 OR E6 AND E7 AND E8 OR E9) OR E5 AND E4 AND E3 AND E7 AND E8);
]]></xhtml:p>
  </ST>
</body>
```

Figura 4.20 — Excerto do documento XML (saída) do programa da Figura 4.18 em ST

Facilmente se percebe que a expressão presente na Figura 4.19 e na Figura 4.20 produzem o mesmo resultado, isto através das propriedades das expressões booleanas (distribuição).

4.3.2 -Blocos Realimentados

Na presença de programas com blocos realimentados (como por exemplo o programa FBD na Figura 4.21) o Conversor XML apresenta o código ST que se encontra na Figura 4.22.

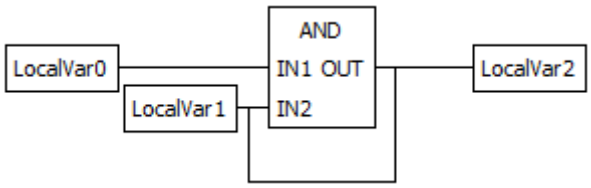


Figura 4.21 — Programa FBD com bloco "AND" realimentado

```
<body>
  <ST>
    <xhtml:p><![CDATA[
        AND1_OUT := AND(LocalVar0, LocalVar1 OR AND1_OUT);
        LocalVar2 := AND1_OUT;
]]></xhtml:p>
  </ST>
</body>
```

Figura 4.22 — Excerto do documento XML (saída) do programa da Figura 4.21 em ST

De notar que o Conversor XML "responde" de forma correta a este tipo de blocos, resultando num código ST válido.

4.3.3 -Variáveis Diretamente Ligadas

Em certos programas FBD ou LD podem existir variáveis (variáveis de entrada, saída e/ou variáveis de entrada/saída) que estejam diretamente ligadas entre si (Figura 4.23), sem estarem ligadas a qualquer bloco. Assim, temos, no código ST, uma atribuição bastante simples que pode ser vista através do resultado, do Conversor XML, expresso na Figura 4.24.

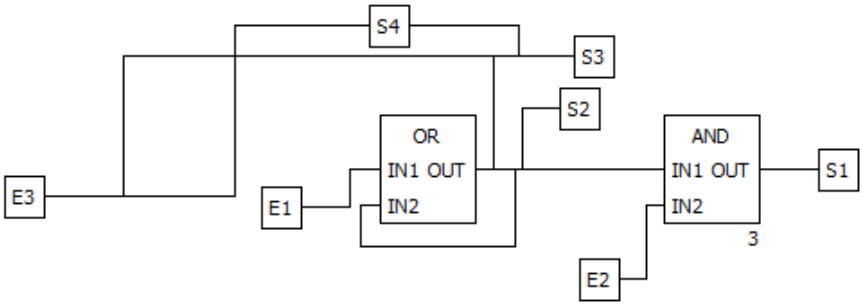


Figura 4.23 — Programa FBD com variáveis ligadas diretamente

```
<body>
  <ST>
    <xhtml:p><![CDATA[
AND16_OUT := AND(OR12_OUT, E2);
S1 := AND16_OUT;
OR12_OUT := OR(E1, OR12_OUT);
S2 := OR12_OUT;
S4 := E3;
S3 := E3 OR OR12_OUT OR S4;
]]></xhtml:p>
  </ST>
</body>
```

Figura 4.24 — Excerto do documento XML (saída) do programa da Figura 4.23 em ST

Através da Figura 4.24 pode-se concluir que o Conversor XML funciona para estes casos, visto que estas simples atribuições se encontram todas declaradas no código ST presente nessa mesma figura.

4.3.4 -Utilização de "*Jumps*"

Como referido no Subcapítulo 2.8 podem ser inseridos "saltos" (elemento "*connector*" no caso do FBD) e "pontos de destino" (elemento "*continuation*" no caso do FBD) entre ligações para tornar os programas gráficos mais simples de visualizar e interpretar. Contudo, o Conversor XML e o Conversor TXT (no caso de programas em SFC) não são compatíveis com esta característica. Assim, se tivermos o programa em FBD presente na Figura 4.25, o mesmo não é corretamente convertido no seu código ST. Sendo assim, trata-se de uma limitação dos projetos desenvolvidos.

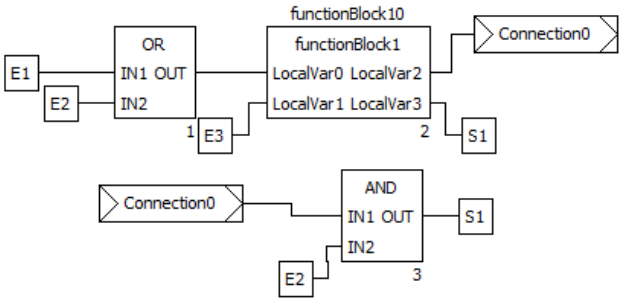


Figura 4.25 — Programa FBD com "saltos" nas ligações

4.3.5 -Variáveis no Programa LD

Nos programas em LD pode-se associar variáveis a contactos (*contacts*), bobinas (*coils*), variáveis de entrada (elemento "*inVariable*"), variáveis de saída (elemento "*outVariable*") e variáveis e entrada/saída (elemento "*inOutVariable*"). As variáveis de entrada, saída e entrada/saída funcionam corretamente com o Conversor XML, desde que as mesmas não se encontrem ligadas diretamente a uma bobina. Ou seja, se por exemplo der entrada no Conversor XML um programa em LD como o da Figura 4.26 que tem uma variável de entrada "E2" ligada à bobina "S1", o mesmo não vai ser convertido corretamente pelo Conversor XML, resultando na omissão da variável "E2".

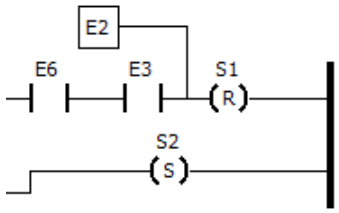


Figura 4.26 — Excerto de um programa LD com variável ligada a uma bobina

Capítulo 5

Conclusões e Trabalhos Futuros

Neste Capítulo é apresentada uma conclusão do Projeto realizado e algumas sugestões para trabalho futuro de forma a melhorar e ampliar este Projeto.

Neste Projeto foram realizados estudos às linguagens presentes na Norma IEC61131-3 e à linguagem XML, assim como, a estruturação TC6-XML compatível com a norma. Assim sendo, foram estudadas as linguagens ST, LD, IL, FBD e SFC quanto às suas regras e sintaxe apresentando sempre que possível exemplos ilustrativos. O mesmo também ocorreu aquando da abordagem do XML, mais especificamente para a estruturação do tipo TC6-XML, relevando os aspetos mais importantes do mesmo.

De seguida foram apresentados os dois grandes Projetos deste Projeto global:

- O Conversor XML;
- O Conversor TXT;

Estes Projetos baseiam-se em algoritmos que foram aqui descritos e descortinados. Na implementação dos algoritmos desenvolvidos pode-se concluir que se trata de um Projeto funcional e que se encontra a produzir documentos segundo a Norma IEC 61131-3. Contudo, estes algoritmos podem ser melhorados ou repensados para os tornar mais robustos e cada vez mais capazes (com mais funcionalidades).

Ambos os Projetos encontram-se capazes de processar qualquer tipo de variável, com ou sem modificador, assim como, programas, funções e blocos de funções criados pelo utilizador. No Conversor TXT está implementada a capacidade de processar *data types* criados pelo utilizador, como também as diferentes configurações dos Projetos de automação criados.

Quanto ao Conversor XML, este encontra-se capaz de processar documentos XML que contenham qualquer linguagem para a declaração dos POUs, ou seja, tem a capacidade de perceber que POUs podem e devem ser convertidos (POUs escritos na linguagem LD e/ou FBD) e quais não devem (POUs escritos na linguagem ST, IL e SFC).

O método de criação/desenvolvimento destes dois Projetos foi baseado no método tentativa erro. Ou seja, a criação de programas de teste ditavam, em certa parte, a implementação e o melhoramento dos algoritmos presentes neste documento. Assim sendo, existe, com certeza, aspetos não abordados que poderão tornar as duas ferramentas criadas, incompletas. Esses aspetos podem ser o facto de, por exemplo, o Conversor XML não abranger a possibilidade de criação/conversão da variável *enable out* e ou *enable in* que podem estar presentes num qualquer bloco para a verificação da atuação do mesmo. Apesar de ser o método mais utilizado, o mesmo, foi também acompanhado de um estudo mais específico das lacunas dos conversores procurando, sempre que possível, a implementação de novas funcionalidades através desse mesmo estudo (mais especificamente o documento contendo a estruturação TC6-XML).

Em suma, pode-se afirmar que o Projeto respondeu em certa parte aos objetivos pretendidos, visto que o mesmo não chegou a ser testado em conjunto com o Projeto matiec (sendo este o objetivo final) e encontra-se com algumas limitações.

Como ideias para Trabalho Futuro de forma a melhorar os algoritmos apresentados e ampliar a capacidade dos mesmos fica:

- Adicionar a possibilidade do Conversor TXT conseguir interpretar outras linguagens que estejam presentes nas condições de transição ou ações nos programas SFC;
- Melhoria das funções que interpretam as variáveis (“*inVariable”, “outVariable” e inOutVariable*) em conjunto com os contactos e bobinas;
- Adicionar a possibilidade de leitura e tradução da saída *enable out* e da entrada *enable in*, presente nos blocos;
- Um melhor método de criação das variáveis de saída de cada bloco;
- Simplificação da operação “OR” na escrita da sua tradução para ST em programas na linguagem LD;
- Adicionar a possibilidade de converter programas que contenham “saltos” nas ligações;
- Teste do Projeto em conjunto com o Projeto matiec e validação do mesmo;

Anexo A

Anexos

A.1 - Diagramas de classe

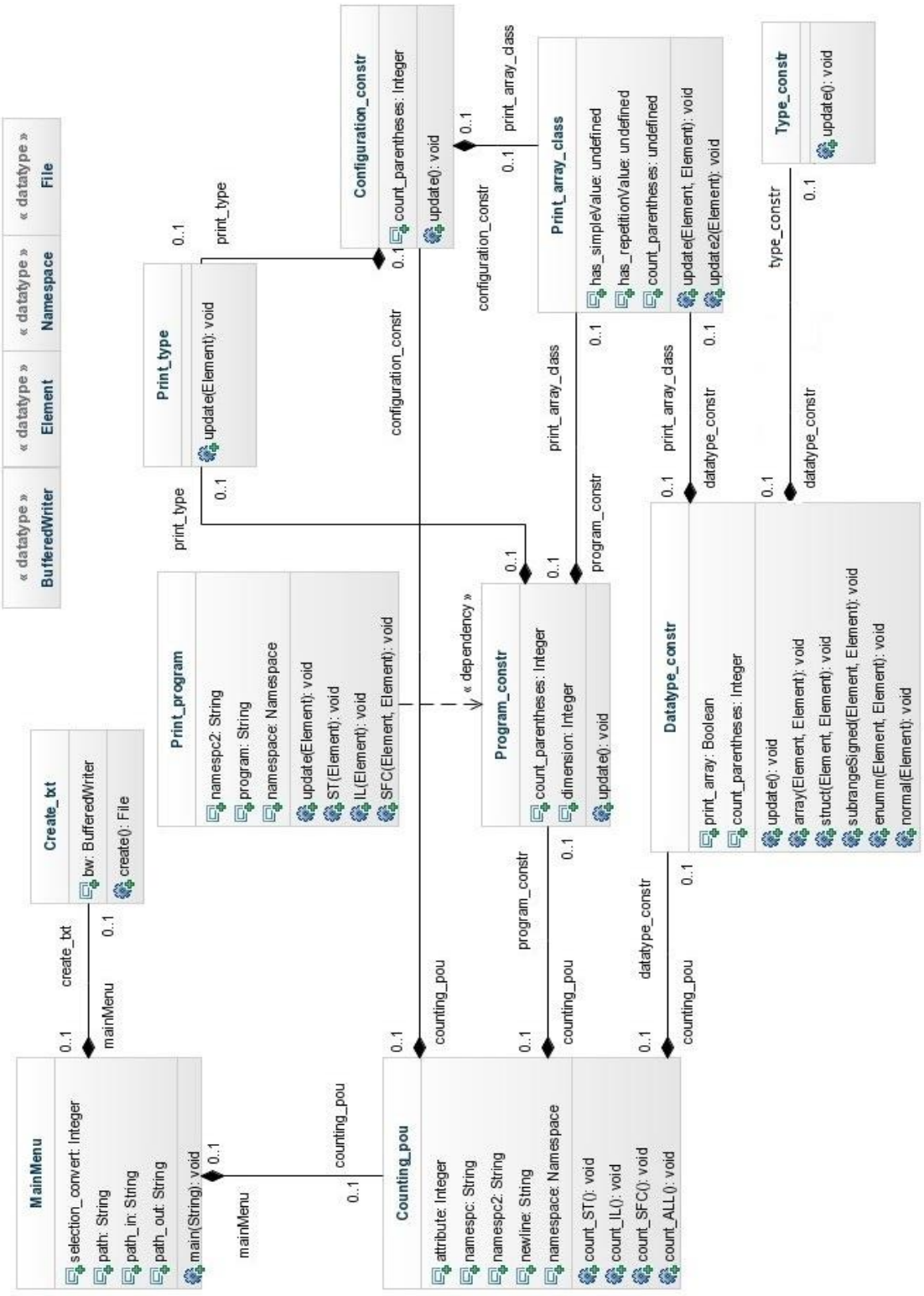


Figura A.1 – Diagrama de classes do Conversor TXT

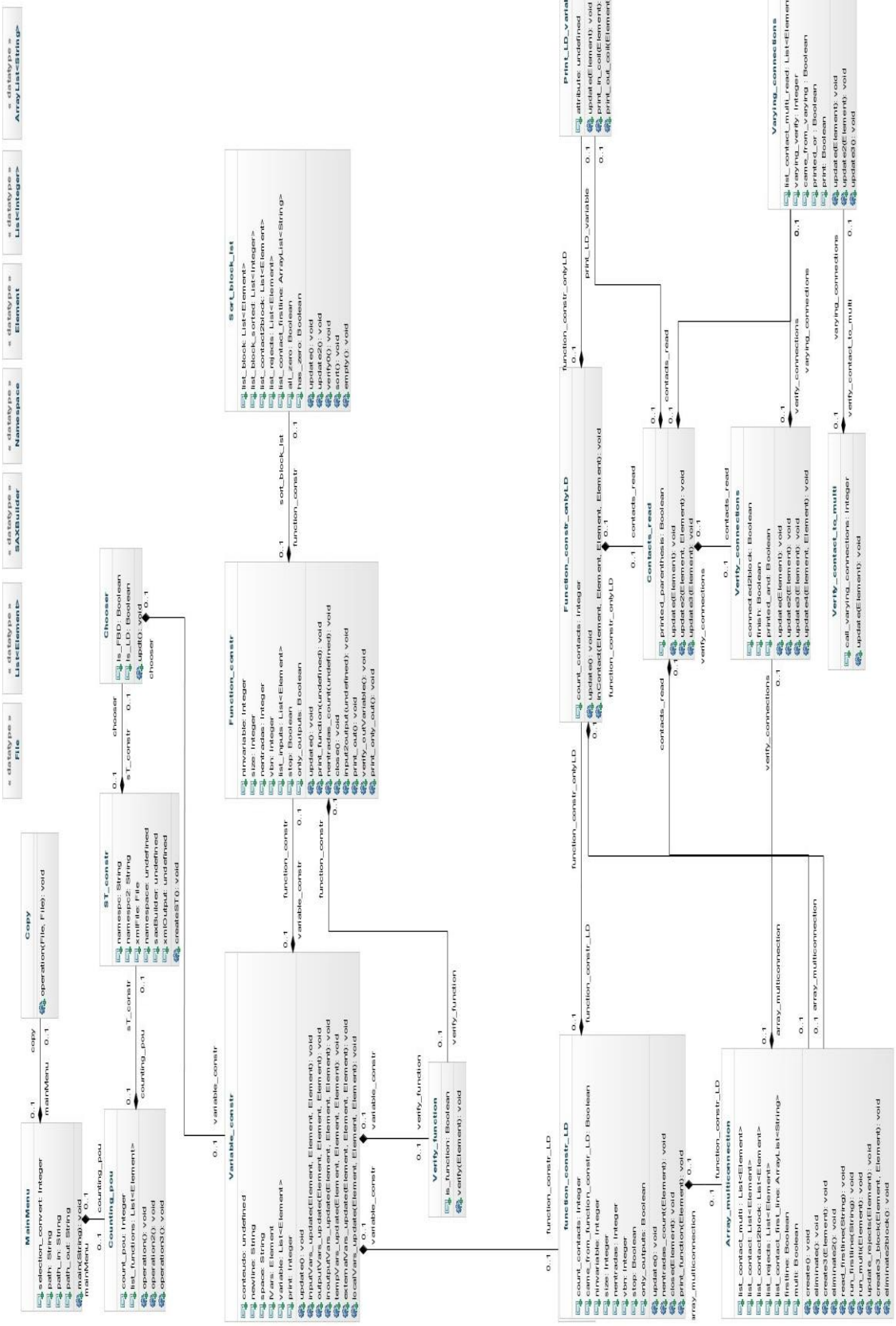


Figura A.2 — Diagrama de classes do Conversor XML

A.2 - Fluxogramas

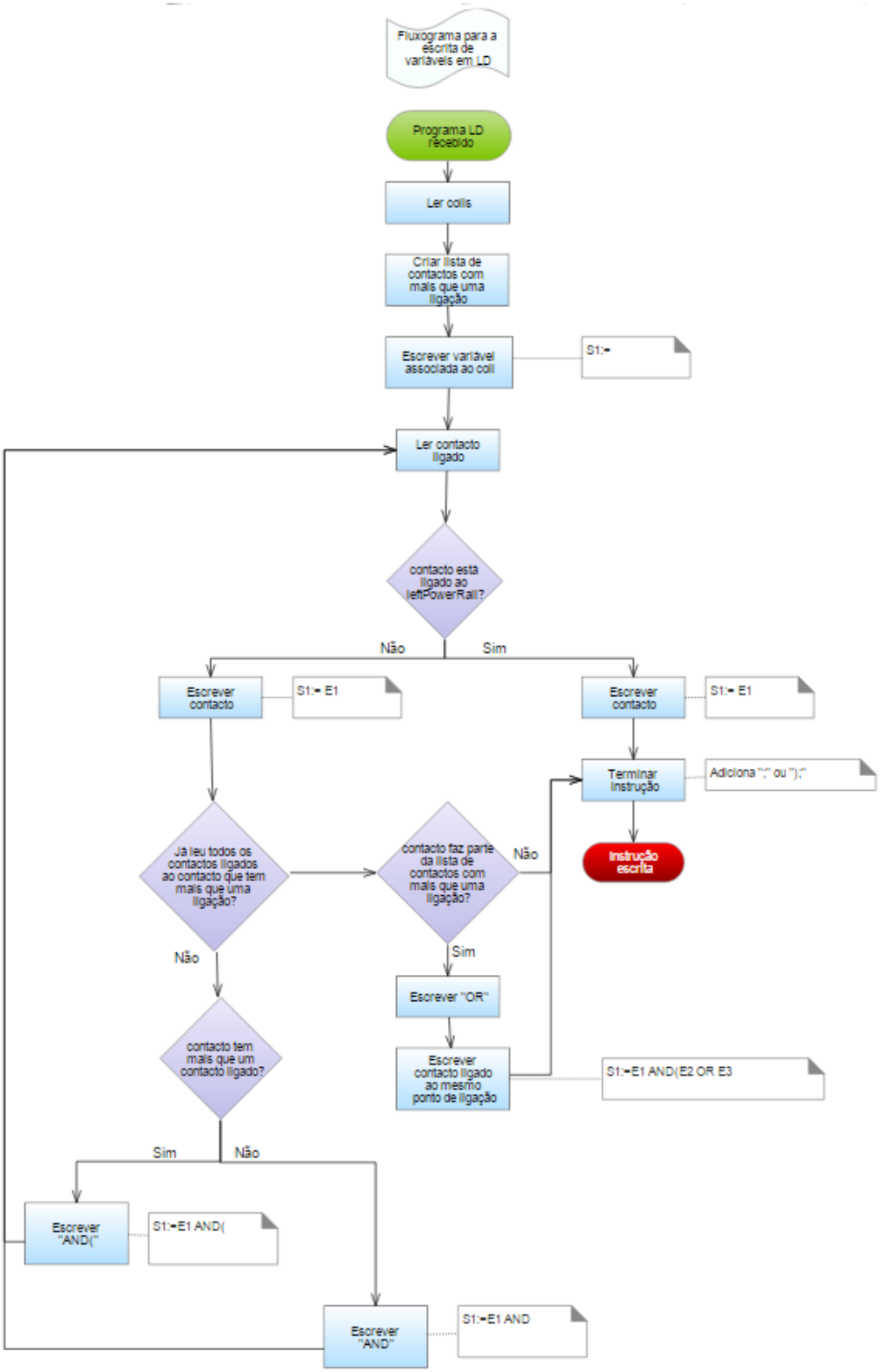


Figura A.3 — Fluxograma para a escrita de variáveis de um programa LD

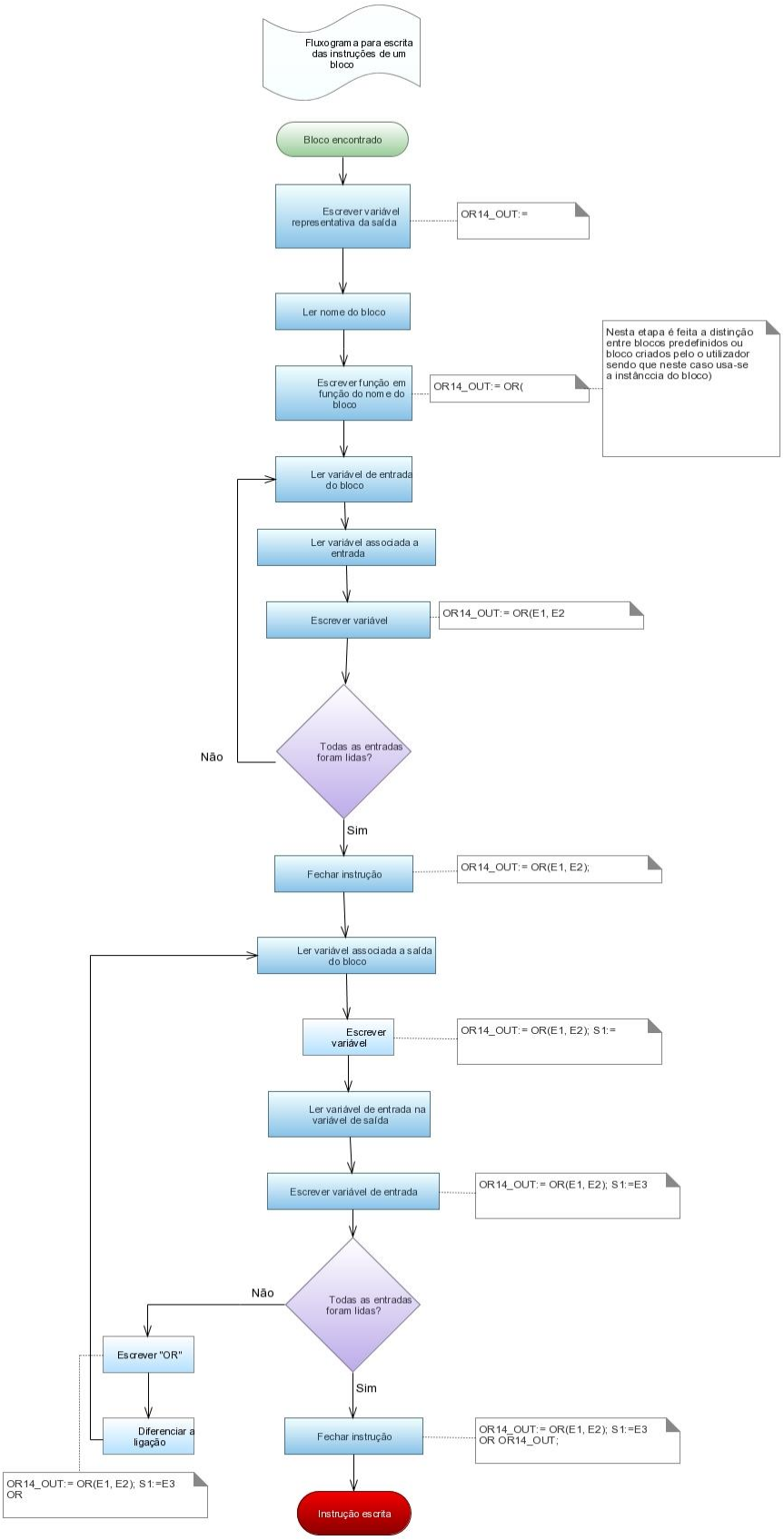


Figura A.4 — Fluxograma para a escrita de instruções de um bloco

A.3 - Documento XML

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://www.plcopen.org/xml/tc6_0201"
xmlns:ns1="http://www.plcopen.org/xml/tc6_0201"
xmlns:xhtml="http://www.w3.org/1999/xhtml"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <fileHeader companyName="Gomes" productName="teste1" productVersion="1"
creationDateTime="2014-05-11T17:15:49" />
  <contentHeader name="teste" modificationDateTime="2015-01-11T23:43:29">
    <coordinateInfo>
      <fbd>
        <scaling x="0" y="0" />
      </fbd>
      <ld>
        <scaling x="0" y="0" />
      </ld>
      <sfc>
        <scaling x="0" y="0" />
      </sfc>
    </coordinateInfo>
  </contentHeader>
  <types>
    <dataTypes>
      <dataType name="datatype4">
        <baseType>
          <BOOL />
        </baseType>
        <initialValue>
          <simpleValue value="TRUE" />
        </initialValue>
      </dataType>
      <dataType name="datatype0">
        <baseType>
          <array>
            <dimension lower="1" upper="2" />
            <dimension lower="1" upper="2" />
            <dimension lower="1" upper="2" />
            <baseType>
              <BOOL />
            </baseType>
          </array>
        </baseType>
        <initialValue>
          <arrayValue>
            <value>
              <arrayValue>
                <value repetitionValue="2">
                  <arrayValue>
                    <value>
                      <simpleValue value="2" />
                    </value>
                    <value>
                      <simpleValue value="2" />
                    </value>
                  </arrayValue>
                </value>
                <value>
```

```
            <arrayValue>
              <value>
                <simpleValue value="2" />
              </value>
              <value>
                <simpleValue value="2" />
              </value>
            </arrayValue>
          </value>
          <value>
            <arrayValue>
              <value>
                <simpleValue value="2" />
              </value>
              <value>
                <simpleValue value="3" />
              </value>
            </arrayValue>
          </value>
        </arrayValue>
      </value>
    </arrayValue>
  </initialValue>
 </dataType>
 <dataType name="datatype2">
  <baseType>
   <array>
    <dimension lower="1" upper="2" />
    <baseType>
     <BOOL />
    </baseType>
   </array>
  </baseType>
  <initialValue>
   <arrayValue>
    <value>
     <simpleValue value="1" />
    </value>
    <value>
     <simpleValue value="3" />
    </value>
   </arrayValue>
  </initialValue>
 </dataType>
</dataTypes>
<pous>
 <pou name="program1" pouType="program">
  <interface>
   <localVars>
    <variable name="E1">
     <type>
      <BOOL />
     </type>
     <initialValue>
      <simpleValue value="False" />
     </initialValue>
    </variable>
    <variable name="E3">
     <type>
      <BOOL />
```

```
  </type>
  <initialValue>
    <simpleValue value="False" />
  </initialValue>
</variable>
<variable name="S3">
  <type>
    <BOOL />
  </type>
  <initialValue>
    <simpleValue value="False" />
  </initialValue>
</variable>
<variable name="E4">
  <type>
    <TOD />
  </type>
</variable>
<variable name="E5">
  <type>
    <BOOL />
  </type>
  <initialValue>
    <simpleValue value="False" />
  </initialValue>
</variable>
<variable name="S2">
  <type>
    <BOOL />
  </type>
  <initialValue>
    <simpleValue value="False" />
  </initialValue>
</variable>
<variable name="E6">
  <type>
    <BOOL />
  </type>
  <initialValue>
    <simpleValue value="False" />
  </initialValue>
</variable>
<variable name="E2">
  <type>
    <derived name="datatype4" />
  </type>
</variable>
<variable name="S1">
  <type>
    <BOOL />
  </type>
</variable>
<variable name="functionBlock01">
  <type>
    <derived name="functionBlock0" />
  </type>
```

```
          </variable>
          <variable name="E7">
            <type>
              <BOOL />
            </type>
          </variable>
          <variable name="NOT23_OUT">
            <type>
              <BOOL />
            </type>
          </variable>
          <variable name="F_TRIG13">
            <type>
              <derived name="F_TRIG" />
            </type>
          </variable>
          <variable name="R_TRIG16">
            <type>
              <derived name="R_TRIG" />
            </type>
          </variable>
        </localVars>
      </interface>
      <body>
        <ST>
          <xhtml:p><![CDATA[
F_TRIG13(CLK := E3);
R_TRIG16(CLK := E4);
NOT23_OUT := NOT(E5 AND E2 OR E2);
S3 := NOT23_OUT;
functionBlock01(LocalVar1 := F_TRIG13.Q AND (E1 AND (E5 OR NOT23_OUT) OR E6),
LocalVar2 := R_TRIG16.Q AND E6);
IF E3 AND E6 AND (F_TRIG13.Q AND (E1 AND (E5 OR NOT23_OUT) OR E6) OR
functionBlock01.LocalVar0) THEN
S1 := FALSE; (*reset*)
END_IF;
IF functionBlock01.LocalVar3 THEN
S2 := TRUE;  (*set*)
END_IF;
]]></xhtml:p>
        </ST>
      </body>
    </pou>
    <pou name="functionBlock0" pouType="functionBlock">
      <interface>
        <outputVars>
          <variable name="LocalVar0">
            <type>
              <BOOL />
            </type>
          </variable>
          <variable name="LocalVar3">
            <type>
              <BOOL />
            </type>
          </variable>
        </outputVars>
        <inputVars>
          <variable name="LocalVar1">
            <type>
```

```
            <BOOL />
           </type>
          </variable>
          <variable name="LocalVar2">
           <type>
            <BOOL />
           </type>
          </variable>
         </inputVars>
        </interface>
        <body>
         <ST>
          <xhtml:p><![CDATA[
LocalVar0 := LocalVar1 and LocalVar2;
LocalVar3 := LocalVar1 and LocalVar2;]]></xhtml:p>
         </ST>
        </body>
       </pou>
       <pou name="program0" pouType="program">
        <interface>
         <localVars>
          <variable name="LocalVar0">
           <type>
            <BOOL />
           </type>
          </variable>
          <variable name="LocalVar1">
           <type>
            <BOOL />
           </type>
          </variable>
          <variable name="LocalVar2">
           <type>
            <BOOL />
           </type>
          </variable>
          <variable name="LocalVar3">
           <type>
            <BOOL />
           </type>
          </variable>
          <variable name="LocalVar4">
           <type>
            <BOOL />
           </type>
          </variable>
          <variable name="LocalVar5">
           <type>
            <BOOL />
           </type>
          </variable>
          <variable name="functionBlock00">
           <type>
            <derived name="functionBlock0" />
           </type>
          </variable>
```

```
        </localVars>
       </interface>
       <body>
        <ST>
         <xhtml:p><![CDATA[
functionBlock00(LocalVar1 := LocalVar0, LocalVar2 := LocalVar2);
LocalVar1 := LocalVar0;
LocalVar4 := functionBlock00.LocalVar0;
LocalVar5 := functionBlock00.LocalVar3;
]]></xhtml:p>
        </ST>
       </body>
     </pou>
   </pous>
  </types>
  <instances>
   <configurations>
    <configuration name="config">
     <resource name="resource1">
      <task name="tsk" priority="0" interval="T#1d0h0m0s0ms">
       <pouInstance name="sds" typeName="program1" />
      </task>
      <globalVars>
       <variable name="LocalVar0">
        <type>
         <derived name="datatype4" />
        </type>
       </variable>
       <variable name="LocalVar1">
        <type>
         <INT />
        </type>
       </variable>
      </globalVars>
     </resource>
    </configuration>
   </configurations>
  </instances>
</project>
```

A.4 - Documento TXT

```
TYPE
 datatype4 : BOOL := TRUE;
 datatype0 : ARRAY [1..2, 1..2, 1..2] OF BOOL := [[2([2, 2]),[2, 2] ,[2, 3]]];
 datatype2 : ARRAY [1..2] OF BOOL := [1, 3];
END_TYPE


PROGRAM program1
 VAR
  E1 : BOOL := False;
  E3 : BOOL := False;
  S3 : BOOL := False;
  E4 : TOD;
  E5 : BOOL := False;
  S2 : BOOL := False;
  E6 : BOOL := False;
  E2 : datatype4;
  S1 : BOOL;
  functionBlock01 : functionBlock0;
  E7 : BOOL;
  NOT23_OUT : BOOL;
  F_TRIG13 : F_TRIG;
  R_TRIG16 : R_TRIG;
 END_VAR

  F_TRIG13(CLK := E3);
  R_TRIG16(CLK := E4);
  NOT23_OUT := NOT(E5 AND E2 OR E2);
  S3 := NOT23_OUT;
  functionBlock01(LocalVar1 := F_TRIG13.Q AND (E1 AND (E5 OR NOT23_OUT) OR E6),
LocalVar2 := R_TRIG16.Q AND E6);
  IF E3 AND E6 AND (F_TRIG13.Q AND (E1 AND (E5 OR NOT23_OUT) OR E6) OR
functionBlock01.LocalVar0) THEN
  S1 := FALSE;
  (*reset*)
  END_IF;
  IF functionBlock01.LocalVar3 THEN
  S2 := TRUE;
  (*set*)
  END_IF;
END_PROGRAM

FUNCTION_BLOCK functionBlock0
 VAR_INPUT
  LocalVar1 : BOOL;
  LocalVar2 : BOOL;
 END_VAR
 VAR_OUTPUT
  LocalVar0 : BOOL;
  LocalVar3 : BOOL;
 END_VAR

  LocalVar0 := LocalVar1 and LocalVar2;
  LocalVar3 := LocalVar1 and LocalVar2;
END_FUNCTION_BLOCK

PROGRAM program0
 VAR
  LocalVar0 : BOOL;
  LocalVar1 : BOOL;
```

```
  LocalVar2 : BOOL;
  LocalVar3 : BOOL;
  LocalVar4 : BOOL;
  LocalVar5 : BOOL;
  functionBlock00 : functionBlock0;
 END_VAR

  functionBlock00(LocalVar1 := LocalVar0, LocalVar2 := LocalVar2);
  LocalVar1 := LocalVar0;
  LocalVar4 := functionBlock00.LocalVar0;
  LocalVar5 := functionBlock00.LocalVar3;
END_PROGRAM

CONFIGURATION config
 RESOURCE resource1 ON PLC
 VAR_GLOBAL
  LocalVar0 : datatype4;
  LocalVar1 : INT;
 END_VAR
  TASK tsk(INTERVAL := T#1d0h0m0s0ms, PRIORITY := 0);
  PROGRAM sds WITH tsk: program1;
 END_RESOURCE
END_CONFIGURATION
```

Referências

[1] "APIs para ler documentos XML," ed.

[2] (2014, 10/07/2014). *XML DOM Tutorial*. Available: http://www.w3schools.com/Dom/

[3] (2010, 10/07/2014). *TinyXML Tutorial*. Available: http://www.grinninglizard.com/tinyxmldocs/tutorial0.html

[4] PLCopen, "XML Formats for IEC 61131-3," p. 80, May 08, 2009 2009.

[5] L. A. B. E. A. Bryan, *Programmable Controllers Theory and Implementation*. United States of America: Industrial Text Company, 1997.

[6] I. I. E. Comission), "Programmable controllers - Part 3: Programming languages.," in *Norma Internacional.*, ed: I. 61131-1, 2003.

[7] N. Maravitsas. (2013, 09/07/2014). *Modify XML File in Java using JDOM parser example*. Available: http://examples.javacodegeeks.com/core-java/xml/jdom/modify-xml-file-in-java-using-jdom-parser-example/

[8] F. O. J. Asensio, M. Damas, H. Pomares, "Industrial Automation Programming Environment with a New Translation Algorithm among IEC 61131-3 Languages Based on the TC6-XML Scheme," *International Journal of Automation and Control Engineering*, vol. 2, p. 9, May 2013 2013.

[9] H. C. F. GUIMARÃES, "NORMA IEC 61131-3 PARA PROGRAMAÇÃO DE CONTROLADORES PROGRAMÁVEIS: ESTUDO E APLICAÇÃO," DEPARTAMENTO DE ENGENHARIA ELÉTRICA, UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO CENTRO TECNOLÓGICO, 2005.

[10] M. T. K.-H. John, "Building Blocks of IEC 61131-3," in *IEC 61131-3: Programming Industrial Automation Systems*, 2nd edition ed Verlag Berlin Heidelberg: Springer, 2010.

[11] M. d. Sousa. (2011). *Overview of IEC 61131-3*. Available: https://sigarra.up.pt/feup/pt/conteudos_service.conteudos_cont?pct_id=196667&pv_cod=36aRg0GazaPR

[12] "PL7 - Structured Text Language," ed, p. 8.

[13] "Texto Base-Linguagem em Ladder 1," vol. 2.

[14] W. Bolton, "Ladder and Functional Block Programming," p. 30.

[15] *Programação de Autómatos Utilizando a norma IEC 61131-3*. Available: https://sigarra.up.pt/feup/pt/conteudos_service.conteudos_cont?pct_id=95375&pv_cod=44GoHdmanvIq

[16] W. Wei, "Merging of XML documents," MQ97459 M.C.S., Carleton University (Canada), Ann Arbor, 2004.

[17] K. H. Goldberg, *XML: Visual QuickStart Guide*: Pearson Education, 2010.

[18] J. M. N. Coelho, "Processamento de XML em Programação em Lógica," Departamento de Ciência de Computadores, Universidade do Porto, 2002.

[19] B. M. Jason Hunter. (2000, 09/07/2014). *Easy Java/XML integration with JDOM, Part 1*. Available: http://www.javaworld.com/article/2076294/java-se/easy-java-xml-integration-with-jdom--part-1.html
[20] (07/05/2014). *Beremiz*. Available: http://www.beremiz.org/