UNIVERSIDADE DO VALE DO RIO DOS SINOS

CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS

PROGRAMA INTERDISCIPLINAR DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO APLICADA - PIPCA

Uma Arquitetura de Interoperabilidade para as Linguagens da Norma IEC 1131-3 Usando XML

por

JOAQUIM GOMES DA COSTA EULÁLIO DE SOUZA

Dissertação submetida a avaliação
como requisito parcial para a obtenção do grau de
Mestre em Computação Aplicada

Prof. Dr. Sérgio Crespo Coelho da Silva Pinto
Orientador

São Leopoldo, Fevereiro de 2004

Agradecimentos

Agradeço a meus pais, em memória, pela educação que me propiciaram e pela pessoa que me ensinaram a ser. Tenho orgulho de ser filho deles.

Agradeço a minha família e amigos pelo incentivo a realizar um trabalho como este e pela compreensão da ausência em várias ocasiões como conseqüência do tempo dedicado nesta tarefa.

Agradeço a meus amigos e colegas da Altus, não só pelo apoio técnico mas principalmente pelo interesse demonstrado para que este trabalho resultasse em sucesso, e pela compreensão das dificuldades e transtornos que possam ter causado.

Agradeço a empresa Altus e seus diretores pela oportunidade proporcionada e pelo apoio oferecido.

Agradeço a meu orientador pela lucidez e boa vontade em procurar e desenvolver um tema relevante e de interesse do aludo, da universidade e da empresa.

Agradeço a todos os meus colegas de mestrado e PIPCA pela ajuda que sempre foi oferecida com boa vontade.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

BNF	Backus Normal Form
CI	Circuito Integrado
CLP	Controlador Lógico Programável
CP	Controlador Programável
CTT	Mnemônico da instrução contato na linguagem Ladder
DSL	Domain Specific Language
DTD	Document Type Definition
FBD	Function Block Diagram
HTML	HyperText Markup Language
IEC	International Electro-Technical Commission
IEEE	Institute of Electrical and Electronics Engineers
IL	Instruction List
ILA	Interpretador de Linguagem Algoritmica
LD	Ladder Diagram
MFC	Microsoft Foundation Classes
PLC	Programmable Logic Controller
POU	Program Organization Unit
SFC	Sequential Function Chart
SGML	Standard Generalized Markup Language
ST	Structured Text
UML	Unified Modeling Language
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits
W3C	World Wide Web Consortium
XML	Extensible Markup Language

Resumo

A norma IEC 1131-3 define e padroniza cinco linguagens de programação de controladores programáveis e diz que os programas escritos em uma linguagem podem ser convertidos para outra, mas não diz nada sobre como fazer isto. Esta dissertação apresenta a proposta de uma arquitetura XML que permite a interoperabilidade para as linguagens ST, FBD e LD desta norma.

É apresentado uma revisão bibliográfica sobre linguagens específicas de domínio, conceitos de XML e uma visão geral da norma IEC 1131-3, bem como alguns trabalhos relacionados com controladores programáveis, linguagem XML e linguagens de programação da norma IEC 1131-3.

É feita uma comparação entre as linguagens ST, LD e FBD e uma proposta de representação textual para os diagramas gráficos LD e FBD. A linguagem CLPScript é definida a partir da linguagem ST. São definidos as *tags* e atributos utilizadas para a representação de diagramas LD e FBD e programas CLPScript, formando a arquitetura XML proposta.

Foi desenvolvido o programa CLPStoXML, um compilador que lê um programa CLPScript e gera a arquitetura XML correspondente.

Como prova de conceito, foi realizado um estudo de caso onde se converteram alguns programas fonte escritos em CLPScript para XML e deste XML para a linguagem LD da Altus S. A., uma empresa fabricante de controladores programáveis. Para realizar esta conversão foi desenvolvido o programa XMLtoLD, especificamente para os controladores programáveis da Altus.

Abstract

The international standard IEC 1131-3 defines five programming languages for programmable controllers. The standard says that it is possible to translate a program developed with one language to any other, but doesn't say how. This master dissertation presents the proposal of a XML architecture that allows this kind of interchange for languages ST, FBD and LD.

It is presented a bibliographical revision on domain specific languages, main concepts of XML and general aspects of IEC 1131-3 standard. Some works related with programmable controllers, XML and IEC 1131-3 programming languages are presented.

It is made a comparison between ST, LD and FBD languages and a proposal of textual representation for LD and FBD graphical diagrams. The CLPScript language is defined from ST language. It is defined XML *tags* and attributes used for the representation of LD and FBD diagrams and CLPScript programs, forming the XML architecture.

The CLPStoXML program was developed, a compiler who reads a CLPScript program and generates corresponding XML architecture. A case study was developed where some CLPScript programs were converted to XML and then to LD language of Altus S. A., a manufacturer company of programmable controllers. To carry through this conversion the XMLtoLD program was developed, specifically for Altus's programmable controllers.

1 Introdução

Resumo

Este capítulo apresenta uma introdução possibilitando uma visão geral do trabalho a ser realizado, destacando-se o contexto do trabalho, o problema e o objetivo a ser alcançado.

1.1 Introdução

O Controlador Programável (CP, CLP ou PLC) é um equipamento da área de automação industrial ([NAT00]) que surgiu nos anos 60 para substituir os caros e complexos painéis de relés da época. Seus requisitos iniciais eram ([FES00]):

- ser de fácil programação;
- possibilitar alterações do programa sem alterar o resto do sistema, ou seja, sem modificações internas da fiação;
- ser menor, mais barato e mais seguro que os controladores a relés;
- ter manutenção fácil e barata.

Desde então se passaram três décadas e o enorme desenvolvimento da microeletrônica também acabou influenciando os CPs, aumentando bastante sua tecnologia ([MOR01]).

Atualmente o CP ([GEO00]) está presente em praticamente todas as áreas da automação industrial. Dentre alguns dos principais fabricantes, pode-se citar :

- ABB - Sweden - www.abb.com
- Altus S. A. - Brazil - www.altus.com.br
- Rockwell Software - USA - www.rockwell.com
- Siemens - Germany - www.siemens.com
- Schneider Automation - Germany - www.schneider-electric.com
- Yokogawa Electric - Japan - www.yokogawa.com

Uma vez que os CPs surgiram para substituir os painéis de controle a relés, uma linguagem de programação que mais se aproximasse da experiência de técnicos e engenheiros da época seria a solução mais adequada para o desenvolvimento de programas aplicativos ([FES00]).

Neste contexto, a linguagem *Ladder* ([BON97], [LEW95]) se tornou uma das mais utilizadas ao longo do tempo, pela sua simplicidade e semelhança com um esquema elétrico.

Durante os últimos 20 anos, uma variedade de linguagens e técnicas de programação foram sendo criadas e utilizadas para escrever programas para controle de aplicações industriais, e cada fabricante as implementou da sua própria forma. A linguagem *Ladder*, por exemplo, foi adotada pela maioria dos fabricantes de CPs, mas cada um deles implementou seu próprio dialeto ([MOR01]). O resultado final é que todos estas linguagens são na verdade diferentes, tanto semântica quanto sintaticamente.

Para o pessoal envolvido com estes sistemas, sejam técnicos, engenheiros, projetistas, operadores ou pessoal de manutenção, este resultado é muito ineficiente, pois é preciso estar treinado com equipamentos de diferentes fabricantes, cada qual com sua própria linguagem de programação e particularidades.

A norma IEC 1131-3 surgiu em 1993 ([IEC93]) devido a este requisito de padronização na área de linguagens de programação para controladores programáveis. As suas principais características são:

- padronização dos programas aplicativo dos controladores programáveis;
- define 5 linguagens de programação;
- especifica a sintaxe e semântica de cada uma destas linguagens;
- permite projeto de *software* estruturado;
- facilidades de reutilização de código e manutenção;
- permite desenvolvimento independente de fabricante.

Das cinco linguagens definidas, duas são gráficas, *Ladder Diagram* (LD) e *Function Block Diagram* (FBD), duas são textuais, *Instruction List* (IL) e *Structured Text* (ST), e uma quinta serve como base de organização para as demais, *Sequential Function Chart* (SFC).

1.2 Motivação e Contexto do Trabalho

A IEC 1131-3 é uma norma internacional que padroniza as linguagens de programação para controladores programáveis na área de automação industrial. Definindo um conjunto de linguagens gráficas e textuais correlacionadas, traz diversos benefícios para todas os envolvidos, sejam engenheiros, integradores, técnicos, operadores ou pessoal de manutenção.

A norma marca o início do uso de programas bem estruturados, reutilizáveis e de fácil manutenção na área de controle de processo industrial. As empresas fabricantes de *software* e hardware precisam se adaptar a norma para que possam ser competitivas neste mercado, e muitas ainda não o fizeram completamente. Dentro deste contexto o trabalho foi desenvolvido.

1.3 Problema

A norma IEC 1131-3 surgiu com o objetivo de padronização dos programas aplicativo dos controladores programáveis. Ela especifica 5 linguagens de programação, sendo que os programas podem ser escritos na linguagem que o projetista achar mais adequada para a aplicação.

A norma determina que qualquer programa escrito em qualquer das 4 linguagens (LD, FBD, ST ou IL) **pode ser convertido** em um programa equivalente nas outras linguagens, mas não diz nada sobre **como fazer isto**. Nenhum algoritmo ou esquema de conversão está documentado ([TOU97]).

Sendo assim, alguns fabricantes de CPs que implementaram mais de uma linguagem da IEC 1131-3 desenvolveram seus próprios sistemas de conversão entre linguagens, com algoritmos proprietários, não padronizados e que também não são públicos, pois a maioria destes *softwares* é de uso comercial.

Os *softwares* de programação de controladores programáveis ([ALL94, ALT03, ROC99]) são em sua grande maioria proprietários, e seus fornecedores não disponibilizam os detalhes de implementação e muito menos o código fonte.

Além disto, As linguagens LD e FDB são linguagens gráficas, padronizadas, para que possam ser utilizadas da mesma forma em CPs de diferentes fabricantes. Entretanto, o formato binário no qual os programas desenvolvidos nestas linguagens são armazenados é proprietário, não padronizado e diferente a cada fabricante. O programa gráfico LD ou FBD feito com o *software* de determinado fabricante só pode ser utilizado naquele fabricante. Para ser utilizado em outro, é preciso digitá-lo novamente no *software* programador deste outro fabricante, ou seja, reprogramar toda a aplicação com todo o custo de um processo desta natureza.

1.4 Questão de Pesquisa

Como permitir uma interoperabilidade entre as linguagens LD, FBD e ST, conforme previsto pela norma IEC 1131-3, por meio de uma arquitetura única de forma que o trabalho realizado em uma dessas linguagens possa ser utilizado ou complementado em outra linguagem ?

1.5 Objetivo

Os objetivos deste trabalho são:

a) especificar e implementar uma arquitetura XML, tendo por base uma ontologia para unificar e permitir a integração das linguagens LD, FBD e um sub conjunto da linguagem ST da norma IEC 1131-3;
b) especificar e implementar a linguagem específica de domínio CLPscript que gera e formaliza esta arquitetura XML.

Salienta-se que as linguagens IL e SFC da norma IEC 1131-3 estão fora do escopo deste trabalho.

A arquitetura XML proposta é uma alternativa que permite a interoperabilidade entre as linguagens definidas pela norma IEC 1131-3. Um programa armazenado neste formato poderá ser convertido para as demais linguagens, independente da linguagem original com que foi criado. Esta arquitetura pode fazer com que programas LD ou FBD de diferentes fabricantes sejam compatíveis entre si, desde que o *software* correspondente esteja programado para entender e utilizar o XML proposto.

A linguagem CLPScript é oferecida como uma forma mais confortável de gerar esta arquitetura XML. Usando-se a CLPScript, o programa desenvolvido já será salvo com este recurso de interoperabilidade, no formato XML correto. Caso não se deseje utilizar CLPScript, o editor da linguagem utilizada é que será responsável pela criação do XML proposto.

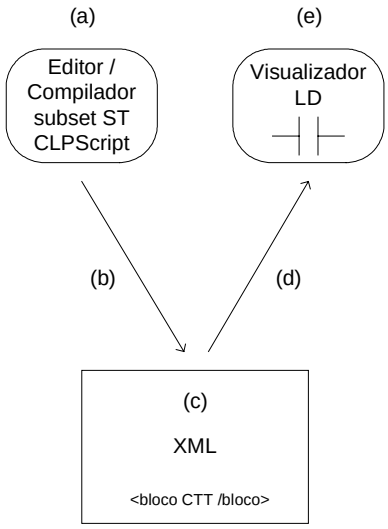


Figura 1-1: Arquitetura da Solução Proposta

A Figura 1-1 representa a arquitetura da solução proposta. Um programa CLPScript é criado (a) com um editor de texto, compilado (b) e armazenado em XML (c). Em seguida é convertido (d) e apresentado (e) em um visualizador LD.

Como forma de validação do trabalho, foi feito um estudo de caso onde o programa CLPScript compilado gera o XML, que por sua vez foi convertido para ser visualizado em um editor LD *MasterTool* ([ALT03]) da Altus S. A., conforme a seqüência mostrada na Figura 1-1.

1.6 Organização do Trabalho

Este volume está organizado em nove capítulos, sendo que o Capítulo 2 apresenta uma revisão bibliográfica de linguagens específicas de domínio, noções básicas de XML e uma introdução a norma IEC 1131-3, principalmente nos aspectos relacionados com as linguagens ST, LD e FBD e suas semelhanças.

O Capítulo 3 apresenta alguns trabalhos relacionados com controladores programáveis, linguagens de programação da IEC 1131-3 e linguagem XML.

O Capítulo 4 faz uma comparação das linguagens ST, LD e FBD da IEC 1131-3, ressaltando suas semelhanças e diferenças e aspectos de portabilidade entre elas. Além disto, neste capítulo é proposta uma representação textual para as linguagens gráficas LD e FBD, utilizada posteriormente na arquitetura XML

O Capítulo 5 define a linguagem CLPScript, usada para gerar e formalizar a arquitetura XML proposta.

O Capítulo 6 apresenta a arquitetura XML, mostrando as razões da escolha desta linguagem, definindo as *tags* e atributos XML para armazenar programas LD, FBD e CLPScript. Faz uma comparação da arquitetura XML com a linguagem IL.

O Capítulo 7 descreve a implementação da linguagem específica de domínio CLPScript que gera e formaliza a arquitetura XML proposta, através do programa CLPStoXML.

O Capítulo 8 apresenta o estudo de caso que visa mostrar que os controladores programáveis Altus e seu *software* programador conseguem ler o XML da arquitetura proposta. Apresenta o *software* XMLtoLD, desenvolvido para realizar esta prova de conceito.

O Capítulo 9 apresenta as conclusões da realização do trabalho, as contribuições e os trabalhos futuros.

2 Revisão Bibliográfica

Resumo

Este capítulo apresenta uma revisão bibliográfica sobre linguagens específicas de domínio, uma introdução aos principais conceitos do XML e uma visão geral da norma IEC 1131-3.

2.1 Linguagens Específicas de Domínio

2.1.1 Definição

As linguagens de domínio específico (DSL) são especificações ou linguagens de programação que oferecem bastante poder focado em uma particular família de problemas ([DEU00]). Em vez de ser uma linguagem genérica, uma DSL captura precisamente a semântica do domínio ([SPI00]) através de abstrações e notações apropriadas.

As DSLs são linguagens de uso específicas, ao contrário de linguagens tradicionais de uso genérico tipo C++ ([STR00]) ou Java ([MOR00]). Representam uma alternativa ao uso destas linguagens tradicionais, embora um sistema possa utilizar as duas abordagens simultaneamente, cada uma na forma mais conveniente: as linguagens tradicionais para construir os componentes ou partes do *software*, e as DSLs para fazer a integração ou conexão destes componentes ([OUS98, PEP01]).

Alguns exemplos de DSLs conhecidas são *lex yacc*, HTML, VHDL, o shell e comando *make* do Unix. Em [DEU00], Arie van Deursen apresenta diversas referências para DSLs nas áreas de engenharia de *software*, sistemas de *software*, multimídia e telecomunicações.

2.1.2 Vantagens de Uso

Os benefícios do uso de uma DSL são:

- permitem soluções expressadas no idioma e no nível de abstração do domínio do problema ([DEU00]);
- utilizam expressões concretas no domínio do conhecimento ([SPI00]), capturando as funcionalidades específicas em uma forma concreta e legível;
- permitem que os próprios especialistas do domínio possam entender, validar, modificar ou até mesmo desenvolver os programas, sem que seja necessário aprender C++ ou Java ou outra linguagem genérica qualquer;
- os programas DSL são concisos, auto documentados e podem ser reutilizados para diferentes propósitos;
- as DSLs aumentam a produtividade, confiança e a portabilidade, além de facilitar a manutenção dos sistemas ([DEU00]).

2.1.3 Implementação

A implementação de uma DSL difere da implementação com linguagens genéricas tradicionais.

Os compiladores tradicionais são tipicamente estruturados em analisador léxico, sintático, semântico, otimizador e gerador de código.

As implementações de DSL possuem escopo mais limitado e utilizam estratégias diferentes ([SPI00]). Tipicamente processam o programa fonte usando expressões regulares e não precisam de todas as etapas da compilação tradicional, como por exemplo a etapa de geração de código *assembly*.

Uma referência completa sobre compiladores pode ser encontrada em [AHO86], descrevendo as etapas de análise léxica, sintática e semântica, código intermediário, otimização e geração de código.

Em [SPI00], Diomidis Spinellis apresenta alguns design patterns ([SHA02]) para DSLs, dentre os quais cita-se *pipeline*, *lexical processing*, *language extension*, l*anguage specialization*, *source-to-source transformation* e *data structure representation*.

Um *framework* para a construção de uma variedade de DSLs é apresentado por Markus Fromahers em [FRO97], mostrando como estas linguagens podem ser construídas e porque este *framework* é apropriado para esta tarefa.

2.2 XML

XML é uma abreviação de *eXtensible Markup Language*, ou linguagem de marcação extensível. Pode ser formalmente descrita como uma metalinguagem de segunda geração: segunda geração porque é derivada de SGML (*Standard Generalized Markup Language*), norma ISO 8879; uma metalinguagem porque, assim como a SGML, XML é uma linguagem de marcação projetada para descrever outras linguagens de marcação ([TIT02]).

Linguagens de marcação compreendem todas aquelas que possibilitam a formatação de elementos por meio de *tags* e atributos como o HTML (*HyperText Markup Language*). Por ser **extensível**, o XML possibilita a criação de elementos, ou seja, você mesmo pode inventar as suas *tags*. Este é um dos fatores que tornam a linguagem preferida na transação e armazenamento de dados ([SIL01]).

Uma das principais diferenças entre o HTML e o XML é que o HTML é usado para formatação e exibição das informações, enquanto que o XML é usado para descrever e armazenar essas informações ([HAR99]).

Os padrões que compõem o XML são definidos pelo W3C (*World Wide Web Consortium*).

A flexibilidade do XML provém da possibilidade de transportar qualquer tipo de dados, mantendo-os estruturalmente coesos e inteligíveis, como binários através da estrutura de marcação "<*tag*>valor</*tag*>". Devido a esta estrutura, é também possível combinar num mesmo documento, vários objetos com tipos de dados diferentes.

O XML é considerado de grande importância na Internet e em grandes intranets porque provê a capacidade de interoperabilidade dos computadores por ter um padrão flexível e aberto e independente de dispositivo. As aplicações podem ser construídas e atualizadas mais rapidamente e também permitem múltiplas formas de visualização dos dados estruturados.

2.2.1 Documentos XML

Os documentos XML são formados por **elementos**, que são delimitados por ***tags*** de início e fim, e podem conter outros elementos e textos. Uma ***tag*** de início é composta de uma *tag* de nome e um conjunto de **atributos**. Uma *tag* de fim contém apenas o nome do elemento, precedido de '/'.

Um **atributo** é um par nome/valor separado por um sinal de igual. O valor de um atributo deve estar envolto em sinais de aspas simples ou duplas. O espaçamento entre o nome do elemento e os atributos bem como o espaçamento entre nomes e valores de atributos é irrelevante ([ARC02]).

```
<local>
   <universidade> nome="Unisinos" cidade= "São Leopoldo" </universidade>
</local>
```

Figura 2-1: Exemplo de *Tags* e Atributos em XML

A Figura 2-1 mostra dois exemplos de *tags*, "local" e "universidade", com suas marcas de início e fim. Mostra também dois atributos da *tag* "universidade", chamados "nome" e "cidade", com seus valores, respectivamente "Unisinos" e "São Leopoldo".

Um documento XML deve seguir algumas regras básicas de formatação. O aspecto mais importante é o aninhamento correto. Um elemento está corretamente aninhado se as suas *tags* de início e fim estão dentro do mesmo elemento pai.

Os documentos XML começam sempre pela declaração <?xml version="1.0"?> .

2.2.2 Validação

Um documento XML é **bem formado** se ele seguir as regras de sintaxe de formatação resumidas na seção anterior. Um documento XML é **válido** quando é bem formado e segue as regras de validação definidas pelas DTDs (*Document Type Definitions*).

Uma DTD é um conjunto de regras que define que tipo de dados e entidades fazem parte de um documento XML, a sua ordem e atributos, e seu conteúdo. Essas regras podem ser definidas externamente, em arquivos DTD, ou internamente, dentro do próprio arquivo XML. Um analisador de documentos pode checar os dados do arquivo

XML analisando as regras contidas na DTD para ter certeza de que está estruturado corretamente. As DTD são opcionais.

As DTDs são uma ferramenta excelente para uma enorme variedade de aplicativos em que o grau de formalismo necessário é moderado e definições como “o conteúdo são dados de caracteres” são suficientes. Outros aplicativos, porém, requerem definições muito mais precisas, tais como “O conteúdo são dados de caracteres representando um ponto flutuante menor que 9.69”. Para tais aplicativos, o W3C criou o XML *Schema* ([ARC02]).

O XML *Schema* é uma linguagem para a definição de estruturas complexas e tipos de dados para documentos XML. Usando o XML *Schema*, pode-se definir, derivar, expandir e compor construções muito precisas fornecendo, assim, uma semântica de alto nível no documento e uma validação automática maior a partir no nível do analisador sintático ([ARC02]).

2.3 A Norma IEC 1131-3

A IEC 1131-3 ([IEC93]) é uma norma internacional para padronização de linguagens de controladores programáveis (CP) na área de automação industrial. Foi desenvolvida em resposta a pressões da indústria por maior compatibilidade entre os CPs. O objetivo foi especificar uma linguagem portável, extensível, que eliminasse as barreiras de *software* proprietários e seus custos de treinamento associados. Esta norma também é referenciada como IEC 61131-3.

A norma define cinco linguagens, sendo duas gráficas, *Ladder Diagram* (LD) e *Function Block Diagram* (FBD), duas textuais, *Instruction List* (IL) e *Structured Text* (ST), e uma quinta que serve como base de organização para as demais, *Sequential Function Chart* (SFC).

Variando de baixo até alto nível, estas cinco linguagens oferecem ao programador diversos recursos para que ele use suas habilidades conforme os requisitos de cada aplicação. Diferentes partes da aplicação podem ser programadas em diferentes linguagens, formando um único programa aplicativo a ser executado no CP.

Os principais benefícios que a norma traz são ([LEW95, BON97]):

- melhora a qualidade do *software* aplicativo, utilizando-se *software* estruturado e modernas técnicas de programação;
- aproveita técnicas existentes agregando ferramentas de alta produtividade;
- redução do tempo de treinamento, teste e manutenção;
- uso de módulos de *software* desenvolvidos por terceiros;
- estruturas de dados poderosas;
- desenvolvimento independente de fabricante.

2.3.1 Elementos Comuns

A norma define elementos que são comuns às 5 linguagens padronizadas. Esta seção apresenta os tipos de dados existentes, as variáveis e também os *Program Organization Units* (POU). Conforme a norma, um **POU** é um **Programa**, uma **Função** ou um **Bloco de Função** ([LEW95]), descritos mais adiante neste capítulo.

A norma define uma série de funções e Blocos de Função padronizados para tarefas típicas de controle. Além destas, a norma permite que sejam criados funções e blocos de função definidos pelo usuário, podendo então ser utilizados da mesma forma que os padronizados.

2.3.1.1 Tipos de Dados

A norma prove alguns tipos de dados básicos para lidar com os valores típicos de variáveis de aplicações industriais, mostrados na Tabela 2-1.

Palavra-Chave	Descrição	Bits	Faixa
BOOL	Boolean	1	FALSE ou TRUE
SINT	Short integer	8	- 128 ..127
USINT	Unsigned Short integer	8	0 .. 255
INT	Integer	16	- 32.768 .. 32.767
UINT	Unsigned integer	16	0 .. 65.535
DINT	Double Integer	32	$-2^{31} .. +2^{31}-1$
REAL	Real	32	$\pm 10^{\pm 38}$
LREAL	Long Real	64	$\pm 10^{\pm 308}$
TIME	Time Duration		
DATE	Calendar Date		
STRING	Character Strings		

Tabela 2-1: Tipos de Dados Básicos

Podem ser definidos novos tipos de dados a partir dos tipos básicos. Também existe definições para estruturas, enumerados, *sub-ranges* e *arrays* de uma ou mais dimensões.

A declaração de tipos possuiu a forma mostrada na Figura 2-2.

```
TYPE
        <lista de novos tipos...> : <tipo_de_dado_básico>;
        PRESSURE:       REAL;
END_TYPE
```

Figura 2-2: Declaração de Novos Tipos

2.3.1.2 Variáveis

Conforme a norma, uma variável é uma área de memória que armazena um determinado tipo de dado. Todas as variáveis devem ser declaradas antes de serem usadas. Isto é feito no início da função ou do programa principal.

A declaração de variáveis possui a forma mostrada na Figura 2-3.

```
VAR
        [CONSTANT] <lista de variáveis...> : <tipo_de_dado_1> [ := <valor> ] ;
        A,B,C: REAL;
END_VAR
```

Figura 2-3: Declaração de Variáveis

O escopo das variáveis é limitado ao POU em que foram declarados, permitindo que os nomes possam ser reutilizados em outras partes sem nenhum conflito, eliminando uma possível fonte de erros.

2.3.1.3 Função

Uma função é um trecho de código que poderá ser chamado diversas vezes pelo programa principal, por um bloco de função ou por outras funções. Ela se caracteriza por ter diversas entradas, mas apenas uma saída. Quando chamada com os mesmos parâmetros de entrada, a função sempre retorna o mesmo valor na saída.

Uma função pode ser declarada da forma mostrada na Figura 2-4.

```
FUNCTION <nome> : <tipo_retorno>
        [ VAR_INPUT ... END_VAR ]
        [ VAR ... END_VAR]
        <corpo_da_função>
END_FUNCTION
```

Figura 2-4: Declaração de Função

O <*corpo_da_função*> corresponde ao código da função, que pode ser escrito em qualquer uma das 5 linguagens. O código da função irá conhecer somente as suas variáveis declaradas em VAR_INPUT e VAR. Não será possível para a função conhecer as variáveis declaradas em outras funções ou no programa principal.

Antes que a função retorne à rotina que a chamou, deverá ser atribuído o valor de retorno.

2.3.1.4 Bloco de Função

Um Bloco de Função é um trecho de código que poderá ser chamado diversas vezes pelo programa principal, por outro bloco de função ou por outras funções. Ele se caracteriza por ter diversas entradas, e uma ou mais saídas. Quando chamada com os mesmos parâmetros de entrada, um bloco de função pode retornar valores diferentes nas saídas, pois estas dependem não somente das entradas mas também de variáveis que determinam seu estado interno.

O bloco de função é equivalente a um circuito integrado (CI), representando uma função especializada de controle. Ele possui internamente dados e algoritmo, podendo portanto manter o histórico do passado. Possuem uma interface bem definida e variáveis internas, da mesma forma que um CI caixa preta.

Um bloco de função pode ser declarado da forma mostrada na Figura 2-5.

```
FUNCTION_BLOCK <nome>
        [ VAR_INPUT ... END_VAR ]
        [ VAR_OUTPUT ... END_VAR ]
        [ VAR ... END_VAR]
        <corpo_do_bloco_de_função>
END_FUNCTION_BLOCK
```

Figura 2-5: Declaração de Bloco de Função

O <corpo_do_bloco_de_função> corresponde ao código do bloco de função, que pode ser escrito em qualquer uma das 5 linguagens.

Os blocos de função precisam ser instanciados para que possam ser chamados, pois cada instancia armazena seu próprio conjunto de variáveis de estado interno.

2.3.1.5 Programas

Os programas são a unidade de execução principal em um CP compatível com IEC 1131-3. Um programa é muito similar a um bloco de função, conforme mostra a Figura 2-6.

```
PROGRAM <nome>
        [ VAR_INPUT ... END_VAR ]
        [ VAR_OUTPUT ... END_VAR ]
        [ VAR ... END_VAR]
        <corpo_do_programa>
END_PROGRAM
```

Figura 2-6: Declaração de Programa

O <corpo_do_programa> corresponde ao código do programa, que pode ser escrito em qualquer uma das 5 linguagens da norma.

2.3.2 A Linguagem LD

Os controladores programáveis surgiram para substituir painéis de controle a relés. Neste contexto, uma linguagem de programação que mais se aproximasse da experiência de técnicos e engenheiros seria a solução mais adequada para desenvolvimento de programas aplicativos de CPs ([ALT03]).

A linguagem LD ([IEC93]), *Ladder Diagram*, também conhecida por linguagem de diagrama de relés, é uma linguagem gráfica onde os elementos principais de seu diagrama são contatos e bobinas de vários tipos, muito semelhante à linguagem de descrição dos painéis de controle a relé.

A linguagem LD foi especificada pela IEC considerando os símbolos e terminologias mais utilizados pelos fabricantes de CPs. A principal vantagem da utilização deste tipo de linguagem é seu rápido aprendizado, pois assemelha-se muito com os esquemas elétricos convencionais.

Um diagrama LD sempre possui uma **barra de energização** vertical a esquerda que intuitivamente fornece energia para os contatos e bobinas ao longo das ligações horizontais à direita. O fluxo de "energia" da barra é sempre da esquerda para a direita.

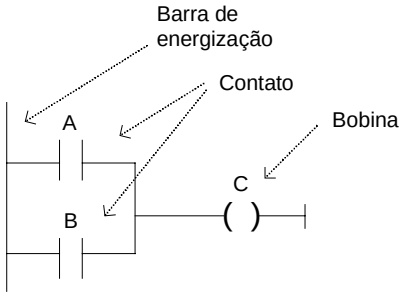


Figura 2-7: Linguagem LD

Considera-se **energizado** o valor lógico 1, e **desenergisado** o valor lógico 0. A Figura 2-7 mostra um diagrama LD com a barra de energização, dois contatos e uma bobina. Esta simbologia representa que a variável "C" associada à bobina só vai ser energizada se um dos dois contatos "A" ou "B" estiver ativo.

A Figura 2-8 mostra os principais tipos de contatos e bobinas da LD ([FES00]):

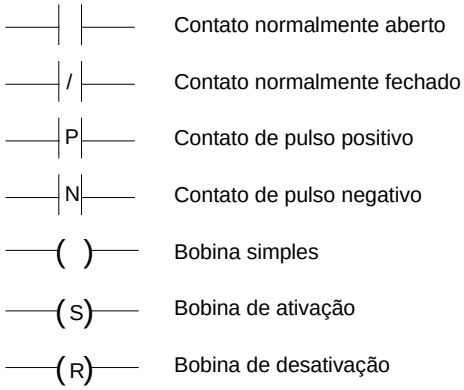


Figura 2-8: Elementos da Linguagem LD

Um contato normalmente aberto fornece o valor lógico 1 em sua saída quando sua entrada está energizada e sua respectiva variável ("A" e "B" na Figura 2-8) valer 1. Um contato normalmente fechado fornece o valor lógico 1 em sua saída quando sua entrada está energizada e sua respectiva variável valer 0.

Um contato ou relé de pulso positivo fornece o valor 1 em sua saída para a transição de 0 para 1 na sua entrada. O contato de pulso negativo opera de maneira inversa.

A bobina simples atribui o valor 1 para sua respectiva variável (“C” na Figura 2-8) se sua entrada estiver energizada, caso contrário atribui o valor 0. A bobina de ativação ou bobina liga atribui o valor 1 caso sua entrada esteja energizada, caso contrário não atribui nada, o valor da variável associada fica inalterado. A bobina de desativação ou bobina desliga atribui o valor 0 caso sua entrada esteja energizada.

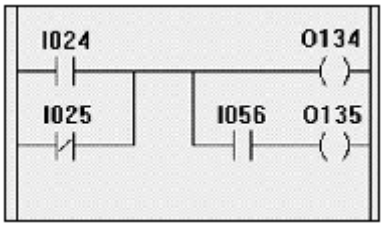


Figura 2-9: Exemplo de LD

A linguagem LD é indicada para aplicações de intertravamento[1] e lógicas combinatórias em geral. A Figura 2-9 mostra um exemplo de LD com dois contatos normalmente aberto (I024 e I056), um contato normalmente fechado (I025) e duas bobinas simples (O134 e O135).

2.3.3 A Linguagem FBD

A linguagem FBD ([IEC93]), *Function Block Diagram*, é uma linguagem gráfica que permite descrever um processo por um conjunto de blocos interconectados de maneira semelhante a um circuito eletrônico.

A norma IEC1131-3 ([IEC93]) inclui uma variedade de blocos de função padronizados para diversas operações, e é possível incluir novos blocos definidos pelo usuário.

[1] Intertravamento, no contexto de automação industrial, é um ação executada em um equipamento em função de determinadas condições do processo no qual se encontra. Por exemplo, pode-se dizer que uma lógica de intertravamento deve desligar o motor X quando o nível do tanque tornar-se menor que Y, ou que o intertravamento deve abrir imediatamente a válvula de segurança X caso a pressão da caldeira ultrapasse o valor limite Y.

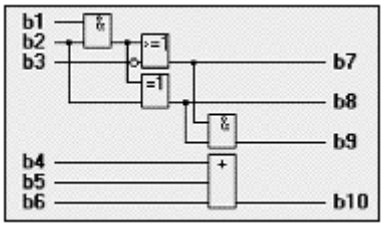


Figura 2-10: Exemplo de FBD

A Figura 2-10 mostra um exemplo de FBD. Os blocos são conectados representando o fluxo dos sinais entre os elementos, sendo que sinais conectados devem ter o mesmo tipo de dado. O fluxo de dados é sempre da esquerda para a direita. As entradas são representadas na borda esquerda do bloco e as saídas são representadas na borda direita do bloco.

2.3.4 A Linguagem ST

A linguagem ST ([IEC93]), *Structured Text*, é uma linguagem textual de alto nível, poderosa, utilizada para implementar procedimentos complexos que não podem ser facilmente expressos pelas linguagens gráficas.

É uma linguagem fortemente tipada, semelhante ao PASCAL, possuindo as seguintes características :

- comandos de atribuição (:=);
- comandos de seleção IF, THEN, ELSE, CASE OF;
- comandos de iteração FOR, WHILE e REPEAT;
- chamadas de funções e de blocos de função;
- comandos de controle RETURN e EXIT.

A Figura 2-11 mostra um trecho de programa em linguagem ST.

```
if (level <= level_max)
then
   out_valve := false;
   m_vlv := (vlv23 + dbh18) / 2;
else
   alarm_level := true;
end_if;
```

Figura 2-11: Exemplo de ST

2.3.5 A Linguagem IL

A linguagem IL ([IEC93]), *Instruction List*, é uma linguagem textual de baixo nível utilizada para implementar procedimentos simples. É constituída por instruções no estilo *assembly*, conforme mostra o trecho de código da Figura 2-12:

```
start_cmd:  LD      bi101
            ADD     10
mul_ope:    MUL(    i_bcmd
            SUB     bo100
            )
            ST      bcmd
            JMPNC   mul_op
```

Figura 2-12: Exemplo de IL

Um programa IL é uma lista de instruções, onde cada instrução tem um *label*, um operador, um ou dois operandos e um comentário opcional.

2.3.6 A Linguagem SFC

A linguagem SFC ([IEC93]), *Sequential Function Chart*, é uma linguagem gráfica utilizada para descrever o comportamento seqüencial do programa. Derivada de *Redes de Petri* ([MOR01]), é a linguagem ideal para se escrever máquinas de estado.

Através da SFC pode-se dividir um programa em (a) passos, (b) ações e (c) transições bem definidas, formando um algoritmo seqüencial de controle. A Figura 2-13 mostra um esboço de programa em SFC.

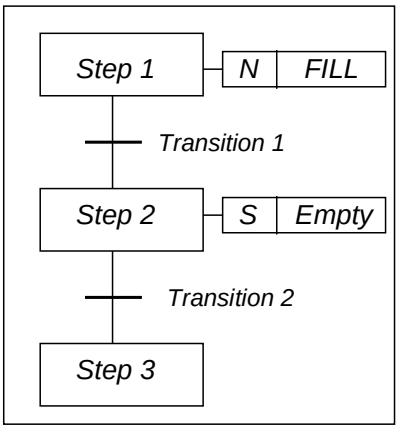


Figura 2-13: Exemplo de SFC

Enquanto a SFC define os blocos de controle e a forma que estão interconectados, qualquer outra das 4 linguagens pode ser utilizada para codificar as ações associadas a cada passo e as condições de cada transição.

2.4 Conclusão

Este capítulo apresentou uma revisão bibliográfica sobre linguagens específicas de domínio, uma introdução aos principais conceitos do XML e uma visão geral da norma IEC 1131-3.

Foram apresentadas noções básicas da linguagem XML que serão importantes no desenvolvimento do trabalho, podendo-se citar os conceitos de *tags* e atributos, a diferença entre um documento bem formado e um documento válido, as DTDs e principalmente a característica de ser uma linguagem extensível, onde nós mesmos podemos criar nossos próprios *tags*.

A norma IEC 1131-3 ([IEC93]) surgiu com o objetivo de padronizar a área de linguagens de programação para controladores programáveis. Foram apresentadas as 5 linguagens da norma, os pontos em comum, as características de cada uma e aplicações apropriadas.

Os conceitos de linguagens específicas de domínio foram aproveitados na definição das linguagens CLPScript e ProLD, criadas neste trabalho. O XML aparece no centro da arquitetura proposta no Capítulo 6 e a norma IEC 1131-3 é um dos focos deste trabalho.

O próximo capítulo apresenta alguns trabalhos relacionados com controladores programáveis, linguagens de programação da IEC 1131-3 e linguagem XML

3 Trabalhos Relacionados

Resumo

Este capítulo apresenta trabalhos e projetos relacionados com controladores programáveis, linguagem XML e linguagens de programação da norma IEC 1131-3.

3.1 Linguagem SIPN

Em [FRE01], Georg Frey apresenta a linguagem ***Signal Interpreted Petri Net*** (SIPN), utilizada para a programação de controladores programáveis.

SIPN é uma linguagem gráfica capaz de descrever o comportamento seqüencial e concorrente do processo a ser controlado. É baseada em um tipo especial de *Redes de Petri*, e utiliza os símbolos mostrados na Tabela 3-1:

Símbolo	Descrição
	Posição / lugar
	Transição
	Arco orientado

Tabela 3-1: Símbolos da Linguagem SIPN

Em SIPN as **transições** estão associadas a uma função booleana dos sinais de entrada gerando a condição de transição, enquanto que as **posições** estão associadas a uma função que atribui valores aos sinais de saída.

O comportamento dinâmico do SIPN é dado pelo fluxo de **marcas** (*tokens*) através da rede. Este fluxo é realizado pelo disparo das transições, removendo as marcas das pré-

posições e inserindo-as em cada uma das pós-posições. Para que um disparo de transição aconteça, existem quatro regras:

- a transição está habilitada se todos as pré-posições estão marcadas e todas as pós-posições estão "não marcadas";
- uma transição é disparada imediatamente se está habilitada e se a condição de transição está atendida;
- todas as transições que podem disparar disparam imediatamente;
- o processo de disparo é iterativo e continua até que uma marcação estável é atingida.

A Figura 3-1 mostra um exemplo de SIPN para controle de uma máquina furadeira. Durante um ciclo de operação, o objetivo do controle é ligar o motor da furadeira, comandar a descida da broca para executar a perfuração e, após concluída, comandar a subida de volta a posição inicial.

Para isto, o sistema que possui uma saída digital para ligar/desligar seu motor, uma para movimentá-la para cima e outra para baixo. Possui também uma entrada digital para indicar que a máquina atingiu a posição limite inferior e outra para a posição superior (o exemplo original apresenta outras entradas e saída, omitidas do texto em razão de maior clareza).

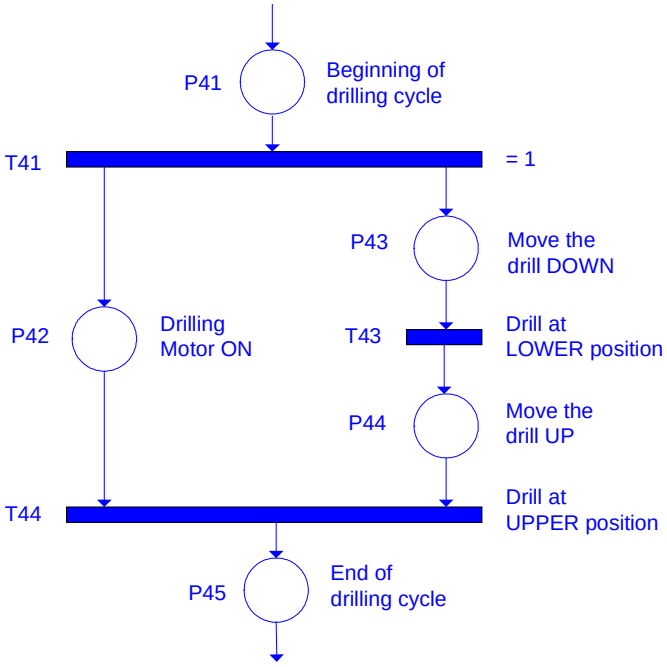


Figura 3-1: Exemplo de SIPN de Máquina Furadeira

Durante a execução da rede, após a transição T41 (ver Figura 3-1), as posições P42 e P43 são marcadas (semelhante a um estado ativo da linguagem SFC). Em P42 o motor é ligado e em P43 a saída “Mover para baixo” é acionada. Tudo permanece nesta situação até que a transição T43 seja disparada, o que acontece quando a máquina atinge o limite inferior e a entrada digital correspondente é acionada. A posição P43 é então desmarcada, a P44 é marcada e a P42 fica inalterada. A posição P44 marcada desliga a saída “Mover p/ baixo” e liga a “Mover p/ cima”. Novamente tudo permanece nesta situação até que a próxima transição, T44, seja disparada pela entrada digital que indica que a máquina atingiu o limite superior, quando então o motor é desligado e encerra-se o ciclo.

3.1.1 Geração de Código em Instruction List (IL)

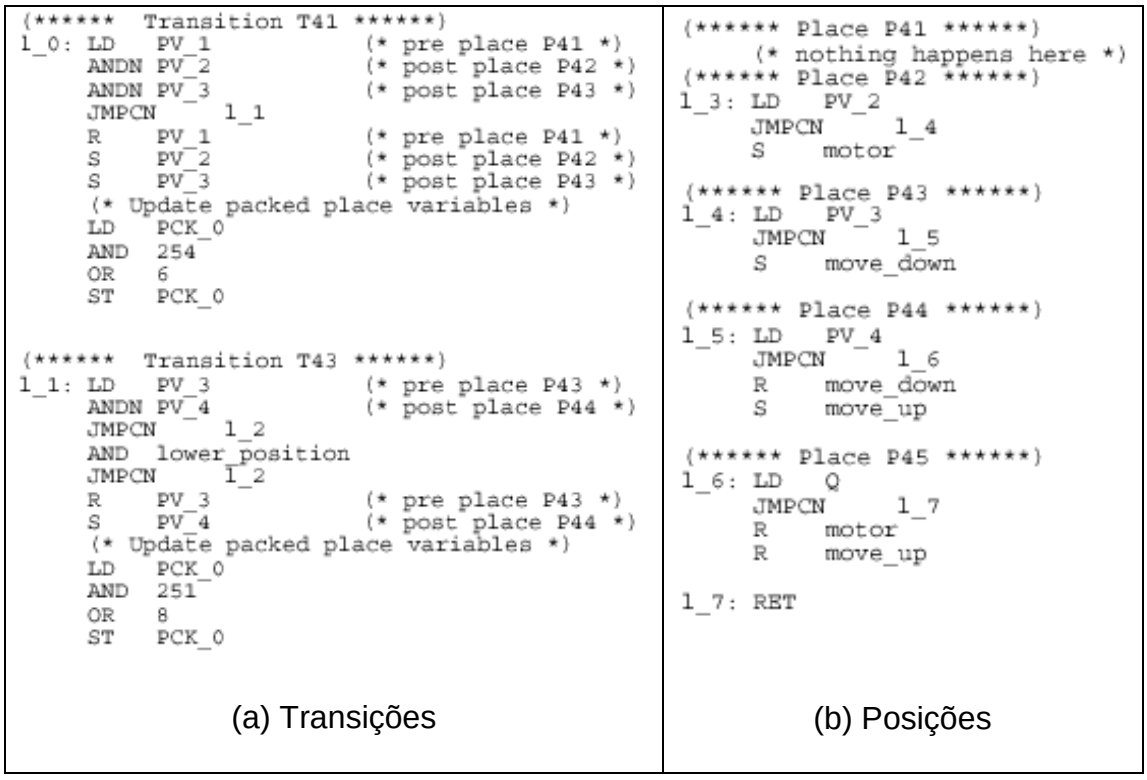
A linguagem SIPN é convertida na linguagem *Instruction List* (IL) da IEC 1131-3. Conforme Georg Frey, a implementação direta de um compilador iria funcionar para apenas um tipo especial de CP, enquanto que convertendo-se para a linguagem IL pode-se executar o programa em qualquer CP que atenda a especificação IEC 1131-3.

A Figura 3-2 mostra a declaração das entradas e saídas digitais do exemplo apresentado. Esta forma de declaração é a mesma para todas as linguagens da IEC 1131-3.

```
VAR_GLOBAL
     start_button      at %IX0.0 : bool ;
     part_in_position  at %IX0.1 : bool ;
     lower_position    at %IX0.2 : bool ;
     upper_position    at %IX0.3 : bool ;
     active            at %QX0.0 : bool ;
     motor             at %QX0.1 : bool ;
     move_down         at %QX0.2 : bool ;
     move_up           at %QX0.3 : bool ;
END_VAR
```

Figura 3-2: Declaração das Entradas e Saídas em IL

A Figura 3-3 mostra o trecho de código IL gerado para as transições T41 e T43 e para as posições P41, P43, P44 e P45.



```
(******  Transition T41 ******)
l_0: LD   PV_1            (* pre place P41 *)
     ANDN PV_2            (* post place P42 *)
     ANDN PV_3            (* post place P43 *)
     JMPCN     l_1
     R    PV_1            (* pre place P41 *)
     S    PV_2            (* post place P42 *)
     S    PV_3            (* post place P43 *)
     (* Update packed place variables *)
     LD   PCK_0
     AND  254
     OR   6
     ST   PCK_0


(******  Transition T43 ******)
l_1: LD   PV_3            (* pre place P43 *)
     ANDN PV_4            (* post place P44 *)
     JMPCN     l_2
     AND  lower_position
     JMPCN     l_2
     R    PV_3            (* pre place P43 *)
     S    PV_4            (* post place P44 *)
     (* Update packed place variables *)
     LD   PCK_0
     AND  251
     OR   8
     ST   PCK_0
```

(a) Transições

```
(****** Place P41 ******)
     (* nothing happens here *)
(****** Place P42 ******)
l_3: LD   PV_2
     JMPCN     l_4
     S    motor

(****** Place P43 ******)
l_4: LD   PV_3
     JMPCN     l_5
     S    move_down

(****** Place P44 ******)
l_5: LD   PV_4
     JMPCN     l_6
     R    move_down
     S    move_up

(****** Place P45 ******)
l_6: LD   Q
     JMPCN     l_7
     R    motor
     R    move_up

l_7: RET
```

(b) Posições

Figura 3-3: Código de SIPN gerado em linguagem IL

3.1.2 A Ferramenta de Edição do SIPN

O uso do SIPN é disponibilizado através de uma ferramenta para edição, visualização e compilação. Esta ferramenta, *SIPN-Editor* ([SIP01]), foi implementada usando o DiaGen, *Diagram Editor Generator*, uma framework para geração de editores gráficos desenvolvido na *Universität Erlangen-Nürnberg*, Alemanha ([DIA01]). A Figura 3-4 mostra o *SIPN-Editor*, que é disponibilizado como um *Applet Java* que pode ser executado em browsers padrão de qualquer PC conectado à internet.

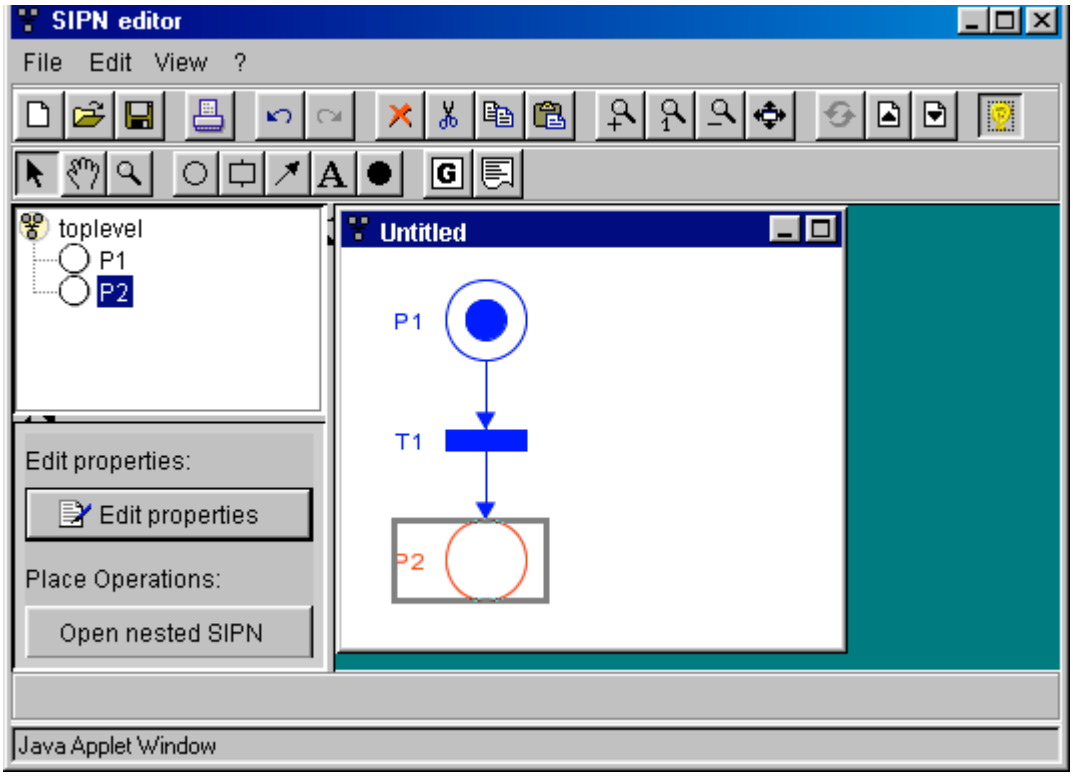


Figura 3-4: A Ferramenta SIPN-Editor

O principal componente da ferramenta é o editor gráfico que permite a criação de redes SIPN de maneira direta. As transições e estados podem ser inseridos s partir de ícones na *tool bar* do programa, da mesma forma que os arcos de conexão. As ações de cada estado e transição podem ser inseridos através da propriedade destes objetos.

O editor verifica a validade da rede sendo criada, gerando um *feedback* visual em caso de erro, conforme pode-se observar pela cor vermelha do objeto P2 na Figura 3-4. Após a edição de um programa SIPN completo, o código IL pode ser gerado e está pronto para ser executado em qualquer CP compatível.

3.1.3 SIPN e XML

A ferramenta SIPN-Editor oferece uma interface baseada em XML para exportar a rede SIPN editada em um formato de trocas que pode ser utilizado por outros programas. O formato selecionado é o *Petri Net Markup Language* (PNML), que se encontra em definição ([WEB02]).

O PNML é uma proposta de um formato de trocas baseado em XML para *Redes de Petri*. É um formato aberto que distingue as diferentes características de todos os tipos de *Redes de Petri* e também características específicas de tipos especiais de redes.

A Figura 3-5 mostra um exemplo de uma rede SIPN com uma posição, uma transição e um arco e o XML de armazenamento correspondente em PNML:

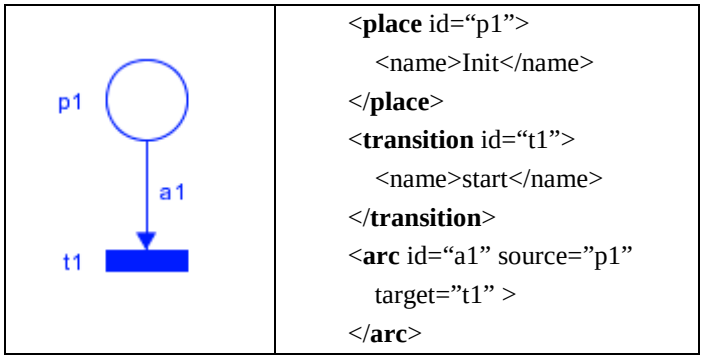


Figura 3-5: Um exemplo de PNML

3.2 PLCOpen

PLCopen ([PLC92]) é uma associação mundial não vinculada a fabricantes ou produtos que visa resolver tópicos relacionados com programas de controle e normas internacionais na área de automação industrial.

Possui vários comitês técnicos, mas o foco principal de atuação é relacionado com a norma IEC 1131-3. Possui diversos membros associados em diversos países, como mostra a Figura 3-6. Fundada em 1992, possui escritório central na Holanda e escritórios de suporte no Canadá, Japão e outros países (PLC92]).

• **ABB Sweden** • **Altus Information Brazil** • Atos industrial Automation Brazil • ATS International The Netherlands • Baumüller Germany • Beckhoff Germany • Berger Lahr Germany • Bosch Rexroth Germany • B&R Industrie-Elektronik Austria • Control Techniques United Kingdom • Danfoss Drives A/S • Digital Electronics Japan • Elau Germany • Fuji Electric Japan • Giddings & Lewis CMS USA • Honeywell SMS The Netherlands • ISA Triplex Canada • Infoteam Software Germany • Keba Austria • Kirchner Soft GmbH • Kloeckner Tevopharm Netherlands • Kuka Germany	• KW – Software Germany • Lenze Germany • LG Industrial Systems South Korea • Matsushita Electric works Germany • Mitsubishi Electric Europe Germany • Nyquist Industrial Control The Netherlands • Omron Co. Japan • Online Development USA • Parker Automation Germany • Philip Morris USA • Phoenix Contact Germany • **Rockwell Software USA** • Selectron Systems AG Switserland • Softing Germany • **Siemens Germany** • SMS Demag AG Germany • **Schneider Automation Germany** • Team Spain • Teco Czech Republic • Triconex USA • Wago USA • **Yokogawa Electric Japan**

Figura 3-6: Empresas Membro da PLCOpen

A PLCOpen promove o ambiente da norma IEC 1131-3, desenvolve diretrizes comuns de implementação e níveis de conformidade, e define laboratórios de certificação. Os membros da PLCopen são parceiros na definição e uso desta norma.

Os comitês técnicos existentes são:

- TC1 – Padronizações;
- TC2 – Funções: define bibliotecas comuns de funções e blocos de função para áreas específicas;
- TC3 – Certificação e Testes de Conformidade;
- TC4 – Comunicações: redes de campo Profibus, CANOpen;
- TC5 – “Safe Software”: relacionado com as normas IEC 61508 e 61511;
- **TC6 – XML.**

A próxima seção aborda os trabalhos do comitê técnico TC6, sobre XML.

3.2.1 Comitê Técnico TC6: Interface XML

A necessidade de representação textual padronizada de linguagens gráficas relaciona-se com o objetivo deste comitê técnico: disponibilizar as informações lógicas das linguagens em um formato XML, incluindo opcionalmente características gráficas.

O TC6 ([PLC03]) trabalha na definição de *schemas* XML para todas as linguagens da IEC 1131-3. Os trabalhos encontram-se em desenvolvimento.

3.3 Linguagem VHDL

A linguagem VHDL é uma linguagem de descrição de hardware, surgida a partir de diversos trabalhos efetuados para aumentar o nível de abstração do projeto de sistemas digitais. Este trabalhos foram motivados pela necessidade do mercado de lançar novos produtos em tempos cada vez menores e pelo rápido avanço tecnológico que vem permitindo circuitos com mais e mais componentes ([CAR01]).

A VHDL pode ser vista como uma DSL no domínio de "descrição de hardware". VHDL significa *VHSIC Hardware Description Language*, e VHSIC significa *Very High Speed Integrated Circuits.*

Originalmente a VHDL era voltada para a descrição e simulação de circuitos digitais e sistemas eletrônicos. Atualmente ela também é voltada para a **síntese** de *chips* customizáveis por *software* pelo usuário. Em [CAR01], Luigi Carro cita as seguintes vantagens no uso de uma linguagem de descrição de hardware :

- permitem maior poder de abstração ao projetista, tornando possível que projetos maiores possam ser desenvolvidos;
- o uso de síntese torna o projeto independente das tecnologias de fabricação de *chips*. Assim, um projeto pode beneficiar-se da última fronteira de tecnologia um migrar para outra técnica de implementação;
- o uso de síntese melhora a produtividade e perde muito pouco de qualidade em relação a um projetista humano. A síntese é rápida, vários estilos de descrição e várias soluções podem ser tentadas em curto espaço de tempo, o que é impossível em um projeto manual;

- facilidades de manutenção, pois a linguagem serve também para documentar o hardware sintetizado.

A seguir é mostrado um exemplo de um circuito eletrônico simples e a descrição VHDL correspondente de um de seus componentes.

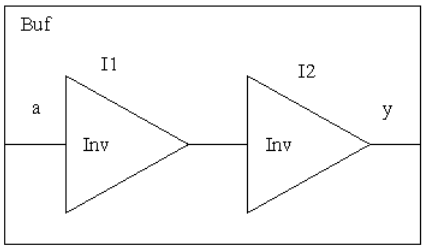


Figura 3-7: Circuito Eletrônico com Dois Inversores

A Figura 3-7 mostra um circuito eletrônico com dois inversores conectados. Um inversor é um componente eletrônico que ativa sua saída quando sua entrada não está ativa, e desativa a saída quando a entrada é ativada.

```
Entity inversor is port
        (       in1 :  in std_logic;
                out1 : out std_logic
        );
end inversor;

architecture comportamento  of inversor is
BEGIN
        out1  <=  not  in1;
end comportamento;
```

Figura 3-8: Linguagem VHDL

Em VHDL, uma entidade é qualquer componente VHDL que tenha um conjunto de portas de comunicação com outras entidades, e uma arquitetura é um conjunto de primitivas VHDL que fazem a efetiva descrição do hardware.

A Figura 3-8 mostra uma descrição VHDL do inversor da Figura 3-7. As palavras reservadas estão em negrito. A entidade "inversor" é definida a partir de ***entity*** como uma porta de uma entrada e uma saída. O comportamento do inversor é definido a partir da palavra reservada ***architeture***, e o símbolo '<=' é utilizado com a semântica de atribuição.

A linguagem VHDL descreve, de forma textual, os blocos gráficos e as conexões dos circuitos digitais. Pode-se fazer uma analogia desta representação com os blocos de função e suas conexões, da linguagem FBD. Os dois exemplos a seguir visam mostrar a maneira que o VHDL descreve textualmente estes blocos e conexões. Os exemplos foram implementados com o *software* MAX+PLUS II ([ALT97]), da Altera. O *software* gera o VHDL automaticamente, a partir de um diagrama lógico correto.

3.3.1 VHDL de Portas Lógicas AND, OR, XOR e NOT

A Figura 3-9 mostra um diagrama lógico com duas portas AND, uma OR, uma XOR e uma porta NOT, e as conexões entre elas. A pergunta relevante aqui é como o VHDL representa este diagrama e as conexões através de uma linguagem textual ?

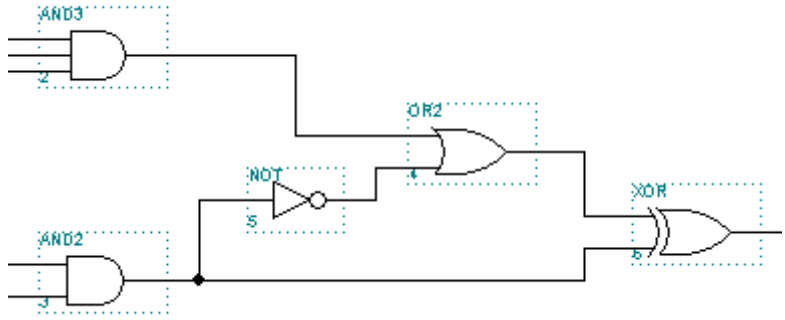


Figura 3-9: Diagrama Lógico com Portas AND, OR, XOR e NOT

O programa VHDL correspondente ao diagrama da Figura 3-9, gerado pelo MAX+PLUS II ([ALT97]), é mostrado na Figura 3-10.

```
ARCHITECTURE bdf_type OF exemplo01 IS
        signal   SYNTHESIZED_WIRE_0 :  STD_LOGIC;
        signal   SYNTHESIZED_WIRE_1 :  STD_LOGIC;
        signal   SYNTHESIZED_WIRE_5 :  STD_LOGIC;
        signal   SYNTHESIZED_WIRE_3 :  STD_LOGIC;

BEGIN
        SYNTHESIZED_WIRE_0 <= in_2 AND in_1 AND in_0;
        SYNTHESIZED_WIRE_5 <= in_3 AND in_4;
        SYNTHESIZED_WIRE_3 <= SYNTHESIZED_WIRE_0 OR SYNTHESIZED_WIRE_1;
        SYNTHESIZED_WIRE_1 <= NOT(SYNTHESIZED_WIRE_5);
        out_0 <= SYNTHESIZED_WIRE_3 XOR SYNTHESIZED_WIRE_5;
END;
```

Figura 3-10: VHDL do Diagrama de Portas Lógicas

Observa-se a partir da Figura 3-10 que o programa VHDL não instancia as cinco portas (ou blocos) utilizados no diagrama gráfico, mas sim a fiação, ou conexões, através da palavra reservada *signal*. São criadas as conexões WIRE_0, WIRE_5, WIRE_1 e WIRE_3, conforme mostra a Figura 3-11.

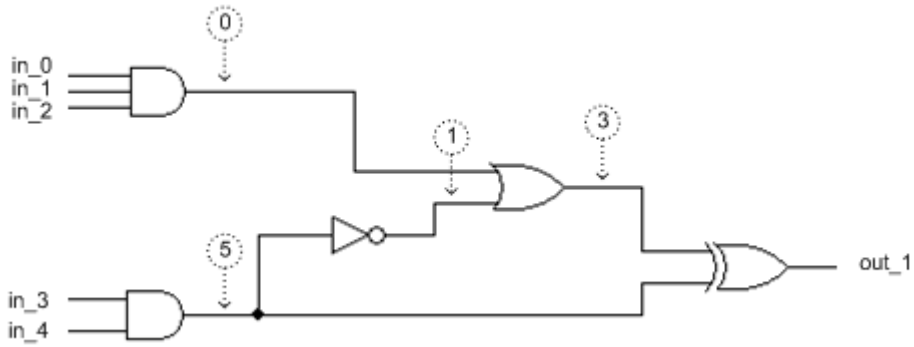


Figura 3-11: Conexões Criadas no VHDL

Cada um destas conexões possui uma linha no VHDL da Figura 3-10, onde seu valor é atribuído (no escopo do BEGIN e END). A última linha é a atribuição do valor da saída *out_1*. Observa-se que a referência às portas (ou blocos) é feita nestas expressões de atribuição. No diagrama original existem cinco portas lógicas, esperava-se encontrar cinco referências às mesmas no VHDL, mas isto não acontece. O que acontece é que as conexões é que são modeladas, e as referências às portas aparecem apenas nas expressões de atribuição das conexões.

3.3.2 VHDL de Blocos Contador e Comparador

A Figura 3-12 mostra um diagrama lógico com um bloco contador (A) e dois blocos comparadores (B e C) . A saída *q* do contador está conectada às entradas *dataa* dos comparadores (D). O VHDL representa de forma textual estes blocos gráficos e esta conexão.

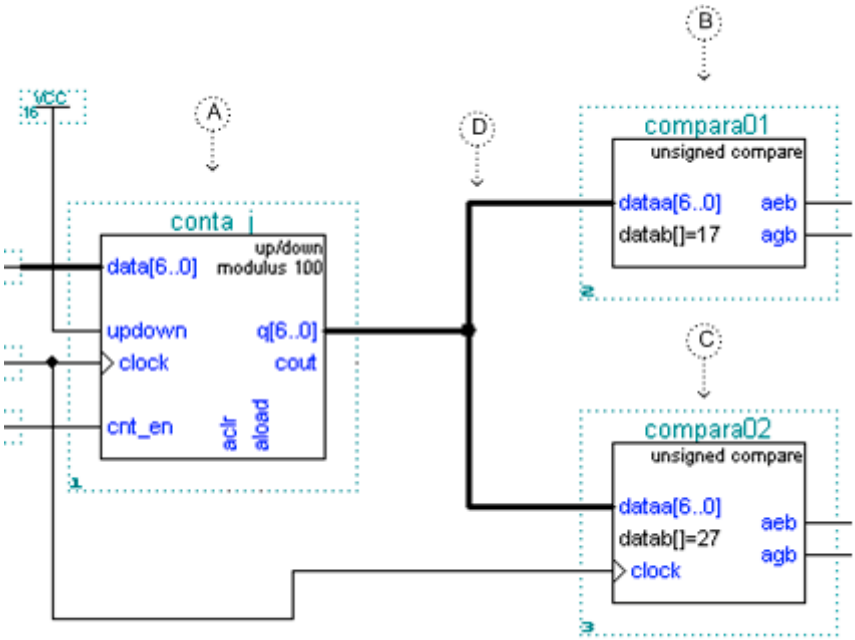


Figura 3-12: Diagrama Lógico de Contador e Comparadores

A Figura 3-13 mostra o VHDL do diagrama lógico da Figura 3-12. O bloco contador (A) e os dois blocos comparador (B e C) são instanciados com o comando PORT_MAP. A conexão entre os três é representada pelo “signal WIRE_3” (D).

```
ARCHITECTURE bdf_type OF exemplo02 IS

        signal    SYNTHESIZED_WIRE_0 :  STD_LOGIC;
        signal    SYNTHESIZED_WIRE_3 :  STD_LOGIC_VECTOR(6 downto 0);

BEGIN
        SYNTHESIZED_WIRE_0 <= '1';

        b2v_inst : conta_j
        PORT MAP
        (       updown => SYNTHESIZED_WIRE_0,
                clock => clock,
                ...
                q => SYNTHESIZED_WIRE_3
        );

        b2v_inst2 : compara02
        PORT MAP
        (       clock => clock,
                dataa => SYNTHESIZED_WIRE_3,
                ...
        );

        b2v_inst3 : compara_01
        PORT MAP
        (       dataa => SYNTHESIZED_WIRE_3,
                ...
        );
END;
```

A → conta_j (PORT MAP) ← D
C → b2v_inst2 : compara02 (PORT MAP) ← D
B → b2v_inst3 : compara_01 (PORT MAP) ← D

Figura 3-13: VHDL do Diagrama Lógico de Contador e Comparadores

3.4 Uma Avaliação das Linguagens da IEC 1131-3

Em [TOU97], Konstantinos Tourlas faz uma avaliação das linguagens FBD e ST da norma IEC 1131-3 considerando os blocos de função e identificando alguns problemas na norma.

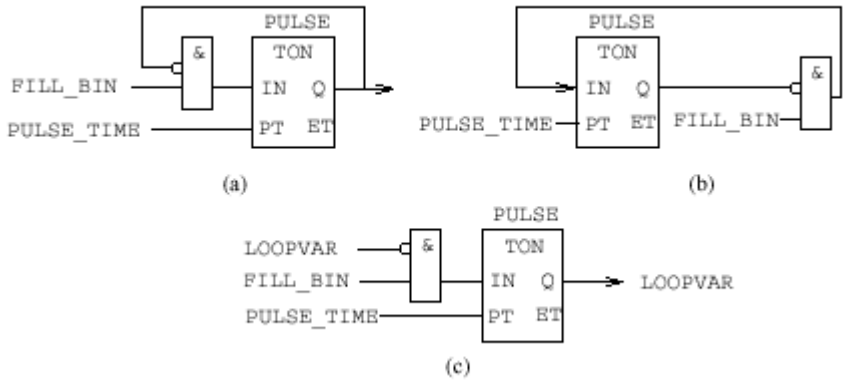


Figura 3-14: Exemplo de Loops em FBD

Considerando que os diagramas gráficos precisam ser convertidos para algum tipo de linguagem textual antes de serem compilados, Tourlas mostra que a norma usa a notação de "precedência" para se referir à ordem de execução dos blocos FBD. A Figura 3-14 mostra um diagrama com dois blocos conectados mutualmente, sendo que a parte (b) mostra o mesmo diagrama da parte (a) com a ordem de execução dos blocos invertida. Tourlas mostra que a norma trata esta situação dizendo simplesmente "o usuário deveria ser capaz de definir a ordem de avaliação selecionando *feedback variables* para formar um *loop* implícito". Este *loop* implícito está mostrado na parte (c) da Figura 3-14.

Tourlas alega que a ordem de execução não deveria ser importante para a avaliação do diagrama, e propõe uma solução baseada no que ele denomina *decorated* para provar a equivalência dos diagramas.

Em [TOU00], outro trabalho de Konstantinos Tourlas defende uma tese que conduz uma investigação sobre a representação de diagramas em linguagens específicas de domínio, usando técnicas matemáticas e estudos de caso. Nos estudos de caso foram utilizados os blocos de função da linguagem FBD da IEC 1131-3.

3.5 Conclusão

Este capítulo apresentou alguns trabalhos relacionados com controladores programáveis, linguagem XML e linguagens de programação da norma IEC 1131-3.

O SIPN é uma linguagem gráfica capaz de descrever o comportamento seqüencial e concorrente de um processo, gerando código na linguagem IL da norma IEC 1131-3 ou exportando código através de uma interface baseada em XML. Criada na *Universität Kaiserlautern*, Alemanha, cita a PNML, uma proposta de formato de trocas para *Redes de Petri* baseado em XML.

PLCopen ([PLC92]) é uma associação mundial não vinculada a fabricantes ou produtos que atua na área de controladores programáveis e normas, principalmente a IEC 1131-3. Um de seus comitês técnicos tem foco na necessidade de representação textual padronizada de linguagens gráficas e trabalha na definição de *schemas* XML para todas as linguagens da norma.

A linguagem VHDL é uma linguagem de descrição de hardware voltada originalmente para a descrição e simulação de circuitos digitais e sistemas eletrônicos e atualmente para síntese de *chips* customizáveis por *software*. Foi estudada porque era uma maneira de descrever de forma textual os blocos gráficos de uma linguagem de diagrama de blocos tipo FBD.

Todos estes trabalhos foram pesquisados por terem alguma semelhança com o desenvolvido aqui. O próximo capítulo inicia as contribuições desta dissertação, apresentando uma comparação entre as linguagens ST, LD e FBD e com uma proposta de representação textual para armazenar diagramas gráficos LD ou FBD.

4 Desenvolvimento do Trabalho

Resumo

Este capítulo faz uma comparação das linguagens ST, LD e FBD da norma IEC 1131-3, ressaltando suas semelhanças e diferenças e aspectos de portabilidade entre elas. Além disto, é proposta uma representação textual para as linguagens gráficas LD e FBD, utilizada posteriormente na arquitetura XML

4.1 Comparação Entre LD, FBD e ST

As linguagens LD e FBD são linguagens gráficas, apropriadas para a formulação de operações básicas e para controles simples que podem ser descritos com lógica booleana. Por outro lado, ST é uma linguagem textual, de alto nível, apropriada para a elaboração de módulos de *software* com conteúdo matemático e algoritmos envolvendo estruturas de dados mais complexas.

4.1.1 LD e FBD

As linguagens gráficas LD e FBD baseiam-se na (1) inserção e parametrização de contatos e bobinas (LD) ou blocos (FBD) e na (2) conexão entre eles.

Em LD e FBD, as conexões são feitas entre a saída de um contato ou bloco com a entrada de outro. Não é permitido que nenhuma entrada fique desconectada, ao contrário das saídas. Cada entrada deve estar conectada a uma única saída, ou configurada com um valor constante. Neste segundo caso, o valor constante fica armazenado na própria instancia do bloco e é sempre usado quando o bloco é executado. Uma saída pode estar conectada a mais de uma entrada, mas não é permitido conectar mais de uma entrada à mesma saída. Não é permitido que duas entradas sejam conectadas somente entre si, é preciso que pelo menos uma saída esteja conectada junto.

Na LD, a conexão sempre carrega uma informação booleana, indicando se a próxima instrução vai ou não estar energizada. Na FBD, o tipo da informação carregada na conexão é determinado pela saída que está sendo conectada, e pode ser, além de booleano, inteiro, real, data, hora e string.

Na FBD, a ordem de execução dos blocos é importante e pode afetar o resultado do programa. O mesmo vale para os programas desenvolvidos em LD. Em geral os programas LD existentes no mercado executam em uma ordem fixa, definida pelo fabricante (de cima para baixo e da esquerda para a direita), enquanto que os programas FBD possuem um parâmetro por bloco que determina a ordem de execução deste bloco em relação aos demais.

Os blocos de um programa FBD podem ser conectados de tal forma que aconteça uma realimentação. Em LD isto não é possível, pois os contatos e bobinas possuem apenas uma entrada. A Figura 4-1 mostra um exemplo de realimentação em FBD.

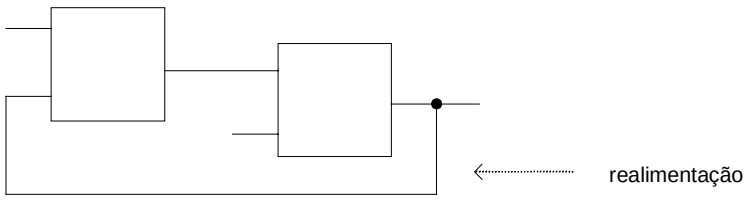


Figura 4-1: Realimentação em FBD

Em LD é possível conectar-se diretamente duas saídas de instruções contato, formando uma conexão ou ligação horizontal. Para se conectar duas saídas booleanas em FBD e obter o mesmo efeito que uma ligação horizontal em LD deve-se inserir um bloco OR. A Figura 4-2 mostra uma ligação horizontal em LD e o bloco OR equivalente em FBD.

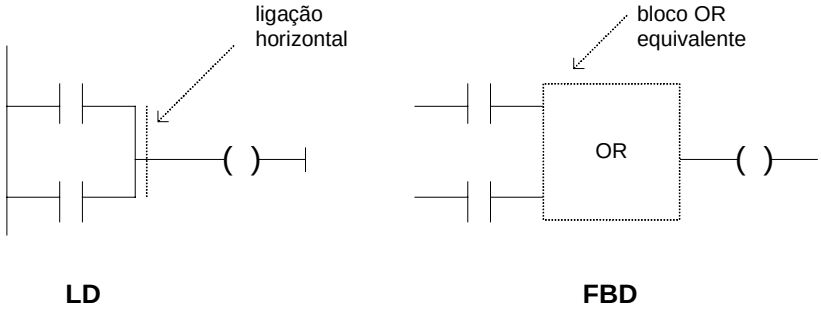


Figura 4-2: Ligação horizontal e Bloco OR Equivalente

Abstraindo-se a ligação horizontal de programas LD como um bloco OR “invisível” no diagrama original, pode-se utilizar a mesma estrutura de dados para representar os programas LD e FBD. Indo mais além, **se as instruções contato e bobina forem também abstraídas como blocos, um diagrama LD passa a ser um caso particular de um diagrama FBD**. A Figura 4-3 exemplifica esta equivalência.

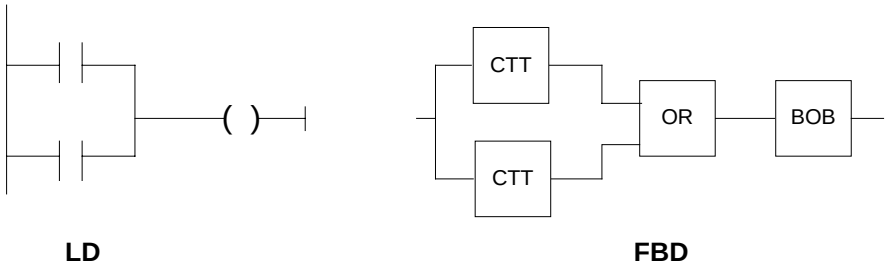


Figura 4-3: Equivalência entre Diagramas LD e FBD

4.1.2 ST, LD e FBD

A linguagem ST possui diversos recursos existentes em outras linguagens de alto nível tais como Pascal e C. Dentre estes, pode-se citar os comandos IF, FOR, WHILE, atribuição, variáveis, estruturas, chamadas de função e outros. A seguir é feita a comparação da linguagem ST com as linguagens FBD e LD.

Um comando de atribuição em ST corresponde a uma bobina (bobina liga ou desliga) no LD ou a uma chamada do bloco de movimentação no FBD, conforme mostrado na Figura 4-4. Neste caso, “A3” representa uma variável inteira cuja declaração é feita da mesma forma nas três linguagens.

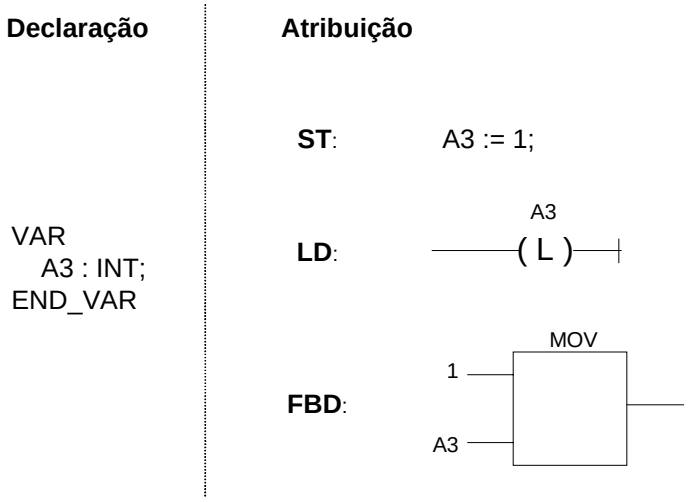


Figura 4-4: Comando Atribuição em ST, LD e FBD

Uma expressão lógica em ST pode ser representada por um diagrama de contatos equivalentes em LD, utilizando-se os conceitos de ligação em série e paralelo da engenharia elétrica ([FIT81]). A expressão OR corresponde a uma ligação dos contatos em paralelo e a expressão AND corresponde a uma ligação em série. Em FBD, a expressão lógica ST é representada diretamente pelos blocos equivalentes OR e AND, conforme mostra Figura 4-5.

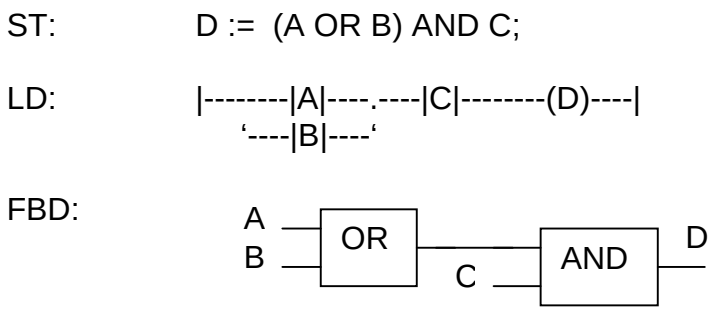


Figura 4-5: Expressão Lógica em ST, LD e FBD

Pode-se fazer uma analogia deste tipo de expressão lógica LD com um circuito elétrico: a barra de energização corresponde a fonte de alimentação e os contatos correspondem aos resistores. A Figura 4-6 mostra os contatos A, B e C, equivalentes aos resistores R1, R2 e R3, respectivamente.

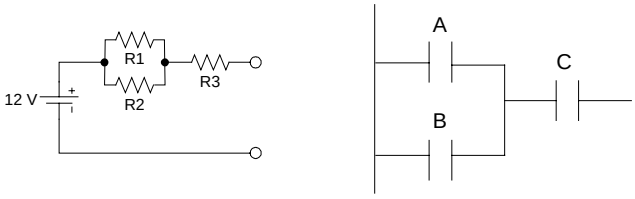


Figura 4-6: Analogia entre Circuito Elétrico e Expressão Lógica LD

De acordo com ([HAY73]), William H. Hayt mostra que o "Teorema de Thévenin" permite-nos substituir todo o circuito por uma fonte de tensão independente em série com um resistor equivalente, conforme mostra a Figura 4-7.

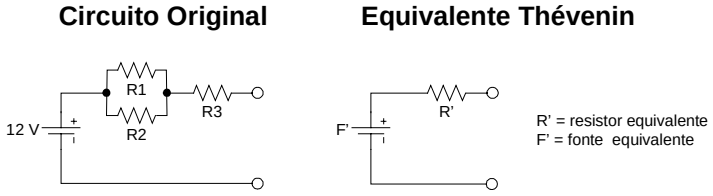


Figura 4-7: Equivalente Thévenin de Circuito Elétrico

De forma análoga, uma expressão LD pode ser representada pelo seu equivalente Thévenin, conforme mostra a Figura 4-8.

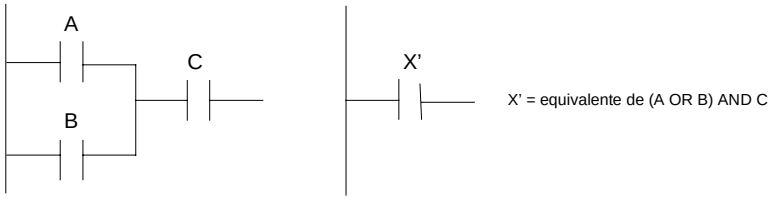


Figura 4-8: Equivalente Thévenin de Expressão Lógica

Os comandos IF da ST são definidos pela sintaxe *IF <expressão> THEN <comando>*. A expressão lógica de todo comando IF pode ser representada pelo equivalente Thévenin. Os comandos de atribuição dentro de IFs podem ser representados em LD por bobinas liga ou desliga, e em FBD por blocos de movimentação.

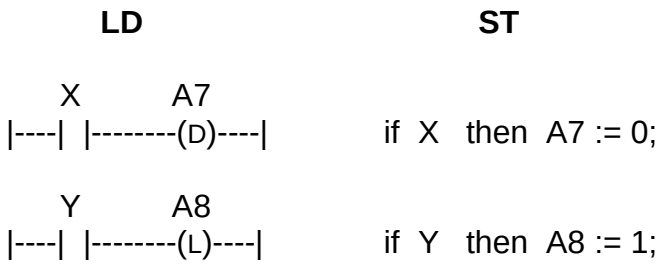


Figura 4-9: Comando IF em ST e LD

Em LD, uma bobina desliga atribui o valor 0 ao seu operando quando energizada, e uma bobina liga atribui o valor 1. A Figura 4-9 mostra dois comandos IF em ST e seus equivalentes em LD. Na primeira linha o operando A7 está sendo utilizado com uma bobina desliga e na segunda linha o operando A8 está sendo utilizado com uma bobina liga.

Os comandos CASE são casos particulares do comando IF, podendo ser tratados da mesma forma. Os comandos FOR, REPEAT e WHILE são expandidos em comandos IF e GOTO. A Tabela 4-1 exemplifica esta expansão.

Comando ST	Exemplo em ST	Expandindo em IFs

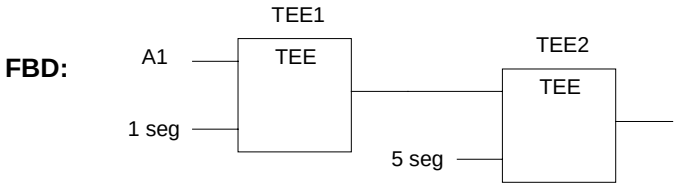
CASE	**CASE** I **OF**
	15: A7 := A7 + 1;
	30: A8 := A8 +1;
	END_CASE;
		IF I = 15 **THEN**
		A7 := A7 + 1;
		END_IF;
		IF I = 30 **THEN**
		A8 := A8 + 1;
		END_IF;

FOR	**FOR** I := 1 to 10 **DO** A8 := A8 +1; **END_FOR**;	I := 1; INI_FOR: **IF** I > 10 **THEN** **GOTO** END_FOR; **END_IF**; A8 := A8 +1; I := I +1; **GOTO** INI_FOR; END_FOR:

Tabela 4-1: Exemplo de Comandos CASE e FOR Expandidos em Comando IF

As chamadas de função em ST correspondem diretamente a um bloco equivalente em FBD. Para cada linha que chama a função na ST existirá um bloco instanciado no FBD,

e os parâmetros passados no ST correspondem às entradas do bloco no FBD. A Figura 4-10 mostra dois blocos TEE instanciados com os nomes TEE1 e TEE2, no diagrama FBD. No ST equivalente, são feitas duas chamadas de bloco de função usando-se estes mesmos nomes. Na chamada de TEE1 passam-se a variável "A1" e a constante 1, representando 1 segundo. Na chamada TEE2 passa-se a referência à saída de TEE1 (uma conexão) e a constante 5.



ST:

```
TEE1 ( A1 , 1);
TEE2 ( TEE1.OUT , 5);
```

Figura 4-10: Chamada de Bloco de Função em FBD e ST

Em LD e FBD todos os blocos são sempre executados, mesmo se suas entradas de habilitação estiverem desenergizadas (LD). No ST isto nem sempre acontece, pois o conteúdo de um comando IF pode não ser executado. Uma alternativa, para que a conversão ST – LD/FBD seja possível, é assumir que todos os blocos possuem uma entrada e uma saída de habilitação: quando energizada o bloco executa normalmente e quando desenergizada o bloco executa mas não faz nada, desenergizando sua saída *habilita*. A Figura 4-11 mostra um comando IF, em ST, com uma chamada de bloco de função que pode não ser executado e seu equivalente em FBD.

ST:

```
if  A
then  CALC1( B , C );
```

LD/FBD:

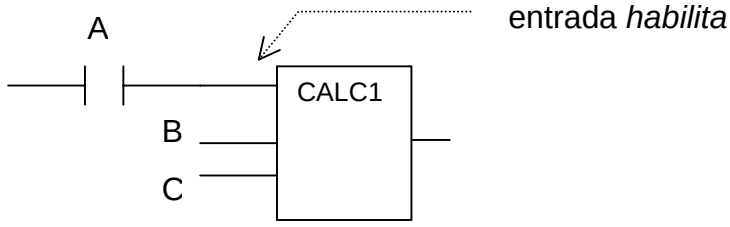


Figura 4-11: Blocos FBD com Entrada Habilita

As instruções contato e bobina do LD também podem ser representadas como um bloco contento uma entrada de habilitação, conforme mostra a Figura 4-12:

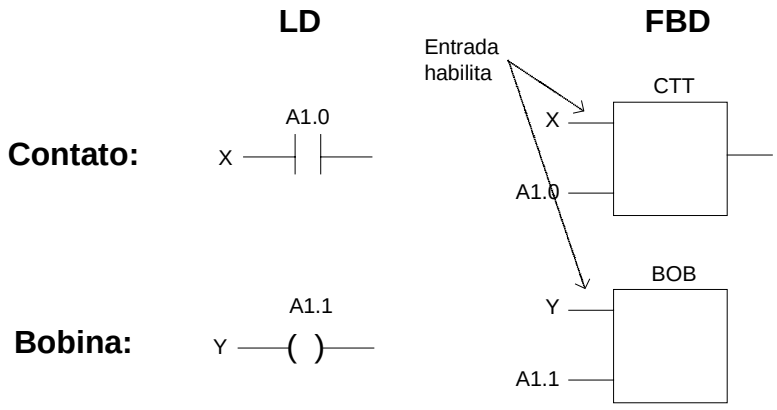


Figura 4-12: Contatos e Bobinas em LD e FBD

Note-se que o bloco bobina não possui nenhuma saída, pois ele é a última instrução possível de uma lógica LD.

4.2 Representação Textual de LD e FBD

Esta seção apresenta uma forma de representação textual que pode ser usado como alternativa para as linguagens gráficas LD e FBD. Esta forma de representação será adaptada para o formato XML a ser definido.

4.2.1 Modelagem dos Blocos

Um bloco é caracterizado pelo seu nome e pelas suas entradas e saídas. Por exemplo, o bloco SOM da Figura 4-13 possui duas entradas (IN1 e IN2) e duas saídas (OUT1 e OUT2).

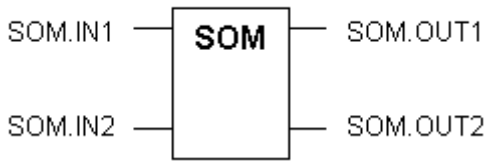


Figura 4-13: Nomes das Entradas e Saídas dos Blocos

As entradas são numeradas seqüencialmente, e cada qual pode ser referenciada pelo conjunto <nome do bloco> + <caracter “.” > + <string “IN”> + <número entrada> . O procedimento para se referenciar as saídas é o mesmo, utilizando-se “OUT” em vez de “IN” .

Quando existir mais de um bloco do mesmo tipo de bloco, pode-se diferenciá-los através do número de ocorrência do bloco, conforme mostrado na Figura 4-14.

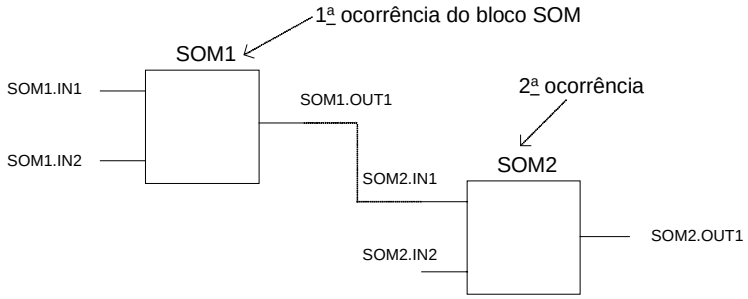


Figura 4-14: Número de Ocorrência dos Blocos

4.2.2 Modelagem das Conexões

Como representar em uma linguagem textual as conexões de blocos gráficos ? Através da referência às entradas e saídas definidas na seção anterior. Esta seção mostra inicialmente um diagrama exemplo em LD, que em seguida é adaptado para FBD e finalmente representado de forma textual.

A Figura 4-15 mostra um diagrama LD com dois blocos TEE (temporizadores). O TEE de 5 segundos está sempre habilitado, enquanto que o de 1 segundo depende do estado do contato “X8” . Se este contato estiver energizado, a bobina “X9” é acionada após 1 segundo, caso contrário após 5 segundos.

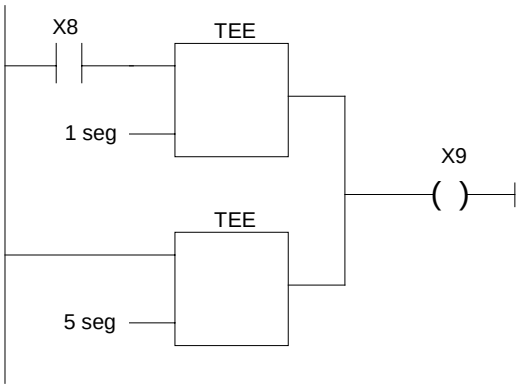


Figura 4-15: Exemplo de LD com dois blocos TEE

Antes de se modelar as conexões, pode-se redesenhar este diagrama LD pelo seu equivalente FBD, com as seguintes modificações:

- substitui-se o contato e a bobina pelos seus blocos equivalentes em FBD
- inclui-se o número de ocorrência nos dois blocos TEE
- inclui-se o OR invisível que abstrai a ligação horizontal

A Figura 4-16 mostra as modificações efetuadas :

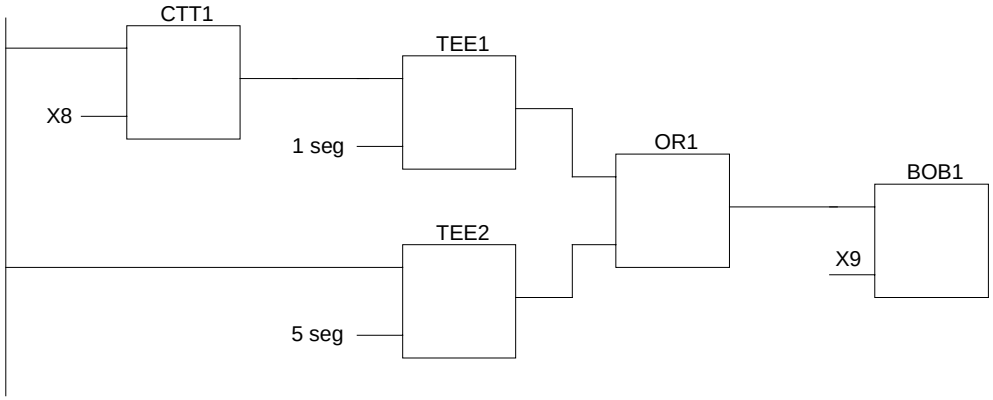


Figura 4-16: Exemplo de FBD com Dois Blocos TEE

Para modelar este diagrama FBD em linguagem textual, cada bloco precisa ser instanciado na linguagem textual, procedimento análogo a declaração de objetos nas linguagens de programação orientada a objetos ([DEI02]). Cada instância possui uma ou mais variáveis de estado que correspondem ao valor da(s) saída(s) do bloco. As entradas constantes, sem conexão, são consideradas variáveis locais e seu valor também é armazenado na instância do bloco. Por outro lado, as entradas conectadas a uma saída de outro bloco não são consideradas variáveis locais desta instância pois seu valor já está armazenado no bloco ao qual está conectado.

A informação de conexão fica armazenada no bloco que possui a entrada conectada e não no que possui a saída. Isto facilita a estrutura de dados de armazenamento, uma vez que uma entrada só pode estar conectada a uma única saída, enquanto que uma saída poderá estar conectada a zero, uma ou várias entradas.

Cada conexão é informada na mesma linha de instanciação do bloco, da mesma forma que as entradas locais (constantes), através do identificador do bloco de saída, que é a outra ponta da conexão. A Figura 4-17 mostra o diagrama textual correspondente :

```
CTT1 ( TRUE , X8 )
TEE1 ( CTT1.OUT1 , 1 )
TEE2 ( 1 , 5 )
OR1 ( TEE1.OUT1 , TEE2.OUT2 )
BOB1 ( OR1.OUT1 , X9 )
```

Figura 4-17: Representação Textual do FBD

Comparando-se a linguagem textual da Figura 4-17 com o diagrama FBD correspondente da Figura 4-16, pode-se observar que os cinco blocos do FBD (CTT1, TEE1, TEE2, OR1 e BOB1) estão instanciados nas cinco linhas do textual correspondente. Este mesmo comando de instanciação vai determinar também a execução do bloco, após a compilação, que é vista mais adiante.

O primeiro bloco, CTT1, aparece no FBD da Figura 4-16 com a primeira entrada energizada (conectada a barra) e a segunda com a variável “X8”. Estas duas entradas aparecem como parâmetros na mesma linha de CTT1, entre parêntesis, conforme mostra a Figura 4-18.

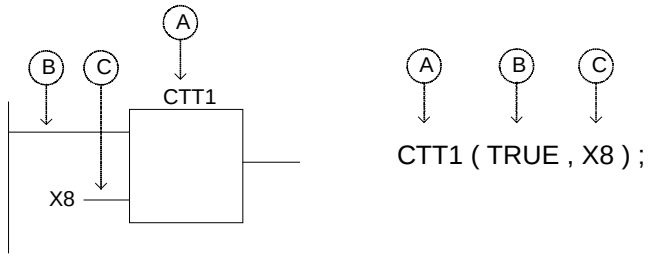


Figura 4-18: Instância de Bloco

O segundo bloco, TEE1, aparece no FBD da Figura 4-16 com a primeira entrada conectada a saída do bloco CTT1 e a segunda com a constante 1. Na forma textual, estas duas entradas aparecem como parâmetros na mesma linha de TEE1, entre parêntesis. A conexão é informada na posição correspondente, através do nome do bloco com o qual é feita a conexão, seguido do caracter “.” e do identificador da saída, conforme mostra a legenda “B” na Figura 4-19.

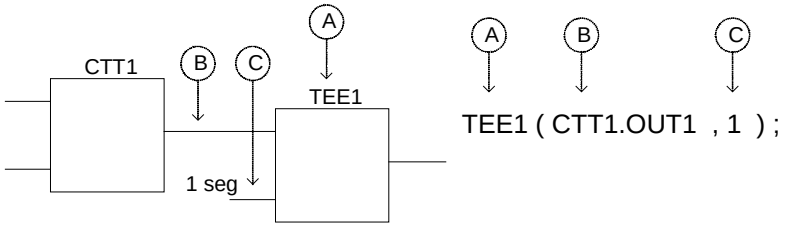


Figura 4-19: Conexão de Bloco

Os demais blocos são instanciados e conectados da mesma forma descrita até aqui. **A linguagem textual da Figura 4-17 possui toda a informação necessária para redesenhar completamente o diagrama original FBD ou LD**.

4.3 Conclusão

Este capítulo apresentou uma comparação das linguagens ST, LD e FBD da norma IEC 1131-3.

Da comparação entre FBD e LD concluiu-se que a representação textual de um diagrama gráfico LD se torna um caso particular do FBD, desde que as instruções contato e bobina sejam assumidas como se fossem blocos e que as ligações horizontais sejam substituídas por um bloco OR equivalente

Da comparação com ST nasce a proposta de representação do comando IF e demais comandos condicionais da linguagem.

Para a representação textual de um diagrama gráfico de blocos conectados percebe-se que não é necessário nenhuma estrutura de dados mais elaborada tipo grafo ou lista encadeada, desde que assuma-se que cada saída de bloco possui memória alocada para armazenar seu valor e que a informação de conexões fique sempre armazenada na entrada dos blocos. Com isto apresenta-se uma representação textual que é usada mais adiante na arquitetura XML.

O próximo capítulo apresenta a linguagem CLPScript a partir da qual a arquitetura XML será gerada.

5 A Linguagem CLPScript

Resumo

Este capítulo define a linguagem CLPScript, uma extensão de um sub-conjunto da linguagem ST usada para gerar e formalizar a arquitetura XML proposta.

A linguagem CLPScript permite escrever programas que são compilados para o formato XML que permite a interoperabilidade entre as linguagens da norma IEC 1131-3. Após uma análise das linguagens da norma, optou-se por utilizar ST como linguagem hospedeira de CLPScript.

As razões para utilização da linguagem ST como hospedeira são:

- criar uma nova linguagem do zero é muito trabalhoso e desnecessário;
- os programadores e o mercado já conhecem linguagens no estilo ST;
- seu uso é simples;
- pode-se utilizar uma versão mais enxuta da ST.

Como exemplo de outras linguagens que são baseadas em uma linguagem hospedeira, pode-se citar, dentre outras, JSP, que utiliza Java ([MOR00]) como linguagem hospedeira, *WebCompose* ([CRE00]), que utiliza Lua como linguagem hospedeira e *GlueScript* ([HAN03]), que utiliza Java/JSP como linguagem hospedeira. Estas duas últimas citadas, *WebCompose* e *GlueScript*, foram desenvolvidas no contexto do Laboratório de Engenharia de Software do PIPCA da Universidade do Vale do Rio dos Sinos.

A linguagem CLPScript é portanto uma extensão de um sub-conjunto da linguagem ST.

Esta seção define as partes do ST que fazem parte da CLPScript. As extensões incluídas são as relacionadas com os operandos do CP e outras facilidades para a geração do XML.

5.1 Elementos Comuns da IEC-1131-3

Esta seção define os elementos comuns da IEC-1131-3 utilizados na CLPScript. São itens presentes em todas as linguagens padronizadas pela norma.

5.1.1 Notações Léxicas

A norma IEC 1131-3 define diversas notações para representar valores e dados. Esta seção descreve as notações utilizadas.

Item	**Descrição**
Letra	qualquer letra de ‘a’ - ‘z’, maiúscula ou minúscula
digito	qualquer digito de ‘0’ - ‘9’
digito binário	qualquer digito de ‘0’ - ‘1’
digito octal	qualquer digito de ‘0’ - ‘7’
digito hexadecimal	qualquer digito de ‘0’ - ‘9’, ou qualquer letra ‘a’ - ‘z’, maiúscula ou minúscula

Tabela 5-1: Notações Léxicas

5.1.1.1 Identificadores

Um identificador é usado para nomear diferentes elementos dentro da linguagem, sendo um elemento único dentro do seu escopo. É formado por uma seqüência de letras, números ou do caracter subscrito ‘_’. Deve começar por letra ou pelo caractere de subscrito.

Apenas os primeiros 32 caracteres da seqüência são considerados, os demais não terão significado e são ignorados. O uso de letras maiúsculas ou minúsculas fazem diferença no identificador, ou seja, “Nivel_Vaso1”, “nivel_vaso1” ou “Nivel_vaso1” não são o mesmo identificador.

5.1.1.2 Uso de Espaço, Tabulação e Quebras de Linha

Serão desprezados todos os caracteres de espaço, tabulação e quebras de linha entre os identificadores, palavras chave, literais ou delimitadores.

5.1.1.3 Comentários

Comentários no código fonte devem ser feitos entre os caracteres ‘(*’ e “*)’. Podem ser comentadas várias linhas formando um único bloco de comentários.

5.1.1.4 Literais Numéricos

Existem dois tipos de literais numéricos: inteiros e reais.

Um número inteiro também pode ser expresso em base binária, octal ou hexadecimal. Para isto utiliza-se os prefixos **2#, 8#** e **16#** antes do número, respectivamente.

Literais numéricos reais contém o ponto decimal “.” entre os números. Opcionalmente pode-se especificar o expoente da base 10 utilizado os prefixos ‘E’ ou ‘e’.

5.1.1.5 Literais Booleanos

Para literais booleanos podem ser usado as palavras reservadas TRUE ou FALSE.

5.1.1.6 Operandos do CP

Os operandos do CP, para serem referenciados nos programas, devem ser mapeados em variáveis. Todos os operandos podem ser mapeados através da instrução AT. O acesso a subdivisão dos operandos só poderá ser feita a nível de bit.

5.2 Tipos de Dados

A CLPScript utiliza os tipos de dados definidos na Tabela 5-2.

Tipo	Descrição	Bits
BOOL	Booleano	1
BYTE	Seqüência de 8 bits	8
WORD	Seqüência de 16 bits	8
DWORD	Seqüência de 32 bits	8
SINT	Short integer	8
USINT	Unsigned short integer	8
INT	Integer	16
UINT	Unsigned integer	16
DINT	Double integer	32
UDINT	Unsigned double integer	32
REAL	Real	32

Tabela 5-2: Tipos de Dados da CLPScript

5.3 Variáveis

Uma variável é uma área de memória que armazena um tipo de dado definido na seção *Tipos de Dados*. Todas as variáveis devem ser declaradas antes de serem usadas.

5.3.1 Declaração de Variáveis

Cada declaração pode conter diversas variáveis, separadas por vírgula. Neste caso, todas serão de um mesmo tipo.

<Var1>, <Var2>, ... , <VarN> : <tipo> [:= <Literal>];

Vetores unidimensionais podem ser declarados usando para palavra reservada ARRAY. Os limites inferiores e superiores podem ser livremente especificados.

<vetor> : ARRAY [<limite_inferior> .. < limite_superior>] OF <tipo>;

5.3.2 Categorias de Variáveis

Uma categoria define o tipo de acesso e o escopo de uma variável. É identificada por duas palavras chaves que definem o inicio e o fim da declaração de variáveis de uma categoria, conforme mostra a Figura 5-1.

```
<Categoria>
<declaração1> ; <declaração 2> ; .. ; <declaração N>;
END_VAR
```

Figura 5-1: Categorias de Variáveis em CLPScript

Existe três categorias possíveis:

- **VAR_GLOBAL**: Permite acesso de leitura e escrita. Variáveis declaradas nesta categoria possuem o escopo global, podendo ser acessadas por todas funções e pelo o programa principal.
- **VAR_INPUT**: Permite acesso somente de leitura. É utilizada para definir os parâmetros de entrada de uma função. Por definição, uma variável de entrada não pode ser inicializada ou associada com operandos do CP com a cláusula AT. Também não é permitido declarar vetores como parâmetro de entrada. Variáveis declaradas nesta categoria possuem escopo local, ou seja, só podem ser acessadas pela função onde foi declarada.
- **VAR**: Permite acesso somente de leitura. É utilizada para definir variáveis internas de uma função ou do programa. Variáveis declaradas nesta categoria possuem escopo local, ou seja, só podem ser acessadas pela função ou programa onde foi declarada.

5.3.2.1 Mapeando Operandos do CP em Variáveis

É possível mapear um endereço físico ou operando do CP diretamente para uma variável. O mapeamento possibilita a interação do módulo gerado pelo compilador com os demais módulos de programa existentes no CP. É na verdade a única forma de acessar operandos do CP dentro de um programa CLPScript.

Todo mapeamento é feito na declaração de variáveis com a instrução AT. Por mapear um endereço global para todos os módulos de programação do CP, esta operação tem as seguintes restrições:

- só poderá ser usado dentro de VAR e VAR_GLOBAL;
- só será permitido uma variável por declaração;
- variáveis declaradas com mapeamento de operandos não são inicializadas;
- o uso da palavra reservada CONSTANT será permitido, ele serve para indicar que a variável não será modificada no decorrer do programa.

A sintaxe de declaração da instrução AT é:

<variável> **AT** <operando_CP> **:** <tipo> **;**

Os tipos permitidos na declaração devem ser compatíveis com os operandos.

5.4 Funções

Uma função é um trecho de código que poderá ser chamado diversas vezes pelo programa principal ou por outras funções. Ela se caracteriza por ter diversas entradas mas apenas uma saída.

5.4.1 Funções Definidas pelo Usuário

Uma função pode ser declarada da seguinte forma mostrada na Figura 5-2:

```
FUNCTION <nome> : <tipo_retorno>
        [ VAR_INPUT ... END_VAR ]
        [ VAR ... END_VAR]
        <corpo_da_função>
END_FUNCTION
```

Figura 5-2: Declaração de Função em CLPScript

Antes que a função retorne à rotina que a chamou, deverá ser configurado um valor de retorno. Esta operação poderá ser feita atribuindo-se um valor para o nome da função.

Uma função pode chamar outra função definida pelo usuário desde de que ela já tenha sido declarada. Uma função não poderá chamar a si mesma, isto é, não pode ser recursiva.

5.5 Programa

O programa é trecho de código onde começa a execução do módulo. Um programa pode acessar todas as variáveis globais e funções definidas no módulo. É declarado da forma mostrada na Figura 5-3:

```
PROGRAM <nome>
        [ VAR ... END_VAR]
        <corpo_do_programa>
END_ PROGRAM
```

Figura 5-3: Declaração de Programa em CLPScript

5.6 Comandos

As próximas seções definem os comandos da linguagem CLPScript.

5.6.1 Expressões

Compostas por diversos operandos e operadores, as expressões são utilizadas para calcular ou avaliar valores. Os operandos podem ser variáveis, literais ou chamadas de função.

Os operadores podem utilizar um ou dois operandos. Quando utilizam apenas um operando são chamados de unários. Neste caso, sempre se localizam antes do operando. Quando utilizam dois operandos são chamados de binários. Neste caso o operador deverá estar entre os operandos.

Os dois operandos usados em operações binárias devem ser do mesmo tipo. Caso não seja, deverá ser usado uma função de conversão de tipo.

Ambos os operadores, unários e binários, retornam o valor com mesmo tipo usado nos operandos. Se o operando que receber o resultado for de um tipo diferente deverá ser usado um função de conversão de tipos.

5.6.1.1 Operadores Matemáticos

Os operadores mostrados na Tabela 5-3 realizam operações matemáticas entre dois operandos. Os operandos podem ser de qualquer tipo numérico, mas ambos devem ter o mesmo tipo. O operador matemático sempre retorna o mesmo tipo dos operandos usados.

Operador	Descrição	Aplicação
+	Adição	ANY_NUM + ANY_NUM
-	Subtração	ANY_NUM – ANY_NUM
-	Negação (menos unário)	- ANY_NUM
*	Multiplicação	ANY_NUM * ANY_NUM
/	Divisão	ANY_NUM / ANY_NUM

Tabela 5-3: Operadores Matemáticos

5.6.1.2 Operadores Relacionais

Os operadores relacionais mostrados na Tabela 5-4 executam uma comparação entre dois tipos numéricos. Os operandos devem ser do mesmo tipo e a operação retorna sempre um tipo BOOL.

Operador	Descrição	Aplicação
<	Menor que	ANY_NUM < ANY_NUM
>	Maior que	ANY_NUM > ANY_NUM
<=	Menor ou igual que	ANY_NUM <= ANY_NUM
>=	Maior ou igual que	ANY_NUM >= ANY_NUM
=	Igual a	ANY = ANY
<>	Diferente de	ANY <> ANY

Tabela 5-4: Operadores Relacionais

5.6.1.3 Operadores Lógicos e Bit-a-Bit

Estes operadores executam duas operações diferentes: lógica booleana e lógica bit-a-bit. A seleção da operação é feita de acordo com os tipos dos operandos usados.

Operações de lógica booleana são executadas entre operandos do tipo BOOL. A Tabela 5-5 representa o resultado de uma operação booleana para os operadores AND, OR e XOR. O resultado sempre será do tipo BOOL.

Operando A	Operando B	AND	OR	XOR
FALSE	FALSE	FALSE	FALSE	FALSE
FALSE	TRUE	FALSE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE	TRUE
TRUE	TRUE	TRUE	TRUE	FALSE

Tabela 5-5: Operadores Lógicos

Operações lógicas bit-a-bit são executadas quando os operandos são do tipo BYTE, WORD e DWORD, sendo que ambos devem ser do mesmo tipo. A operação bit-a-bit realiza uma operação booleana individualmente em cada bit dos operandos. Estas operações retornam o mesmo tipo dos operandos usados.

Operador	Descrição	Aplicação
AND, &	“E”	ANY_BIT **AND** ANY_BIT ANY_BIT **&** ANY_BIT
XOR	“OU” exclusivo	ANY_BIT **XOR** ANY_BIT
OR	“OU”	ANY_BIT **OR** ANY_BIT
NOT	Complemento	**NOT** ANY_BIT

Tabela 5-6: Operadores Bit-a-Bit

5.6.1.4 Precedência de Operadores

A avaliação da expressão é feita de acordo com a precedência dos operadores, definido na Tabela 5-7. Operadores de maior precedência são avaliados primeiro. Caso os operadores tenham a mesma precedência, o que estiver mais a esquerda será o primeiro a ser avaliado.

Precedência	**Operador**	**Descrição**
0 (maior)	(expressão)	Expressão entre parênteses
	função (p1, p2, ... , pN)	Avaliação de função
1	**	Elevação a um expoente inteiro
2	-	Negação
	NOT	Complemento
3	*	Multiplicação
	/	Divisão
	MOD	Resto
4	+	Adição
	-	Subtração
5	< , > , <= , >=	Comparação
6	=	Igualdade
	<>	Desigualdade
7	AND, & XOR	Operação “E” booleana Operação “OU” exclusivo booleana
8 (menor)	OR	Operação “OU” booleana

Tabela 5-7: Precedência de Operadores

5.6.2 Literais Numéricos

Os literais numéricos inteiros da Tabela 5-8 podem ser usados em operações com inteiros, desde que o valor do literal não ultrapasse o limite do tipo da faixa.

Faixa	Tipos compatíveis
0 a 127	SINT, USINT, INT, UINT, DINT, UDINT, BYTE, WORD e DWORD
-1 a -128	SINT, INT e DINT
128 a 255	USINT, INT, UINT, DINT, UDINT, BYTE, WORD e DWORD
256 a 32.767	INT, UINT, DINT, UDINT, WORD e DWORD
-129 a –32.768	INT e DINT
32.768 a 65.535	UINT, DINT, UDINT, WORD e DWORD
-32767 a -2.147.483.648	DINT
65.536 a 2.147.483.647	DINT, UDINT e DWORD
2.147.483.648 a 4.294.967.296	UDINT DWORD

Tabela 5-8: Literais Numéricos

5.6.3 Comandos de Atribuição

A atribuição é usada para escrever um determinado valor em uma variável.

<variável> **:=** <expressão>

Uma atribuição também pode ser usada para configurar o valor retornado de uma função. Neste caso a variável deve ser substituída pelo nome da função. Esta operação não irá encerrar a função, apenas irá configurar o seu valor de retorno.

5.6.4 Comando RETURN

Um função sempre retorna para a rotina que a chamou após a execução da última afirmação. Porém, é possível forçar o retorno em qualquer ponto do código, através do uso da palavra reservada **RETURN.**

5.6.5 Comando IF

O comando IF mostrado na Figura 5-4 executará as afirmações após o **THEN** se a condição <*expressão_booleana*> de teste for verdadeira. Opcionalmente é possível inserir outras condições de teste com a clausula **ELSIF**, sendo que apenas o grupo de afirmações onde o primeiro teste for verdadeiro é que será executado.

```
IF <expressão_booleana> THEN <afirmações>
{ ELSIF <expressão_booleana>  THEN <afirmações> }
[ ELSE <afirmações> ]
ENDIF;
```

Figura 5-4: Comando IF em CLPScript

Opcionalmente é possível especificar um bloco de afirmações para ser executado caso todos os testes falhem através da clausula **ELSE**.

5.6.6 Comando CASE

O comando **CASE**, mostrado na Figura 5-5, executa apenas um bloco de afirmações dentre vários possíveis. A seleção é feita pela comparação do valor inteiro de *<expressão_inteira>* com os valores dos *<casos>*.

```
CASE <expressão_inteira> OF
        <casos> :  <afirmações>
        { <casos> :  <afirmações> }
         [ ELSE <afirmações> ]
END_CASE
```

Figura 5-5: Comando CASE em CLPScript

Opcionalmente é possível especificar um bloco de afirmações para ser executado caso todos os testes falhem através da clausula **ELSE**.

5.6.7 Comandos de Iteração

Um comando de iteração executa repetidamente um bloco de afirmações. O número de vezes que é executado depende do tipo de iteração, que pode ser: o comando **WHILE**, o comando **FOR** ou o comando **REPEAT**.

Para todos os comandos, é possível interromper o laço da iteração prematuramente através do comando **EXIT**. Este comando só pode ser usado dentro do laço da iteração.

5.6.7.1 Comando WHILE

O comando WHILE, mostrado na Figura 5-6, executa o bloco de comandos enquanto expressão de avaliação *<expressão_booleana>* for verdadeira. O comando WHILE sempre testa a função de avaliação antes de executar o bloco. Assim se na primeira iteração o teste resultar em falso, o bloco de afirmações não será executado.

```
WHILE <expressão_booleana> DO
        <afirmações>
END_WHILE
```

Figura 5-6: Comando WHILE em CLPScript

5.6.7.2 Comando REPEAT

O comando REPEAT, mostrado na Figura 5-7, executa o bloco de comandos até que a expressão de avaliação <*expressão_booleana*> seja verdadeira. Diferente do comando WHILE, o comando REPEAT executa primeiro o bloco de afirmações e depois testa a expressão de avaliação. Assim o bloco de afirmações é executado pelo menos uma vez.

```
REPEAT
<afirmações>
UNTIL <expressão_booleana>
END_REPEAT
```

Figura 5-7: Comando REPEAT em CLPScript

5.6.7.3 Comando FOR

O comando FOR, mostrado na Figura 5-8, permite executar um bloco de afirmações repetidas vezes. O número de repetições é controlado por uma <*variável_controle*>. Esta variável deve ser do tipo SINT ou INT e não pode ser um operando do CP.

```
FOR <variável_controle> := <expr_inicial> TO <expr_final> BY
<expr_inc> DO
<afirmações>
END_FOR
```

Figura 5-8: Comando FOR em CLPScript

Primeiramente, <*variável_controle*> é inicializada com o valor de <*expr_inicial*>. No inicio de cada repetição, é verificado se o valor de <*variável_controle*> excedeu o valor definido por <*expr_final*>. Se não excedeu, o bloco de afirmações é executado. Caso contrário, o comando é encerrado. No fim da execução do bloco <*variável_controle*> é incrementado em 1, ou pelo valor definido por <*expr_inc*>. Tanto a <*variável_controle*> como as expressões <*expr_inicial*>, <*expr_inicial*> e <*expr_final*> devem ser do mesmo tipo (SINT ou INT).

5.7 Itens Desconsiderados da Norma

A definição da CLPScript desconsiderou os seguintes aspectos da linguagem ST da norma IEC 1131-3:

- strings;
- inicialização de variáveis;
- tipos de dado *struct*, enumerados e *sub-range*;
- *function blocks;*
- *resource*, *task* e *configuration.*

5.8 Conclusão

Este capítulo apresentou a definição da linguagem CLPScript, criada a partir da linguagem ST da norma IEC 1131-3.

Procurou-se manter em CLPScript um mínimo de comandos imprescindíveis em uma linguagem de alto nível e retirou-se opções mais complexas tais como estruturas de dados (*structs*) e criação de novos tipos de dados definidos pelo usuário. No contexto da automação industrial, este conjunto de comandos selecionados já é de grande utilidade.

O próximo capítulo utiliza a CLPScript para formalizar a arquitetura XML e o seguinte apresenta o programa CLPScript.EXE implementa esta arquitetura.

6 A Arquitetura XML

Resumo

Este capítulo apresenta a arquitetura XML, mostrando as razões da escolha desta linguagem, definindo as *tags* e atributos XML para armazenar programas LD, FBD e CLPScript. Faz uma comparação da arquitetura XML com a linguagem IL

6.1 Porque XML ?

A linguagem XML foi selecionada para este trabalho devido às seguintes razões:

- é uma linguagem de marcação amplamente difundida na Web que permite garantir a interoperabilidade entre dados e aplicações;
- permite que se definam estruturas de dados simples e complexas, auto descritivas, de fácil compreensão por quem está utilizando;
- foi projetada para descrever informação através de *tags* que **não são** pré definidas, mas criadas conforme a necessidade de cada aplicação
- é extensível, podendo descrever dados de uma enorme variedade de aplicações

Existem outras linguagens que poderiam ter sido utilizadas para esta mesma finalidade, como por exemplo o VHDL. O XML foi selecionado em vez de VHDL pelas seguintes razões:

- a VHDL foi projetada com o foco em descrição de hardware, e embora pudesse perfeitamente ser utilizada aqui, o XML é mais flexível, pois foi projetado para garantir a interoperabilidade entre dados e aplicações de qualquer área de aplicação;
- o XML foi projetado pelo W3C para ser amplamente utilizado na Web, enquanto que a VHDL foi criada numa época em que este requisito não era relevante;

- as ferramentas mais poderosas de VHDL pertencem a empresas privadas que as comercializam, enquanto que para XML existem diversas ferramentas de qualidade e de domínio público, tais como parsers para leitura e validação de um documento;
- o XML é uma proposta mais atual, uma tendência de mercado e desperta um maior interesse da comunidade de engenharia de *software* da qual fazemos parte.

6.2 Representações Comuns a Todas as Linguagens

A norma IEC 1131-3 define elementos que são comuns às 5 linguagens, conforme visto anteriormente. Dentre estes estão os tipos de dados e a declaração de variáveis, funções e programas.

Os tipos de dados e trechos das declarações que aparecem no XML possuem os mesmos nomes definidos na norma. Procurou-se, sempre que possível, manter no XML a mesma nomenclatura original da IEC.

A Tabela 6-1 mostra as *tags* e atributos utilizados na declaração de variáveis, funções e programas.

Tag no XML	**Atributos Obrigatórios**	**Atributos Opcionais**
ARQ	Nome	-
FUNCTION	Nome, Type	-
PROGRAM	Nome	-
VAR VAR_INPUT VAR_GLOBAL	Nome, Type	At, Atrib

Tabela 6-1: *Tags* e Atributos XML de Declarações

O atributo **“Nome”** define o nome do arquivo origem, função, programa ou variável, conforme a *tag* em que se encontra. O atributo **“Type”** define o tipo da função ou da variável. O atributo **“Atrib”** pode assumir os valores “CONSTANT” ou “RETAIN” e o atributo **“At”** especifica o endereço de memória ou operando do CP associado a variável declarada.

(a)

```
FUNCTION Calcular: INT
        VAR_INPUT
                x: real;
        END_VAR
        VAR
                y1,y2: INT;
                R at F13: REAL;
        END_VAR
END_FUNCTION
```

(b)

```
<?xml version="1.0" ?>
<ARQ nome="D:\DISS4\BIN\T1.XML">
 <FUNCTION nome="Calcular" type="INT">
   <VAR_INPUT nome="x" type="REAL" />
   <VAR nome="y1" type="INT" />
   <VAR nome="y2" type="INT" />
   <VAR nome="R" type="REAL" at="F13" />
 </FUNCTION>
</ARQ>
```

Figura 6-1: XML de Declarações

A Figura 6-1a mostra um exemplo de declaração de uma função com um parâmetro de entrada e três variáveis locais e a Figura 6-1b mostra sua versão correspondente compilada em XML. Note-se que cada variável é declarada em uma linha diferente do XML, mesmo que no programa fonte não seja assim. Isto facilita a implementação dos programas que lêem este XML, e também permite que atributos individuais sejam acrescentados a cada variável individualmente, tal como o "at=F13" da variável "R".

Outra razão para manter uma declaração variável por linha no XML é quanto a futuras expansões, com novos atributos. Um exemplo possível é o offset de cada variável, conforme existe no *software* Step 7 da Siemens, mostrado na Figura 6-2.

O atributo **"Type"** aceita como valor os tipos básicos definidos na norma IEC 1131-3, e também o tipo *array*, descrito mais adiante na seção 6.4.1.1, "XML de Array nas Expressões".

Os arquivos desta arquitetura XML podem possuir uma ou mais funções e/ou um programa.

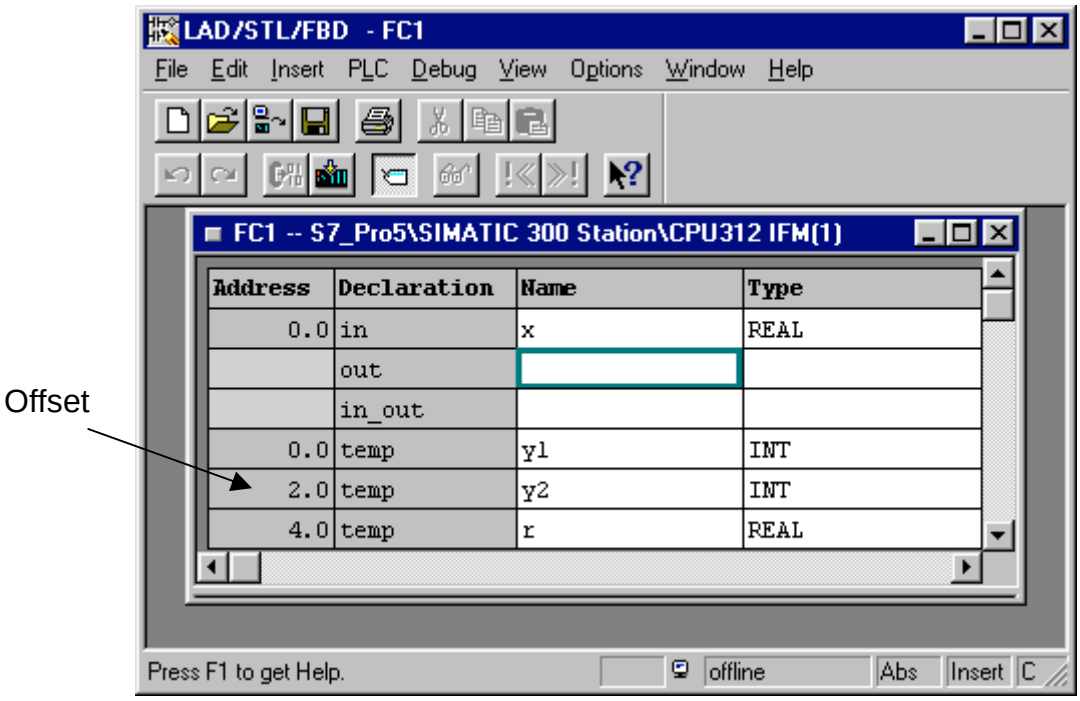


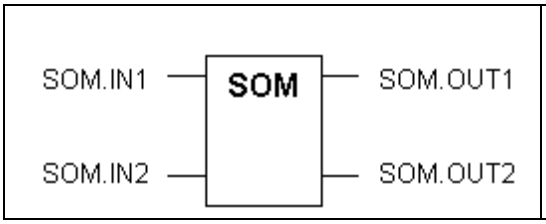
Figura 6-2: Declaração de Variáveis no Step 7

6.3 Representação de LD e FBD

A representação de diagramas LD e FBD em XML é baseada na modelagem de blocos descrita no capítulo 4. Um bloco elementar possui as seguintes *tags* e atributos:

- *tag* **BlocoNNN**: define o início do bloco com número de ocorrência NNN;
- atributo **instr**: instrução ou nome do bloco;
- atributos **in1**, **in2**, ... : definem as entradas do bloco.

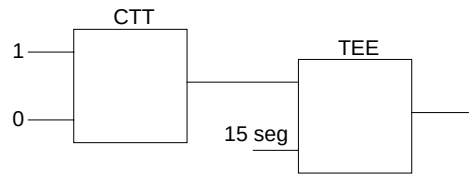
A Figura 6-3 a seguir é uma reprodução da Figura 4-13, mostrando um bloco elementar e o trecho de XML correspondente.




```
<Bloco1 instr="SOM" in1="" in2="" />
```

Figura 6-3: XML de um Bloco Elementar

As saídas não são representadas diretamente no XML, elas aparecem referenciadas nas conexões de entrada, quando for o caso. A Figura 6-4 mostra a representação XML de dois blocos com uma conexão entre eles.




```
<Bloco1 instr="CTT" in1="1" in2="0" />
<Bloco2 instr="TEE" in1="Bloco1.OUT"
        in2="15" />
```

Figura 6-4: XML de Dois Blocos Conectados

Mostrou-se que a representação de um diagrama LD é um caso particular da representação de um diagrama FBD. Este tipo de representação, portanto, pode ser utilizada para armazenar os diagramas destas duas linguagens.

Em situações típicas são aceitas as seguintes simplificações:

- em blocos com uma única saída, omite-se seu número, referenciando-a apenas por "OUT", em vez de "OUT1", por exemplo;
- blocos sem nenhuma saída podem ser descritos sem o número de ocorrência. Isto pode facilitar a geração de código executável a partir do XML no aspecto de alocação de memória: em blocos com saídas é preciso alocar memória para armazenar o valor da(s) saída(s), enquanto que em bloco sem saídas isto não é necessário;
- nomes de entrada ou saída conhecidos podem ser utilizados no lugar de "in1", "in2", "out1" e demais nomes *default* de entrada e saída.

6.3.1 Exemplos de LD e FBD

A seguir são apresentados 3 exemplos de diagramas LD/FBD criados em *softwares* de diferentes fabricantes, enfatizando que esta representação XML, apesar de compacta, pode ser utilizada para representar diagramas destas duas linguagens.

Cada exemplo mostra dois blocos com uma conexão entre si. Cada bloco representa alguma instrução a ser executada pelo respectivo controlador programável. Observe-se que, conforme o fabricante, cada bloco pode possuir uma ou mais entradas, uma ou mais saídas e um ou mais parâmetros, conforme resumido na Tabela 6-2.

Fabricante	Nº Entradas	Nº Saídas	Nº Parâmetros
RSLogix - Rockwell	uma única	várias	vários
MasterTool - Altus	várias	várias	vários
Step7 - Siemens	várias	várias	nenhum

Tabela 6-2: Comparação de Blocos Entre Fabricantes

Um bloco, conforme a norma a norma IEC 1131-3, é um POU, ou seja, um programa, função ou bloco de função. Existem blocos pré definidos e blocos criados pelo usuário. O que a norma não define, porém, são parâmetros de blocos. Os fabricantes que os utilizam, portanto, estão fora da norma. Em termos de armazenamento no XML, entretanto, pode-se tratar parâmetros de forma semelhante a uma entrada do bloco.

6.3.1.1 Exemplo de LD da Rockwell

A Figura 6-5 mostra um diagrama LD criado no *software* programador RSLogix da Rockwell ([ROC99], [ROC00]) com um bloco temporizador TON conectado a um bloco comparador GRT.

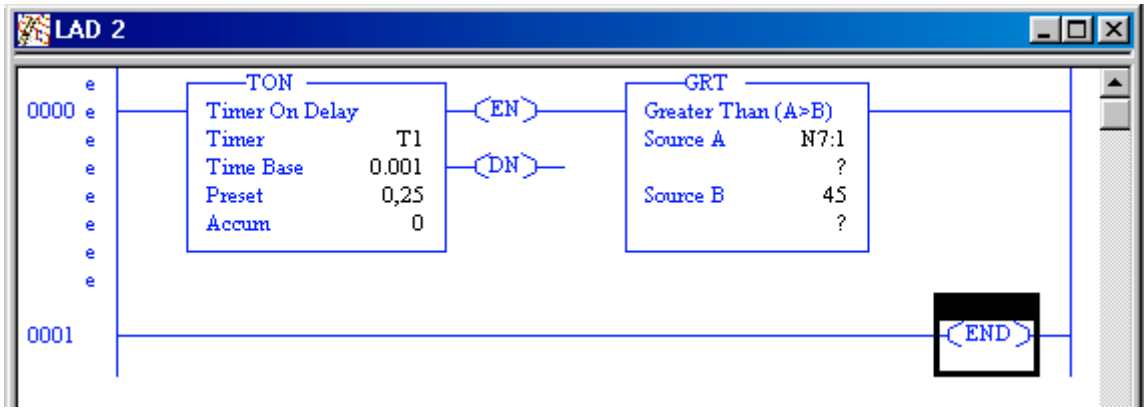


Figura 6-5: Diagrama LD no RSLogix da Rockwell

A Figura 6-6 mostra a representação XML correspondente. Os parâmetros do bloco TON (*Timer*, *Time Base*, *Preset*, *Accum*) e os do bloco GR (*Source A* e *Source B*) foram representados como se fossem uma entrada normal do bloco.

```
<Bloco1 instr="TON" en="1" Timer="T1" TimeBase="0.001" Preset="0,25"
        Accum="0" />
<Bloco2 en="Bloco1.EN" instr="GRT" SourceA="N7:1" SourceB="45" />
```

Figura 6-6: XML do Diagrama LD no RSLogix da Rockwell

Observe-se que esta representação engloba as entradas e saídas utilizadas, os parâmetros (na forma de entradas) e as conexões existentes.

6.3.1.2 Exemplo de LD da Altus

A Figura 6-7 mostra um diagrama LD criado no *software* programador MasterTool MT4100 da Altus ([ALT02]) com um bloco temporizador TEE conectado a um bloco de armazenamento CAR, por sua vez conectado a um comparador MAIOR.

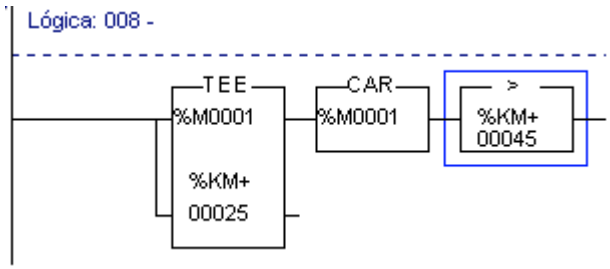


Figura 6-7: Diagrama LD no MasterTool MT4100 da Altus

A Figura 6-8 mostra a representação XML correspondente. Da mesma forma que no exemplo anterior, os parâmetros do bloco TEE foram representados como se fossem entradas normais do bloco.

```
<Bloco1 instr="TEE" in1="1" in2="1" param1="%M0001" param2="25" />
<Bloco2 instr="CAR" in1="Bloco1.Out1" param1="%M0001" />
<Bloco3 instr=">" in1="Bloco2.Out" param1="45" />
```

Figura 6-8: XML do Diagrama LD no MasterTool MT4100 da Altus

6.3.1.3 Exemplo de FBD da Siemens

A Figura 6-9 mostra um diagrama FBD criado no *software* programador Step7 da Siemens ([SIE99a]) com um bloco temporizador S_ODT conectado a um bloco de movimentação MOVE.

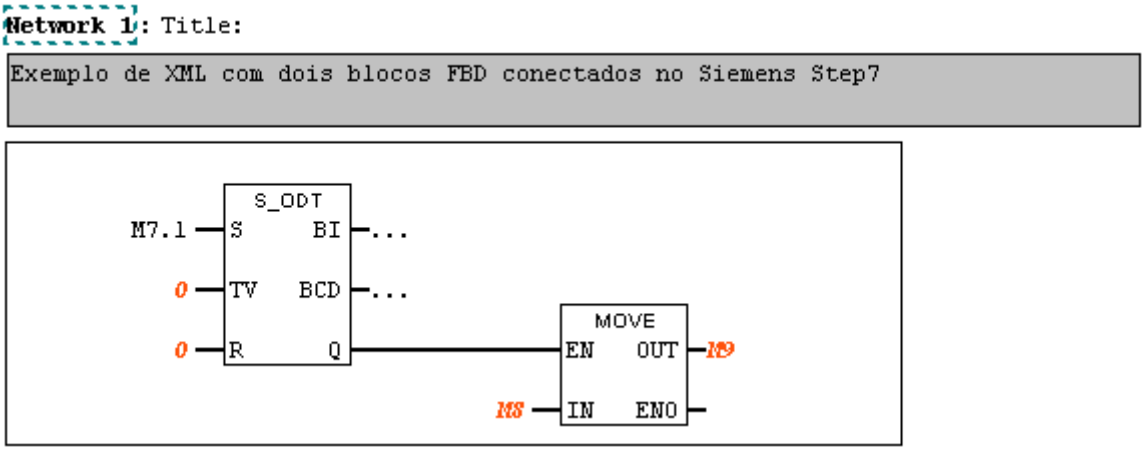


Figura 6-9: Diagrama FBD no Step7 da Siemens

A Figura 6-10 mostra a representação XML correspondente.

```
<Bloco1 instr="S_ODT" S="M7.1" TV="0" R="0" />
<Bloco2 instr="MOVE" EN="Bloco1.Q" IN="M8" />
```

Figura 6-10: XML do Diagrama FBD no Step7 da Siemens

6.3.2 Expansões no XML

A escalabilidade do XML permite que a arquitetura definida aqui possa ser facilmente expandida, incluindo-se novos atributos nas *tags* "**BlocoNNN"** para novas funções, sem que isto cause nenhum conflito com os existentes anteriormente.

Como exemplo destas expansões, pode-se citar:

- coordenadas, cores e outros atributos gráficos dos blocos: embora não necessários para regenerar o diagrama original, podem ser implementados de forma a restaurar configurações feitas pelo usuário;
- comentários configuráveis pelo usuário de determinado conjunto de blocos de um diagrama, tal como o texto "Exemplo de XML ..." da Figura 6-9 do Step7 da Siemens.
- descrição de variáveis e blocos, configuráveis pelo usuário
- o tipo de dado das saídas de cada bloco: isto pode facilitar a implementação do programa que lê e interpreta o XML.

A Tabela 6-3 mostra um exemplo XML para cada uma das expansões sugeridas acima. Os novos atributos são mostrados com um caractere subscrito '_' na frente.

Expansão	**Exemplo de XML**
Gráficos	<Bloco1 ... **_X**="179" **_Y**="25" Colour="Black" />
Comentários	<Bloco1 ... **_Description**="Exemplo de XML com dois blocos ..." />
Descrição	<Bloco1 ... **_M8**="Tempo de Motor Ligado" />
Tipo de dado	<Bloco2 instr="MOVE" **_OutDataType**="Real" **_EnoDataType** ="Bool" />

Tabela 6-3: Exemplo de Expansões no XML

6.4 Representação da CLPScript

Esta seção apresenta maneiras de se armazenar em XML uma linguagem de alto nível tipo CLPScript, com recursos tais como comandos IF, FOR, WHILE, atribuição, variáveis, estruturas e chamadas de função.

Um programa CLPScript deve ser quebrado em trechos pequenos e relevantes para serem armazenados no XML, de preferência em um formato semelhante ao utilizado para armazenar os blocos LD e FBD. Esta operação é na verdade uma atividade de compilação, onde um programa fonte é transformado em outro.

A parte de declaração de variáveis, funções e programas é a mesma já apresentada, uma vez que utilizam o formato comum padronizado na norma IEC 1131-3. Resta então definir a maneira que são armazenados os comandos ou *statements* da linguagem.

6.4.1 Tags XML de Expressões

Uma expressão lógica ou aritmética deve ser quebrada em um conjunto de operações básicas, sucessivas, envolvendo um ou dois operandos e um operador.

Por exemplo, a expressão "y1 + (x-3) * 5.6" deve ser quebrada em "x-3", seguido de "* 5.6" e de "+y1". Cada uma destas operações básicas será então representada como sendo a chamada de um bloco correspondente, com os dois operandos como entrada, conforme mostra a Figura 6-11. A representação destes blocos é a mesma descrita anteriormente.

y1 + (x-3) * 5.6	<Bloco1 instr="**SUB**" in1="x" in2="3" /> <Bloco2 instr="**MUL**" in1="Bloco1.OUT" in2="5.6" /> <Bloco3 instr="**ADD**" in1="Bloco2.OUT" in2="y1" />

Figura 6-11: XML de Expressão Aritmética

O atributo "**instr**" de expressões pode assumir os seguintes valores:

- EQ, GT, GE, LE, LT, NE
- AND, OR, XOR
- ADD, SUB, MUL, DIV, EXPT

Os atributos "EQ" até "XOR" correspondem a uma operação lógicas e representam blocos que possuem saída ".OUT" do tipo bool. Os atributos "ADD" até "EXPT" correspondem a uma operação aritmética e representam blocos nos quais o tipo da saída ".OUT" varia com o tipo das entradas. A Figura 6-12 mostra o exemplo de uma expressão lógica.

(A>B) and (A>=C)	<Bloco4 instr="**GT**" in1="A" in2="B" /> <Bloco5 instr="**GE**" in1="A" in2="C" /> <Bloco6 instr="**and**" in1="Bloco4.OUT" in2="Bloco5.OUT" />

Figura 6-12: XML de Expressão Lógica

6.4.1.1 XML de Array nas Expressões

A norma IEC define a sintaxe de declaração e utilização de arrays em suas linguagens. Para declarar um array em XML, utiliza-se a mesmo *tag* "VAR" descrito anteriormente, conforme mostra a Figura 6-13.

VAR M : array[1..10] of real; END_VAR	<VAR nome="M" type="'ARRAY[1..10] OF REAL'" />

Figura 6-13: XML de Declaração de Array

A utilização de variáveis do tipo array no XML acontece conforme mostra a Figura 6-14: em a) está se atribuindo a constante real "12.34" para uma posição do array indexada pela constante "3", em b) uma atribuição para uma posição indexada pela

variável “i”, em c) uma atribuição para uma posição indexada por uma expressão, e neste caso é preciso avaliar primeiro a expressão e indexar o array com o resultado “Bloco1.OUT”. Em d), é uma posição do array que está sendo atribuída para uma variável.

```
A)  M[3] := 12.34;
B)  M[i] := 56.78;
C)  M[i+1] := 9.01;

D) R := M[3];
```

```
A)  <Bloco instr="MOVE" in1="12.34" in2="M[3]" />
B)  <Bloco instr="MOVE" in1="56.78" in2="M[i]" />
C)  <Bloco1 instr="ADD" in1="i" in2="1" />
    <Bloco instr="MOVE" in1="9.01" in2="M[Bloco1.OUT]" />
D)  <Bloco instr="MOVE" in1="M[3]" in2="r" />
```

Figura 6-14: XML de Array em Expressões

Pode-se ter estruturas mais complicadas tipo um array cujo índice é uma chamada de função com parâmetros de entrada dependentes de outro ou do mesmo array. Por exmplo, a Figura 6-15 mostra o XML da expressão “M[1 + REAL_TO_INT(2.3 * Sin(4-M[5]))] := 6.7;”

```
<ATRIBUICAO>
  <Bloco1 instr="SUB" in1="4" in2="M[5]" />
  <Bloco2 instr="CHF" nome="Sin" qtdIn="1" in1="Bloco1.OUT" type="REAL" />
  <Bloco3 instr="MUL" in1="2.3" in2="Bloco2.OUT" />
  <Bloco4 instr="CHF" nome="REAL_TO_INT" qtdIn="1" in1="Bloco3.OUT" type="INT" />
  <Bloco5 instr="ADD" in1="1" in2="Bloco4.OUT" />
  <Bloco instr="MOVE" in1="6.7" in2="M[Bloco5.OUT]" />
</ATRIBUICAO>
```

Figura 6-15: XML de Array em Expressão Complexa

6.4.2 Tags XML de Comando Atribuição

Um comando atribuição pode ser dividido em duas partes: na avaliação da expressão a ser atribuída e na atribuição propriamente dita.

A avaliação da expressão foi definida na seção anterior, enquanto que a atribuição é feita com um novo bloco MOVE. Uma *tag* “**ATRIBUICAO**” é inserida envolvendo todos os blocos, os da expressão e os do MOVE, conforme mostra a Figura 6-16.

x := Y1 + (x-3) * 5.6 ;	<**ATRIBUICAO**> <Bloco1 instr="SUB" in1="x" in2="3" /> <Bloco2 instr="MUL" in1="Bloco1.OUT" in2="5.6" /> <Bloco3 instr="ADD" in1="Bloco2.OUT" in2="y1" /> <Bloco instr="MOVE" in1="Bloco3.OUT" in2="x" /> </**ATRIBUICAO**>

Figura 6-16: XML de Comando Atribuição

Observe-se que a *tag* do bloco MOVE não possui número de ocorrência, pois corresponde a um bloco que não possui nenhuma saída. A presença da *tag* "ATRIBUICAO" visa unicamente possibilitar que o programa original possa ser restaurado a partir do XML.

6.4.3 Tags XML de Chamada de Função

Uma chamada de função em CLPScript corresponde diretamente a uma chamada de bloco em LD ou FBD. Utilizam uma *tag* "Bloco" com os seguintes atributos:

- "Instr" : recebe o valor "CHF", significando chamada de função;
- "Nome" : recebe o nome da função a ser chamada;
- "QtdIn" : quantidade de parâmetros de entrada;
- "in1" , "in2", ... : valor de cada parâmetro de entrada;
- "Type" : tipo de retorno da função.

A Figura 6-17 mostra um exemplo de XML para a chamada da função "SIN" . Caso o parâmetro de entrada não fosse uma constante simples, mas uma expressão, então a expressão teria de ser avaliada antes da chamada, conforme mostra a Figura 6-18.

sin(5.6);	<Bloco4 instr="**CHF**" nome="**sin**" qtdIn="1" in1="5.6" type="REAL" />

Figura 6-17: XML de Chamada de Função

sin(x + 3.1416/2);	<Bloco5 instr="DIV" in1="3.1416" in2="2" /> <Bloco6 instr="ADD" in1="x" in2="Bloco5.OUT" /> <Bloco7 instr="CHF" nome="sin" qtdIn="1" in1="Bloco6.OUT"

Figura 6-18: XML de Chamada de Função com Expressão de Entrada

6.4.4 Tags XML de Comando IF

Um comando IF pode ser dividido em duas ou três partes: a avaliação da expressão condicional, o(s) statement(s) da seção THEN e opcionalmente o(s) *statement(s)* da

seção ELSE. A forma de representar no XML recai em casos já vistos. A condição do IF é a avaliação de uma expressão, e as seções THEN e ELSE são statements, normalmente comandos de atribuição ou chamada de função. Para demarcar estas regiões, insere-se no XML as *tags* “IF”, “THEN” e “ELSE”, conforme mostrado na Figura 6-19.

```
if r > 50
        then x := 60;
        else x := 70;
end_if;
```

```
<IF>
  <Bloco1 instr="GT" in1="r" in2="50" />
  <THEN>
    <ATRIBUICAO>
      <Bloco instr="MOVE" in1="60" in2="x" />
    </ATRIBUICAO>
  </THEN>
  <ELSE>
    <ATRIBUICAO>
      <Bloco instr="MOVE" in1="70" in2="x" />
    </ATRIBUICAO>
  </ELSE>
</IF>
```

Figura 6-19: XML de Comando IF

Observe-se que as *tags* “THEN” e “ELSE” estão dentro da *tag* “IF”, e que a avaliação da condição do IF fica após a *tag* “IF” e antes da *tag* “THEN”, exatamente como na linguagem original.

6.4.5 Tags XML de Comando CASE

O Comando CASE é inserido no XML através da *tag* “CASE” e da *tag* “OF” junto com seu atributo “valor”. Entre a *tag* “CASE” e o primeiro “OF” devem estar os blocos de avaliação da expressão associada ao CASE. Dentro de cada “OF” aparecem o xml dos comandos correspondentes, conforme mostra a Figura 6-20.

```
CASE  i+10  OF
        15:   x := 10;
        30:   x  := 20;
END_CASE;
```

```
<CASE>
  <Bloco1 instr="ADD" in1="i" in2="10" />
  <OF valor=15>
    <Bloco instr="MOVE" in1="10" in2="x" />
  </OF>
  <OF valor=30>
    <Bloco instr="MOVE" in1="20" in2="x" />
  </OF>
/CASE>
```

Figura 6-20: XML de Comando CASE

6.4.6 Tags XML de Comando FOR

O Comando FOR é inserido no XML através da *tag* “FOR” e dos atributos “VAR” e “TO”. A variável de controle é especificada no atributo “VAR”, e sua inicialização é feita no comando atribuição que precede a *tag* “FOR”. O(s) statement(s) a serem executados ficam aninhados dentro da *tag* “FOR”, conforme mostra a Figura 6-21.

```
VAR
  I: INT;
  nivel: array[1..10] of
                        real;
END_VAR
...
for i := 0 to 9
  do nivel[i+1] := 0;
end_for;
```

```
<ATRIBUICAO>
   <Bloco instr="MOVE" in1="0" in2="i" />
</ATRIBUICAO>
<FOR var="i" to="9">
   <ATRIBUICAO>
     <Bloco1 instr="ADD" in1="i" in2="1" />
     <Bloco instr="MOVE" in1="0" in2="nivel[Bloco1.OUT]" />
   </ATRIBUICAO>
 /FOR>
```

Figura 6-21: XML de Comando FOR

6.4.7 Tags XML de Comando WHILE

O Comando WHILE é inserido no XML através das *tags* “WHILE” e “DO”. Entre estas duas *tags* aparece(m) o(s) bloco(s) com a avaliação da expressão de condição de entrada no laço WHILE. Dentro da *tag* “DO” aparece o xml dos comandos correspondentes, conforme mostra a Figura 6-22.

```
WHILE  x < 7
DO
          x := x + 1;
END_WHILE
```

```
<WHILE>
   <Bloco1 instr="LT" in1="x" in2="7" />
   <DO>
     <Bloco2 instr="ADD" in1="x" in2="1" />
     <Bloco instr="MOVE" in1="Bloco1.OUT" in2="x" />
   </DO>
 /WHILE>
```

Figura 6-22: XML de Comando WHILE

6.4.8 Tags XML de Comando REPEAT

O Comando REPEAT é inserido no XML através das *tags* “REPEAT” e “UNTIL”. Dentro da *tag* “UNTIL” aparece(m) o(s) bloco(s) com a avaliação da expressão de

condição de saída do laço REPEAT. Entre as *tags* “REPEAT” e “UNTIL” aparece o xml dos comandos correspondentes, conforme mostra a Figura 6-23.

```
REPEAT
        x := x + 1;
UNTIL x < 7
END_REPEAT
```

```
< REPEAT >
   <Bloco2 instr="ADD" in1="x" in2="1" />
   <Bloco instr="MOVE" in1="Bloco1.OUT" in2="x" />
   <UNTIL>
      <Bloco1 instr="LT" in1="x" in2="7" />
   </UNTIL >
 / REPEAT >
```

Figura 6-23: XML de Comando REPEAT

6.4.9 Tags XML de Comandos RETURN e EXIT

Os comandos RETURN e EXIT são inseridos no XML como *tags* de mesmo nome, sem conter nenhum atributo ou qualquer outro elemento XML.

6.5 Resumo das Tags e Atributos da Arquitetura XML

A Tabela 6-4 mostra as *tags* e atributos utilizados na arquitetura XML.

Tag no XML	Atributos Obrigatórios	Atributos Opcionais
ARQ	nome	-
FUNCTION	nome, type	-
PROGRAM	nome	-
VAR, VAR_INPUT, VAR_GLOBAL	nome, type	at, atrib
ATRIBUICAO	-	-
BlocoNNN, Bloco	instr, in1	in2, type, nome, qtdIn
IF, THEN, ELSE	-	-
FOR	var, to	-
CASE	-	-
OF	valor	-
REPEAT, UNTIL	-	-
WHILE, DO	-	-
RETURN	-	-
EXIT	-	-

Tabela 6-4: *Tags* da Arquitetura XML

A Tabela 6-5 mostra os valores possíveis que cada atributo pode assumir na arquitetura XML.

Atributo	Valor
nome	Nome do arquivo origem, função, programa ou variável
type	Tipo básico INT, DINT, REAL, ... ou ARRAY
at	Endereço de memória ou operando do CP
atrib	CONSTANT ou RETAIN.
instr	EQ, GT, GE, LE, LT, NE AND, OR, XOR ADD, SUB, MUL, DIV, EXPT CHF
in1, in2	Valores de entrada do bloco
var	Nome da variável do FOR (não confundir com a *tag* "VAR")
to	Valor limite da variável do FOR
valor	Valores possíveis de cláusulas OF de um comando CASE

Tabela 6-5: Valores dos Atributos na Arquitetura XML

Nesta arquitetura não estão representadas a entrada EN e saída ENO opcionais dos blocos, conforme define a norma. Seria possível, entretanto, inseri-las no XML através de dois atributos a mais na *tag* correspondente.

Também não está representado aqui os *Function Blocks*, cujas principais diferenças em relação aos *functions* são:

1) possuem variáveis internas de estado;
2) precisam ser instanciados antes de serem executados;
3) a chamada é feita pelo nome da variável instanciada e não pelo nome do *function block*.

Os *Function Blocks* poderiam ser incluídos no XML de uma forma muito semelhante ao *Function*.

6.6 DTD da Arquitetura XML

A Figura 6-24 e Figura 6-25 mostram a DTD da arquitetura XML. Estão presentes todos as *tags* e atributos definidos neste capítulo.

```
<!ELEMENT ARQ (FUNCTION* , PROGRAM?)>
<!ATTLIST ARQ nome CDATA #REQUIRED>

<!ENTITY % tiposIEC "BOOL | BYTE | WORD | DWORD | SINT | USINT |
                     INT | UINT | DINT | UDINT | REAL">

<!ELEMENT FUNCTION ANY>
<!ATTLIST FUNCTION nome    CDATA          #REQUIRED
                   type    (%tiposIEC;)   #REQUIRED>

<!ENTITY % atribsVAR  "
                   nome    CDATA                        #REQUIRED
                   type    CDATA                        #REQUIRED
                   at      CDATA                        #IMPLIED
                   atrib   (CONSTANT |  RETAIN)         #IMPLIED
                   ">
<!ELEMENT VAR (#PCDATA)>                <!ATTLIST VAR %atribsVAR;>
<!ELEMENT VAR_INPUT (#PCDATA)>          <!ATTLIST VAR_INPUT %atribsVAR;>
<!ELEMENT VAR_GLOBAL (#PCDATA)>         <!ATTLIST VAR_OUTPPUT %atribsVAR;>

<!ELEMENT IF ANY>
<!ELEMENT THEN ANY>
<!ELEMENT ELSE ANY>
<!ELEMENT ATRIBUICAO ANY>

<!ELEMENT PROGRAM ANY>
<!ATTLIST PROGRAM nome CDATA #REQUIRED>

<!ELEMENT CASE ANY>
<!ELEMENT OF ANY>
<!ATTLIST OF            valor   CDATA          #REQUIRED>

<!ELEMENT REPEAT ANY>
<!ELEMENT UNTIL ANY>
<!ELEMENT WHILE ANY>
<!ELEMENT DO ANY>
```

Figura 6-24: DTD da Arquitetura XML (1ª parte)

A sintaxe desta DTD foi verificada com o *parser* MSXML da Microsoft, disponível no *browser* Internet Explorer a partir da versão 5. Para executar estes testes, foram utilizados os arquivos da arquitetura XML gerados pelos exemplos descritos no Capítulo 8, Estudo de Caso.

```
<!ELEMENT FOR ANY>
<!ATTLIST FOR          var      CDATA          #REQUIRED
                       to       CDATA          #REQUIRED>

<!ELEMENT RETURN EMPTY>
<!ELEMENT EXIT EMPTY>

<!ENTITY % atribsBloco          "
                instr      (EQ | GT | GE | LE | LT | NE  | AND | OR | XOR |
                            ADD | SUB | MUL | DIV | EXPT | CHF | MOVE) #REQUIRED
                in1 CDATA #REQUIRED
                in2 CDATA #IMPLIED
                type CDATA #IMPLIED
                nome CDATA #IMPLIED
                qtdIn CDATA #IMPLIED">

<!ELEMENT Bloco EMPTY>
<!ATTLIST Bloco %atribsBloco;>
```

Figura 6-25: DTD da Arquitetura XML (2ª parte)

6.7 Comparação com IL

Em ([LEW95]), R. W. Lewis cita que “a linguagem IL é muitas vezes referenciada como sendo uma linguagem base para a qual todas as outras linguagens da norma IEC 1131-3 podem ser convertidas” .

A norma, entretanto, não impõe que nenhuma linguagem particular seja tratada como uma linguagem base. A linguagem IL é simplesmente outra linguagem que pode ser usada, se desejado, para solucionar determinados tipos de problema.

O *software* Step7, da Siemens, adotou a IL como linguagem base. Nele, os programas escritos em linguagem LD e FBD sempre podem ser convertidos para IL, mas o contrário não é verdadeiro. Em determinadas situações, a conversão de IL para LD ou FBD é possível, em outras não. Outros fabricantes não tratam a IL como linguagem base, ou simplesmente não a possuem.

A questão relevante aqui é o quanto a arquitetura XML definida neste trabalho é semelhante a linguagem IL. O XML é parecido com a linguagem *assembly*, a IL também, seus conteúdos não seriam o mesmo, então ? Nosso XML não seria simplesmente o IL convertido em XML ?

A primeira vista parece que sim. A declaração de variáveis é idêntica, a razão disto é que esta parte da norma é comum às 5 linguagens, logo a declaração de variáveis é sempre idêntica, qualquer que seja a linguagem utilizada.

Uma analise mais profunda mostra que a arquitetura XML oferece ou pode oferecer as seguintes funcionalidades que não seriam possíveis com a linguagem IL:

- o XML permite restaurar o programa fonte ST original através das *tags* IF, FOR, CHF, *et cetera*;
- a linguagem IL não possui a ordem de execução dos blocos, que pode ser importante em diagramas LD e FBD, enquanto que o XML possui;
- a linguagem IL não representa a conexão entre blocos da forma que acontece no XML;
- o XML possibilita expansões a critério do implementador que não geram incompatibilidades com o formato anterior.

6.8 Conclusão

XML é uma linguagem de marcação amplamente difundida na Web que permite garantir a interoperabilidade entre dados e aplicações. Foi particularmente útil na definição desta arquitetura porque, por ser extensível, permitiu que as *tags* e atributos fossem criados com os nomes mais apropriados.

As *tags* e atributos que formam a arquitetura XML foram definidos neste capítulo. A representação de LD e FBD foi baseada na representação textual descrita no Capítulo 4. Na representação de CLPScript e dos aspectos comuns a todas as linguagens da norma IEC 1131-3 procurou-se , sempre que possível, manter no XML a mesma nomenclatura original da IEC.

O próximo capítulo descreve a implementação da CLPScript que gera e formaliza a arquitetura XML proposta, através do programa CLPStoXML desenvolvido neste trabalho.

7 A Implementação da Arquitetura XML com CLPScript

Resumo

Este capítulo descreve a implementação do programa CLPStoXML, um compilador que lê um programa CLPScript e gera a arquitetura XML proposta.

Este capítulo apresenta o programa **CLPStoXML.EXE**, um compilador CLPScript para XML desenvolvido durante os trabalhos desta dissertação. O programa CLPStoXML lê o arquivo de entrada selecionado, compila-o conforme a sintaxe da linguagem CLPScript definida no capítulo 5, “A Linguagem CLPScript”, e gera o arquivo XML com a arquitetura definida no capítulo 6, “A Arquitetura XML”.

O CLPStoXML envolve o conhecimento de toda uma área de compiladores ([AHO86], [GRU01]): analisadores léxico, sintático, semântico, tabela de símbolos, gerador de código, tratamento de erros, ficando de fora, neste caso, gerador de código intermediário e otimizador de código.

7.1 O programa CLPStoXML

Um compilador é um programa que lê um programa escrito numa linguagem – a linguagem fonte – e o traduz num programa equivalente numa outra linguagem – a linguagem alvo ([AHO86]).

O programa **CLPStoXML** é um compilador de CLPScript para XML. Foi desenvolvido em Visual C++ 6.0 ([KRU97], [DAT00]) utilizando MFC (*Microsoft Foundation Classes*) ([KAI99]) da Microsoft.

Foi implementado um analisador sintático *top-down* preditivo ([GRU01], [PRI00]), escrito manualmente, sem usar geradores de código tais como *lex & yacc* ([LEV92]), por exemplo. Consiste em um conjunto de rotinas recursivas, onde cada rotina corresponde a uma regra na gramática da linguagem. Tal analisador é chamado analisador descendente recursivo, e funciona com gramáticas LL(1)[2] ([GRU01]).

O CLPStoXML foi escrito baseado no código fonte de outro *software*, o ILA - Interpretador de Linguagem Algoritmica ([ILA91]). o Projeto ILA foi desenvolvido na Universidade do Vale do Rio dos Sinos, sob orientação do Prof. Dr. Sérgio Crespo, como uma alternativa de minimizar problemas de construção de algoritmos . ILA não é propriamente um ambiente, mas sim um interpretador, pequeno e versátil, que permite o teste e execução de algoritmos em um português estruturado ([ILA91]). O ILA versão 1.01 está disponível para download a partir do site http://www.inf.unisinos.br/~crespo .

O compilador **CLPStoXML** possui as seguintes diferenças em relação ao ILA:

- o ILA possui os tipos “Numérico” e “Caracter”, enquanto que o CLPStoXML não possui o “Caracter” mas possui diferentes tipos numérico, tais como REAL, INT, SINT, UINT, LREAL, dentre outros, e faz a verificação de tipo entre eles por ser uma linguagem fortemente tipada;
- o ILA não tem conversão de tipos, nem explícita, através de rotinas de conversão, e nem implícita (coerção);
- o ILA é um interpretador, e para no primeiro erro que encontra. CLPStoXML, por ser um compilador, não deve para no 1º erro, tenta se recuperar e continuar compilando;
- o ILA processa as funções a medida em que são chamadas. O CLPStoXML deve processa-las na ordem que se encontram no fonte;
- o ILA não gera código, ele executa o código e armazena os valores das variáveis. O CLPStoXML gera o código XML.

[2] LL é uma classe de gramática de linguagens que permite a utilização de *parsers top-down* sem a necessidade de *backtracking*. O primeiro "L" refere-se a busca da esquerda para a direita e o segundo "L" refere-se a derivação mais a esquerda. Normalmente é referenciada na forma LL(k), onde "k" é o número de *tokens look-ahead* necessários na análise de uma sentença da linguagem ([MUC97]).

7.1.1 As Funções Pré Definidas do CLPStoXML

Dentre as funções pré definidas especificadas na norma IEC 1131-3, as listadas a seguir foram implementadas no CLPStoXML:

- ABS
- SQRT
- LN
- LOG
- EXP
- SIN
- COS
- TAN
- ASIN
- ACOS
- ATAN
- INT_TO_REAL
- INT_TO_DINT
- DINT_TO_REAL
- DINT_TO_INT
- REAL_TO_INT
- REAL_TO_DINT

O compilador CLPStoXML faz a verificação de quantidade de parâmetros e tipo de cada um nas chamadas de função, tanto para as funções pré definidas quanto para as criadas pelo usuário.

7.1.2 A Interface do CLPStoXML

A Figura 7-1 mostra o programa CLPStoXML com duas janelas: a primeira onde visualiza-se o programa fonte, e a segunda, com os resultados da compilação. Esta figura mostra que o programa gerou um erro na linha 9, pois a variável “i” estava sendo usada sem ter sido declarada.

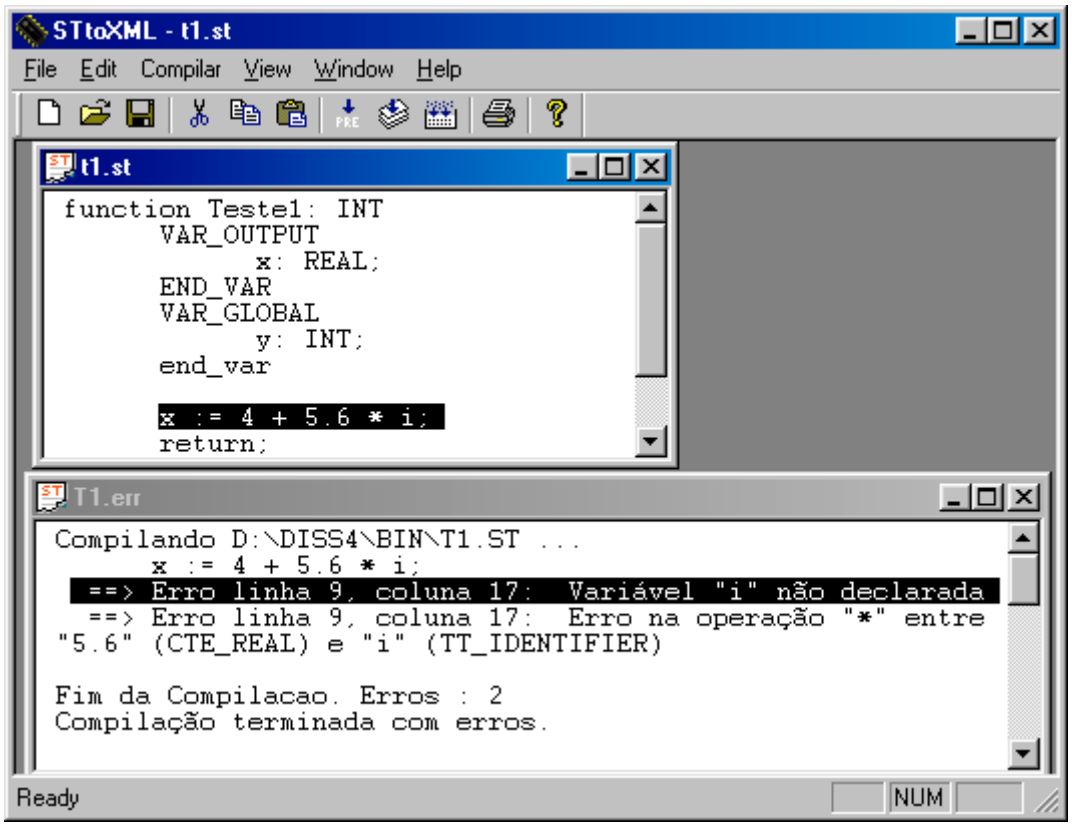


Figura 7-1: Compilador CLPStoXML

O programa CLPStoXML lê o arquivo de entrada selecionado, compila-o conforme as regras da linguagem CLPScript definidas no capítulo 5, e gera o arquivo XML com o programa compilado.

A seleção do arquivo de entrada é feita da forma padrão do Windows, pelo menu “Arquivo \ Abrir”. A operação de compilação é iniciada a partir do menu “Compilar”, que possui as seguintes opções:

- **Pré Compilador**: executa o pré compilador, que retira os comentários antes de compilar;
- **Léxico**: executa o analisador léxico, que quebra o arquivo original em uma seqüência de tokens correspondentes;
- **Sintático**: executa os analisadores sintático e semântico, que fazem a compilação propriamente dita;
- **Semântico**: mesma opção que a anterior, mantida apenas para deixar referenciado as principais etapas de um compilador.

7.1.3 Diagrama UML do CLPStoXML

A Tabela 7-1 mostra as classes principais do programa CLPStoXML. Não estão listadas as classes relacionadas com o ambiente de programação Windows.

Classe	Descrição
CToken	Armazena um token. Um token é uma seqüência caracteres que possui um significado coletivo
CDataType	Representa os diversos tipos de dados básicos e arrays
CLexico	Contém o analisador léxico, que lê o programa de entrada e gera os tokens correspondentes
CParametros	Representa a lista de parâmetros passada para uma rotina e também a declarada em uma rotina
CSintatico	Contém o analisador sintático e semântico, é a classe principal, gera os erros de compilação e o XML final
CTabelaFuncoes	Armazena as funções pré definidas da IEC 1131-3
CTabelaSimbolos	Armazena todas as informações relacionadas às variáveis e funções criados ao longo do programa fonte
CCLPScriptXML	Centraliza tudo relacionado com XML, contém todos os *tags* e atributos usados.
CSTCompiler	Executa os analisadores léxico e sintático e as demais operações da compilação
CUtil	Rotinas utilitárias genéricas

Tabela 7-1: Classes Principais do Programa CLPStoXML

A classe *CSTCompiler* instancia um objeto analisador léxico, classe *CLexico*, e um objeto analisador sintático/semântico, classe *CSintatico,* conforme mostra a Figura 7-2. Inicialmente uma pré compilação é executada, onde retira-se todos os comentários do fonte. Em seguida, executa-se o analisador léxico, que lê o arquivo pré compilado quebrando-o em uma lista de tokens correspondentes, classe *CToken*. Esta lista de tokens é passada para o analisador sintático/semântico, que mantém um ponteiro para o token atual e um para o próximo. Estes dois tokens são os únicos necessários para verificar se a sintaxe e semântica estão de acordo com a BNF da linguagem, uma vez que o analisador é do tipo *top-down* preditivo.

A classe *CSintatico* possui uma tabela de funções e uma tabela de símbolos, classes *CTabelaFuncoes* e *CTabelaSimbolos*, respectivamente. A tabela de funções contém as funções pré definidas especificadas na seção 7.1.1 e vai sendo acrescentada com novas funções definidas pelo usuário presentes no arquivo sendo compilado. Uma função contém sempre uma lista de parâmetros, *CParametros*, que é comparada pelo analisador para verificar se os parâmetros passados na chamada correspondem aos parâmetros

declarados na função, em termos de quantidade e tipo, este último determinado pela classe *CDataType*, que trata tipos básicos e arrays de tipos básicos. A tabela de símbolos vai sendo preenchida durante a compilação, conforme as variáveis e funções vão sendo criados. Cada símbolo contém seu escopo, token original e tipo.

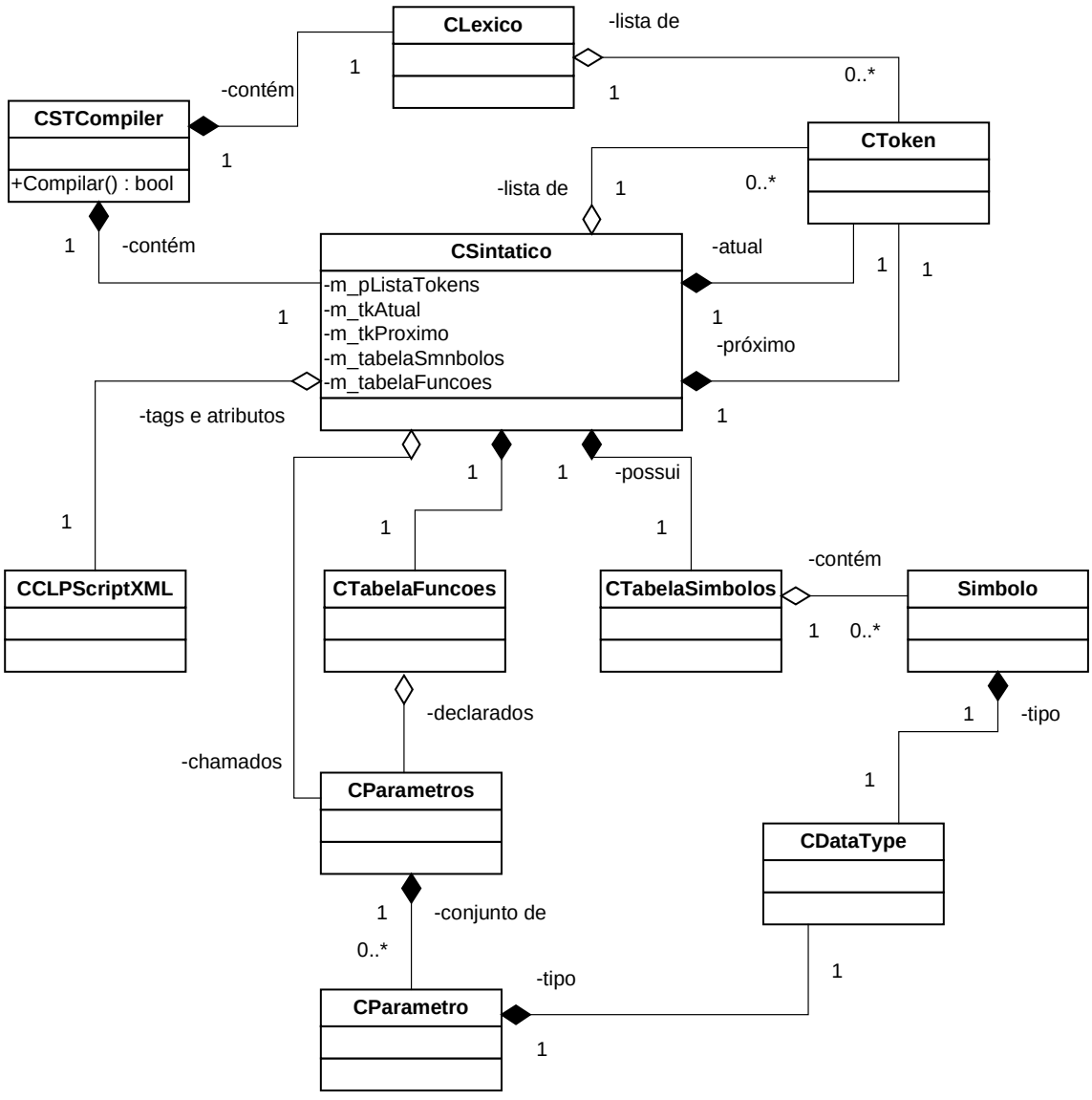


Figura 7-2: Diagrama de Classes do Programa CLPStoXML

O arquivo XML destino vai sendo montado por *CSintatico* durante sua execução, através de um objeto do tipo *CCLPScriptXML*. Caso algum erro de compilação seja encontrado, ele é mostrado na janela de erros e o XML não é gerado. A Figura 7-2

mostra o digrama de classes UML do programa CLPStoXML, resumido por questões de clareza. A Figura 7-3 mostra a definição completa da classe principal do compilador CSintatico.

Figura 7-3: Definição da Classe *CSintatico* de CLPStoXML

7.2 BNF da Linguagem CLPScript

Esta seção apresenta a BNF da linguagem CLPScript, gerada durante a implementação do compilador CLPStoXML. Alguns termos utilizados nesta documentação possuem intencionalmente o mesmo nome de métodos da classe *CSintatico*.

A Tabela 7-2 mostra a BNF de identificadores e constantes. As demais tabelas complementam a BNF, dividida por categorias.

Letter	::=	‘A’ - ‘Z’ | ‘a’ - ‘z’
Digit	::=	‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’
OctalDigit	::=	‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’
cte	::=	`Digit {Digit}*`
ID	::=	`Letter {Letter | Digit}*`

Tabela 7-2: BNF de Identificadores e Constantes da CLPScript

7.2.1 Declaração de Variáveis

A Tabela 7-3 mostra a BNF de declaração de variáveis da CLPScript.

VarSection	::=	**VAR** | **VAR_INPUT** | **VAR_GLOBAL**
Atributo	::=	**RETAIN | CONSTANT**
DeclarVar	::=	VarSection [Atributo] { ID {**,** ID}* [Location]**:** TIPO **;** }* **END_VAR**
TipoBasico	::=	**BOOL** | **INT** | **DINT** | **REAL**
Array	::=	**ARRAY [** cte **..** cte **] OF** TipoBasico
Tipo	::=	TipoBasico | Array
Location	::=	**AT** DirectVariable | OperandoAltus
DirectVariable	::=	**ID**
OperandoAltus	::=	‘M’ cte | ‘F’ cte | ‘D’ cte | ‘A’ cte.OctalDigit

Tabela 7-3: BNF de Declaração de Variáveis da CLPScript

7.2.2 Functions e Programs

A Tabela 7-4 mostra a BNF de *Functions* e *Programs* da CLPScript.

Function	::=	**FUNCTION** ID **:** Tipo { DeclarVar }* { Statement }* **END_FUNCTION**
Program	::=	**PROGRAM** ID { DeclarVar }* { Statement }* **END_PROGRAM**

Tabela 7-4: BNF de *Function* e *Program* da CLPScript

7.2.3 Expressões

A Tabela 7-5 mostra a BNF de expressões da CLPScript.

Operator	::=	'=' | '<>' | '>' | '>=' | '<' | '<=' | AND | OR | XOR | '+' | '-' | '*' | '/'
ChamadaFuncao	::=	ID '(' Expressão {**,** Expressão }* ')'
IDArray	::=	ID '[' Expressão {**,** Expressão }* ']'
Expressão	::=	ID | cte | ChamadaFuncao | IDArray | '(' Expressão ')' | Expressão Operator Expressão

Tabela 7-5: BNF de Expressões da CLPScript

7.2.4 Statements

A Tabela 7-6 mostra a BNF de *Statements* da CLPScript.

Statement	::=	ComandoAtribuicao | ComandoIf | ComandoReturn | ComandoFor | ComandoExit | ComandoChamadaFuncao
Statements	::=	Statement | {Statement}*
ComandoAtribuicao	::=	ID | IDArray **:=** Expressão **;**
ComandoIf	::=	**IF** Expressão **THEN** Statements [**ELSE** Statements] **END_IF ;**
ComandoReturn	::=	**RETURN ;**
ComandoFor	::=	**FOR** ID **:=** Expressão **TO** Expressão **DO** { Statements } **END_FOR**;
ComandoExit		**EXIT ;**
ComandoChamadaFuncao		ChamadaFuncao **;**

Tabela 7-6: BNF de *Statements* da CLPScript

7.2.5 Simbologia Utilizada

A Tabela 7-7 mostra a simbologia utilizada na descrição da BNF da linguagem CLPScript.

Símbolo	**Significado**
Barra vertical ‘|’	Significa “ou”
Parênteses ‘(‘ e ‘)’	Usados para agrupar expressões
Asterisco ‘*’	Significa “zero ou mais ocorrências”
Colchetes ‘[‘ e ‘]’	Significam “zero ou uma ocorrência”, conforme norma IEC 1131-3, página 269.
::=	Significa “produção”.
Sem negrito	Símbolos “não terminais”
negrito	Símbolos “terminais”, palavras reservadas
Chaves ‘{‘ e ‘}’	Significam “zero ou mais ocorrências”, conforme norma IEC 1131-3, página 269.

Tabela 7-7: Simbologia utilizada na BNF da CLPScript

7.3 Conclusão

Este capítulo descreveu a implementação da linguagem específica de domínio CLPScript através do programa CLPStoXML.

O CLPStoXML é um compilador lê um programa CLPScript, conforme definido no Capítulo 5, e gera a arquitetura XML correspondente., definida no Capítulo 6. Foi escrito em Visual C++ 6.0 da Microsoft, baseado no *software* ILA – Interpretador de Linguagem Algoritmica.

Internamente, as classes mais importantes são as que contêm os analisadores léxico, sintático e semântico, estes dois últimos implementados juntos. O analisador sintático/semântico é do tipo *top-down* recursivo, e funciona com gramáticas LL(1). O XML foi gerado diretamente pelo código, nenhum ferramenta externa foi utilizada neste caso.

O programa CLPStoXML mostrou-se complexo de ser implementado, porque demandou muito conhecimento da área de compiladores. O próximo capítulo mostra um estudo de caso que visa validar esta arquitetura XML gerada aqui.

8 Estudo de Caso

Resumo

O estudo de caso deste capítulo selecionou um fabricante de controladores programáveis para certificar que seu *software* programador consegue ler o XML proposto. Um *software* específico é desenvolvido para converter do XML para a linguagem LD utilizada por este fabricante, visando prova de conceito.

Com o objetivo de certificar e validar a arquitetura XML apresentada e verificar sua usabilidade, foi realizado um estudo de caso onde se converteu alguns programas fonte escritos em CLPScript para XML e deste XML para a linguagem LD de uma empresa fabricante de controladores programáveis, a Altus S. A.

A escolha da Altus S. A. se deve a duas razões principais: primeiro, é uma empresa bem conhecida no mercado de automação industrial, cujo *software* programador possui a linguagem LD e entendeu-se como melhor alternativa utilizar o visualizador de LD deste *software* em vez de construir do zero um novo visualizador. Segundo, o autor desta dissertação é colaborador da Altus a mais de 5 anos, tendo portanto um conhecimento prévio dos produtos da empresa e acesso à documentações internas necessárias a este tipo de implementação. Salienta-se que parte desta documentação é informação estratégica da empresa, tecnologia própria, patenteada e tratada com sigilo.

O estudo de caso visa mostrar que os controladores programáveis Altus e seu *software* programador conseguem ler o XML, uma vez que esta arquitetura pode ser adaptada para qualquer fabricante. Para realizar esta prova de conceito, desenvolveu-se o *software* **XMLtoLD**, descrito neste capítulo, que converte do XML para o LD Altus. Salienta-se que o XMLtoLD foi desenvolvido especificamente para o Altus. A partir do XML, cada fabricante é responsável por gerar seu próprio XMLtoLD.

Considerando que o foco do estudo de caso é a prova de conceito e não o volume de experimentos, neste capítulo são apresentados dois exemplos de conversão de XML para LD Altus. No CDROM que acompanha esta dissertação podem ser encontrados outros exemplos.

8.1 As Ferramentas Necessárias

Para tornar este estudo de caso possível, foram desenvolvidas duas ferramentas em Visual C++ :

- **CLPStoXML**: compilador CLPScript para XML, descrito no capítulo anterior;
- **XMLtoLD**: conversor de XML para LD Altus

O programa XMLtoLD lê um arquivo XML gerado pelo CLPStoXML, interpreta suas *tags* e atributos e gera um arquivo no formato binário contendo o *assembly* utilizado pelos controladores programáveis Altus ([ALT98a]) e pelo respectivo *software* de programação *MasterTool Programming MT4100* ([ALT03]). A Figura 8-1 mostra o MasterTool editando uma instrução de chamada de função.

O XMLtoLD envolve o conhecimento do *assembly* do microcontrolador Intel 8051 e 80C251 ([INT96]), utilizado nos controladores programáveis Altus, e do formato interno das instruções LD geradas pelo *software* programador MasterTool MT4100. Se o *assembly* não é gerado exatamente como o original, o MasterTool não o reconhece e não consegue desenhar o diagrama LD correspondente.

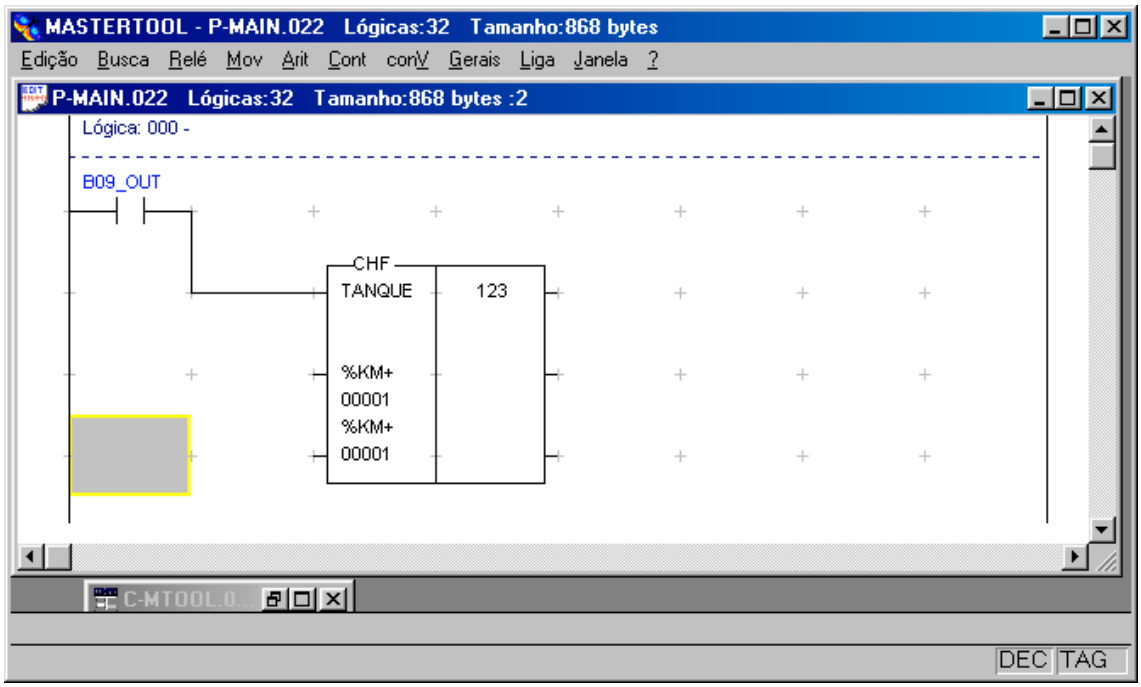


Figura 8-1: Programador de CPs Altus MasterTool MT4100

8.2 A Linguagem LD Altus

A Linguagem LD, nos controladores programáveis Altus, é denominada *Linguagem de Diagramas de Relés*. Um programa aplicativo é um programa LD que executa em um CP, que no caso da Altus é composto por 4 elementos básicos ([ALT03]):

- módulos
- lógicas
- instruções
- operandos

Um programa aplicativo é composto por diversos **módulos**, permitindo uma melhor estruturação das rotinas de acordo com as suas funções. Os módulos são programados em linguagem de relés ([ALT03]). Um módulo é um arquivo cujo nome possui até 6 caracteres, com extensão numérica, contendo uma função ou um programa.

Um módulo de programa aplicativo é dividido em **lógicas de programação**. O formato de uma lógica de programa aplicativo utilizado nos CPs Altus permite até oito elementos em série e até quatro caminhos em paralelo.

As **instruções** são utilizadas para executar determinadas tarefas por meio de leituras e/ou alterações do valor dos operandos. A Tabela 8-1 mostra algumas das instruções dos CPs Altus.

Instrução	Descrição	Visualização
RNA	Contato normalmente aberto	“--[ ]--"
RNF	Contato normalmente fechado	“--[/]--"
BOB	Bobina simples	“--( )--"
SLT	Bobina de salto	“--(S)--"
MOV	Movimentação de operandos simples	“MOV"
MOP	Movimentação de partes de operandos	“MOP"
MOT	Movimentação de tabela de operandos	“MOT"
SOM	Soma	“SOM"
SUB	Subtração	“SUB"
MUL	Multiplicação	“MUL"
DIV	Divisão	“DIV"
AND	Função "e" binário entre operandos	“AND"
OR	Função "ou" binário entre operandos	“OR-"
XOR	Função "ou exclusivo" binário entre operandos	“XOR"
CAR	Carrega operando	“CAR"
IGU	Igual	“=“
MEN	Menor	“<“
MAI	Maior	“>“
CHF	Chama Módulo Função	“CHF"
LGH	Ligação horizontal	“"
LGV	Ligação vertical	“"

Tabela 8-1: Instruções de CPs Altus

Cada uma destas instruções ocupa uma quantidade determinada de linhas e colunas, e possui uma quantidade determinada de operandos parametrizáveis e de entradas e saídas. Esta informação é essencial no momento da geração do código *assembly* da linguagem LD ([ALT98b]).

Os **operandos** identificam diversos tipos de variáveis e constantes utilizadas na elaboração de um programa aplicativo, podendo ter seu valor modificado de acordo com a programação realizada. A Tabela 8-2 mostra os tipos de operandos existentes em CPs da Altus.

Tipo	Operando	Tamanho em Bytes
%E	Relés de Entrada	1
%S	Relés de Saída	1
%R	Endereço no Barramento	1
%A	Relés Auxiliares	1
%M	Memórias	2
%D	Decimais	4
%F	Reais	4
%KM	Constantes Memórias	2
%KD	Constantes Decimais	4
%KF	Constantes Reais	4
%TM	Tabelas Memórias	2 (por posição)
%TD	Tabelas Decimais	4 (por posição)
%TF	Tabelas Reais	4 (por posição)

Tabela 8-2: Operandos de CPs Altus

8.3 O programa XMLtoLD: Conversor XML para LD Altus

O programa **XMLtoLD** lê um arquivo XML gerado pelo CLPStoXML, interpreta suas *tags* e atributos e gera um arquivo no formato binário contendo o *assembly* dos microcontroladores Intel 8051 e 80C251 ([INT96]) utilizado pelos controladores programáveis Altus.

O programa XMLtoLD foi desenvolvido em Visual C++ 6.0 ([HOL01]) com MFC ([KAI99]) e dividido estruturalmente em três grandes módulos:

- **Módulo 1: A leitura do arquivo XML**, efetuada utilizado-se um *parser* XML de domínio público;
- **Módulo 2: A geração de um arquivo texto intermediário** com os mnemônicos das instruções *assembly* Altus;
- **Módulo 3: A geração do código binário** LD Altus a partir deste arquivo intermediário.

Denominou-se este arquivo intermediário com os mnemônicos *assembly* Altus de **ProLD**. Seu objetivo é representar as instruções LD na forma textual para em seguida serem convertidas para código binário; Os arquivos deste tipo são gerados com a extensão “.PLD” .

Este processo é semelhante ao utilizado em compiladores tradicionais, onde a linguagem fonte original é convertida em um arquivo ASM com os mnemônicos do processador e então o **montador** *assembly* é chamado para gerar o código binário final, a partir deste ASM. O benefício ganho com esta divisão é que isola-se as particularidades dos códigos binários do *assembly* da etapa de leitura e interpretação do XML. O problema original fica dividido em dois problemas menores, com uma interface bem definida, facilitando a implementação das partes, que ficam independentes uma da outra. Facilita também a depuração, pois é muito mais simples verificar se um arquivo texto foi gerado corretamente do que um arquivo binário.

Os programas aplicativos Altus possuem a restrição de uma única função por arquivo, nomes de função com no máximo de 6 caracteres e limite máximo de 10 parâmetros de entrada e 10 de saída. Sendo assim, o programa XMLtoLD pode gerar mais de um arquivo destino para um mesmo XML de origem, conforme o número de funções presentes no XML. Os nomes de função com mais que 6 caracteres são truncados.

As seções a seguir descrevem os três módulos citados do XMLtoLD.

8.3.1 Módulo 1: A Leitura do XML

A leitura do arquivo XML foi implementada utilizando-se o *parser* **expat** ([EXP98]), uma ferramenta de domínio público para trabalhar com arquivos XML em C++, disponível para download em http://www.jclark.com/xml/expat.html. O expat é um analisador sintático de não validação, ou seja, ele verifica se o XML é bem formado, mas não consiste as regras da DTD.

O expat trabalha com o conceito de processamento dirigido a eventos: seu analisador sintático notifica o programa principal dos eventos que ocorrem enquanto o documento XML é lido. Alguns exemplos destes eventos são encontrar uma nova *tag* de início ou encontrar mais dados de caracteres ([ARC02]).

Ao utilizar o *parser* expat, um programa deve registrar os eventos para os quais deseja receber notificação. Notificar um evento significa chamar a rotina registrada, ou seja, quando o evento acontece, o *parser* é quem chama uma rotina no programa principal (atendimento ao evento).

No caso do XMLtoLD, os eventos registrados são “startElement” e “endElement”, correspondendo a eventos de início e fim de *tag*, respectivamente.

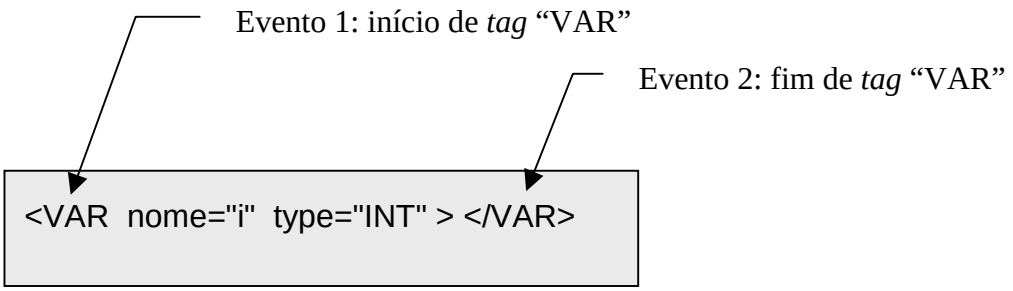


Figura 8-2: Exemplo de Eventos do Parser Expat

A Figura 8-2 mostra um exemplo de eventos com o *parser* expat. Ao processar o trecho de código XML da figura, o analisador sintático do expat gera os eventos de 1) início de *tag* “VAR” e 2) fim de *tag* “VAR”, chamando as rotinas “startElement” e “endElement”, respectivamente. Os atributos “nome” e “type” são passados como parâmetro para a rotina “startElement”, junto com seus valores “i” e “INT” correspondentes.

Os arquivos referentes ao *parser* expat incluídos no projeto do Visual C++ são “Xmlparse.lib”, “xmlparse.h”, “xmlparse.dll” e “xmltok.dll”.

8.3.2 Módulo 2: A geração do ProLD

Cada *tag* e atributo lido do arquivo XML pode disparar a alocação de um novo operando do CP ou a geração de uma ou mais instruções LD correspondentes. Estas instruções são gravadas no arquivo texto ProLD, para que numa terceira etapa sejam convertidas no formato de código binário Altus.

8.3.2.1 Alocação de Operandos

Durante a leitura do XML, todas as variáveis declaradas (*tags* “VAR”, “VAR_INPUT” e “VAR_GLOBAL”) devem ser alocadas em operandos do CP correspondentes, pois no contexto do CP não existem “variáveis”, mas sim operandos, conforme a Tabela 8-2, mostrada anteriormente.

Para esta alocação devem ser selecionados operandos correspondentes em termos de tamanho, formato interno e funcionalidade. O programa XMLtoLD faz as alocações para operandos Altus da seguinte forma:

Tipo Básico IEC 1131-3	Operando Altus correspondente	Operando Inicial
INT	%M	%M10
DINT	%D	%D10
REAL	%F	%F10
BOOL	bit de %A	%A0.0

Tabela 8-3: Alocação de Operandos para as Variáveis

A Tabela 8-3 mostra a alocação de operandos Altus para as variáveis encontradas no XML. Os operandos iniciais também são mostrados na figura, e podem ser configurados no XMLtoLD. Por exemplo, a primeira variável do tipo "INT" encontrada no XML é alocada para o operando "M10", a segunda para o "M11", *et cetera*.

Os Blocos (*tags* "BlocoNNN") também precisam de operandos alocados para suas saídas. A arquitetura XML utiliza blocos de duas entradas e apenas uma saída, logo apenas um operando deve ser alocado a cada bloco. O tipo deste operando pode variar conforme o bloco ou conforme os operandos de suas entradas, conforme mostra a Tabela 8-4. Por exemplo, um bloco "ADD" com uma entrada do tipo "INT" e outra "REAL" vai ter uma saída do tipo "REAL".

Blocos do XML	Tipo Alocado para a Saída
EQ, GT, GE, LE, LT, NE	BOOL
AND, OR, XOR	BOOL
ADD, SUB, MUL, DIV, EXPT	Depende das entradas
CHF	Definido no XML

Tabela 8-4: Alocação de Operandos para Saída de Blocos

8.3.2.2 Gerando as Instruções

Cada atributo "instr" da *tag* "BlocoNNN" gera no arquivo ProLD uma ou mais instruções LD Altus de função equivalente, conforme resumido a seguir:

- ADD, SUB, MUL, DIV: correspondem diretamente a instruções Altus;

- EQ, GT, GE, LT, LE e NE: correspondem a várias instruções Altus, conforme mostra a Figura 8-3;
- AND, OR, XOR: correspondem às instruções Altus conforme mostra a Figura 8-4;
- CHF: corresponde diretamente a instrução CHF.

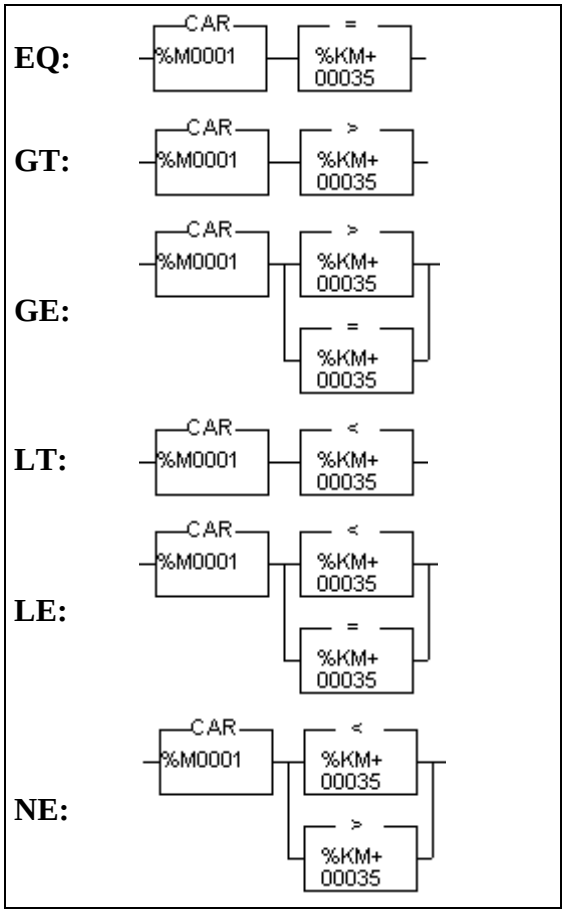


Figura 8-3: Instruções EQ, GT, GE, LT, LE e NE

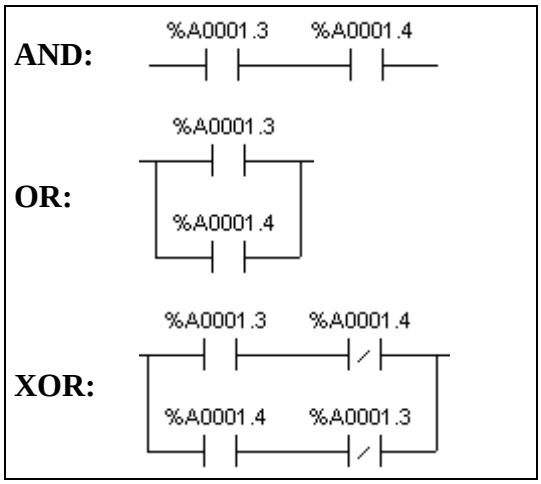


Figura 8-4: Instruções AND, OR e XOR

8.3.2.3 Gerando os Comandos IF

Cada *tag* "IF" encontrado no XML aloca um novo operando do tipo BOOL, que recebe o resultado da avaliação da expressão de condição do IF.

Uma *tag* "THEN" faz com que o programa XMLtoLD passe a acrescentar no ProLD uma instrução contato RNA, com o resultado da condição do IF, em série com os demais blocos que se localizam dentro do contexto desta *tag* "THEN", no XML.

Da mesma forma, uma *tag* "ELSE" após o final de uma "THEN" (o gerador de arquitetura CLPScript garante que foi gravado nesta ordem, senão geraria erro de compilação) faz com uma instrução contato RNF seja acrescentada, em vez de uma RNA.

O fim da *tag* "THEN", quando encontrado no XML, termina este processo. Quando existirem IFs aninhados, os contatos RNA e/ou RNF serão acrescentados em seqüência, até um limite de 7 IFs aninhados.

Os demais comandos condicionais são gerados de forma semelhante ao comando IF.

8.3.3 Módulo 3: A conversão para LD

Este módulo é responsável por gerar o código binário LD de uma função Altus a partir do arquivo texto ProLD, gerado na etapa anterior. Cada linha do ProLd gera uma ou mais instruções LD Altus.

É neste módulo que aparecem as particularidades do *assembly* Altus, tais como número de linhas e colunas de cada lógica, ligações verticais e horizontais, marcas de início de colunas, marcas de início de nova lógica, cabeçalhos de arquivo, *checksum, et cetera*.

Uma outra particularidade é o chamado otimizador de código existente no MasterTool, que, por questões de desempenho, procura ao longo do programa instruções que possam ser suprimidas sem que causem alterações na execução do programa. Pode-se citar, como exemplo, duas instruções MOV consecutivas movendo o mesmo valor para o mesmo registrador. Uma destas instruções poderia ser removida, e é isto que o otimizador de código faz. O programa XMLtoLD precisou levar em consideração esta informação para que o código gerado pudesse ser reconhecido no MasterTool.

O módulo 3 depende unicamente do arquivo ProLD para ser executado. O arquivo ProLD possui as seguintes características:

- uma linha corresponde a ou várias instruções e seus parâmetros, inserida numa nova lógica. Para colocar duas instruções conectadas na mesma lógica, utiliza-se o caracter ';' na mesma linha de ProLD;
- não contem nomes de variável, apenas operandos do CP. Todas as variáveis foram alocadas em operandos do CP no módulo 2;
- linhas começando com "//" significam comentários, e portanto são desprezadas.

A Tabela 8-5 mostra as instruções aceitas no arquivo ProLD e o número de operandos esperado em cada uma. A palavra “Macro” foi usada para ressaltar que uma linha ProLd vai gerar mais do que uma instrução Altus.

Descrição	Formato da Instrução ProLD
Instruções de 1 operando	RNA op1 RNF op1 BOB op1
Macros de 2 operandos de entrada e um de saída	EQ op1 op2 op3 GT op1 op2 op3 GE op1 op2 op3 LE op1 op2 op3 LT op1 op2 op3 NE op1 op2 op3 AND op1 op2 op3 OR op1 op2 op3 XOR op1 op2 op3
Instruções de 2 operandos	MOVE op1 op2
Instruções de 3 operandos	ADD op1 op2 op3 SUB op1 op2 op3 MUL op1 op2 op3
Instruções de 4 operandos	DIV op1 op2 op3 op4
Outras	CHF op1 op2 op3 op4 (paramIn) (paramOut)

Tabela 8-5: Instruções Aceitas no Arquivo ProLD

8.3.4 A Interface do XMLtoLD

A Figura 8-7 mostra a interface do programa XMLtoLD.

Par utilizar o programa XMLtoLD, primeiro deve-se selecionar o arquivo XML a ser utilizado, em seguida gerar o arquivo ProLD (módulos 1 e 2), e por último gerar o arquivo LD Altus (módulo 3). Estes três comandos podem ser executados através do menu “File” ou da barra de ícones, conforme mostra a Figura 8-5.

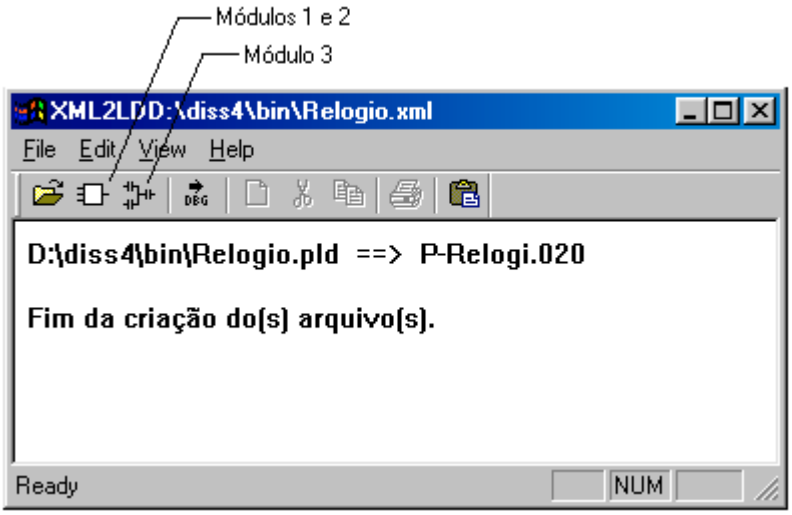


Figura 8-5: O programa XMLtoLD

Após gerar o LD Altus, o(s) módulo(s) criado(s) pode(m) ser visualizado(s) ou editado(s) no MasterTool. Se desejado, pode-se inclusive carregar um ou vários destes módulos em um controlador programável e executá-lo(s).

O programa XMLtoLD cria também um arquivo texto com a descrição dos operandos alocados, no formato que o MasterTool conhece e sabe importar, de tal forma que pode-se visualizar o diagrama LD com as descrições originais dos nomes de variáveis CLPScript.

8.4 Estudo de Caso 1

Este estudo de caso inicia-se a partir de um pequeno programa CLPScript, que é convertido para a arquitetura XML e em seguida para o LD Altus.

A Figura 8-6a mostra um pequeno programa CLPScript com um trecho de código de um *software* de contagem de tempo. Este programa foi compilado com o programa CLPStoXML gerando automaticamente o XML mostrado na Figura 8-6b.

a) Programa CLPScript	b) Arquitetura XML

Program Relogio
Var
hora: INT;
minuto: INT;
end_var
 	
if minuto >= 59
then
minuto **:=** 0;
hora **:=** hora +1;
else
minuto **:=** minuto +1;
end_if;
end_program
	<?xml version="1.0"?>
	<ARQ nome="D:\DISS4\BIN\RELOGIO.XML">
	<**PROGRAM** nome="Relogio" >
	<**VAR** nome="hora" type="INT" > </VAR>
	<**VAR** nome="minuto" type="INT" > </VAR>
	<**IF**>
	<Bloco1 instr="**GE**" in1="minuto" in2="59" />
	<**THEN**>
	<ATRIBUICAO>
	<Bloco instr="**MOVE**" in1="0" in2="minuto" />
	</ATRIBUICAO>
	<ATRIBUICAO>
	<Bloco2 instr="**ADD**" in1="hora" in2="1" />
	<Bloco instr="**MOVE**" in1="Bloco2.OUT" in2="hora" />
	</ATRIBUICAO>
	</**THEN**>
	<**ELSE**>
	<ATRIBUICAO>
	<Bloco3 instr="**ADD**" in1="minuto" in2="1" />
	<Bloco instr="**MOVE**" in1="Bloco3.OUT" in2="minuto" />
	</ATRIBUICAO>
	</**ELSE**>
	</**IF**>
	</**PROGRAM**>
	</ARQ>

Figura 8-6: Programa CLPScript e XML Correspondente

Em seguida utilizou-se o programa XMLtoLD para converter do XML para um módulo LD Altus. Os módulo 1 e 2 do XMLtoLD geram o arquivo intermediário “relogi.pld”, conforme mostra a Figura 8-7.

```
// PROGRAM Relogio
// IF1
//Bloco1: GE minuto 59
GE M0011 59 A01.0
RNA A01.0 ; BOB A00.0
// THEN1
//Bloco: MOVE 0 minuto
RNA A00.0 ; MOVE 0 M0011
//Bloco2: ADD hora 1
RNA A00.0 ; ADD M0010 1 M0012
//Bloco: MOVE Bloco2.OUT hora
RNA A00.0 ; MOVE M0012 M0010
// ELSE1
//Bloco3: ADD minuto 1
RNF A00.0 ; ADD M0011 1 M0013
//Bloco: MOVE Bloco3.OUT minuto
RNF A00.0 ; MOVE M0013 M0011
// FIM IF1
```

Figura 8-7: Arquivo Intermediário ProLD “relogi.pld”

O programa XMLtoLD gera também o arquivo de descrição do operandos alocados, mostrado na Figura 8-8.

```
M0010 ; hora ; (VAR_INPUT INT)
M0011 ; minuto ; (VAR_INPUT INT)
A00.0 ; IF1 ; ( BOOL)
A01.0 ; B01_OUT ; (Bloco1: GE minuto 59  BOOL)
M0012 ; B02_OUT ; (Bloco2: ADD hora 1  INT)
M0013 ; B03_OUT ; (Bloco3: ADD minuto 1  INT)
```

Figura 8-8: Arquivo Texto “relogi.txt”

Observe-se que dos 6 operandos alocados, dois correspondem às variáveis declaradas (“hora” e “minuto”), dois foram utilizados na avaliação do IF e os últimos dois são as saídas dos blocos ADD gerados a partir do CLPScript original.

O módulo 3 do programa XMLtoLD gera o arquivo final “P-relogi.020”, que pode ser utilizado no MasterTool e nos controladores programáveis Altus. Este arquivo contém o código binário LD correspondente ao “relogi.pld” da Figura 8-7, e é mostrado nas 3 próximas figuras.

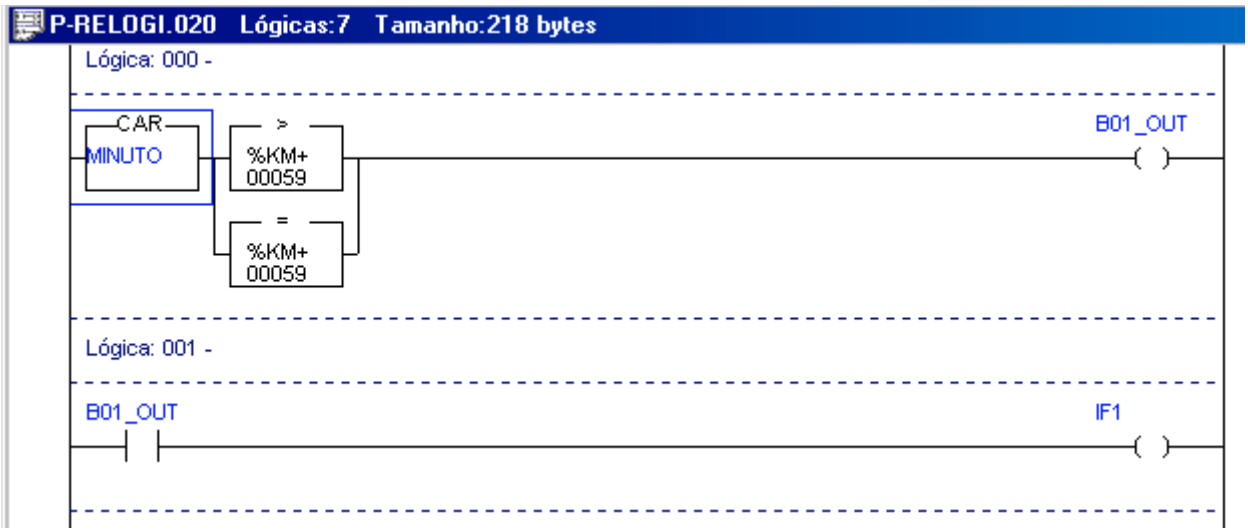


Figura 8-9: Diagrama LD Gerado, Lógicas 000 e 001

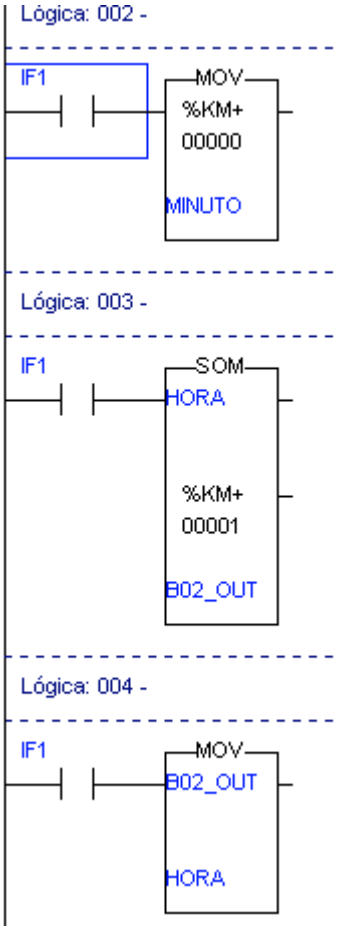


Figura 8-10: Diagrama LD Gerado, Lógicas 002, 003 e 004

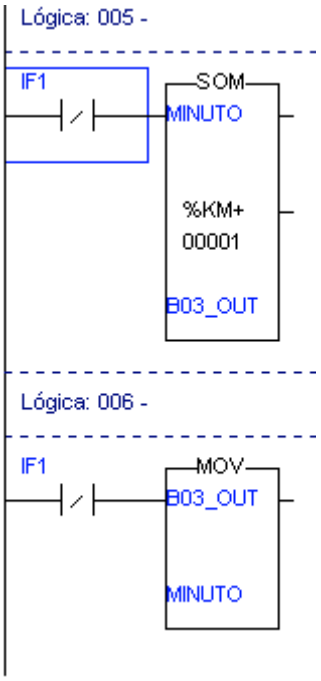


Figura 8-11: Diagrama LD Gerado, Lógicas 005 e 006

A Figura 8-9 mostra o diagrama LD correspondente ao trecho de código de avaliação da expressão do IF, “if minuto >= 59”.

A Figura 8-10 mostra o diagrama LD correspondente ao THEN deste IF, onde são feitas as atribuições “minuto := 0;” e “hora := hora +1;” . Observe-se que “hora + 1” é considerado uma expressão, que é avaliada antes da atribuição. Observe-se também que as três lógicas correspondentes ao **THEN** possuem um contato **normalmente aberto** do operando IF correspondente em série com todos os blocos pertencentes a este THEN.

A Figura 8-11 mostra o diagrama LD correspondente ao **ELSE**, onde são feitas a atribuição “minuto := minuto +1;”. Observe-se que estas lógicas possuem um contato **normalmente fechado** em série com todos os blocos seguintes, representando a condição ELSE do IF.

8.5 Estudo de Caso 2

Este estudo de caso mostra uma função de linearização e um programa que faz a média de 20 medições analógicas escritos em CLPScript. As medições são valores do tipo “REAL” que estão em um “ARRAY” de 20 posições, a linearização é feita chamando-se a função 20 vezes, em um “FOR”, e acumulando-se o resultado para então calcular a

média. A Figura 8-12a mostra este programa e a Figura 8-12b mostra o XML correspondente, gerado automaticamente pelo CLPStoXML.

a) Programa CLPScript	b) Arquitetura XML

function linrza: real
  var_input
  medicao: real;
  end_var
 	
  if medicao > 0
  then
  linrza :=100*
  sqrt(medicao);
  else
  linrza := 0;
  end_if;
end_function
 	
program exemp2
  var
  acum, media: real;
  nivel: array[1..20]
  of real;
  i: int;
  end_var
  acum := 0;
  for i:=1 to 20
  do
  acum := acum +
  linrza(nivel[i]);
  end_for;
  media := acum / 20;
end_program
	<?xml version="1.0"?>
	<ARQ nome="D:\DISS4\CDROM DISSERTACAO\BIN\EX2.XML">
	<**FUNCTION** nome="**linrza**" type="real" >
	<VAR_INPUT nome="medicao" type="REAL" > </VAR_INPUT>
	<IF>
	<Bloco1 instr="GT" in1="medicao" in2="0" />
	<THEN>
	<ATRIBUICAO>
	<Bloco2 instr="CHF" nome="sqrt" qtdIn="1" in1="medicao" type="REAL" />
	<Bloco3 instr="MUL" in1="100" in2="Bloco2.OUT" type="REAL" />
	<Bloco instr="MOVE" in1="Bloco3.OUT" in2="linrza" />
	</ATRIBUICAO>
	</THEN>
	<ELSE>
	<ATRIBUICAO>
	<Bloco instr="MOVE" in1="0" in2="linrza" />
	</ATRIBUICAO>
	</ELSE>
	</IF>
	</FUNCTION>
	<**PROGRAM** nome="**exemp2**" >
	<VAR nome="acum" type="REAL" />
	<VAR nome="media" type="REAL" />
	<VAR nome="nivel" type="'ARRAY[1..20] OF REAL'" />
	<VAR nome="i" type="INT" />
	<ATRIBUICAO><Bloco instr="MOVE" in1="0" in2="acum" /> </ATRIBUICAO>
	<ATRIBUICAO><Bloco instr="MOVE" in1="1" in2="i" /> </ATRIBUICAO>
	<FOR var="i" to="20" >
	<ATRIBUICAO>
	<Bloco4 instr="CHF" nome="linrza" qtdIn="1" in1="nivel[i]" type="REAL" />
	<Bloco5 instr="ADD" in1="acum" in2="Bloco4.OUT" type="REAL" />
	<Bloco instr="MOVE" in1="Bloco5.OUT" in2="acum" />
	</ATRIBUICAO>
	</FOR>
	<ATRIBUICAO>
	<Bloco6 instr="DIV" in1="acum" in2="20" type="REAL" />
	<Bloco instr="MOVE" in1="Bloco6.OUT" in2="media" />
	</ATRIBUICAO>
	</PROGRAM>
	</ARQ>

Figura 8-12: Programa CLPScript e XML correspondente

Observe-se que em um mesmo arquivo XML ficam declaradas a função “linrza” e o programa “exemp2”. A Figura 8-13 mostra os arquivos ProLD correspondentes, gerados pelo programa XMLtoLD.

(a)

```
// FUNCTION linrza: real
// IF1
//Bloco1: GT medicao 0
GT F0011 0 A01.0
RNA A01.0 ; BOB A00.0
// THEN1
//Bloco2: CHF sqrt
RNA A00.0 ; CHF sqrt 123 1 1 (F0011 ) (F0012)
//Bloco3: MUL 100 Bloco2.OUT
RNA A00.0 ; MUL 100 F0012 F0013
//Bloco: MOVE Bloco3.OUT linrza
RNA A00.0 ; MOVE F0013 F0010
// ELSE1
//Bloco: MOVE 0 linrza
RNF A00.0 ; MOVE 0 F0010
// FIM IF1
```

(b)

```
// PROGRAM exemp2
// Variáveis:
//Bloco: MOVE 0 acum
MOVE 0 F0014
//Bloco: MOVE 1 i
MOVE 1 M0011
// FOR i <= 20  (IF1_FOR)
LE M0011 20 A02.0
//Bloco4: CHF linrza
RNA A02.0 ; ADD M0011 100 M0010; CHF linrza 123 1 1 (M0010*F] ) (F0016)
//Bloco5: ADD acum Bloco4.OUT
RNA A02.0 ; ADD F0014 F0016 F0017
//Bloco: MOVE Bloco5.OUT acum
RNA A02.0 ; MOVE F0017 F0014
// FIM IF1_FOR
ADD M0011 KM1 M0011
RNA A02.0 ; SLT –5
//Bloco6: DIV acum 20
DIV F0014 20 F0018 M0
//Bloco: MOVE Bloco6.OUT media
MOVE F0018 F0015
```

Figura 8-13: Arquivos ProLD a) “Linrza.pld” e b) “Exemp2.pld”

O arquivo de descrição do operandos alocados, gerado pelo XMLtoLD, é mostrado na Figura 8-14. As 20 posições do array “nivel” foram alocadas na faixa de operandos F0101 até F0120, inclusive.

```
F0010 ; linrza ; (FUNCTION real)
F0011 ; medicao ; (VAR_INPUT REAL)
A00.0 ; IF1 ; ( BOOL)
A01.0 ; B01_OUT ; (Bloco1: GT medicao 0  BOOL)
F0012 ; B02_OUT ; (Bloco2: CHF sqrt  REAL)
F0013 ; B03_OUT ; (Bloco3: MUL 100 Bloco2.OUT  REAL)
F0014 ; acum ; (VAR REAL)
F0015 ; media ; (VAR REAL)
F0101 ; ; array nivel[1]
F0120 ; ; array nivel[20]
M0010 ; ; índice do array nivel
M0011 ; i ; (VAR INT)
A02.0 ; IF1_FOR ; ( BOOL)
F0016 ; B04_OUT ; (Bloco4: CHF linrza  REAL)
F0017 ; B05_OUT ; (Bloco5: ADD acum Bloco4.OUT  REAL)
F0018 ; B06_OUT ; (Bloco6: DIV acum 20  REAL)
```

Figura 8-14: Arquivo Texto com a Descrição dos Operandos Alocados

Em seguida, o XMLtoLD converteu estes dois arquivos ProLD nos respectivos módulos LD Altus, denominados “P-Linrza.020” e “P-Exemp2.021”. Apenas o LD do 2º módulo é mostrado aqui, nas figuras a seguir. Todo o estudo de caso, entretanto, pode ser encontrado no CDROM que acompanha esta dissertação.

A Figura 8-15 mostra as lógicas criadas para inicializar a variável “acum” com zero e a variável “i” do FOR com o valor 1.

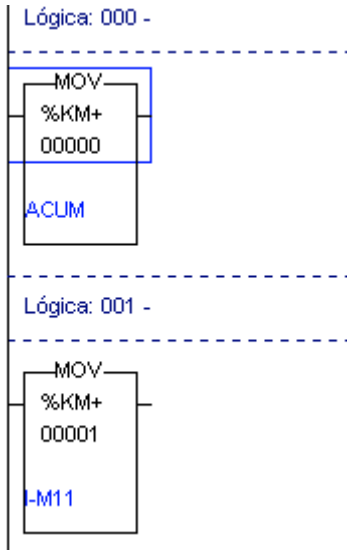


Figura 8-15: Lógicas 000 e 001 do Módulo “P-Exemp2.021”

A Figura 8-16 mostra o LD do teste da condição do FOR que verifica se a variável “i” é menor ou igual a 20, e a primeira instrução dentro do FOR, que faz a chamada da função “linrza” passando a posição “i” do array “nivel” como parâmetro. Para isto, utilizou-se o modo de **acesso indexado** dos CPs Altus, no qual o valor de um operando X qualquer é que determina o operando Y a ser usado na instrução ([ALT03]). No caso, o operando correspondente a variável “i” (M11) é somado com o valor 100 (local alocado para a primeira posição do array “nivel”) e o resultado, colocado em M10, gera o índice de acesso indexado ao array.

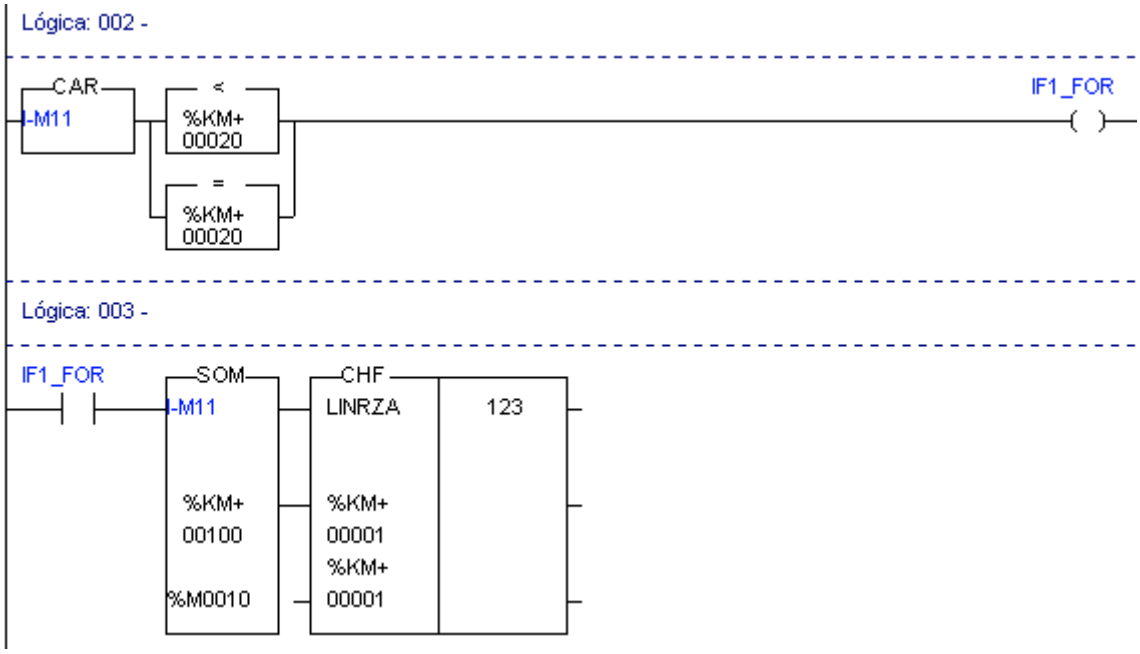


Figura 8-16: Lógicas 002 e 003 do Módulo “P-Exemp2.021”

A Figura 8-17 mostra a tela de configuração dos parâmetros de entrada e saída da instrução CHF, no MasterTool. No caso, está sendo mostrado o operando de acesso indexado “M0010 * F”, que significa que na instrução será utilizado o operando “F” determinado pelo conteúdo do operando “M0010”.

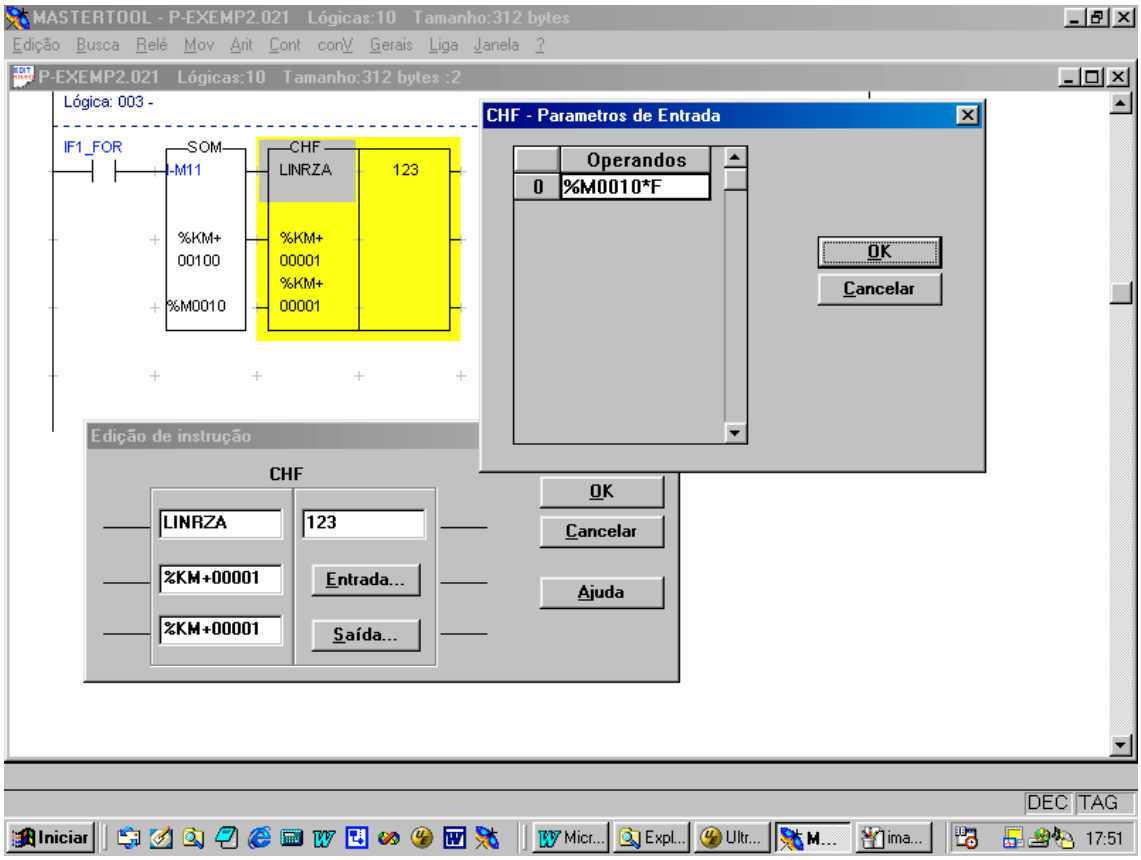


Figura 8-17: Parâmetros de Entrada da Instrução CHF

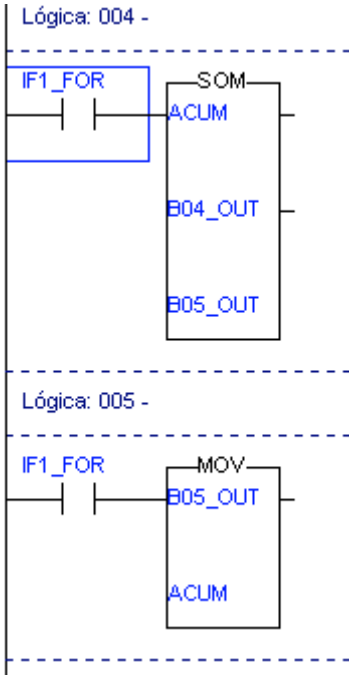


Figura 8-18: Lógicas 004 e 005 do Módulo “P-Exemp2.021”

A Figura 8-18 mostra as lógicas que acumulam, dentro do FOR, o valor de retorno da função “linrza” na variável “acum”, alocada no operando F0014.

A Figura 8-19 mostra o incremento da variável de “i” de controle do FOR e o salto negativo de 5 lógicas, recaindo no início do FOR, feito através da instrução bobina de salto “—(S)—” dos CPs Altus.

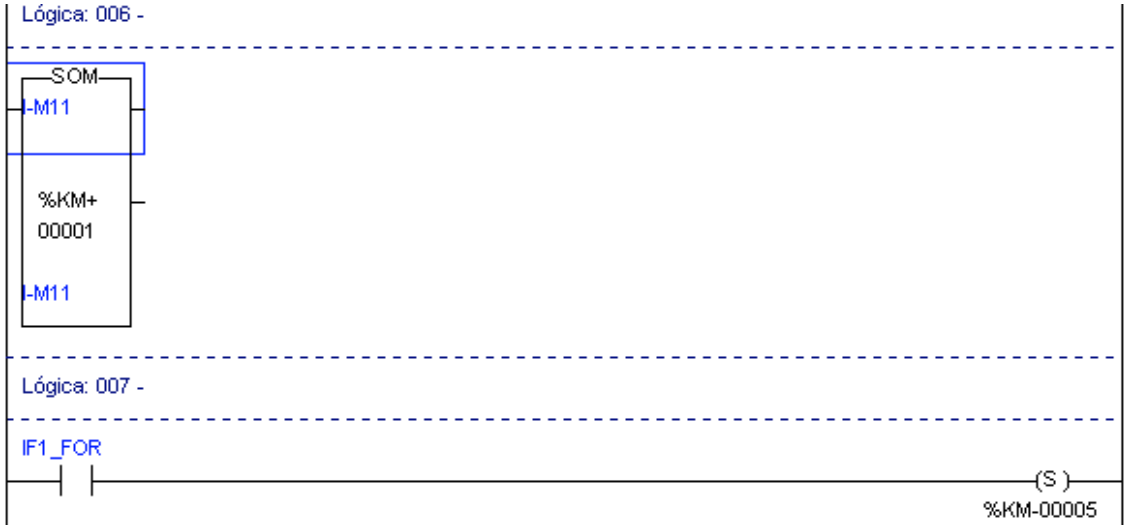


Figura 8-19: Lógicas 006 e 007 do Módulo “P-Exemp2.021”

A Figura 8-20 mostra as lógicas fazem a divisão do total acumulado para obter o valor médio desejado, na variável “media”.

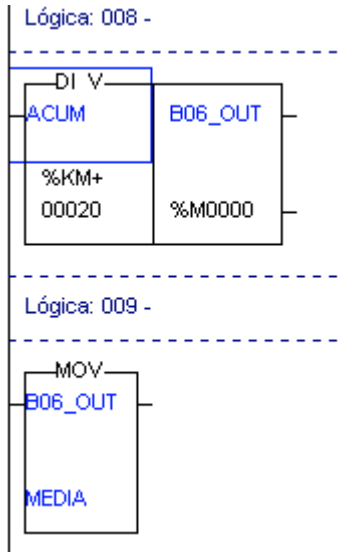


Figura 8-20: Lógicas 008 e 009 do Módulo “P-Exemp2.021”

8.6 Conclusão

Este capítulo apresentou um estudo de caso onde programas CLPScript armazenados na arquitetura XML foram convertidos para a linguagem LD dos controladores programáveis da Altus S.A.

Para realizar esta conversão foi desenvolvido o programa XMLtoLD, em Visual C++ 6.0. Este programa lê um arquivo XML utilizando um *parser* XML bem conhecido e de domínio público, o *parser* expat. O programa identifica as *tags* e atributos da arquitetura XML, faz a conversão para diagrama LD e grava um novo arquivo no formato Altus.

O XMLtoLD também se mostrou bastante complexo de ser implementado pois demandou o conhecimento do *assembly* do microcontrolador 80C251 e do formato interno das instruções LD dos controladores programáveis Altus.

Após o LD ser gerado pelo XMLtoLD, ele pode ser visualizado pelo *software* programador MasterTool da Altus. Pode inclusive ser editado ou até mesmo executado em um controlador programável. Após ter sido gerado, não há como distinguir se foi criado pelo MasterTool ou pelo XMLtoLD.

O XMLtoLD foi desenvolvido especificamente para o Altus, para ser usado como prova de conceito. A partir do XML, cada fabricante é responsável por gerar seu próprio XMLtoLD.

9 Conclusões e Trabalhos Futuros

9.1 Conclusões

A norma IEC 1131-3 definiu e padronizou 5 linguagens de programação para desenvolvimento de programas aplicativo de controladores programáveis: LD, FBD, ST, IL e SFC. Segundo a norma, os programas podem ser escritos na linguagem que o projetista achar mais adequada, e qualquer programa escrito em qualquer das 4 linguagens (LD, FBD, ST ou IL) pode ser convertido em um programa equivalente nas outras linguagens.

Esta dissertação abordou o problema da interoperabilidade entre as linguagens da norma e entre diferentes fabricantes, uma vez que a norma diz que a conversão entre as linguagens é possível mas não diz nada sobre como fazer isto. Como forma de amenizar este problema, este trabalho sugere uma arquitetura que permite a interoperabilidade baseada em XML.

Foi apresentado uma revisão bibliográfica sobre linguagens específicas de domínio, uma introdução aos principais conceitos do XML e uma visão geral da norma IEC 1131-3. Os conceitos de linguagens específicas de domínio foram aproveitados na definição das linguagens CLPScript e ProLD, criadas neste trabalho. O XML aparece no centro da arquitetura proposta no Capítulo 6 e a norma IEC 1131-3 é um dos focos principais deste trabalho.

Foram apresentados alguns trabalhos relacionados com controladores programáveis, linguagem XML e linguagens de programação da norma IEC 1131-3, tais como o a linguagem gráfica SIPN que descreve o comportamento seqüencial de um processo gerando código IL, o comitê técnico TC6 da associação mundial PLCopen que trabalha na definição de *schemas* XML para todas as linguagens da norma e a linguagem VHDL, dentre outros. Todos estes trabalhos foram pesquisados por terem alguma semelhança com o desenvolvido aqui.

As contribuições deste trabalho iniciam-se com a comparação entre as linguagens ST, LD e FBD e com a proposta de representação textual para armazenar um diagrama gráfico LD ou FBD. Destes estudos concluiu-se que, em determinadas condições, a representação textual de um diagrama gráfico LD pode se tornar um caso particular do diagrama FBD. Este detalhe pode ser muito importante no projeto de um *software* editor gráfico, pois, se for levado em consideração, consegue-se que o mesmo editor FBD possa ser utilizado para apresentar diagramas LD.

Para a representação textual de um diagrama gráfico de blocos conectados percebeu-se que não é necessário nenhuma estrutura de dados mais elaborada tipo grafos ou lista encadeada, desde que assuma-se que cada saída de bloco possui memória alocada para armazenar seu valor e que a informação de conexões fique sempre armazenada na entrada dos blocos (e não nas saídas). Esta representação textual acaba sendo usada na arquitetura XML.

A linguagem CLPScript é definida a partir da linguagem ST da norma IEC 1131-3. Procurou-se manter um mínimo de comandos imprescindíveis em uma linguagem de alto nível e retirou-se opções mais complexas e não tão relevantes. Para o contexto da automação industrial, entendeu-se que este conjunto de comandos selecionados já é de grande utilidade. A arquitetura XML foi definida a partir da CLPScript.

A linguagem XML é uma linguagem de marcação amplamente difundida na Web que permite garantir a interoperabilidade entre dados e aplicações. Foi particularmente útil na definição da arquitetura XML porque, por ser extensível, permitiu que as *tags* e atributos fossem criados com os nomes que entendeu-se mais apropriados.

A arquitetura XML define as *tags* e atributos utilizadas para a representação de diagramas LD e FBD e programas CLPScript. A representação de LD e FBD foi baseada na representação textual descrita no Capítulo 4, enquanto que na representação de CLPScript e dos aspectos comuns a todas as linguagens da norma IEC 1131-3 procurou-se , sempre que possível, manter no XML a mesma nomenclatura original da IEC.

Foi desenvolvido o programa CLPStoXML, um compilador lê um programa CLPScript, conforme definido no Capítulo 5, e gera a arquitetura XML correspondente, definida no Capítulo 6. Foi escrito em Visual C++ 6.0 da Microsoft, baseado no *software* ILA –

Interpretador de Linguagem Algoritmica. O programa CLPStoXML mostrou-se bastante complexo de ser implementado, porque demandou muito conhecimento da área de compiladores.

Com o objetivo de certificar e validar a arquitetura XML, foi realizado um estudo de caso onde se converteu alguns programas fonte escritos em CLPScript para XML e deste XML para a linguagem LD de uma empresa fabricante de controladores programáveis, a Altus S. A.

Para realizar esta conversão foi desenvolvido o programa XMLtoLD, em Visual C++ 6.0. Este programa lê um arquivo XML utilizando o *parser* expat, identifica as *tags* e atributos da arquitetura XML, faz a conversão para diagrama LD e grava um novo arquivo no formato Altus.

Durante os trabalhos iniciou-se o desenvolvimento de um software visualizador para as linguagens gráficas LD e FBD. Passadas cerca de três semanas percebeu-se que esta alternativa seria por demais complexa, e optou-se então por utilizar um visualizador que já estivesse pronto, no caso o da Altus, em vez de construir um do zero. Desta opção surgiu o programa XMLtoLD.

O XMLtoLD também se mostrou bastante complexo de ser implementado, pois demandou o conhecimento do *assembly* do microcontrolador 80C251 e do formato interno das instruções LD dos controladores programáveis Altus.

Após o LD ser gerado pelo XMLtoLD, ele pode ser visualizado pelo *software* programador MasterTool da Altus. Pode inclusive ser editado ou até mesmo executado em um controlador programável. Após ter sido gerado, não há como distinguir se foi criado pelo MasterTool ou pelo XMLtoLD.

Salienta-se que o XMLtoLD foi desenvolvido especificamente para o Altus, para ser usado como prova de conceito. A partir do XML, cada fabricante é responsável por gerar seu próprio XMLtoLD.

A escolha da Altus S. A. se deu por esta ser uma empresa bem conhecida no mercado de automação industrial e pelo fato do autor desta dissertação atuar na área de automação industrial e ser colaborador da Altus a mais de 5 anos, tendo portanto um conhecimento prévio dos produtos da empresa e acesso à documentações internas necessárias a este

tipo de implementação. Salienta-se que parte desta documentação é informação estratégica da empresa, tecnologia própria, patenteada e tratada com sigilo.

O tema desta dissertação foi escolhido porque a adequação a norma IEC 1131-3 é uma área estratégica para todas as empresas que fabricam software para controladores programáveis. O assunto selecionado acabou se mostrando uma boa escolha pois conciliou perfeitamente o interesse do aluno, da universidade e da empresa envolvida.

Entende-se que os pré-requisitos para o desenvolvimento de um trabalho deste porte são:

- Conhecimento da área de automação industrial e experiência em desenvolvimento de programas aplicativos para controladores programáveis;
- Conhecimento da norma IEC 1131-3;
- Conhecimento da linguagem XML, dos parsers e das ferramentas de validação disponíveis;
- Conhecimento das técnicas utilizadas na implementação de compiladores;
- Conhecimento da linguagem assembly utilizada no controlador programável e do formato interno de armazenamento da linguagem LD deste controlador;
- Experiência em desenvolvimento de software em alguma linguagem de programação orientada a objetos, tal como C++ .

Para se chegar nos resultados descritos neste trabalho foi realizado um grande esforço de programação devido ao tamanho e a complexidade dos softwares desenvolvidos.

As maiores dificuldades deste trabalho foram na definição e na implementação dos programas CLPStoXML e XMLtoLD. Em CLPStoXML a geração do XML foi relativamente simples, mas o compilador ST foi bastante complexo de ser implementado, e não o teria feito sem a utilização dos fontes do ILA ([ILA91]) e de bons livros de compiladores ([AHO86], [MUC97]). Quanto ao XMLtoLD, a implementação só foi possível porque teve-se acesso a documentações internas do formato das instruções LD e também aos fontes do programador MasterTool.

Como herança deste trabalho, pode-se dizer que a arquitetura XML desenvolvida aqui poderá ser aproveitada no desenvolvimento de softwares programadores de controladores programáveis que pretendem ter suporte para mais de uma linguagem de programação, conforme a norma. A Altus pretende lançar uma nova versão de seu

programador onde o XML será utilizado como formato intermediário de armazenamento das linguagens.

Todos os programas utilizados e gerados no estudo de caso, o código fonte dos programas CLPStoXML e XMLtoLD, e arquivos pdf com manuais e características técnicas de produtos utilizados podem ser encontrados no CDROM que acompanha esta dissertação.

9.2 Contribuições

Este trabalho propiciou as seguintes contribuições:

- o projeto da arquitetura XML para permitir a interoperabilidade para as linguagens da norma IEC 1131-3, visto que a norma determina que um programa escrito em qualquer das 4 linguagens LD, FBD, ST ou IL pode ser convertido em um programa equivalente nas outras linguagens, mas não diz nada sobre como fazer isto;
- a linguagem CLPScript, que gera e formaliza esta arquitetura XML junto com seu compilador CLPStoXML;
- o estudo comparativo das linguagens LD, FBD, ST da norma IEC 1131-3 mostrando suas similaridades estruturais e o estudo da representação textual de diagramas gráficos, que permitiram pensar na arquitetura XML;
- o estudo comparativo da linguagem IL com a arquitetura XML, mostrando que o XML é mais adequado que a IL para ser a linguagem intermediária na qual todas as demais podem ser convertidas, porque com o XML é possível que se restaure a linguagem original, enquanto que com a IL não é possível.

9.3 Trabalhos Futuros

Este trabalho possibilita os seguintes trabalhos futuros:

1) a arquitetura XML para representação dos programas aplicativos pode ser convertida no formado binário nativo de determinado processador para que o programa armazenado possa ser executado no hardware correspondente. Esta arquitetura XML poderia ser expandida com detalhes específicos do processador e hardware necessários para esta conversão;

2) a arquitetura XML para representação dos programas aplicativos pode ser convertida em um formado binário que poderia ser diretamente utilizado pelos CPs para executar o programa aplicativo através de um interpretador executando no próprio CP. Esta é uma idéia equivalente a uma máquina virtual Java executando o *byte-code* dos programas compilados.

Neste segundo item, o código objeto gerado pode ser executado por um interpretador específico de cada CP, de forma análoga ao que acontece com um programa Java. Em Java, o programa fonte é compilado para um código intermediário chamado de *byte code (*[MOR00]). Este código intermediário é executado por um interpretador Java, chamado de máquina virtual Java. De forma análoga, é preciso uma máquina virtual no CP para executar programas compilados neste código objeto.

As vantagens de se utilizar uma máquina virtual em vez de executar o código *assembly* nativo são as mesmas da linguagem Java comparadas com linguagens compiladas tradicionais tais como C e Pascal. Dentre outras, pode-se citar (1) simulador e interpretador mais simples, (2) programa independente de plataforma e (3) complexidades do *assembly* específico só aparecem no interpretador, e não no editor gráfico/compilador.

Bibliografia

[AHO86] Aho, Alfred V.; Sethi, Rari; Ullman, Jeffrey D. *Compiler: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[ALL94] Allen-Bradley. *Controlador Programável CLP – 5™ Manual de Montagem & Instalação*. Milwaukee, USA: [s.n.], 1994. (ABT-1785-6.1.1)

[ALT97] ALTERA. MAX+PLUS II Getting Started Manual. USA, 1997. (PN : 25-04803-03)

[ALT98a] ALTUS S. A . *Manual de Características Técnicas*. [S.l.: s.n.], 1998. CD-ROM.

[ALT98b] ALTUS S. A *Descrição de software: Código Executável para Diagrama de Relés*. Porto Alegre, Brasil: [s.n.], 1998. (DS6003.028.7 ver D)

[ALT02] ALTUS S. A *Manual de Utilização MasterTool*. Porto Alegre, Brasil: [s.n.], 2002. (MP-6299-101.7 ver C)

[ALT03] ALTUS S. A *Manual de Programação MasterTool*. Porto Alegre, Brasil: [s.n.], 2003. (MP-6399-100.4 ver D)

[ARC02] Arciniegas, Fabio. *C++ XML*. Makron Books, São Paulo, 2002.

[Bom97] Bonfatti, Monari e Sampieri. *IEC 1131-3 Programming Methodology*. CJ international, 1997.

[CAR01] Carro, Luigi. Projeto e Prototipação de Sistemas Digitais. Ed. Universidade/UFRGS, 2001.

[CRE00] Crespo, Sérgio C. S. P.: *Composição em WebFrameworks.* Tese de Doutorado. Pontifícia Universidade Católica do Rio de Janeiro. Agosto, 2000.

[DAT00] Dattatri, Kayshav. *C++ Effective Object-Oriented Software Construction, Second Edition*. Prentice-Hall, New Jersey, 2000.

[DEI02] Deitel, Paul J.; Deitel, Harvey M. *C++ How To Program*. Editora Prentice Hall, 2002..

[DEU00] Deursen, Arie van; Klint, Paul; Visser, Hoost. *Domain-specific languages*. CWI, Amsterdam, 2000.

[DIA01] DiaGen. *The Diagram Editor Generator*. Disponível em http://www2.informatik.uni-erlangen.de/DiaGen/. Último acesso – Setembro – 2003.

[EXP98] Expat. *Expat - XML Parser Toolkit*. Disponível em http://www.jclark.com/xml/expat.html. Último acesso, Janeiro 2004.

[FES00] Festo Automação LTDA. *Introdução a Controladores Lógicos Programáveis*. Festo Didatic – Brasil, 2000.

[FIT81] Fitzgerald, Arthur Eugene; Higginbotham, David E.; Grabel, Arvin. *Engenharia Elétrica*. McGraw-Hill, 1981.

[FRE01] Frey, Georg; Minas, Mark: *Internet-based Development of Logic Controllers Using Signal Interpreted Petri Nets and IEC 61131*. Universität Kaiserlautern, Alemanha, 2001.

[FRO97] Fromherz, Markus; Gupta, Vineet; Saraswat, Vijay. *cc – A Generic Framework for Domain Specific Languages*. POPL'97, Paris, 1997.

[GEO00] Georgini, Marcelo. *Automação Aplicada*. Érica, São Paulo, 2000.

[GRU01] Grune, Dick; Bal, Henri E.; Jacobs, Ceriel J. H.; Langendoen, Koen G. *Projeto Moderno de Compiladores*. Editora Campus, Rio de Janeiro, 2001.

[HAN03] Hansen, Roseli P. *GlueScript: Uma linguagem específica de Domínio para composição de Web Services*. Dissertação de Mestrado. Universidade do Vale do Rio dos Sinos. Fevereiro, 2003.

[HAR99] Harold, Elliotte Rusty. *XML Bible*. ADG Books Worldwide Inc., Foster City, 1999.

[HAY73] Hayt, William H.; Kemmerly, Jack E. *Análise de Circuitos em Engenharia*. McGraw-Hill, 1973.

[HOL01] Holzner, Steven. *C++ Black Book*. Makron Books Ltda, 2001.

[IEC93] International Electro-Technical Commission. *IEC International Standard 1131-3 Programmable Controllers, Part 3: Programming Languages*. 1993.

[ILA91] PINTO, S. C. C. S.; SILVA, J. L. T.; COUTINHO, H.. *Construção de um interpretador de linguagem algorítmica*. In: III Salão de iniciação científica da UFRGS, 1991, Porto Alegre. Proceedings of III Salão de Iniciação Científica da UFRGS, 1991. v. 1. p. 1-2 .

[INT96] Intel®, *8XC251SA, 8XC251SB, 8XC251SP, 8XC251SQ Embedded Microcontroller User's Manual*. Intel, 1996.

[KAI99] Kain, Eugène. *The MFC Answer Book*. Addison-Wesley, Indianapolis, 1999

[KRU97] Kruglinski, David. *J. Inside Visual C++. 4th.ed*. Redmond, Washington: Microsoft Press, 1997.

[LEI99] Leijen, Daan; Meijer, Erik. *Domain Specific Embedded Compilers*. USENIX, 1999.

[LEV92] Levine, John R.; Mason, Tony; Brown, Doug. lex & yacc. O'Reilly, 1992

[LEW95] Lewis, R. W. *Programming Industrial Control Systems Using IEC 1131-3*. Institution of Electrical Engineers, London, 1995.

[MOR00] Morgan, Michael. *Java 2 para Programadores Profissionais*. Editora Ciência Moderna Ltda., 2000.

[MOR01] Moraes, C. Cícero; Castrucci, L. Plínio: *Engenharia de Automação Industrial*. LTC – Livros Técnicos e Científicos Editora, Rio de Janeiro, 2001.

[MUC97] Muchnick, Steven S.. *Advanced Compiler Design and Implementation*. Academic Press, 1997.

[NAT00] Natale, Ferdinando. *Automação Industrial*. Érica, São Paulo, 2000.

[OUS98] Ousterhout, John K. *Scripting: Higher-Level Programming for the 21st Century*. Sun Microsystems Laboratories, 1998.

[PEP01] P. Pepper; M. Cebulla; K. Didrich; W. Grieskamp. *From program languages to software languages*. Berlin, Germany, 2001.

[PLC92] PLCOpen. *Standardization in Industrial Control Programming*. Disponível em http://www.plcopen.org . Último acesso – Setembro – 2003.

[PLC03] PLCopen TC6 . T*echnical Committee TC6: XML*. Disponível em http://www.plcopen.org/plcopen/TC6_XMl.htm. Último acesso – Setembro – 2003.

[PRI00] Price, Ana Maria de Alencar; Toscani, Simão Sirineo. Implementação de Linguagens de Programação: Compiladores. Editora SagraLuzzatto, Porto Alegre, 2000.

[ROC99] Rockwell Automation. *Software RSLogix 5™ Versão 3.2 Guia de Procedimentos*. Milwaukee, USA: [s.n.], 1999.

[ROC00] Rockwell Automatio. *RSLogix 5000™ Configuration and Programming for the Logix5000 family of controllers*. USA: [s.n.], 2000.

[SCO98] Scowen, R. S. *Extended BNF – A generic base Standard*. 1998.

[SHA02] Shalloway, Alan; Trott, R. James; *Design Patterns Explained*. Addison-Wesley, 2002.

[SIE99a] Siemens. *Programming with Step 7 V5.0* .[s.n.], 1999.

[SIE99b] Siemens. *Working with Step 7 V5.0* .[s.n.], 1999.

[SIL01] Silva, Osmar J.. *XML: Aplicações Práticas*. Érica, São Paulo, 2001.

[SIP01] SIPN-Editor. *Editor for Hierarchical Signal Interpreted Petri Nets*. http://www2.informatik.uni-erlangen.de/DiaGen/SIPN/. Último acesso – Setembro – 2003.

[SPI00] Spinellis, Diomidis. *Notable design patterns for domain-specific languages*.

University of the Aegean, 2000.

[STR00] Stroustrup, Bjarne. *The C++ Programming Language Special Edition*. AT&T, 2000.

[TIT02] Tittel, Ed. *XML*. The McGraw Hill Companies, Inc., 2002.

[TOU97] Tourlas, Konstantinos. *An Assessment of the IEC 1131-3 Standard on Languages for Programmable Controllers*. SafeComp' 97, 1997.

[TOU00] Tourlas, Konstantinos. *Diagrammatic Representations in Domain-Specific Languages*. University of Edinburgh, 2000.

[WEB02] Weber, Michael; Kindler, Ekkard: *The Petri Net Markup Language*. Technische Universität München, 2002.

[WYK00] Wyk, Eric Van. *Domain Specific Meta Languages*. ACM, Oxford University Computing Laboratory, 2000.