Universidade do Porto

Faculdade de Engenharia

FEUP

Desenvolvimento de Plataforma para Apoio de Aplicações de Controlo Industrial

Contribuição na Avaliação das Linguagens de Programação Normalizadas para Utilização no Domínio

Mário Jorge Rodrigues de Sousa

Mário Jorge Rodrigues de Sousa

Dissertação submetida para a obtenção do grau de
Doutor em Engenharia Electrotécnica e de Computadores
pela Faculdade de Engenharia da Universidade do Porto

Trabalho realizado sob a orientação do Professor Doutor

Adriano da Silva Carvalho

Universidade do Porto
Faculdade de Engenharia
Departamento de Engenharia Electrotécnica e de Computadores
2004

Apoios:

O autor foi,
durante a realização desta dissertação,
bolseiro do PRAXIS XXI, BD/13726/97

Durante toda a realização deste trabalho recebi provas de amizade e solidariedade por parte de algumas pessoas. Assim desejo expressar os meus sinceros agradecimentos:

- ao Professor Doutor Adriano de da Silva Carvalho, na qualidade de orientador desta tese, pelo total apoio e constantes disponibilidade sempre manifestados durante o desenvolvimento daquela;
- aos Professores Pedro Souto e Francisco Vasques pelas suas sugestões;
- aos amigos Jiri Baum, Andrey Romanenko e Juan Carlos Orozco pelas discussões mantidas através de email;
- à Faculdade de Engenharia da Universidade do Porto e ao Instituto de Sistemas e Robótica pela cedência de equipamentos e instalações imprescindíveis à realização deste trabalho;

Mais do que todos, à minha família, muito especialmente à minha filha que sofreu a ausencia do pai durante muios fins-de-semana os quais nunca mais poderão ser recuperados.

Dedico este trabalho ao meu Pai, pelo sua paciência e dedicação.

Resumo

A crescente automatização dos processos de fabrico resultou na proliferação do recurso a PLCs (Programmable Logic Controllers) para o seu controlo. Com o intuito de aproximar os modos de funcionamento, utilização e programação destes, foram criados diversos standards que têm tido alguma aceitação por parte dos fornecedores de PLCs.

Tirando partido destes standards, e com o intuito de libertar os utilizadores da subjugação dos fornecedores de PLCs, foi criado um projecto que visa desenvolver um PLC em software que possa ser executado sobre qualquer sistema operativo POSIX. Nasceu assim o MatPLC.

O presente trabalho descreve a arquitectura modular adoptada para o MatPLC, e o modo de interacção entre os referidos módulos. Foi adoptada a filosofia do exo-núcleo para o controlo de acesso aos recursos comuns, incluindo zonas de memória partilhada e mecanismos de sincronização. São ainda apresentados alguns testes que permitem avaliar o nível de execução do código resultante.

A par do projecto principal, foi ainda desenvolvido um compilador para as linguagens IL e ST definidas no standard IEC 61131-3. No decorrer deste trabalho foi então efectuada uma análise ao referido standard, tendo sido encontradas diversas falhas ou incongruências no mesmo. São apresentadas sugestões de como as questões identificadas poderão ser ultrapassadas. É ainda apresentada uma análise da adequabilidade das referidas linguagens na programação de aplicações que se pretende sejam de elevada integridade.

Abstract

The growing automation of industrial production processes has resulted in the increased use and dependence on PLCs (Programmable Logic Controllers) to implement the control logic. With the intention of creating a common experience when it comes to configuring, programming and using these devices, several standards have been written and approved, having been well accepted and adopted by the PLC manufacturers.

Taking advantage of these standards, the MatPLC project was created with the intention of freeing the end users form the control of the PLC manufacturers. This project intends to develop a PLC in software capable of executing over any POSIX compliant operating system.

The present work describes the modular architecture adopted for the MatPLC, as well as the mechanisms used for the synchronization and sharing of information between the modules. Additionally, the results of a few tests are presented which allow an evaluation of the resulting code base.

Along with the main project, a compiler was also developed for the IL and ST languages defined in the IEC 61131-3 standard. During the course of this work, several issues were found relating to the definition of these languages. Suggestions on how to overcome these inconsistencies are presented, and an overall evaluation of the languages is made regarding their applicability for use in the coding of high integrity applications.

Resumé

La croissante automatisation des processus de fabrication a résulté dans la prolifération et la dépendance aux PLCs (Programmable Logic Controllers), utilisez por implementer la logique de commande. Avec l'intention de créer une expérience commune en relation la configuracion, la programmacion et la utilizacion de ces dispositifs, plusieurs normes ont été écrites et approuvées. Celles-ci ont eté bien acceptées et adoptées par les fabricants de PLC.

Tirant profit de ces normes, le projet MatPLC a été créé avec l'intention de libérer l'utilisateurs de la commande des fabricants de PLC. Ce projet vise à développer un PLC en logiciel qui puisse être exécuté sur quelconque système operatife POSIX.

Le travail actuel décrit l'architecture modulaire adoptée pour le MatPLC, et les mécanismes utilisés pour la synchronisation et le partage d'information entre les modules. La philosophie de l'exo-noyau a été adoptée pour le contrôle d'accès aux ressources communes. Encore sont présentés quelques essais qui permettent d'évaluer le niveau d'exécution du code résultant.

Avec le projet principal, un compilateur a été développé pour les langues IL et ST définies dans la norme du CEI 61131-3. Pendant ce travail, plusieurs imperfections ont été trouvées concernant le defitiion de ces langues. Sugestions sur la façon dont surmonter ces contradictions sont présentés, et une évaluation globale des langues est faite concernant leur applicabilité pour leur usage pour coder des applications d'intégrité élevées.

Índice

Resumo.....ix
Abstract.....xi
Resumé.....xiii

Índice.....xv
Índice de Figuras.....xix

Capítulo 1
Introdução.....1
1.1 Motivação e Contexto.....1
1.2 Contribuições.....3
1.3 Estrutura.....4

Capítulo 2
Tecnologias de Controlo Programável.....5
2.1 História dos PLCs.....5
2.2 COMEDI.....8
2.3 Controlador EMC.....9
2.4 OROCOS.....11

Capítulo 3
O MatPLC.....17
3.1 Arquitectura Geral.....18
3.2 Micro-Núcleo vs Núcleo Monolítico vs Exo-Núcleo.....20
3.3 O núcleo do MatPLC.....25

3.3.1 O CMM....26
3.3.2 O GMM....28
3.3.3 A Secção SYNCH....35
3.3.4 A Secção STATE....41
3.3.5 A Secção PERIOD....45
3.3.6 A Secção RT....54
3.3.7 A Secção LOG....60
3.3.8 A secção CONF....61
3.4 Exemplos de Módulos....62
3.4.1 O DSP....62
3.4.2 O ModBus....66
3.4.3 A Biblioteca de Funções para E/S....68

Capítulo 4
Compilador IEC 61131-3 71
4.1 O Standard IEC 61131....72
4.1.1 O Standard IEC 61131-3....73
4.1.2 Blocos de Programação....75
4.1.3 As Linguagens de Programação....78
4.1.4 Tipos de Dados....84
4.1.5 Localização de Variáveis....86
4.1.6 As Configurações....87
4.1.7 A Linguagem ST....89
4.1.8 A linguagem IL....91
4.2 O Compilador de IL e ST....94
4.2.1 Gerador da Arvore de Sintaxe Abstracta....98
4.2.2 O Analisador Semântico....104
4.2.3 Gerador de Código....105
4.2.4 Mapeamento dos Blocos para C++....108
Conversão de Variáveis Localizadas....114
Conversão dos Tipos de Dados....116
Conversão de Código ST....118
A Conversão da Linguagem IL....120
4.3 Questões Relacionadas com o IEC 61131-3....125
4.3.1 Nomes de Variáveis....125
4.3.2 Operadores de Expressões....131
4.3.3 Operadores IL....132
4.3.4 Sequências de Funções Bloco....133
4.3.5 Identificadores em Configurações....135
4.3.6 Inicialização de Tarefas....136

4.3.7 Persistência de Variáveis em Funções Bloco....139
4.3.8 Declaração de Variáveis em Configurações....139
4.4 A Segurança das Linguagens IL e ST....140
4.4.1 Avaliação das Linguagens IL e ST....141
4.4.2 Resumo da Classificação....154

Capítulo 5
Aplicações Distribuídas....157
5.1 O IEC 61499....157
5.1.1 Porquê um Novo Standard?....157
5.1.2 O Modelo do IEC 61499....158
5.1.3 Funções Bloco Básicas....161
5.1.4 Funções Bloco Compostas....165
5.1.5 Aplicações e Sub-aplicações....166
5.1.6 Funções Bloco de Serviços de Interface....166
Serviços de Comunicação....167
Serviços de Gestão e Configuração....168
5.1.7 Interface de Adaptação....168
5.1.8 Relação com o IEC 61131-3....169
5.2 Mapeamento na Arquitectura do MatPLC....170
5.3 Protocolos de Rede....172
5.3.1 IDA – Interface para Automação Distribuída....173

Capítulo 6
Conclusões....175
6.1 O MatPLC....175
6.2 O Compilador IEC 61131-3....177
6.3 Desenvolvimentos Futuros....178

ANEXO A....clxxxi
ANEXO B....clxxxix
ANEXO C....cxci

Referências Bibliográficas....cxciii

Índice de Figuras

Figura 2.1 - Estrutura interna do EMC....11
Figura 2.2 - Primitivas estruturais da arquitectura: componentes, portos, conectores....12
Figura 2.3 - A arquitectura do sistema de controlo....13
Figura 2.4 - Duas malhas de controlo em cascata....14
Figura 3.1 - Arquitectura geral do MatPLC....18
Figura 3.2 - Um sistema operativo com núcleo monolítico....22
Figura 3.3 - Um sistema operativo com micro-núcleo....22
Figura 3.4 - Um sistema operativo com um exo-núcleo....22
Figura 3.5 - O MatPLC com arquitectura monolítica....23
Figura 3.6 - O MatPLC com um micro-núcleo....23
Figura 3.7 - O MatPLC com um exo-núcleo....23
Figura 3.8 - A arquitectura do MatPLC....26
Figura 3.9 - Tempos necessários para efectuar a sincronização dos mapas globais local e partilhado, para valores crescentes de memória....35
Figura 3.10 - Tempos necessários para efectuar a sincronização dos mapas globais local e partilhado, para valores crescentes de memória....35
Figura 3.11 - Estados internos do MatPLC e do Módulo são implementados com recurso à Rede de Petri da secção de sincronização....43
Figura 3.12 - Teste 1 - Período 50 ms, máquina caravela com ambiente gráfico em funcionamento....49
Figura 3.13 - Teste 1 - Período 50 ms, máquina caravela com ambiente gráfico em funcionamento....49
Figura 3.14 - Teste 2 - Período 50 ms (10 módulos), máquina caravela com ambiente gráfico em funcionamento....50
Figura 3.15 - Teste 2 - Período 50 ms (10 módulos), máquina caravela com ambiente

gráfico em funcionamento....50
Figura 3.16 - Teste 3 - Período 50 ms, máquina caravela com actividade intensa de rede e disco....51
Figura 3.17 - Teste 3 - Período 50 ms, máquina caravela com actividade intensa de rede e disco....51
Figura 3.18 - Teste 4 - Período 50 ms, máquina caravela sem outras actividades (modo utilizador único)....52
Figura 3.19 - Teste 4 - Período 50 ms, máquina caravela sem outras actividades (modo utilizador único)....52
Figura 3.20 - Teste 5 - Período 50 ms, máquina macau com ambiente gráfico em funcionamento (sem prioridades Tempo-Real)....53
Figura 3.21 - Teste 5 - Período 50 ms, máquina macau com ambiente gráfico em funcionamento (sem prioridades Tempo-Real)....53
Figura 3.22 - Resumo dos resultados dos testes 1-5....54
Figura 3.23 - Máquina macau com ambiente gráfico em funcionamento (não RT)....58
Figura 3.24 - Máquina macau com ambiente gráfico em funcionamento (não RT)....58
Figura 3.25 - Máquina macau com ambiente gráfico em funcionamento (RT)....59
Figura 3.26 - Máquina macau com ambiente gráfico em funcionamento (RT)....59
Figura 3.27 - Construção interna de um bloco PID....64
Figura 3.28 - Composição interna do filtro digital....65
Figura 4.1 - Invocação de funções e funções-bloco a partir de funções, funções-bloco, e programas....75
Figura 4.2 - Exemplos da declaração de função, função-bloco, e de programa....78
Figura 4.3 - Exemplo de um programa em LD....79
Figura 4.4 - Exemplo de um programa em FBD....81
Figura 4.5 - Exemplo de um programa em IL....83
Figura 4.6 - Exemplo de um programa em ST....84
Figura 4.7 - Tipos de dados elementares....84
Figura 4.8 - Os componentes de uma configuração....87
Figura 4.9 - Sintaxe das declarações de iteração....91
Figura 4.10 - A chamada de funções em IL....93
Figura 4.11 - Estrtura do compilador IL e ST....95
Figura 4.12 - Representação da Arvore de Sintaxe Abstracta resultante da análise da declaração em ST 'vel := 42 + (pos1 – pos2) / delta_t;'....97
Figura 4.13 - Maquina de estados do interpretador léxico....101
Figura 4.14 - Máquina de estados do intepretador léxico, com distinção entre IL e ST....102
Figura 4.15 - Diagrama de classes do gerador de código C++....107
Figura 4.16 - Exemplo de declaração de função....108
Figura 4.17 - Conversão para C++ da função na fig. 4.16....109
Figura 4.18 - Exemplo de declaração de função bloco....110

Figura 4.19 - Classe em C++ correspondente à conversão da função bloco da fig. 4.18. .110
Figura 4.20 - Classe C++ correspondentes à conversão da configuração da fig. 4.21.....114
Figura 4.21 - Exemplo de declaração de um programa e de uma configuração.....114
Figura 4.22 - Classe modelo utilizada no mapeamento de variáveis com mapeamento directas.....115
Figura 4.23 - Conversão dos tipos de dados elementares para C++.....117
Figura 4.24 - Conversão dos tipos de dados derivados.....117
Figura 4.25 - Declaração de função utilizada nos exemplos de conversão seguintes.....119
Figura 4.26 - União de tipos de dados elementares utilizada para a declaração da variável comum da linguagem IL.....120
Figura 4.27 - Exemplo de código IL.....121
Figura 4.28 - Código C++ correspondente à conversão do código IL da fig. 4.27.....121
Figura 4.29 - Exemplo de código IL.....122
Figura 4.30 - Código C++ correspondente à conversão do código ST da fig. 4.29.....122
Figura 4.31 - Exemplo de sintaxe de invocação de funções em IL não suportadas.....123
Figura 4.32 - Possivel conversão para C++ da invocação de função da fig. 4.31.....123
Figura 4.33 - Exemplo de código IL por enquanto ainda não suportado.....124
Figura 4.34 - Possivel conversão para C++ do código IL da fig. 4.33.....124
Figura 4.35 - As palavras chave definidas no anexo C do standard IEC 61131-3.....127
Figura 4.36 - Exemplo de utilização de nomes de instruções IL para identificar funções declaradas pelo utilizador.....132
Figura 4.37 - Extensão à sintaxe sugerida para a invocação de funções bloco armazenadas em um array.....135
Figura 4.38 - Sugestão de alterações à definição formal da sintaxe necessária para a invocação de funções bloco em arrays.....135
Figura 4.39 - Resumo de classificação das linguagens IL e ST.....155
Figura 5.1 - Modelo do IEC 61499.....159
Figura 5.2 - Diagrama de Funções Bloco do IEC 61499.....160
Figura 5.3 - Associações entre eventos e dados.....162
Figura 5.4 - Gráfico de controlo de Execução.....163

Capítulo 1

Introdução

1.1 Motivação e Contexto

Toda a tecnologia que permita aumentar o rendimento e produtividade tem tido naturalmente uma grande aceitação em ambientes industriais. Não seria pois de esperar outra alternativa que não a adopção de micro-controladores e micro-processadores para o controlo e automatização de processos de fabrico. Devido aos ambientes fisicamente hostis nos quais estes necessitam de operar, e ainda para facilitar a sua adopção por parte de técnicos menos qualificados, os micro-processadores foram integrados em equipamentos que mais tarde se vieram a chamar de PLC (Programmable Logic Controllers), conhecidos em Português por autómatos programáveis.

A evolução dos PLCs tem levado a que estes adoptem cada vez mais standards, tanto do ponto de vista do hardware como do software. Considerando o hardware, os PLCs têm estado cada vez mais a adoptar a arquitectura do PC (Personal Computer), que hoje em dia se pode considerar um standard *de facto*. Já do ponto de vista do software, e muito embora cada PLC execute o seu próprio sistema operativo, os programas que estes sistemas operativos executam são cada vez mais escritos recorrendo a linguagens e a arquitecturas de suporte

estandardizadas. Uma vez que os utilizadores, que desenvolvem os programas que implementam a lógica de controlo, vêem uma arquitectura e um conjunto de linguagens estandardizadas, (e não têm acesso directo aos sistemas operativos divergentes que os PLCs executam), pode-se considerar pois que do ponto de vista do software os PLCs têm também convergido num único standard.

No entanto, e muito embora esta convergência na adopção de standards internacionais, é ainda difícil aos utilizadores substituir PLCs de uma marca por outra, ou transferir programas que executam a lógica de controlo entre PLCs de fabricantes diferentes. Os fabricantes têm assim conseguido prender os utilizadores aos seus produtos através de técnicas de subterfúgio tal como a utilização de ambientes de programação proprietários que utilizam formatos também eles proprietários para armazenar os programas, mesmo sendo estes escritos recorrendo a linguagens de programação estandardizadas. Outras dificuldades prendem-se com a utilização de formas distintas de referenciar as entradas e saídas físicas dos PLCs, e ainda das redes de comunicação suportadas pelos PLCs, que embora tenham sido estandardizadas, existem ainda em numero demasiado extenso para que se possa dizer que tenha havido uma convergência dos diversos fabricantes.

Esta prisão dos utilizadores aos fabricantes de PLCs baseia-se assim em componentes proprietários de um sistema que pretende-se que seja standard. Como forma de combater mecanismos semelhantes aos descritos no âmbito mais alargado de sistemas de software genéricos, o Richard Stallman criou em 1985 o FSF (Free Software Foundation) com o intuito de escrever, do zero, um sistema operativo compatível com o UNIX, mas livre de restrições proprietárias. Para tal criou uma licença sob a qual o software escrito por ele era licenciado. Esta licença, conhecida por GPL (GNU Public License), ao contrário das licenças habituais que têm como objectivo restringir o que se pode legalmente fazer com o produto, diz essencialmente que se pode fazer tudo com o software disponibilizado, com a única condição de que se este for alterado e redistribuído a terceiros, essa redistribuição deve ser feita acompanhada pelo código fonte desse mesmo software [1]. Esta licença garante assim que um programa pode ser usado, alterado e até vendido por terceiros. Mas mais importante ainda, uma vez que estas garantias sejam concedidas, nunca mais podem ser retiradas ao programa.

Desde então muito software tem sido escrito e distribuído sob esta licença, incluindo o conjunto de utilitários que geralmente acompanham um sistema operativo UNIX (awk, flex, ls, csh, vi, ...), uma família de compiladores de varias linguagens de programação (gcc, g++, gnat, ...), e o próprio LINUX, o núcleo de um sistema operativo semelhante ao UNIX.

Com o objectivo semelhante de libertar os utilizadores de PLCs da subjugação dos vendedores destes, e tirando partido da proliferação do standard *de facto* que é o hardware dos PCs, o Curt Wuollet sugeriu que fosse criado um PLC

distribuído sob a licença GPL. Este apelo foi efectuado numa lista de email frequentada por engenheiros e técnicos de alguma forma ligados a automação industrial pelo mundo fora. Obteve uma rápida aceitação por parte destes, e muitas ideias e sugestões rapidamente recomeçaram a surgir. Algumas pessoas começaram a oferecer os seus serviços para implementar uma parte do PLC na qual se encontravam mais a vontade, outros para escrever os manuais, e ainda outros para ajudar a testar o novo programa. No entanto, e apesar de todas as ofertas de ajuda, todos continuavam a dar a sua opinião de como deveria ser feito o programa, sem nunca realmente escrevem uma única linha de código. Este impasse foi ultrapassado após algum tempo, quando o Jíri Baum, na Australia, escreveu uma muito pequena biblioteca de funções que permitiam a partilha de dados entre processos recorrendo a memoria partilhada. Incluía também a possibilidade de definir quais os nomes das variáveis (de apenas um bit) que seriam partilhadas.

Por discordar de alguns aspectos desta implementação embrionária, o autor desta monografia passou a implementar o núcleo do que veio a ser o MatPLC, tendo o código desenvolvido sido aceite por todos. Ao longo da implementação deste projecto foram surgindo questões que exigiram analises teóricas, resultando neste trabalho.

1.2 Contribuições

Esta dissertação apresenta as seguintes contribuições de relevo:

- Construção prática de um softPLC para sistemas operativos POSIX, com capacidade de suportar aplicações com restrições de tempo-real
- Construção prática de um compilador das linguagens IL e ST definidas no standard IEC 61131-3, ver. 2.0.
- Análise do standard IEC 61131-3, e identificação de questões que deverão ser corrigidas no mesmo relativas à definição da sintaxe e semântica das linguagens IL e ST.
- Estudo da aplicabilidade de utilizar as linguagens definidas no IEC 61131-3 para a programação de aplicações de elevada fiabilidade.

1.3 Estrutura

O segundo capítulo começa com uma curta descrição da origem e história da evolução dos PLCs aos longos dos tempos. Neste capítulo é ainda efectuada uma síntese de outros projectos e trabalhos que de alguma forma estão relacionados com a automatização de processos. Alguns dos trabalhos abordados foram desenvolvidos no âmbito de projectos com financiamento comunitário ou mesmo nacional do país de origem. No entanto, projectos existem que surgiram pela simples vontade em colmatar alguma lacuna ou falta de ferramentas que o seus autores sentiram.

O terceiro capítulo é dedicado ao projecto e descrição da implementação do MatPLC, a aplicação que deu origem a esta tese. É descrita a arquitectura global do MatPLC, passando de seguida a descrever alguns dos módulos que se encontram já implementados. É ainda formalizada a arquitectura interna do núcleo do MatPLC, sendo descrito com pormenor o funcionamento de cada componente. São ainda apresentados os resultados de alguns testes da execução de componentes relevantes do MatPLC.

O capítulo seguinte é totalmente dedicado ao standard IEC 61131-3. Em primeiro lugar é efectuada uma rápida descrição do referido standard e das linguagens de programação nele definidas, sendo de seguida descrito o compilador das linguagens IL e ST desenvolvido no âmbito no projecto do MatPLC. São ainda levantadas algumas questões relativas ao que eventualmente se poderão considerar lacunas no standard em si, e apresentadas sugestões de como as questões identificadas poderão ser ultrapassadas. Este capítulo termina com uma análise à adequabilidade das linguagens IL e ST para o desenvolvimento de aplicações de elevada fiabilidade.

O documento termina com um capítulo dedicado a conclusões, sendo aqui ainda apresentadas algumas sugestões relativas à continuação do projecto do MatPLC.

Capítulo 2

Tecnologias de Controlo Programável

2.1 História dos PLCs

O primeiro PLC foi criado por Dick Morley no primeiro dia de 1968, estando na altura a trabalhar para a empresa Bedford Associates, localizada nos EUA. O objectivo inicial era o de substituir os relés que até então eram utilizados no controlo de maquinas ferramentas. Como é comum em desenvolvimento, os primeiros protótipos mostraram-se demasiado lentos e não chegaram a ser lançados no mercado. Isto apesar de as entradas e saídas serem mapeadas directamente na memória do PLC, sem qualquer gestão de algo que se assemelha-se a um sistema operativo. A questão da velocidade foi resolvida ao acrescentar mais uma carta dedicada a executar os algoritmos principais de interpretação do programa de controlo, isto sem recorrer a processadores. O primeiro PLC ficou assim constituído por três módulos (memória, processador, e carta lógica), de onde surgiu o nome MODICON - MOdular DIgital CONtroller.

O primeiro modelo comercializado, o Modicon 084, tinha versões com 1, 2, 3 e 4 Kbytes de memória, e era programado com recurso a uma linguagem chamada de 'ladder lister'. Este modelo não foi no entanto um sucesso comercial, pois muitos engenheiros ainda não estavam convencidos de que um PLC poderia ser

tão robusto quanto os circuitos de relés que iria substituir. No entanto, as capacidades de processamento extras (principalmente as operações matemáticas) introduzidas no modelo seguinte, o 184, tornaram-no num sucesso. Isto porque estas operações não poderiam ser facilmente replicadas pelos circuitos de relés. Foram assim as capacidades extras que tornaram os PLC no sucesso que hoje em dia são.

As capacidades continuaram a expandir com a introdução de subrotinas, que eram utilizadas para implementar operações matemáticas mais complexas, controladores PID, servo controladores, etc... Em 1973 surgiu a capacidade de comunicação com equipamento remoto, sendo o primeiro protocolo o ModBus, ainda utilizado nos dias de hoje. Infelizmente, a ausência de standards bem como o constante desenvolvimento da tecnologia em que se baseiam as redes de comunicação, fez com que outros fabricantes de PLCs desenvolvessem os seus próprios protocolos. O resultado foi uma explosão de protocolos, e uma incompatibilidade tanto ao nível da camada física, bem como nos níveis superiores, das redes de comunicação.

Nos anos 80 surgiu o MAP [2](referenciado em [3]), Manufacturing Automation Protocol. Foi uma tentativa, liderada pela General Motors (GM), de estandardização dos protocolos de comunicação industriais, bem como dos perfis de utilização. Infelizmente esta tentativa acabou por falhar por diversas razões, tais como o facto de ter apostado num nível físico (Token Bus) que não chegou a ser largamente adoptado nos ambientes de escritório, incluir protocolos em muitas das camadas OSI tornando-o pesado e de difícil implementação, incluir uma API (Application Programming Interface) muito pesada e de difícil utilização, bem como as razões comerciais dos fornecedores de equipamento de controlo que desejavam manter os clientes presos aos produtos que comercializavam. Os anos 90 resultaram nalguma redução dos protocolos existentes devido à concentração das soluções sobre as camadas físicas que permaneceram em utilização (RS485, CAN, Ethernet) bem como em mais uma tentativa de estandardização. No entanto, vingaram mais uma vez os interesses comerciais dos fabricantes de PLCs, e vários protocolos de comunicação continuam a coexistir ainda hoje.

A evolução tem sido um pouco diferente no prisma da linguagem de programação dos PLCs. Os primeiros PLCs eram programados recorrendo a uma 'linguagem' hoje em dia conhecida por 'Ladder Logic'. Esta 'linguagem' existia já antes da concepção do primeiro PLC, sendo conhecida na altura como 'Ladder Lister', e utilizada na concepção dos circuitos de relés que os PLCs vieram substituir. Foi por isso natural que os fabricantes de PLCs tenham adoptado uma linguagem semelhante, facilitando assim a adopção dos mesmos pelos técnicos habituados a conceber circuitos de controlo baseados em relés.

Uma segunda linguagem veio a ser adoptada mais tarde, baseada numa lista de instruções simples, assemelhando-se assim às linguagens máquina utilizadas na

programação de micro processadores. Esta segunda linguagem passou a coexistir com o Ladder Logic, que nunca chegou a desaparecer.

Pese embora a utilização do Ladder Logic pelos vários fabricantes de PLCs, a portabilidade dos programas escritos nesta linguagem nunca foi muito grande. A grande razão desta falta de portabilidade prendia-se com pequenas diferenças das linguagens de cada fabricante (tanto no Ladder Logic como na linguagem assemelhada à linguagem máquina), e pelas diferenças na forma de endereçamento das entradas e saídas físicas. Outra razão prende-se com a dificuldade de transferir os programas directamente recorrendo a um formato que o próprio PLC entenda, sendo por isso necessário a reintrodução manual dos programa no novo PLC. Esta ultima dificuldade entendia-se como natural a principio, altura em que a programação dos PLCs era feita recorrendo a consolas especificas de cada fabricante. No entanto, com a proliferação dos computadores pessoais na década de 1990, estes passaram a ser utilizados como plataformas de programação dos PLCs. No entanto a dificuldade de transferência de programas mantinha-se, pois as linguagens diferiam entre os fabricantes dos PLCs.

Como forma de ultrapassar esta barreira, foi criado o standard IEC 61131 [4] em 1998, que define uma arquitectura para a programação dos PLCs, bem como de várias linguagens de programação que podem ser utilizadas nessa arquitectura. A adopção de este standard por parte dos fabricantes tem sido bastante boa, no entanto, e apesar de agora todos os PLCs serem programados por uma mesma linguagem que se encontra estandardizada, a portabilidade dos programas continua a ser baixa. Isto porque cada fabricante recorre a um formato distinto para gravar os programas em ficheiros.

Como seria de esperar, o hardware utilizado pelos PLCs tem também evoluído. O primeiro PLC, como já foi referido, recorria ainda a memórias de ferrite e utilizava, para alem do processador genérico, um circuito lógico especifico para executar algumas instruções mais comuns. Esta arquitectura evolui para o uso exclusivo de micro processadores para a execução do programa lógico. Muitos micro processadores foram utilizados nestes sistemas, desde os AMD 2901 e 2903 nos PLCs da Modicon e da Allen Bradley dos anos 70, passando pelos micro controladores SAB 80535 da Siemens muitos anos mais tarde. Ultimamente, e devido à economia de escala tornada possível pela utilização dos processadores da família x86 da Intel nos computadores pessoais (PCs), também estes têm sido utilizados nos PLCs. Devido à proliferação dos PCs à escala mundial, e mais uma vez com o intuito de tirar proveito da economia de escala dos circuitos auxiliares utilizados em conjunto com os processadores, alguns PLCs de topo de gama começam a ser na realidade computadores com arquitecturas compatíveis com os PCs, correndo eles até sistemas operativos geralmente utilizados nos PCs, como o Windows da Microsoft ® ou o LINUX. Exemplos destes são os PLCs comercializados pela Beckhoff e pela SixnetIO. Outra solução que tem sido adoptada ultimamente é o de utilizar PCs com

software apropriado que os tornam semelhantes a PLCs.

2.2 COMEDI

O projecto COMEDI (Linux Control and Measurement Device Interface) [5], [6] desenvolve 'device drivers' para o sistema operativo Linux, os quais permitem o controlo de dispositivos utilizados na aquisição e geração de sinais eléctricos, tais como cartas de Entradas/Saídas digitais e/ou analógicas para barramentos PCI, ISA, etc. A par disto, são também fornecidas bibliotecas de funções para aceder aos 'device drivers' e ferramentas para a sua configuração e utilização. Todo estes programas são distribuídos sob a licença GPL.

Os 'device drivers' COMEDI são constituídos por um módulo do núcleo do Linux comum a todos os dispositivos e que implementa a funcionalidade comum a todos eles. A par disto, existe ainda um módulo do núcleo do Linux para cada dispositivo distinto que é suportado. De momento existem 'drivers' para vários modelos de cartas de cada um dos seguintes fabricantes: ADLink, Advantech, Amplicon, Analog Devices, Computer Boards, Contec, Data Translation, General Standards, ICP, IOTech, ITL, Inova, Intelligent Instrumentation, Keithley, Keithley Metrabyte, Kolter Electronic, Measurement Computing, Meilhaus, Motorola, National Instruments, Quanser Consulting, Quatec, Real Time Devices, SSV Embedded Systems, WinSystems, e ainda cartas genéricas baseados no circuito integrado 8255 da Intel, e para a porta paralela standard dos PCs.

Por outro lado, a biblioteca COMEDIlib é uma biblioteca de funções que executam em modo de utilizador (i.e. fora do núcleo do Linux), e funcionam como um intermediário entre a aplicação e os 'device drivers' COMEDI. Fornecem uma interface fácil de utilizar por parte do programador de aplicações que necessita de aceder às cartas de Entradas/Saídas.

Deste projecto faz ainda parte o COMEDI, que não é mais do que um módulo do núcleo do Linux, pensado para ser utilizado nas versões de Tempo Real do Linux (Ex. RTAI e RTLinux), permitindo que aplicações com restrições de tempo real possam aceder aos dispositivos de Entrada/Saída de forma determinística.

No fundo a grande contribuição deste projecto é o de disponibilizar uma interface de programação comum a todos os 'device drivers' das cartas de aquisição de dados. Um aplicação que recorra à API comum poderá assim utilizar qualquer uma das cartas de aquisição suportadas sem ser necessário alterar em nada a referida aplicação. Adicionalmente, se novas cartas vieram a ser suportadas, a mesma aplicação poderá utilizá-las mais uma vez sem ser necessário introduzir quaisquer alterações à aplicação.

A API suporta a aquisição de entradas e o controlo de saídas, tanto digitais, como analógicas, e ainda pulsantes (temporizadores, contadores, ...). Cada entrada ou saída, seja ela digital, analógica, ou pulsante, é considerada um canal. Cada canal, para alem do valor da saída/entrada, pode ainda ter um conjunto de parâmetros tais como gama de tensão, referencia de tensão, polaridade, factor de conversão entre o valor e as grandeza física que representa, etc.

Um conjunto de canais com características semelhantes são agrupados em sub-dispositivos, por exemplo, um conjunto de 8 entradas analógicas. Cada sub-dispositivo contem parâmetros tais como o numero e tipo de canais que o compõem. Finalmente, uma carta é mapeada num único dispositivo, o qual pode ser composto por vários sub-dispositivos em função das capacidades da referida carta.

A API da COMEDILIB requer que cada dispositivo seja primeiro aberto como um ficheiro, depois do qual se pode obter informações relativas ao dispositivo tais como a gama de valores das entradas/saídas analógicas, numero de entradas/saídas digitais, etc. Inclui também funções para a leitura e escrita discreta de cada canal, bem como para a conversão dos valores lidos para a grandeza física que este representa. Esta leitura discreta é implementada por uma chamada a uma função síncrona, isto é pela chamada a uma função que bloqueia o programa que a chama ate retornar o valor.

Vários canais podem ser lidos sequencialmente usando uma única função da API, também ela síncrona. Esta função pressupõe uma inicialização prévia de uma estrutura de dados que define quais os canais que deverão ser lidos/escritos, e os locais onde deverão ser guardados os valores lidos, ou os valores que deverão ser escritos. O COMEDILIB inclui ainda uma função semelhante à anterior, mas com funcionamento assíncrono. Isto é, as aquisições vão decorrendo em paralelo com a execução do programa principal. Esta função permite tirar partido da capacidade de algumas placas de aquisição de dados de sincronizarem as aquisições/escritas de dados com a ocorrência de eventos despoletados por relógios internos à própria placa. Ou seja, é o hardware da placa de aquisição que faz o trabalho pesado, podendo por isso o programa executar em simultâneo enquanto decorre a aquisição dos sinais.

2.3 Controlador EMC

O EMC (Enhanced Machine Controller) [7], [8], [9] tem as suas origens num projecto do NIST (National Institute of Standards and Technology), um organismo do departamento de comércio do governo dos Estados Unidos da América, e é distribuído também sob a licença GPL. Este projecto é no fundo um

controlador de CNC (Computer Numerical Control) que corre em Linux e nas suas variantes de Tempo Real. O CNC e utilizado em maquinas ferramenta e robots articulados, e controla de forma coordenada o movimento (nas vertentes de aceleração, velocidade e posição) de vários motores, através dos quais se controla o movimento e posição dos braços do robot, ou as ferramentas de corte das maquinas ferramenta.

Os movimentos pretendidos do robot ou da maquina ferramenta são programados através de uma linguagem de programação especifica, geralmente conhecida de códigos G. Este nome tem origem nos comandos da própria linguagem, sendo a maioria deles precedida da letra G. A linguagem é algo semelhante à linguagem maquina utilizada na programação de micro processadores, constituída por uma sequência de linhas de comandos. Cada linha de comando é constituída por uma ou mais instruções e pelos seus respectivos parâmetros. As instruções são geralmente a letra G ou M seguida de um numero. Também os parâmetros são letras que indicam qual o parâmetro que está a ser alterado, seguidos do seu novo valor.

Por exemplo, a instrução 'G0 X11 Y22 Z33' indica que se deve efectuar um movimento para a posição [X,Y,Z] = [11,22,33], o mais rapidamente possível. Uma instrução 'G1' indica uma deslocação em linha recta (interpolação linear) e 'G2' e 'G3' movimentos circulares com orientação inversa e directa respectivamente. Os códigos 'M' são utilizados para accionar dispositivos auxiliares, tais como abrir portas de protecção, activar bomba de circulação de liquido refrigerante, etc.

O EMC é composto por vários módulos, um dos quais é uma interface com o utilizador. De momento existem várias interfaces a partir do qual o utilizador pode escolher: interfaces baseadas em terminais de texto, para estações gráficas X, baseadas em tcl/tk e também em Java. Através desta interface o utilizador pode introduzir comandos directamente, ou escolher um ficheiro contendo um programa completo. Estes comandos são então passados ao módulo que os interpreta, faz a interpolação de movimentos, e os transforma em comandos canónicos para as tarefas de comando de entradas/saídas e de comando de movimentos.

Existem varias implementações das tarefas de comando de entradas/saídas, em função do hardware que se esteja a utilizar para disponibilizar as saídas/entradas físicas (por exemplo, a porta paralela de um PC). De igual forma, existem também varias implementações da tarefa de comando de movimentos, em função do tipo de motores que se estejam a utilizar no accionamento dos eixos. Por exemplo, existe uma versão para o controlo de motores passo-a-passo que produz ondas rectangulares, e outra versão para motores servo, com saídas analógicas, que no fundo implementa vários controladores PID.

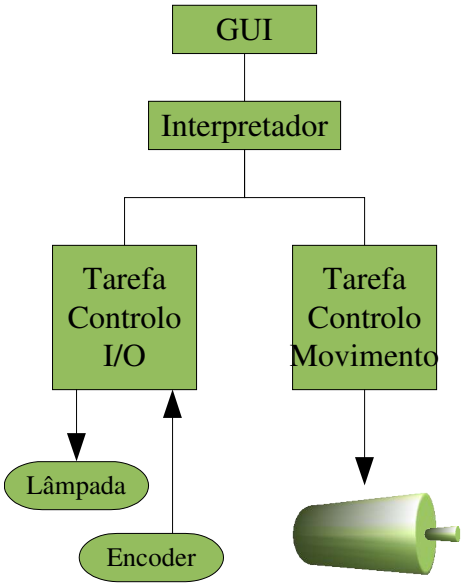

Figura 2.1 - Estrutura interna do EMC.

Este é um projecto bastante maduro, estando já a ser utilizado em ambiente fabril no controlo das mais variadas maquinas.

2.4 OROCOS

OROCOS (Open Robot Control Software) [10], [11], [12] é um projecto que tem como objectivo desenvolver uma infraestrutura de apoio ao desenvolvimento de software de controlo de robots, sendo o software que compõe esta infraestrutura distribuído sob a licença GPL. De facto, a utilidade desta infraestrutura não se limita ao controlo de robots, mas pode ser extendida a aplicações de controlo em malha fechada de diversos dispositivos tais como manipuladores, robots móveis, e maquinas ferramenta. Pode pois ser utilizada independente da arquitectura do sistema de controlo, para aplicações com ou sem restrições de tempo real.

No fundo, este projecto fornece um conjunto de componentes de software que poderão ser interligados para formar uma aplicação de controlo em malha fechada. Os componentes são fornecidos com a infraestrutura desenvolvida pelo projecto, podendo ainda ser extendidos com funcionalidades especiais para um projecto particular. A extensão é baseada na inclusão de 'plug-ins' em cada componente, podendo mais uma vez estes 'plug-ins' serem fornecidos com o projecto, ou serem desenvolvidos para uma aplicação particular. O mesmo componente pode ter vários 'plug-ins' em simultâneo, podendo inclusive cada um ter as suas próprias características quanto a restrições tempo-real e periodicidade de execução.

Cada componente pode ter um ou mais portos. Os portos são objectos de dados que um plug-in desse componente importa ou exporta. O mesmo plug-in pode ter múltiplos portos, sendo que cada porto é unidireccional, i.e. ou importa ou exporta os dados, mas nunca os dois em simultâneo.

A ligação entre portos de vários componentes é efectuada via conectores. Cabe aos conectores providenciarem a difusão da informação de portos de saída por todos os portos de entrada e garantir a consistência dos dados. Esta arquitectura prevê pois que os componentes possam estar distribuídos por vários CPUs, sendo que os conectores encaminham a informação relevante até aos portos finais usando a tecnologia mais adequada.

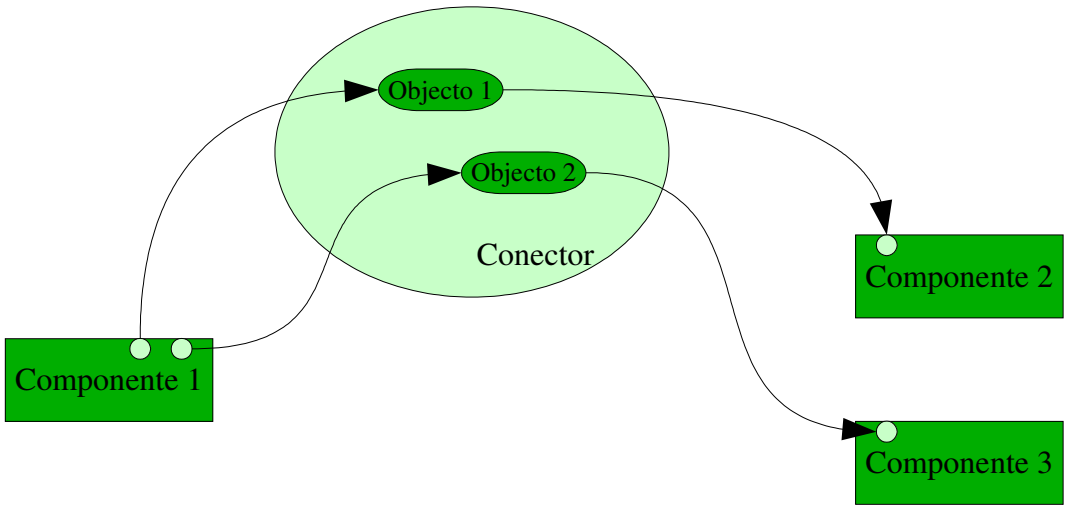

Figura 2.2 - As primitivas estruturais da arquitectura: componentes (quadrados), portos (circulos dentro dos componentes), e conectores (ovais). Os conectores servem de intermediários na transferência de dados entre componentes.

Todos os componentes podem também sincronizar as suas actividades através da troca de eventos. Estes eventos não se encontram descriminados nas figuras seguintes, com o objectivo de não as tornar demasiado densas. Cada componente pode ainda executar várias actividades, cada uma com a sua própria periodicidade de execução.

Para o OROCOS nem todos os componentes são iguais. A arquitectura prevê vários tipos de componentes, sendo que a interligação entre eles está esquematizada na figura seguinte. Varias instâncias desta estrutura base poderão ser criadas no caso da aplicação necessitar de malhas de controlo imbricadas.

O componente 'Sensor' lê valores de entrada do hardware determina o valor das grandezas físicas correspondentes à planta que se encontra a ser controlada.

O 'Estimador' modela a planta a ser controlada, e determina o estado do sistema, incluindo as variáveis de estado que não podem ser medidas directamente, a partir dos valores obtidos pelo 'sensor'.

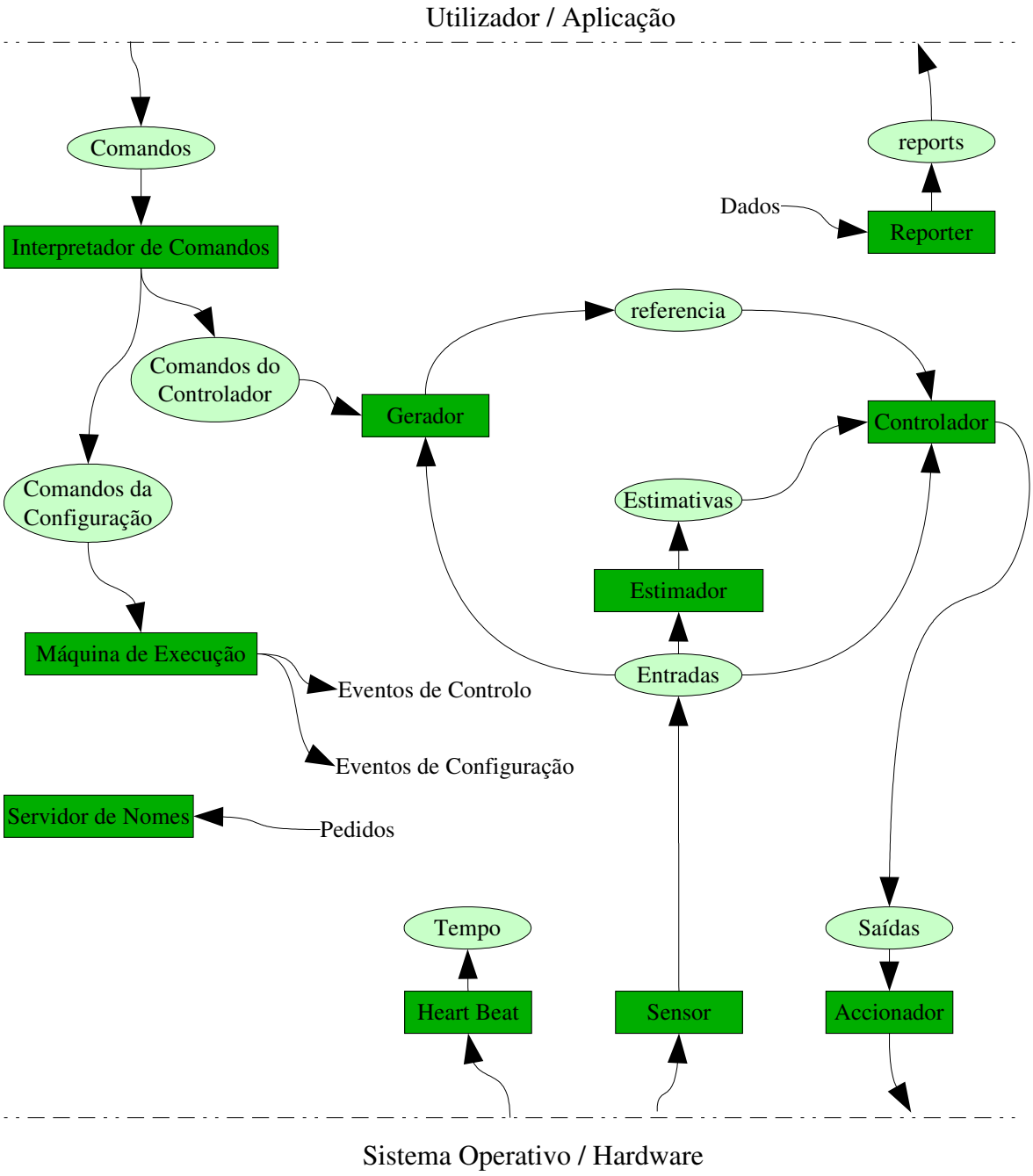

Figura 2.3 - A arquitectura do sistema de controlo: Os componentes (rectangulares) contêm actividades, os Conectores (ovais) contêm objectos de dados. As setas sem conexões (ex. Máquina de Execução) indica que o componente inter-age com (possívelmente) todos os restantes componentes.

O 'Gerador' determina o estado para o qual se deseja que a planta progrida, ou seja o valor que se deseja para as variáveis de estado da planta que se encontra sob controlo.

O 'Controlador' utiliza o estado actual da planta, bem como o estado futuro desejado, para obter os valores que deverão ser colocados na saída por forma a que a planta progrida para o estado desejado. Um exemplo de um plug-in para este componente será o de um controlador PID.

Ao 'Accionador' cabe simplesmente actuar nas saídas do hardware para que lá sejam colocados os valores calculados pelo controlador.

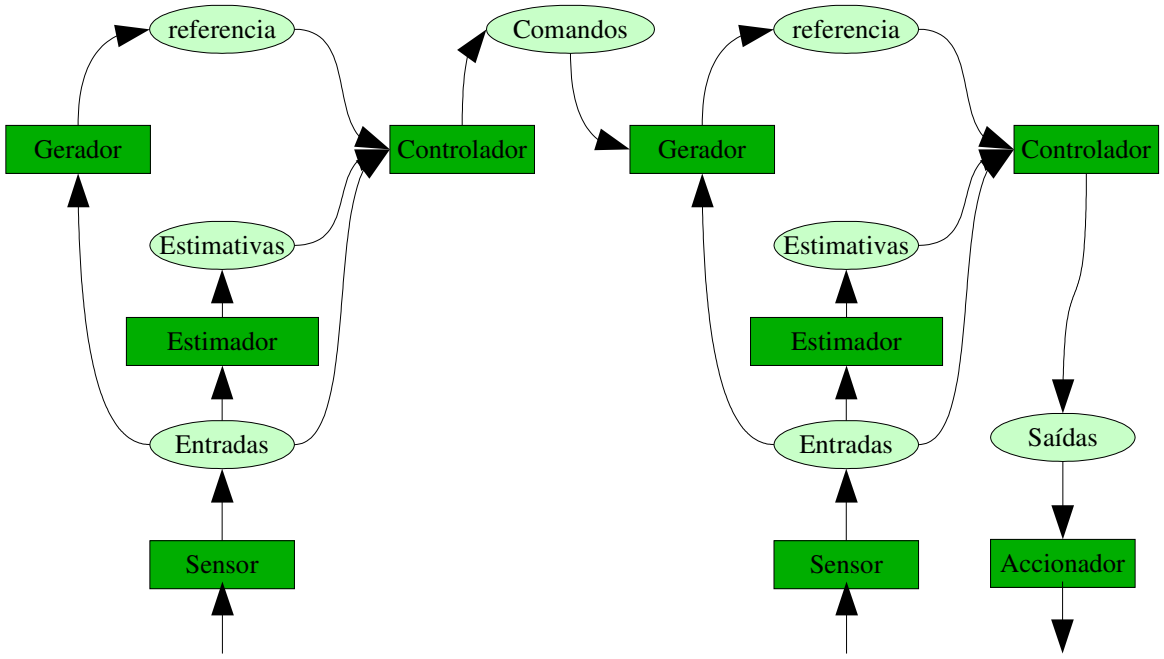

Figura 2.4 - Duas malhas de controlo em cascata.

O 'Interpretador de Comandos' interpreta comandos do utilizador durante a fase de execução da aplicação, e faz com que estes sejam seguidos pelos restantes componentes. Aceita dois tipos de comandos: comandos para o controlador, e comandos de configuração. Os comandos para o controlador especificam as acções desejadas para a aplicação em causa (como por exemplo o movimento desejado para o braço do robot), pelo que a sua sintaxe é dependente da aplicação. Os comandos de configuração são utilizados para especificar quais os 'plug-ins' que uma aplicação irá necessitar, e o instante em que deverão ser activadas estas opções. Estes comandos são executados pela infraestrutura em si, pelo que a sua sintaxe está estandardizada. A alteração de um 'plug-in' durante a fase de execução pode ser utilizada para, por exemplo, alterar o algoritmo de interpolação linear a utilizar. Este componente faz com que os comandos sejam executados através de chamadas a métodos específicos de cada componente (uma vez que os componentes são implementados como objectos na linguagem C++), o que pressupõe que exista canais de transferência de informação entre este componente e os outros que não se encontram explicitamente discriminados na figura 2.3.

A 'Maquina de Execução' contem todos os aspectos de execução que necessitam estar centralizados num único local, tal como a sincronização dos componentes que necessitam de sincronizar as suas acções, o controlo do estado (stop/run) de actividades, configurações, a execução de máquinas de estados, etc. Este componente gera eventos de sincronização e eventos de de configuração para os restantes componentes.

O componente 'Reporter' serve para centralizar a reportagem do estado interno ou a ocorrência de eventos particulares em cada componente.

O 'Heart Beat' é responsável pela sincronização dos relógios caso a aplicação esteja a ser executada num ambiente distribuído.

O 'Servidor de Nomes' permite que cada plug-in seja registado sob um nome que corresponde à função que está a desempenhar. Assim, um plug-in poderá ser substituído por outro que ao assumir o mesmo nome do primeiro, irá parecer com que não tivesse ocorrido qualquer alteração do ponto de vista dos restantes componentes.

O projecto distingue claramente a responsabilidade de cada classe de utilizadores aos quais se destina. Os construtores da infraestrutura são os próprios autores do projecto, que definem a arquitectura da infraestrutura e implementam os serviços necessários à interacção dos componentes, bem como a interface com o sistema operativo, a passagem de eventos, a interpretação de comandos, e gestão de maquinas de estado. Os construtores de componentes são pessoas com conhecimentos profundos numa determinada área, e desenvolvem um ou mais componentes que implementam os algoritmos nos quais são especialistas. Os construtores de aplicações constroem programas de controlo completos para uma determinada aplicação, recorrendo aos componentes e à infraestrutura já desenvolvida. Os utilizadores finais, tais como operadores de maquinas ferramenta, utilizam as aplicações desenvolvidas pelos construtores de aplicações.

Muito embora esta infraestrutura tenha sido desenvolvido no âmbito de um projecto com financiamento da união europeia que já terminou, de momento os componentes 'Maquina de Execução', 'Interpretador de Comandos' e 'Heart Beat' ainda não foram implementados. A restante implementação executa sobre a versão de Linux tempo Real conhecido por RTAI. O projecto OROCOS encontra-se de momento numa segunda fase, tendo conseguido mais financiamento comunitário. O novo projecto, OCEAN, continua o trabalho desenvolvido para um ambiente distribuído, cuja implementação irá ser baseada sobre o CORBA tempo real.

Muito embora o projecto OROCOS tenha como objectivo principal o de ser genérico, e de facto parece atingir o objectivo de isolar a aplicação de controlo do sistema operativo que se encontra a ser utilizado, a sua arquitectura é desnecessariamente rígida no que toca à existência de um numero fixo de componentes com interligações entre eles impostas pela própria arquitectura.

Capítulo 3

O MatPLC

O MatPLC [13] é um programa que implementa a lógica de funcionamento de um PLC. Como tal, irá ser utilizado por três níveis de utilizadores distintos: os programadores, os instaladores, e ainda os operadores. A distinção entre estes níveis de operadores desde logo facilita a compreensão da restante explicação da arquitectura do MatPLC, pois à medida que esta for apresentada poder-se-á imediatamente fazer referencia ao nível dos utilizadores que se espera que venham a fazer uso de determinadas características da arquitectura.

Os programadores são os engenheiros que fazem a concepção do algoritmo de controlo do dispositivo(s) que irão ser controlados/monitorizados pelo PLC, e implementam esse algoritmo utilizando uma linguagem de programação apropriada. Configuram ainda os parâmetros do MatPLC que se tornem necessários para que a aplicação funcione como pretendido.

Os instaladores fazem a instalação física do sistema no ambiente fabril, e colocam o sistema em funcionamento. Estas acções podem envolver a instalação de software no equipamento físico, e a ainda configuração de alguns parâmetros do MatPLC ou do sistema operativo sobre o qual está a correr. Como exemplos, refira-se os endereços de cartas rede Ethernet ou de redes de campo, a configuração das interrupções a utilizar pelas cartas de hardware instalados no

sistema, etc.

Por outro lado, os operadores limitam-se a operar o MatPLC através das interfaces gráficas ou outras (por exemplo, botões de pressão, terminais de texto) disponibilizadas pelo programador.

Em paralelo com os níveis anteriores, pode-se considerar ainda os programadores de sistema, que implementam o MatPLC em si. Tendo em conta que o MatPLC é disponibilizado com o código fonte, este pode ser facilmente adaptado e expandido para fazer face a alguma necessidade especial da aplicação final. No entanto, espera-se que este nível de participação seja relativamente reduzido pois exige conhecimentos profundos de programação e de sistemas concorrentes.

3.1 Arquitectura Geral

Tal como todos os projectos de Engenharia de grandes dimensões, o MatPLC utiliza uma arquitectura modular [14]. O seu desenvolvimento torna-se assim mais fácil, pois os diversos módulos podem ser desenvolvidos em paralelo, e testados de forma isolada ainda antes de serem integrados no conjunto. Sendo que o MatPLC pretende emular em software a execução de um PLC, e que estes últimos apresentam já uma arquitectura modular, torna-se natural utilizar como a divisão entre módulos os mesmos módulos existentes em PLC comerciais. Desta forma, teremos módulos do MatPLC que tratam das entradas ou saídas físicas, módulos que tratam das comunicações via redes de dados, módulos que gerem a interacção com o operador, e ainda módulos que executam a lógica de controlo. Outras funções são ainda possíveis, mas não interessa desde já estarem a ser enumeradas.

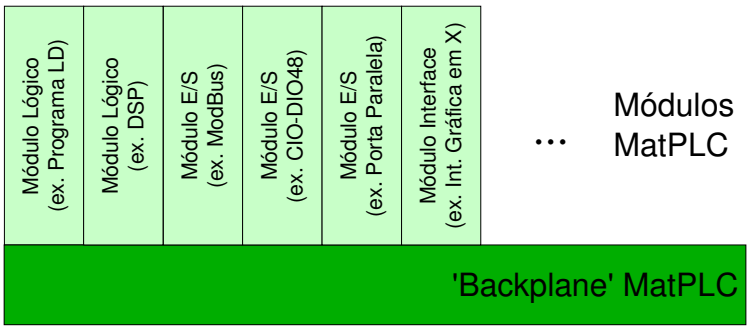

Figura 3.1 - Arquitectura geral do MatPLC.

Os módulos descritos têm de coordenar a sua actividade de forma a agir como um todo. Para tal, necessitam de partilhar informação, e de sincronizar as suas actividades. É esta a tarefa principal do núcleo do MatPLC. Este núcleo está também ele organizado em secções relativamente estanques por forma a facilitar o seu desenvolvimento.

A informação a partilhar entre os módulos distingue-se entre dados de configuração do MatPLC, e dados que representam o estado interno da aplicação que se encontra a ser executada. Estes últimos são organizados em variáveis, que são definidas pelo programador. As variáveis podem ter um comprimento entre 0 a 32 bits, e do ponto de vista do núcleo do MatPLC, são apenas uma sequência de bits sem qualquer significado. Cabe a cada módulo que necessite de ler ou escrever numa determinada variável de lhe dar o significado que intender.

Desta forma, a um módulo que faça a interface com uma carta de entradas/saídas físicas, caberá apenas ler um determinado conjunto de variáveis do MatPLC, e copiar o seu conteúdo para as saídas físicas. De igual forma, irá ler as entradas e escrever o seu estado num outro conjunto de variáveis. Um módulo que executa o programa que implementa a lógica de controlo do sistema a controlar, irá ler as variáveis que foram actualizadas com o estado das entradas, e actualizar o valor das variáveis que serão posteriormente copiadas para as saídas.

Apenas um módulo tem permissão de escrita em cada variável do MatPLC. Todos os outros módulos poderão apenas ler o valor armazenado nessa variável. Cumpre ao instalador configurar qual o módulo com permissão de escrita em cada uma das variáveis do MatPLC.

As variáveis do MatPLC funcionam como uma interface que isola o funcionamento dos módulos. Pode-se, assim, e utilizando o mesmo módulo com a lógica de controlo, rapidamente substituir o módulo que faz a interface com o exterior com um módulo, por exemplo, que faz a interface com uma rede de dados. Desta forma passam as saídas físicas, que eram inicialmente locais, a serem remotas controladas através de uma rede de dados.

Para a implementação desta arquitectura foi escolhido que cada módulo fosse mapeado num processo do sistema operativo, ficando as variáveis armazenadas em memoria partilhada.

Os módulos podem ser caracterizados segundo duas classificações ortogonais: quanto à sua função, e quanto à sua origem. Em relação ao primeiro método de classificar os módulos, e de uma forma genérica, os módulos irão implementar uma de três funções principais: interface física com entradas/saídas, interface com o operador, e lógica de controlo.

Consideram-se módulos de entrada/saída todos os módulos que copiam o estado das variáveis de saída (do ponto de vista do módulo lógico) para local exterior ao do MatPLC, e actualizam o estado das variáveis de entrada com o valor obtido do exterior. Exemplos de módulos de entrada/saída são todos os módulos de interface com cartas de entradas/saídas físicas, módulos de interface com redes industriais de campo, módulos de interface com bases de dados, e ainda módulos de registro ('logging') para ficheiros.

Os módulos de interface com o operador são essencialmente módulos que gerem

uma interface gráfica com o utilizador, ou uma interface baseada em texto. Módulos que fazem a leitura do estado de botões de pressão e accionam lâmpadas que serão observadas pelo operador também poderiam ser consideradas de interface com o operador, mas no nosso caso consideram-se como estando na classificação anterior de módulos de entrada/saída.

Os módulos lógicos são os módulos que actuam apenas no estado das variáveis internas do MatPLC. Podem ser módulos que implementam um filtro digital, um controlador PID ou um controlador de lógica difusa, ou até mesmo programas completos escritos nalguma linguagem de programação de alto nível.

O segundo método de classificação dos módulos é quanto à sua origem: fornecidos com o projecto MatPLC, criados de forma semi-automática por ferramentas fornecidas com o MatPLC, e módulos escritos inteiramente por terceiros ou o programador.

No primeiro caso contam-se módulos tais como o módulo DSP (Digital Signal Processing), que implementa filtros digitais, controladores PID (Proporcional, Integral, Derivativo), e outras funções semelhantes. Este módulo é fornecido com o MatPLC, e necessita apenas que seja configurado com os parâmetros dos filtros, os parâmetros dos controladores PID, as suas variáveis de entrada e de saída, etc. Outro módulo deste género é o Vitrine, um módulo que gere uma interface com o operador à base de texto, que necessita apenas de ser configurada.

Nos módulos produzidos por ferramentas fornecidas com o MatPLC encontra-se o compilador das linguagens IL e ST especificados no standard IEC 61131-3. Neste caso o programador escreve um programa numa das linguagens suportadas, e o compilador gera um módulo do MatPLC.

Os módulos podem ainda ser escritos totalmente pelo utilizador, recorrendo para isso a linguagens de programação como C e Python. Neste ultimo caso o utilizador necessitará de utilizar de forma directa as bibliotecas de funções fornecidas com o MatPLC.

3.2 Micro-Núcleo vs Núcleo Monolítico vs Exo-Núcleo

O núcleo do MatPLC pode ser considerado um sistema operativo próprio para suportar aplicações com a semântica dos PLCs. Neste caso, existem à partida três alternativas de o implementar.

Os núcleos monolíticos são os núcleos nos quais o acesso aos recursos por si geridos é efectuada por apenas o programa que constitui o próprio núcleo. O Linux serve como exemplo dos núcleos que utilizam esta arquitectura. O acesso

aos recursos por si geridos, (i.e. cartas de rede, discos rígidos, portas série, RTC - Real Time Clock, controladores de barramento, etc.) é efectuado apenas pelo próprio núcleo. O núcleo neste caso é um programa com as suas próprias variáveis de estado (ex. lista de processos, lista de semáforos, etc.) às quais só ele pode aceder. Um programa normal que necessite de aceder a estes recursos necessita de fazer uma chamada ao sistema, i.e. ao núcleo monolítico. Enquanto decorre a chamada ao sistema, o programa que o fez é interrompido.

Os micro-núcleos são sistemas operativos minimalistas nos quais o controlo de dispositivos tais como os sistemas de ficheiros, acesso às redes de dados, etc. é efectuada por programas que executam no exterior do núcleo em si. Neste caso, um programa que necessite de aceder a um desses recursos, em vez de fazer uma chamada ao sistema, envia uma mensagem ao programa que controla esse recurso. Cabe ao micro-núcleo fazer a transferência dessa mensagem. Serve como exemplo de um sistema operativo com esta arquitectura o Mach, o QNX, o HURD da GNU, e ainda as primeiras iterações do Windows NT (que hoje em dia a própria Microsoft reconhece que já o deixou de ser).

Um Exo-núcleo [15], [16] é um sistema operativo no qual de facto não existe nenhum programa que constitui o núcleo. Todos os programas têm acesso ao estado partilhado do sistema, bem como podem aceder directamente aos recursos por ele geridos. No entanto, e por forma a garantir que o estado interno não seja corrompido por um qualquer programa, o acesso ao estado partilhado é geralmente efectuado através de um conjunto de funções que são integradas em cada um dos programas quando estes são compilados (na realidade, apenas na fase de 'linkagem'). Não existem sistemas operativos completos com a arquitectura de um exo-núcleo que tenham tido grande divulgação pois estes sofrem da desvantagem de poderem ter o seu estado interno corrompido por um qualquer programa que não siga as regras de aceder ao estado partilhado. Esta característica é geralmente indesejável, sendo pelo contrário preferível isolar os programas dos utilizadores uns dos outros de tal forma que se um se comportar mal, esse comportamento não afectar os restantes programas. No entanto, esta arquitectura tem a vantagem de ser a mais rápida de todas, pois não necessita da transferência de mensagens tais como os micro-núcleos, nem de chamadas ao sistema operativo como os núcleos monolíticos.

Tendo em consideração que o MatPLC não pretende ser um sistema operativo completo, não sendo necessário re-implementar as funcionalidades já oferecidas pelos sistemas operativos existentes, a arquitectura do MatPLC apresenta-se alterada em relação às arquitecturas estandardizadas e que são apresentadas nas figuras 3.2, 3.3, e 3.4. Assim, e usando as mesmas filosofias atrás descritas, mas agora considerando disponível um sistema operativo completo, a arquitectura do MatPLC ficaria como representado nas figuras 3.5, 3.6, e 3.7.

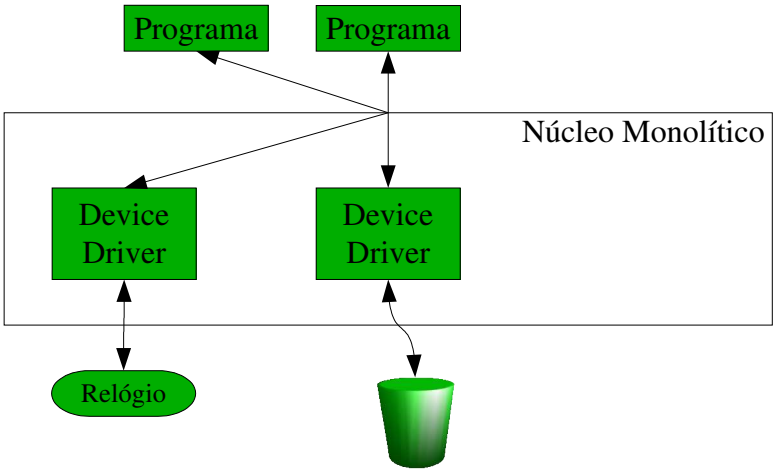

Figura 3.2 - Um sistema operativo com núcleo monolítico.

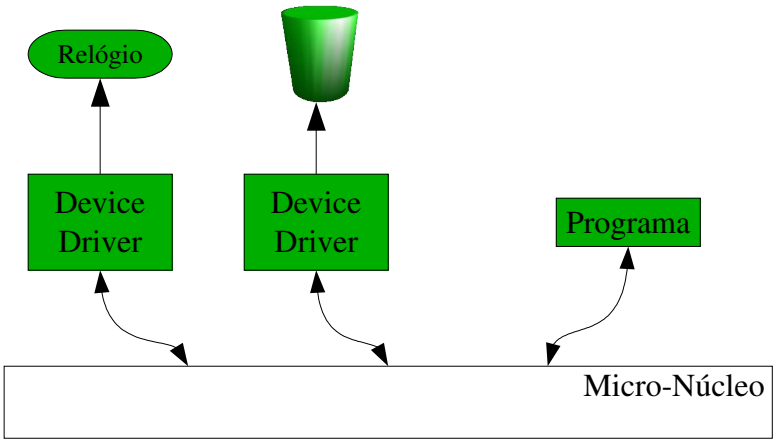

Figura 3.3 - Um sistema operativo com micro-núcleo.

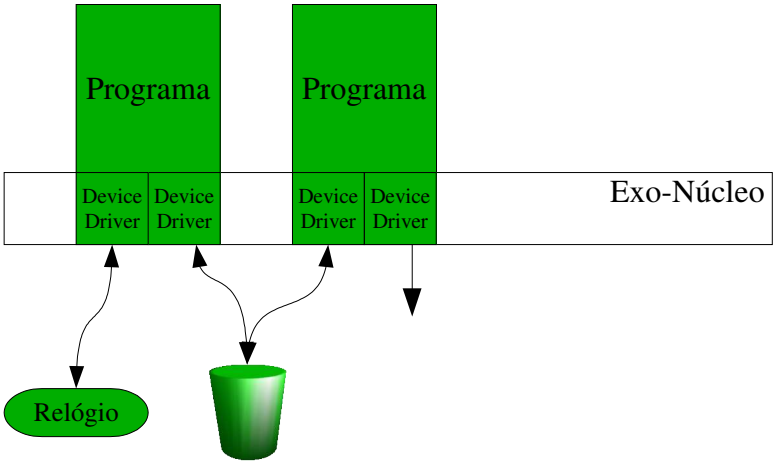

Figura 3.4 - Um sistema operativo com um exo-núcleo.

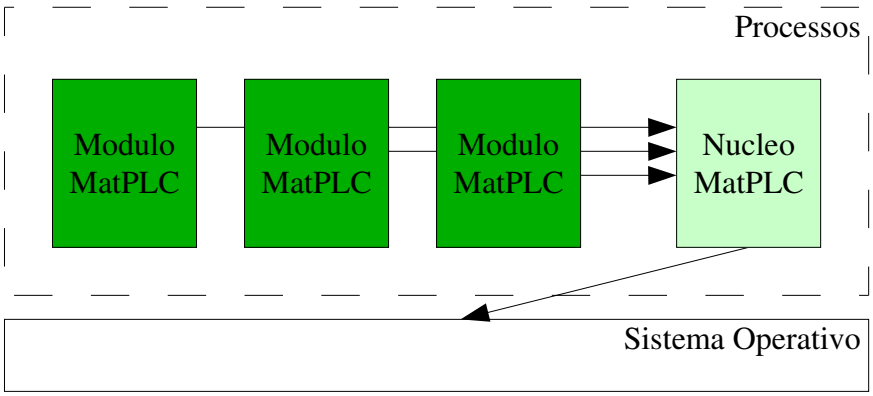

Figura 3.5 - O MatPLC com arquitectura monolítica.

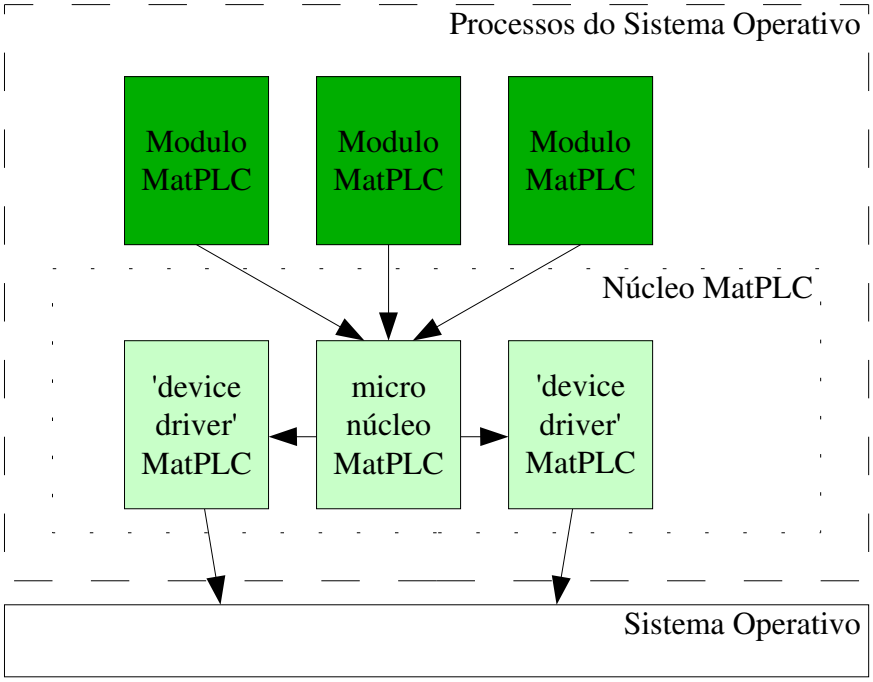

Figura 3.6 - O MatPLC com um micro-núcleo.

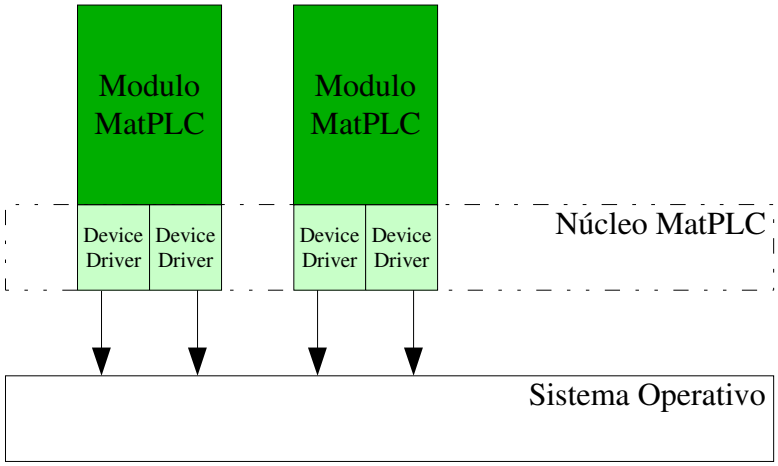

Figura 3.7 - O MatPLC com um exo-núcleo.

A arquitectura de núcleo monolítico seria implementada por um único processo para esse mesmo núcleo, sendo esse processo responsável pelo controlo de acesso a todos os recursos partilhados entre os módulos do MatPLC. Estes recursos poderiam estar tanto dentro do próprio núcleo do MatPLC como dentro do núcleo do sistema operativo.

A arquitectura de micro núcleo do MatPLC seria constituída por vários processos. Um processo principal, e outro processo por cada recurso partilhado pelo MatPLC. Estes últimos processos seriam responsáveis por efectuar todos os acesos ao recurso partilhado por ele gerido. O processo principal seria encarregado de re-encaminhar todos os pedidos de acesso aos recursos, por parte dos módulos do MatPLC, ao processo encarregado de gerir o recurso correspondente.

As duas arquitecturas até agora descritas poderiam ainda, no caso de serem suportadas por um sistema operativo monolítico, terem as componentes do núcleo do MatPLC incorporadas como módulos directamente no sistema operativo em si. Esta alternativa é posta de parte pois implicaria que o MatPLC seria muito difícil de portar para outros sistemas operativos. É de notar que mesmo que o sistema operativo para o qual se pretenderia portar o MatPLC fosse também ele monolítico, as interfaces internas de comunicação entre os módulos do sistema operativo seriam certamente diferentes, podendo até recorrer a semânticas diferentes, ficando assim qualquer tentativa de portar o código mais uma vez comprometida.

As arquitecturas até agora consideradas sofrem ainda da desvantagem de necessitar de uma grande quantidade de processamento que se limita a transferir os pedidos entre os diversos módulos. Cada pedido necessitaria de duas mudanças de contexto por parte do sistema operativo principal para o caso do núcleo monolítico, e quatro mudanças de contexto para o micro núcleo. Tem no entanto a vantagem de manter o núcleo do MatPLC bem compartimentalizado, tornando-o assim mais fácil de desenvolver.

A terceira opção reside em utilizar uma arquitectura tipo exo-núcleo para o núcleo do MatPLC. Neste caso o código de acesso aos recursos partilhados seria replicado por cada módulo do MatPLC, sendo estes autorizados a aceder directamente aos recursos partilhados entre eles. Todos os acessos a estes recursos devem ser efectuados através das funções partilhadas, e nunca directamente. Poder-se-á considerar assim que as funções partilhadas constituem o núcleo do MatPLC.

Esta arquitectura tem a grande vantagem de limitar ao mínimo a transferência de mensagens entre módulos, e ainda a necessidade de mudança de contextos entre processos. Sofre no entanto da desvantagem de não ser possível isolar o código que constitui o módulo, do código utilizado para aceder directamente aos recursos partilhados. Cria-se assim a possibilidade de os acessos ao estado

partilhado serem efectuados de forma directa, contornando as funções do núcleo do MatPLC desenvolvidas de propósito para gerir os acesos ao estado partilhado, o que poderá resultar em potenciais conflitos e estados inconsistentes nas variáveis internas do MatPLC. É de notar também que as funções partilhadas que constituem o núcleo do MatPLC não necessitam de ser obrigatoriamente replicadas em cada módulo. Os sistemas operativos modernos com sistemas de gestão de memória virtual permitem que os diversos módulos acedam à mesma memoria física na qual está armazenado o código, uma vez que esta apenas necessita de ser lida e nunca escrita.

3.3 O núcleo do MatPLC

A opção de utilizar uma arquitectura de exo-núcleo torna o núcleo do MatPLC uma simples biblioteca de funções. No entanto, e com o intuito de manter uma arquitectura compartimentalizada, o núcleo está ele próprio dividido em secções, cada uma com as suas responsabilidades. Para cada recurso que seja partilhado entre os módulos, existe uma secção independente para a gerir. Consegue-se assim ter algumas das vantagens da arquitectura tipo micro-núcleo.

De forma sucinta, o núcleo esta dividido nas seguintes secções:

CMM – Configuration Memory Manager

GMM – General Memory Manager

Synch – Synchronization Manager

Period – Period Manager

State – Sate Manager

RT – Real-Time Parameters Manager

Log – Logging Manager

Conf – Configuration Parser

O CMM gere uma zona de memoria na qual são armazenados dados correspondentes à configuração do MatPLC, bem como o estado interno do MatPLC que tem de ser acessível a todos os módulos. Esta secção torna-se necessária pois a arquitectura do exo-núcleo utilizada pelo MatPLC significa que os módulos não têm acesso directo à memoria dos restantes módulos, e por isso não podem aceder a todas as cópias das variáveis declaradas no código que constitui o núcleo do MatPLC uma vez que cada módulo terá a sua cópia dessas variáveis.

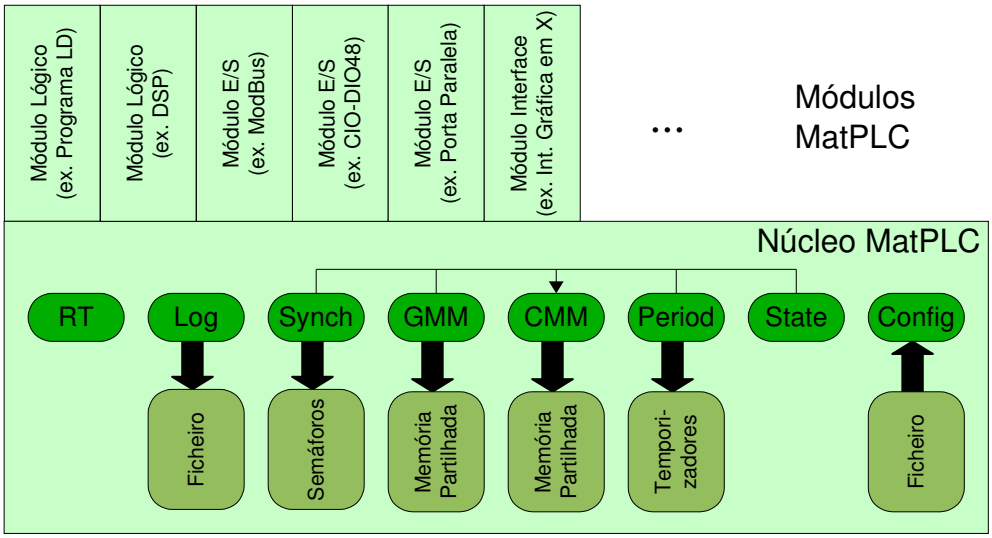

Figura 3.8 - A arquitectura do MatPLC.

O GMM gere a memoria na qual são armazenadas as variáveis utilizadas pelas aplicações de controlo que o MatPLC executa, geralmente referenciados por pontos nos manuais de utilização do MatPLC. Este acesso é protegido por um semáforo de forma a sincronizar os diversos módulos. A secção Synch gere a sincronização dos módulos, i.e. garante que os módulos executam de forma sincronizada e de acordo com as configurações dadas pelo instalador. A secção Period gere a periodicidade de execução de cada módulo, enquanto que a secção State gere o estado destes. A secção Log gere a gravação de todos os eventos relevantes durante a execução de cada módulo, sendo geralmente utilizada para gravar registos da ocorrência de erros. Por último, a secção Conf é responsável pela interpretação das configurações do MatPLC fornecidas pelo instalador/utilizador.

Cada uma das secções atrás descrita é composta por dois conjuntos de funções: as funções que são executadas durante o arranque do MatPLC, e as funções usadas durante a execução do mesmo.

As funções de arranque são responsáveis por inicializar o MatPLC, incluindo a reserva de todos os recursos necessários à sua execução. Estas funções são geralmente apenas utilizadas por programas utilitários que gerem o estado do MatPLC, por exemplo, criando ou destruindo uma determinada instância do MatPLC.

As funções usadas durante a execução são geralmente utilizadas pelos módulos para aceder aos recursos partilhados. Esta distinção, e a sua extensão a cada secção, permite uma melhor organização do código.

3.3.1 O CMM

Tal como já foi referido, o CMM gere uma zona de memoria à qual todos os módulos têm acesso, e que é utilizada para armazenar a configuração actual do

MatPLC, bem como o estado das restantes secções do núcleo do MatPLC ao qual todos os módulos necessitam de aceder. É implementada recorrendo a uma zona de memoria partilhada entre processos ('shared memory'), a qual é gerida pelo sistema operativo sobe o qual executa o MatPLC.

Convém ter em conta que cada secção do núcleo tem as suas próprias variáveis que contêm a sua parte do estado interno do MatPLC. Se as variáveis de estado de todas as secções fossem armazenadas nesta zona de memoria sem serem organizadas em grupos, qualquer alteração a uma variável de estado de uma secção poderia implicar a alteração do código que implementa o CMM. De forma a obviar esta ligação entre secções, e torná-las mais independentes entre si, o CMM organiza a sua memoria em blocos.

O CMM fornece então funções para a reserva de memoria, e ainda uma função para a libertação de memoria caso a memoria alocada não seja mais necessária. A partir do momento que um bloco de memoria e alocado, o alocador poderá fazer o que bem entender com a essa memoria, inclusive utilizá-la para armazenar variáveis de estado. Desta forma, as restantes secções quando necessitam de manter variáveis de estado partilhadas por todos os módulos começam primeiro por requisitar um bloco de memoria com a dimensão adequada ao CMM.

A requisição de memoria é geralmente efectuada pelo grupo de funções de arranque de uma determinada secção. Se as funções de funcionamento necessitam de aceder a essas variáveis, terão primeiro de obter a localização do bloco de memoria anteriormente reservado. Para tal o CMM permite atribuir a cada bloco de memoria um código respeitante ao tipo de dados nele armazenados, bem como uma sequência de caracteres (i.e. uma string). As funções de funcionamento normal podem então obter a localização do bloco de memoria requisitando-o ao CMM, fornecendo para tal o tipo de dados e a string que identificam esse bloco de memoria á função de localização do CMM.

O CMM recorre a uma lista duplamente ligada para gerir os blocos de memoria. Cada bloco de memoria alocado necessita por isso de uma estrutura de dados para armazenar as duas referencias utilizadas para manter a lista duplamente ligada. Esta estrutura é armazenada no inicio do próprio bloco de memoria alocado pelo CMM. Para tal, o CMM aloca sempre os blocos com um comprimento extra suficiente para armazenar a estrutura de dados que contem as referencias. No entanto, o endereço retornado a quem solicitou o bloco de memoria referencia o primeiro byte de memoria livre desse bloco.

A zona de memoria utilizada para conter os blocos de memoria que serão alocados é inicializada com um único bloco de memoria do tipo 'memoria livre'. Pedidos de alocação de memoria são acedidos utilizando a memoria dos blocos de memoria livre. Assim, aquando do pedido de reserva de um bloco de memoria, um bloco de memoria livre é partido em dois, criando um bloco de memoria livre mais pequeno com a memória sobrante, e outro que será

retornado a quem solicitou o bloco de memoria. O algoritmo neste momento utilizado utiliza uma pesquisa de todos os blocos livres, sendo utilizado o bloco mais pequeno de memoria livre que tenha memoria suficiente para satisfazer o pedido.

A libertação de um bloco de memoria começa pela troca do seu tipo para um bloco de memoria livre. De seguida, este novo bloco de memoria livre é fundido com o bloco anterior e/ou posterior se qualquer um destes for também um bloco de memoria livre. Tenta-se desta forma evitar que a memoria fique fragmentada após um numero elevado de alocações e libertações de blocos de memoria.

3.3.2 O GMM

O GMM gere o acesso ás variáveis utilizadas pelas aplicações de controlo executadas pelo MatPLC e que necessitam ser partilhadas entre os módulos (também conhecidos por pontos do MatPLC), contendo assim o estado interno do programa de controlo. Apenas um módulo tem permissão de escrita em cada variável. Os restantes módulos limitam-se a ler as variáveis na qual não tem permissão de escrita.

As variáveis são armazenadas numa zona de memoria partilhada, gerida exclusivamente pela secção GMM. Todos os acessos a esta zona de memoria, também conhecida como mapa global, devem ser efectuados através do conjunto de funções disponibilizadas pelas secção GMM.

As variáveis que deverão ser armazenadas no mapa global são definidas pelo instalador, num ficheiro de configuração do MatPLC. A cada variável é atribuído um nome, o comprimento em bits, e o módulo com direito de escrita nessa variável. Esta informação é transferida para a memoria de configuração (gerida pelo CMM), para garantir que todos os módulos utilizem a mesma configuração, uma vez que o ficheiro de configuração poderia eventualmente vir a ser alterado entre a activação de dois módulos. Os dados de cada variável atrás descrita, bem como a sua localização dentro do mapa global, são armazenadas num bloco de memória do CMM. Isto é efectuado durante o arranque do MatPLC em si, antes de ser lançado qualquer módulo.

A par da zona de memoria partilhada, conhecida pelo mapa global, cada módulo aloca uma zona de memoria à qual apenas esse módulo tem acesso. Esta segunda zona de memoria, com uma dimensão igual ao do mapa global, serve para armazenar um mapa utilizado no controlo de acesso às variáveis na zona de memoria partilhada. É efectuada uma correspondência de um para um entre cada bit destes dois mapas. Em cada bit do mapa global ao qual o módulo tem direito de escrita, o bit correspondente no mapa de controlo é colocado a 1. Todos os restantes bits do mapa de controlo são colocados a 0. É de notar que o conteúdo dos mapas de controlo irá depender do módulo ao qual pertence.

Com recurso ao mapa de controlo, e ainda a operações lógicas simples de 'E',

'OU' e 'OU Exclusivo', torna-se possível fazer acessos ao mapa global de forma rápida e eficaz e garantir simultaneamente que o controlo de acesso é respeitado. Cada copia do mapa de controlo é inicializado aquando da inicialização do módulo ao qual pertence, usando os dados armazenados no CMM que definem quais os pontos existentes, e quais os módulos com permissões de escrita sobre os mesmos.

Uma vez que o MatPLC pretende implementar a semântica habitual dos PLCs, a API *(Application Programming Interface)* disponibilizada pela secção GMM para acesso aos pontos partilhadas reflecte esta opção. A secção define um tipo de variável que referencia um ponto do PLC. Todas as funções de leitura e escrita de um ponto necessitam como parâmetro uma variável que referencia esse ponto. A variável de referência não é mais do que uma estrutura que contem a localização do ponto dentro do mapa global. As variáveis que referenciam pontos devem ser inicializadas por uma função fornecida pelo GMM, a qual necessita apenas como seu parâmetro o nome do ponto que a variável deverá referenciar.

Todos os acessos aos pontos partilhados são controlados de forma a proteger contra os riscos de escrita e leitura simultânea por parte de dois ou mais módulos. Esta protecção é efectuada recorrendo a mecanismos diferentes, consoante o método utilizado para aceder ao mapa global. Existem pois três métodos distintos de ler/escrever os pontos no mapa global: método local, método partilhado, e método isolado. Cada módulo pode utilizar qualquer um dos métodos mencionados, cabendo geralmente ao instalador a decisão de qual o método que cada módulo deverá utilizar. No entanto, uma vez efectuada a escolha do método de acesso por parte do instalador, esse método será utilizado para aceder a todos os pontos que o módulo necessita. Não é pois possível definir um método distinto de acesso dependente do ponto e para um determinado módulo do MatPLC. A opção de acesso é pois efectuada com a granularidade do módulo.

O método local é utilizado por omissão, i.e. quando não é especificado explicitamente na configuração qual o método de acesso ao mapa global que deverá ser utilizado. Cada módulo que utiliza este método gere uma cópia local do mapa global. Esta copia local não é mais do que uma zona de memória, com dimensão idêntica ao do mapa global, que é alocada aquando da inicialização do módulo em causa. Todos os acessos aos pontos, sejam eles para leitura ou escrita, são efectuados sobre a copia local do mapa global. A sincronização entre os dois mapas é efectuado por uma função disponibilizada pelo GMM, a qual deve ser invocada pelo módulo quando este achar oportuno.

Este método local garante uma semântica comum nos PLCs, ao garantir que o estado dos pontos (do ponto de vista do módulo) nunca se alteram entre duas invocações da função de sincronização, a não ser que o próprio módulo escreva um valor no ponto. Os pontos sobre o qual o módulo tem apenas permissão de leitura ficam garantidamente sempre com o mesmo valor entre duas invocações

da função de sincronização. Este mecanismo assemelha-se ao funcionamento dos programas dos PLCs que executam um ciclo infinito, composto por uma fase de sincronização do estado interno com as saídas e entradas físicas, e uma segunda fase de execução do programa o qual acede apenas ao estado interno, que é garantido não sofrer alterações para além daquelas efectuadas pelo próprio programa.

Uma vez que neste método apenas um módulo tem acesso a cada copia local, o acesso a esta copia não necessita de ser protegida contra acessos simultâneos. O mesmo já não acontece com a função que sincroniza as cópias locais com o mapa global. Esta função recorre por isso a um semáforo para controlar o acesso ao mapa global, garantindo assim que em cada instante apenas um módulo pode sincronizar o seu mapa local com o mapa global. Esta sincronização é também atómica e indivisa, não podendo ser interrompida a meio por outro módulo que pretenda também sincronizar o seu mapa local.

Obviamente que o método local sofre do inconveniente de causar potenciais bloqueios a certos módulos enquanto outro módulo sincroniza os seus mapas. Estes tempos de bloqueio poderão ser considerados relativamente elevados uma vez que são atómicos e indivisos. Para tal, decidiu-se fornecer o método partilhado de acesso ao mapa global.

No método partilhado o acesso aos pontos é efectuado directamente sobre o mapa global. Ou seja, os módulos não mantêm a sua cópia local do mapa global, usando como alternativa o próprio mapa global. A função de sincronização de mapas (do mapa global com o agora inexistente mapa local) continua a existir, mas é agora substituída por uma função que simplesmente retorna sem fazer seja o que for.

Para garantir que o acesso aos pontos é efectuado de forma mais rápida possível, não é efectuado qualquer controlo de acesso ao mapa global, podendo por isso existir vários módulos a aceder a este mapa em simultâneo. No entanto, o acesso simultâneo ao mesmo ponto por parte de dois ou mais módulos pode fazer com que o ponto fique com um estado inconsistente. Por este motivo este método de acesso só deverá ser utilizado por módulos que garantidamente executam sozinhos. Poderá eventualmente ainda vir a ser utilizado por módulos que executam em paralelo com outros módulos, mas que acedem a pontos disjuntos no mapa global. Estas garantias de execução poderão ser obtidas por uma correcta configuração da secção de sincronização de módulos.

O método isolado protege ainda mais o recurso partilhado que é o mapa global de eventuais erros ou código maldoso inserido no módulo. Neste caso, o módulo nunca fica com acesso directo ao mapa global, pelo simples facto de que este nunca é mapeado na memória virtual do processo usando os mecanismos habituais de memória partilhada. De facto, os módulos que utilizam o método isolado lançam um segundo processo durante a sua fase de arranque. Este

segundo processo irá servir de intermediário entre o módulo e o mapa global. Uma vez que o processo intermediário executa código fornecido pertencente à própria secção GMM, este pode ser confiado em não transgredir as regras de acesso ao mapa global.

O processo intermediário encarrega-se de criar uma cópia local do mapa de pontos, e de simultaneamente aceder ao mapa global através de memória partilhada. O módulo, que está a utilizar o método isolado, acede apenas à cópia local do processo intermediário, também através de memória partilhada. Todos os acessos aos pontos por parte do módulo resulta num acesso ao mapa local que reside no intermediário. O mapa local é apenas sincronizado com o mapa global quando o módulo chama a função do GMM encarregue de o fazer. De facto, esta é a uma terceira versão da função de sincronização já utilizada nos métodos local e partilhado, pelo que o código do módulo mantém-se inalterado qualquer que seja o método de acesso ao mapa global.

Esta terceira versão da função de sincronização é um pouco mais complexa que as primeiras duas, uma vez que esta executa dentro do processo que é o módulo isolado, e que por isso não tem acesso directo ao mapa global. Esta função limita-se por isso a enviar uma notificação de que foi invocada ao processo intermediário, o qual por sua vez encarrega-se de chamar a versão local desta mesma função de sincronização. O código que efectivamente faz a sincronização entre os dois mapas executa assim dentro do processo intermediário, o qual já tem acesso directo ao mapa global. A notificação da invocação da função de sincronização é de momento efectuada através do mecanismo de comunicação entre processos que é o 'sockets'. A opção de utilizar 'pipes' seria também plausível e correcta.

Uma vez que este método mantém uma cópia local, que tem de ser sincronizada com o mapa global, ele garante também uma semântica idêntica ao do método local. A diferença reside no facto de que o módulo nunca chega a mapear o mapa global na sua memória virtual, o que garante que nunca o poderá corromper.

No entanto espera-se que o tempo de execução da função de sincronização seja significativamente superior ao do método local, uma vez que necessita de comunicar a sua ocorrência através de um socket, e exige ainda duas trocas de contexto entre o processo que executa o módulo e o processo que executa o intermediário. É fundamentalmente por fornecer um serviço mais demorado que não se espera que este método venha a ser utilizado em instalações finais. O método está principalmente vocacionado para módulos que se encontram em fase de testes e de eliminação de erros de programação, ou módulos cujo código foi obtido através de terceiros e no qual não se deposita grande confiança.

Os três métodos de acesso ao GMM fornecem uma interface de programação idêntica, diferindo apenas nos pormenores semânticos que cada função implementa. Ou seja, recorreu-se ao polimorfismo para a implementação dos três

métodos de acesso. Embora a linguagem C, na qual foi escrito o GMM, não forneça mecanismos apropriados de suporte ao polimorfismo, esta técnica continua a ser possível de utilizar directamente na linguagem C.

Com o polimorfismo cada módulo que recorra às funções do GMM não necessita de nenhum código especial para escolher a versão que utiliza. Limita-se a chamar a função comum do API do GMM. O mesmo código do módulo pode assim funcionar com qualquer um dos métodos de acesso ao GMM. Isto permite que a escolha do método de acesso recaia sobre o instalador, em vez do programador do módulo. Ou seja, em vez de o método de acesso ao mapa global ficar programado directamente no código do módulo, o código do módulo passa a ser genérico. O módulo necessita apenas de, durante a sua fase de inicialização, de indicar à função de inicialização do GMM qual o método de acesso ao mapa global que pretende utilizar.

Tendo em conta que o MatPLC utiliza uma arquitectura de exo-núcleo, na qual o seu núcleo se resume a uma biblioteca de funções que é posteriormente ligada (em Inglês: linked) a cada módulo, uma alternativa de implementação do polimorfismo seria a de fornecer três versões diferentes da biblioteca de funções que constitui o GMM. Assim, cada módulo poderia ligar-se com a versão que pretendesse utilizar. No caso de ser utilizada ligação estática (static linking) esta opção resultará na desvantagem de produzir três programas executáveis para cada módulo. Por outro lado, no caso de ligação dinâmica (dynamic linking) será possível utilizar um só programa executável que, em função do método de acesso, faz a ligação durante a fase de inicialização do módulo, com a biblioteca de funções correspondente ao método escolhido.

A segunda alternativa de implementar o polimorfismo é o de recorrer a um conjunto de funções que servem apenas para encaminhar para a versão dessa função que corresponde ao método de acesso escolhido. Aqui é também possível fazer a escolha do método durante a fase de inicialização do módulo, altura em que a tabela de funções activas é criada e inicializada com os endereços das funções correspondentes ao método de acesso escolhido. De momento o MatPLC recorre a esta segunda alternativa, pois permite uma melhor organização das bibliotecas de funções. Ou seja, esta segunda alternativa, ao contrário da anterior, produz uma única biblioteca de funções, tornando-se assim mais fácil de gerir na altura de instalação do MatPLC no sistema.

Na versão local e isolada o MatPLC necessita de sincronizar duas zonas de memoria distintas. Esta operação é tipicamente executada periodicamente por todos os módulos, pelo que deverá ser executada o mais rapidamente possível por forma a diminuir o atraso que introduz na execução desses módulos. Desta forma, não foi descurada a possibilidade de escrever esta parte do código directamente em linguagem maquina, ultrapassando assim potenciais ineficiências introduzidas pelo compilador da linguagem de alto nível (neste caso o C).

Antes de mais, foi efectuada uma analise ao código produzido pelo compilador, e que se apresenta de seguida.

```
        .type   plc_update_local, @function
plc_update_local:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %edi
        pushl   %esi
        pushl   %ebx
        subl    $24, %esp
.LBB14:
        movl    mutex_, %eax
        pushl   %eax
        call    plcmutex_lock
.LBB15:
        xorl    %ecx, %ecx
        addl    $16, %esp
        cmpl    gmm_globalmap_len_, %ecx
        jae     .L91
        movl    gmm_privmap_, %esi
        movl    gmm_globmap_, %ebx
        movl    gmm_mapmask_, %edi
        .p2align 4,,7
.L89:
        movl    (%edi,%ecx,4), %eax
        movl    %eax, %edx
        andl    (%esi,%ecx,4), %eax
        xorl    $-1, %edx
        andl    (%ebx,%ecx,4), %edx
        orl     %eax, %edx
        movl    %edx, (%ebx,%ecx,4)
        movl    %edx, (%esi,%ecx,4)
        incl    %ecx
        cmpl    gmm_globalmap_len_, %ecx
        jb      .L89
.LBE15:
        subl    $12, %esp
        movl    mutex_, %ecx
        pushl   %ecx
        call    plcmutex_unlock
        leal    -12(%ebp), %esp
.LBE14:
        xorl    %eax, %eax
        popl    %ebx
        popl    %esi
        popl    %edi
        popl    %ebp
        ret
```

Como se pode verificar, o código mais demorado, que corresponde à execução do ciclo de sincronização, encontra-se precedido e seguido da chamada à função de bloqueio e desbloqueio do mutex que protege o acesso a essa memoria. A inicialização das variáveis utilizadas no ciclo é efectuado a partir da linha com o identificador '.LBB15', sendo que o ciclo em si encontra-se entre o identificador '.L89' e o salto para esse mesmo identificador na linha 'jb .L89'.

Analisando este código verifica-se que o próprio compilador consegue implementar o ciclo recorrendo apenas aos registos do CPU para guardar as variáveis que são necessárias dentro desse mesmo ciclo. Por outro lado, acede à memoria através de instruções maquina especializadas para aceder a variáveis de um array. Por exemplo a instrução 'movl %edx, (%ebx,%ecx,4)', que guarda o conteúdo do registo edx na zona de memoria com endereço obtido a partir da formula (ebx + ecx*4). Estas instruções resultam numa execução mais rápida do que o incremento de os três apontadores base de 4 em 4 em cada iteração, seguido de um acesso indirecto com instruções como 'movl %edx, (%ebx)'.

Verificas-se assim que a única optimização que é possível introduzir neste código será a substituição de uma contagem crescente do registo ecx para uma contagem decrescente (alteração essa que não afecta o algoritmo), permitindo assim utilizar a instrução 'loop .L89' em substituição das três instruções 'incl %ecx', 'cmpl gmm_globalmap_len_, %ecx' e 'jb .L89'.

Por forma a ter uma noção dos tempos de execução desta secção do código, com um impacto elevado nos tempos de execução finais dos módulos do MatPLC, foram efectuados vários testes. Os gráficos seguintes apresentam os resultados destes testes respeitantes aos tempos de execução do código responsável pela sincronização das zonas de memoria, e para dimensões crescentes da zona de memória a sincronizar. Estes tempos não consideram o tempo que o processo pode eventualmente ficar suspenso à espera que o mutex seja libertado, medindo apenas o tempo de execução do ciclo no qual é efectuada a sincronização da memoria. Por forma a evitar que o processo em causa seja interrompido a meio da sincronização por outro processo, e assim adulterar a medida dos tempos, os testes seguintes foram efectuados com o MatPLC a executar sobre o QNX 6.2.1, um sistema operativo tempo real, com o processo (que executa o módulo do MatPLC utilizado para as medidas) em causa com uma prioridade superior a qualquer outro processo. Foram efectuadas 512 medidas para cada dimensão da memória a sincronizar, sendo depois determinado o valor máximo, mínimo e médio utilizado para a sincronização. Estes testes foram efectuados num PC com um Pentium II a 350 Mhz, e cache nível 2 de 512 Kbytes, e 320 MBytes de memoria RAM.

Analisando os resultados verifica-se que o tempo necessário para fazer a sincronização das zonas de memória aumenta linearmente com a memória, excepto nas transições de 4 para 8 Kbytes, e de 128 para 256 Kbytes. Este aumento deve-se ao facto de as zonas de memória a sincronizar caberem completamente dentro da cache de memória nível 1 até 4 Kbytes, e na cache nível 2 até 128 Kbytes. O tempo necessário para a sincronização sofre aumentos não lineares quando passa a ser necessário recorrer à cache de memória de nível superior, ou à RAM em si, uma vez que os tempos de acesso à memória aumentam à medida que se passa para os níveis superiores.

Verifica-se ainda que a diferença entre o tempo mínimo e máximo aumenta em

valor absoluto com a dimensão da memória, mas reduz em valor relativo. Isto deve-se ao facto do valor máximo incluir não só o tempo de execução do ciclo de sincronização de memória em si, mas também o tratamento de qualquer interrupção que ocorra durante essa sincronização. Como o numero de interrupções que podem ocorrer aumenta à medida que o tempo de sincronização aumenta, a diferença absoluta entre o valor máximo e mínimo aumenta também com a memória a sincronizar.

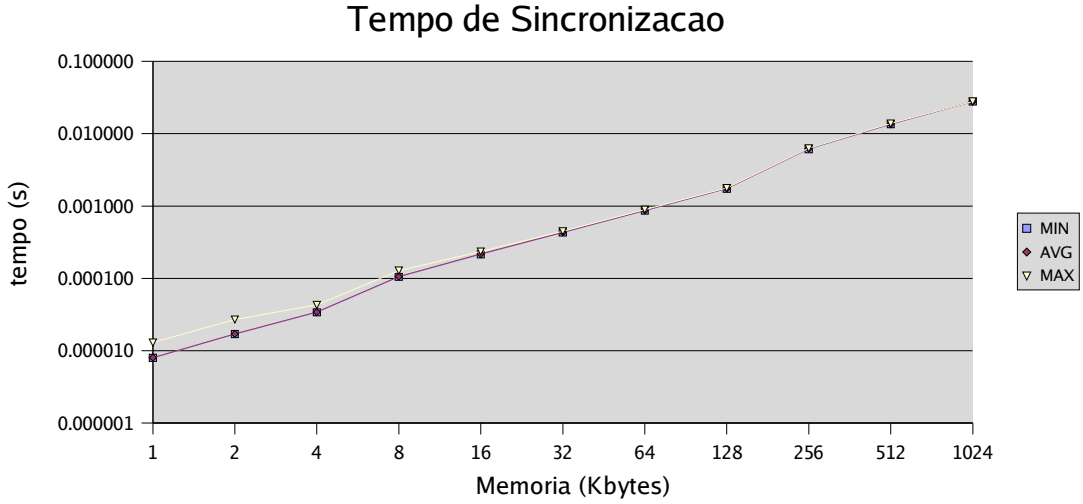

Figura 3.9 - Tempos necessários para efectuar a sincronização dos mapas globais local e partilhado, para valores crescentes de memória.

Memoria (Kbytes)	MIN	AVG	MAX
1	0.000008	0.000008	0.000013
2	0.000017	0.000017	0.000027
4	0.000034	0.000034	0.000043
8	0.000105	0.000106	0.000128
16	0.000215	0.000217	0.000235
32	0.000430	0.000434	0.000449
64	0.000861	0.000868	0.000892
128	0.001736	0.001744	0.001767
256	0.006092	0.006214	0.006266
512	0.013515	0.013538	0.013758
1024	0.027728	0.027757	0.027812

Figura 3.10 - Tempos necessários para efectuar a sincronização dos mapas globais local e partilhado, para valores crescentes de memória.

3.3.3 A Secção SYNCH

A secção synch fornece a opção de sincronizar a execução dos diversos módulos que compõem um MatPLC. Esta secção recorre ao formalismo das Redes de Petri de forma a simplificar a aprendizagem dos conceitos envolvidos na sincronização dos módulos, bem como para tirar proveito das capacidades de modelação das

mesmas redes. O instalador modela então a sincronização usando uma Rede de Petri que é depois implementada pela secção de sincronização.

O mapeamento entre a Rede de Petri e os módulos do MatPLC faz-se através de pontos de sincronização. A cada transição da Rede de Petri fica associada, de forma automática, um ponto de sincronização. Os módulos do MatPLC, quando desejam esperar num ponto de sincronização, invocam uma função disponibilizada pela secção synch. Esta função bloqueia até que o ponto de sincronização seja disparado, altura em que a função retorna. O ponto de sincronização dispara na mesma altura em que a transição da Rede de Petri à qual está associada também dispara.

Tal como os pontos normais do MatPLC, disponibilizados pela secção GMM, os pontos de sincronização são também eles identificados pelo seu nome. Os pontos de sincronização associados a transições herdam o mesmo nome da transição. Um módulo que deseje sincronizar com um ponto de sincronização deve inicializar primeiro uma variável que referencia o respectivo ponto de sincronização, acção para a qual a secção synch disponibiliza uma função que necessita como único parâmetro o nome do ponto de sincronização.

Os módulos estão livres de decidirem em que fases das suas acções e em que local do seu código irão sincronizar com um determinado ponto de sincronização. Dependendo do algoritmo que implementam, módulos diferentes irão sincronizar em alturas diferentes, podendo até nem utilizar os pontos de sincronização. Prevê-se que módulos que necessitem de sincronizar em alturas específicas do seu algoritmo permitam ao instalador configurar qual o ponto de sincronização que deverá ser utilizado em cada local de sincronização. Dá-se assim inteira liberdade de configuração ao instalador, sem que seja necessário alterar qualquer código do módulo. Adicionalmente, qualquer ponto de sincronização poderá ter quaisquer condições de disparo a ela associada, bastando para tal configurar adequadamente a Rede de Petri.

Apesar do elevado potencial de configuração da sincronização dos módulos, deve ser tomado em conta que a grande maioria dos módulos irão executar algoritmos que recorrem ao método de programação assíncrona, no qual o programa executa o mesmo código ciclicamente. Todas as acções desse código são assíncronas, ou seja, nunca bloqueiam à espera da ocorrência de um evento. É este o método tradicional de programação de PLCs, pelo que se depreende também se espera que este seja o método habitual utilizado nos programas executados pelo MatPLC.

Assim sendo, espera-se que os programas assíncronos não irão necessitar dos pontos de sincronização de forma explícita. No entanto, pode se dar o caso do instalador desejar que vários programas, cada um no seu módulo do MatPLC, executem de forma síncrona, i.e. cada um executa uma iteração do seu ciclo infinito, sendo que a iteração seguinte só executa depois de terminada a iteração

anterior em todos os módulos. Para tal será necessário que todos os módulos assíncronos se sincronizem com dois pontos de sincronização, um no inicio da iteração, e outro no final da mesma.

De forma a facilitar a programação destes módulos, a secção synch disponibiliza duas funções que deverão ser chamadas uma no início, e a outra no final, de cada iteração. Estas funções irão sincronizar com um ponto de sincronização, sendo que o nome do ponto de sincronização a ser utilizado em cada local é pré-definido à partida como sendo igual ao nome do próprio módulo acrescido dos sufixos '.beg' e '.end' para os locais de sincronização no inicio e fim de cada iteração respectivamente.

Um programador de um módulo com funcionamento assíncrono necessita assim apenas de chamar as duas funções disponibilizadas pela secção synch para que de forma automática disponibilize ao instalador dois pontos de sincronização. O instalador por sua vez necessita apenas de modelar uma Rede de Petri que defina a sincronização desejada, e dar os nomes apropriados às transições (ao qual ficam associados os pontos de sincronização) de forma a que cada módulo sincronize no local próprio.

Os arcos da Rede de Petri podem ter pesos superiores a um, sendo ainda admitidos arcos nulos, i.e. arcos com peso zero, que só permitem o disparo da transição na qual terminam se a posição de origem do arco se encontrar vazio de testemunhos. Estas funcionalidades permitem já modelar sequências de sincronização complexas, no entanto, e com o intuito de permitir ainda mais liberdade, os pontos de sincronização e as transições são considerados entidades distintas, muito embora exista um mapeamento automático entre cada transição e um ponto de sincronização. Esta separação permitirá que no futuro os pontos de sincronização possam vir a ser associados a outros eventos também relacionados com a Rede de Petri. Por exemplo, um ponto de sincronização poderia ser associado à existência exacta de um determinado numero de testemunhos numa posição definida. Qualquer modulo que se sincronize com um destes pontos de sincronização bloquearia enquanto a posição não tivesse o numero exacto de testemunhos, no entanto o fim do bloqueio do módulo não resultaria em nenhuma alteração ao numero de tokens na posição. Este género de funcionamento é muito difícil de replicar recorrendo apenas às posições e arcos que são agora permitidos, no entanto não é ainda suportada pelo MatPLC uma vez que ainda não se vieram a mostrar necessários.

As regras de evolução da Rede de Petri utilizada para a sincronização segue as regras normais das Redes de Petri, incluindo a não permissão de disparos simultâneos de duas transições. No entanto, e devido ao método na qual é implementada, diverge das regras habituais num pormenor: uma transição só é disparada se existe um módulo à espera no ponto de sincronização associado a essa transição. Ou seja, mesmo que o numero de testemunhos nas posições anteriores a uma transição permitam o disparo dessa transição, esta só será

disparada logo que um módulo se tente sincronizar no ponto de sincronização associado a essa transição. Pelo contrário, caso as regras de evolução da Rede de Petri fossem seguidas, essa transição seria disparada logo que o numero de testemunhos o permitisse. Caso consideremos esta Rede de Petri como sendo interpretada, com condições associadas ao disparo de cada transição na qual só permitem o seu disparo quando um módulo se encontra a tentar sincronizar com essa mesma transição, então não é errado dizer que a Rede de Petri de sincronização segue todas as regras de evolução das habituais Redes de Petri.

A configuração da Rede de Petri por parte do instalador passa então por definir a estrutura da Rede de Petri, o que envolve declarar as posições, as transições, e todos os arcos da Rede de Petri com os respectivos pesos. Os nomes das posições e das transições têm ainda de ser definidas.

Para as configurações mais habituais, em especial nos casos em que se deseja que os módulos se limitem a executar síncronamente, um após o outro, foi previsto ainda um método de configuração mais expedito que consiste apenas em definir quais os módulos que deverão ser lançados logo que termine a execução do ciclo de outro modulo. Neste caso, o próprio MatPLC irá gerar a Rede de Petri necessária para produzir este efeito de forma automática.

A secção synch recorre a mecanismos de sincronização inter-processos para implementar as regras de evolução da Rede de Petri. De facto, de momento existem três implementações alternativas, uma baseada nos semáforos disponibilizados pelos sistemas Unix SysV, e outras duas baseadas nos semáforos, mutexes e variáveis de condição disponibilizados por sistemas operativos que seguem as normas POSIX e as suas extensões para sistemas de tempo-real.

Os semáforos SysV permitem uma implementação mais simples, uma vez que estes são sempre criados em grupos. A sincronização pode ser efectuada sobre um ou mais semáforos em simultâneo, desde que estes pertençam ao mesmo grupo. Desta forma, a Rede de Petri de sincronização é mapeada num grupo de semáforos SysV, com um semáforo por cada posição da Rede de Petri. O valor armazenado no contador de cada semáforo do grupo corresponde ao numero de testemunhos existentes em determinado instante numa posição da Rede de Petri. Este grupo de semáforos é criado e inicializado durante a fase de inicialização do próprio MatPLC. O disparo de uma transição é implementada pela sincronização simultânea em todos os semáforos correspondentes às posições que veriam o número de testemunhos alterados pelo disparo dessa transição. Esta sincronização simultânea inclui tanto a redução de testemunhos das posições a montante, bem como a inserção de testemunhos nas posições a jusante da transição. O próprio semáforo SysV garante que toda a operação é atómica, e só será realizada quando o numero de testemunhos a serem retirados das posições a montante são suficientes para permitir o disparo da transição. A própria semântica dos semáforos SysV permite que sejam implementados os arcos nulos

(de peso zero) da mesma forma.

Uma vez que nem todos os sistemas operativos fornecem os semáforos SysV, foi implementada ainda uma segunda versão da secção synch baseada nas funcionalidades definidas pelo POSIX, incluindo as suas extensões para sistemas de tempo real. Esta versão é bastante mais complexa que a primeira uma vez que a semântica dos semáforos definidos pelo POSIX é bastante mais simples comparada com os semáforos SysV. Ou seja, não é possível efectuar o mapeamento directo da Rede de Petri de sincronização em semáforos POSIX uma vez que estes não podem ser agrupados, não sendo por isso possível a sincronização simultânea e atómica em mais do que um semáforo POSIX.

Por este motivo esta versão baseia-se apenas em mutexes e variáveis de condição. No entanto, e devido ao facto de os módulos do MatPLC executarem cada um no seu próprio processo, esta implementação exige que os mutexes possam ser acedidos de vários processos em simultâneo. Uma vez que o POSIX define esta característica dos mutexes como sendo opcional, não é garantido que o MatPLC possa ser portado como está para qualquer sistema operativo POSIX.

Esta versão da secção synch recorre então a um único mutex, e a uma variável de condição para cada módulo. Utiliza ainda um 'array' de inteiros para contabilizar o numero de testemunhos em cada posição da Rede de Petri. Todos os acessos a este array estão protegidos pelo mutex global.

A sincronização com um ponto de sincronização por parte de um módulo consiste então em primeiro obter o mutex, seguido das alterações no 'array' provocadas pelo disparo dessa transição, e por ultimo a libertação do mutex. No entanto, e ainda antes de ser libertado o mutex, é ainda verificado se existe alguma transição que possa ser disparada devido às alterações introduzidas pelo disparo inicial. Nesta fase apenas são testadas as transições nas quais se encontra um ou mais módulos bloqueados. Caso alguma transição possa ser disparada, as alterações devidas ao seu disparo são introduzidas no 'array', e é indicado ao módulo que este pode prosseguir a sua execução. Apenas quando se verifica que mais nenhuma transição pode ser disparada é então libertado o mutex global e o módulo original que solicitou a sincronização com o ponto de sincronização da primeira transição que foi disparada é permitido prosseguir a sua execução.

Quando a sincronização com um ponto de sincronização exige que o módulo fique bloqueado (devido à falta de testemunhos nas posições a montante da transição associada a esse ponto de sincronização), a função suspende a sua acção na variável de condição do módulo que deseja a sincronização, e em simultâneo liberta o mutex global. Fica bloqueado na variável de condição até que um outro módulo, após o disparo de uma outra transição, verifica que o primeiro módulo já pode continuar, utilizando para isso o algoritmo atrás descrito. Neste caso o segundo módulo irá indicar ao primeiro que pode prosseguir enviando um sinal à variável de condição na qual se encontra

bloqueado o primeiro módulo.

Uma vez que as variáveis de condição utilizam o mesmo mutex global que é utilizado para proteger os acessos ao array, todos os módulos que podem continuar a sua execução são libertados em simultâneo quando o mutex é libertado. Caberá então ao sistema operativo escolher qual dos módulos irá ocupar o CPU e continuar a sua execução baseando-se na prioridade de cada módulo.

A terceira versão recorre apenas a semáforos POSIX, e simultaneamente assemelha-se e partilha o código da versão anterior. Neste caso o mutex é substituído por um semáforo para proteger o acesso ao 'array' que mantém o numero de testemunhos em cada posição da Rede de Petri. As variáveis de condição são substituídas por um semáforo por cada módulo que se sincroniza com a Rede de Petri. Um módulo, que necessite de ser suspenso ao tentar sincronizar com um ponto de sincronização que não se encontra em condições de permitir a continuação da execução desse módulo, irá suspender no semáforo, até que a condição de evolução se verifique e o semáforo seja assinalado para permitir a continuação da execução. Neste caso, e uma vez que não é possível assinalar mais do que um semáforo POSIX de cada vez, não é possível também a libertação de vários módulos em simultâneo para assim deixar ao sistema operativo a responsabilidade de decidir qual o módulo que deverá ocupar o CPU de seguida. Assim, cumpre agora ao código da secção synch garantir a libertação dos módulos, através do sinal ao respectivo semáforo, pela ordem decrescente da prioridade dos mesmos.

Uma vez que o MatPLC recorre a uma arquitectura tipo exo-núcleo, tanto a segunda como a terceira versões baseiam-se na cooperação dos vários módulos envolvidos. Cabe a um módulo que se sincroniza com sucesso num ponto de sincronização libertar todos os outros módulos que deverão ser libertados em consequência da sua própria sincronização. Se esta libertação não ocorrer, as condições de sincronização entre os módulos deixam de respeitar as condições estabelecidas pelo instalador ao definir a Rede de Petri de sincronização. Estas duas versões POSIX sofrem portanto do perigo real de um módulo terminar a sua execução intempestivamente enquanto este mantém bloqueado o semáforo global de protecção do 'array', ou seja, durante a fase crucial de actualização do 'array' e da libertação dos módulos que deverão ser libertados. Neste caso o estado interno do MatPLC que gere a sincronização dos processos pode ficar num estado inconsistente, provocando a dispersão do erro por outros módulos que também recorrem ao serviços fornecidos pela secção synch. Torna-se muito difícil proteger contra este erro, sendo a única alternativa proteger contra a própria ocorrência do evento que provocou a morte súbita e intempestiva do módulo. Tendo em conta que esta morte pode ter como origem o próprio operador que decide matar o processo, a gestão das permissões dadas a cada operador deverão ser cuidadosamente estudadas em função da capacidade deste

compreender as consequenciais de efectuar cada operação sobre o MatPLC, incluindo matar processos de forma descoordenada.

3.3.4 A Secção STATE

A secção State gere o estado de funcionamento do MatPLC em geral, e de cada módulo que executa um programa assíncrono, ou seja que executa um ciclo infinito. Esta secção permite que cada módulo esteja em um de dois estados: HALT e RUN. Por omissão, logo após um módulo terminar a sua inicialização, este entra para o modo RUN.

O estado RUN permite que a execução do ciclo infinito continue normalmente. O estado HALT obriga o módulo a parar a execução do ciclo infinito. A paragem de um ciclo ocorre apenas imediatamente antes do início de um ciclo, e nunca a meio. Isto permite que o módulo termine a execução de um ciclo se a passagem de estado de RUN para HALT ocorra durante a execução do ciclo.

A execução de um ciclo pode assim ser bloqueada tanto pela secção synch tanto pela secção state. Por este motivo, torna-se crucial efectuar uma sincronização correcta da gestão do estado de funcionamento do módulo, com a gestão efectuada pela secção synch na qual é gerida a sincronização do módulo com a Rede de Petri.

De facto, não é correcto verificar primeiro se um módulo pode executar o ciclo tendo em conta o seu estado, para só depois testar se as condições exigidas pela sincronização de módulos se verificam. Isto porque se o módulo estiver no estado RUN, mas não puder continuar a execução devido às condições de sincronização com outros módulos, este irá ficar bloqueado apenas no segundo teste. Se enquanto estiver aqui bloqueado o seu estado se alterar para HALT, este novo estado não será respeitado logo que as condições de sincronização permitam a continuação da execução do módulo, pois o próximo teste ao seu estado só será efectuado no inicio do ciclo seguinte.

No entanto, o contrário também está errado. Ou seja, também não está correcto testar primeiro as condições de sincronização para só depois testar o seu estado. Isto porque pode vir a ocorrer que dois módulos fiquem à espera num mesmo ponto de sincronização em simultâneo, sendo que só um poderá ser libertado quando a condição associada ao ponto de sincronização se verificar. No entanto, se o módulo escolhido para avançar estiver no estado HALT, então ele ficará retido no segundo teste, bloqueando assim o outro módulo que poderia estar no estado RUN e assim a executar o seu ciclo.

Conclui-se então que a única solução correcta será a de verificar em simultâneo e de forma atómica se as suas duas condições (o estado e a sincronização) se verificam. Para tal, a secção state recorre aos recursos providenciados pela secção synch para implementar o seu serviço. Ou seja, a secção state acrescenta uma posição à Rede de Petri de sincronização por cada módulo que seja inicializado.

Esta posição conterá um testemunho quando o modulo está no estado RUN, e vazia quando estiver em HALT. Adicionalmente, esta secção acrescenta dois arcos de peso unitário à Rede de Petri tal que tenham origem e destino na posição que representa o estado do módulo, e a transição associada ao ponto de transição utilizada pelo módulo para sincronizar o inicio de cada ciclo. Estes arcos garantem que a transição associada ao inicio do ciclo só possa disparar quando a posição que representa o estado do módulo contem um ou mais testemunhos, e ainda que o numero de testemunhos dessa posição não sofra alterações com o disparo da própria transição.

Consegue-se assim que o teste do estado seja efectuado de forma simultânea com o teste da sincronização, e re-utilizando os serviços já implementados pela secção synch.

A par do estado de cada módulo, a secção synch gere ainda o estado de funcionamento do MatPLC em geral. À semelhança dos módulos, o MatPLC também poderá estar em um de dois estados: RUN e HALT. No estado HALT todos os módulos param a sua execução. No estado RUN apenas os módulos cujo estado também é RUN executam os ciclos, enquanto que os no estado HALT permanecem parados.

Este estado é também ele implementado com recurso à Rede de Petri de sincronização, pelos mesmos motivos atrás mencionados. Neste caso uma única posição é acrescentada à Rede de Petri para representar o estado do MatPLC. Por cada módulo activo, são ainda acrescentados um par de arcos com origem e destino na posição referida, e a transição associada ao ponto de sincronização utilizado para o controlo do inicio do ciclo. Estes arcos utilizam a técnica também utilizada para gerir o estado individual de cada módulo.

Está ainda previsto que esta secção venha ainda a ser responsável pela gestão de troca de configurações de um módulo, enquanto este continua a sua execução (i.e. reconfiguração on-line). Para tal considera-se provável que os módulos sigam o algoritmo de ler as suas configurações aquando da sua inicialização, sendo estes dados depois guardados em variáveis de estado internas ao processo que executa o módulo de forma a poderem ser acedidas de forma rápida durante a sua execução.

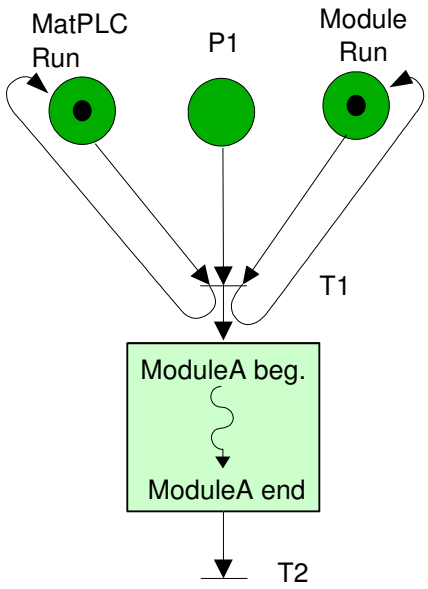

Figura 3.11 - Estados internos do MatPLC e do Módulo são implementados com recurso à Rede de Petri da secção de sincronização.

Existem várias alternativas para permitir a alteração do conteúdo destas variáveis, e assim alterar a configuração do módulo. A primeira reside no módulo guardar estas variáveis em memória partilhada de tal forma que possam ser acedidas por outros processos, e assim alteradas. Esta opção tem a desvantagem que torna-se assim necessário sincronizar o acesso a estas variáveis, visto poderem ser acedidas por mais do um processo concorrente. Uma opção seria a de só permitir a alteração destas variáveis por processos estranhos apenas durante a fase em que o módulo não está a executar o ciclo, ou seja, entre o fim de um ciclo e o inicio do próximo. No entanto mesmo esta variante sofre ainda de outros inconvenientes, em especial o facto de ser difícil garantir que o módulo esteja entre duas execuções do ciclo. De facto, não é possível recorrer ao estado do módulo para o colocar em HALT temporariamente enquanto se fazem alterações às variáveis visto não ser suficiente colocar o módulo em HALT para garantir que este pare imediatamente, sendo este estado apenas respeitado no inicio do próximo ciclo. Por outro lado, torna-se necessário criar um programa auxiliar a cada módulo que saiba qual a estrutura interna das variáveis utilizadas por esse módulo para guardar o estado da sua configuração.

Uma segunda alternativa, à primeira vista mais simples, reside em lançar um segundo processo, que passe novamente pela fase de inicialização, e que portanto leia novamente a nova configuração para que esta seja guardada nas suas próprias variáveis internas. Neste caso, o processo anterior necessita de ser eliminado para que o novo processo, com as configurações novas, tome o seu lugar. Torna-se assim necessário gerir a transferência de responsabilidades entre os dois processos, de forma que o processo que vai ser substituído só seja

eliminado após o novo processo ter terminado toda a sua fase de inicialização que poderá ser demorada. Garante-se assim que a transição entre um processo e o outro não introduza interferências na execução normal do MatPLC.

A gestão da transferência de responsabilidades entre os dois processos torna necessário um esquema no qual o novo processo indica ao processo existente que este ultimo irá ser substituído, e fica depois à espera de autorização para avançar. Várias alternativas podem ser utilizadas para a estas duas comunicações e sincronizações entre os dois processos, no entanto a mais simples recorre a uma variável partilhada entre os dois, e ainda a um semáforo. A variável partilhada encontra-se num bloco de memória alocado ao CMM, e contem sempre um valor 0. A versão nova do processo, após o término da sua fase de inicialização, altera o valor da variável partilhada. Esta acção indica ao processo existente que ele deverá terminar a sua execução pois vai ser substituído. O segundo processo fica então bloqueado no semáforo à espera de autorização para continuar. O processo original, ao verificar que a variável já não tem o valor zero, simplesmente termina a sua própria execução logo após ter autorizado o novo processo a continuar assinalando o semáforo.

Faz sentido efectuar a troca entre os dois processos em duas alturas distintas da execução do ciclo infinito: antes mesmo de verificar se o inicio de ciclo poderá continuar devido ao estado e condições de sincronização entre módulos, ou logo após a verificação das condições anteriores. Considerando que muitos módulos irão passar a maioria do seu tempo à espera de autorização para iniciar o ciclo, a primeira tem a desvantagem de que qualquer alteração só irá sofrer efeitos após o término de um ciclo completo. No entanto, a segunda versão tem o inconveniente de introduzir uma troca de contexto entre dois processos numa altura em que o ciclo deverá iniciar a sua execução. Esta segunda desvantagem irá ser principalmente importante em sistemas Tempo-Real, nos quais se pretende que sejam cumpridos prazos de execução. Nestes casos uma análise aos processos existentes e às suas características (tempos de execução, periodicidade, etc.) permite obter garantias que os prazos irão ser cumpridos, no entanto a introdução de uma mudança de contexto adicional poderá falsificar os pressupostos nos quais se baseiam os cálculos anteriores. A melhor alternativa será a de optar por permitir ao instalador decidir o instante que deverá ser efectuada a troca, utilizando a primeira opção se o instalador não indicar qualquer preferência.

Para que um módulo possa recorrer a este mecanismo de actualização de parâmetros, este deverá no entanto respeitar uma condição fundamental; não poderá armazenar o seu estado de funcionamento em variáveis internas e privadas ao processo. Como alternativa, o estado deverá ser armazenado em memória partilhada que continuará a existir e manter os valores mesmo após o processo terminar a sua execução. Ficam assim disponíveis para serem utilizadas pelo novo processo que tomará o seu lugar.

Esta variante apresenta ainda a séria desvantagem relacionada com a gestão do acesso a hardware por parte de módulos que fazem a gestão da interface com o meio físico. Neste caso, os módulos necessitam de aceder a hardware, sendo provável que este só poderá ser controlado por um processo de cada vez. No caso da troca de um processo por outro, o segundo processo só poderá ter acesso ao hardware, e assim terminar a sua fase de inicialização, após o primeiro ter libertado esse mesmo hardware, o que só ocorrerá na fase de terminação do primeiro processo. No entanto, o primeiro processo só liberta o hardware depois de o segundo estar pronto a assumir as suas responsabilidades, ou seja, só depois do segundo processo terminar a sua inicialização. Torna-se assim difícil fazer a gestão do controlo de acesso ao hardware durante a troca de um processo por outro. É também por este motivo que o suporte da troca de configurações não foi ainda implementado dentro da secção state, sendo provável que a sua implementação seguirá então a primeira opção atrás delineada, e mesmo assim só após a implementação de uma nova secção que assumirá a responsabilidade de gerir a transferência de mensagens assíncronas entre os módulos do MatPLC, as quais poderão ser utilizadas para transferir a informação necessária para actualizar os parâmetros de configuração de um módulo de forma assíncrona.

3.3.5 A Secção PERIOD

A secção period tem como objectivo controlar a periodicidade de execução de cada ciclo dos módulos assíncronos. Cabe ao instalador simplesmente configurar qual a periodicidade desejada para cada módulo, de forma independente. Cada módulo gere assim a periodicidade da sua própria execução, podendo por isso cada módulo ter um período de execução distinto.

No entanto, os módulos cuja execução se encontra também sincronizada pela secção synch poderão sofrer interferências adicionais pelo que poderá não ser respeitado o período de execução desejado. De facto, e uma vez que o controlo do período de execução é efectuado por cada módulo, se dois ou mais módulos estiverem configurados para executarem os seus ciclos de forma sincronizada, um após o outro, o período de execução efectivo será ditado pelo maior período de execução do ciclo de entre todos os módulos cuja execução se encontra sincronizada.

A gestão do período de execução é efectuada ainda antes da verificação das condições de sincronização. Ou seja, quando um módulo se mostra disponível para iniciar novo ciclo de execução, cabe primeiro à secção period verificar se as condições de periodicidade se verificam. Só depois desta secção permitir a execução do ciclo é que é permitida à secção synch verificar as condições de sincronização.

A gestão do período é implementado com recurso a um temporizador POSIX, um mutex e ainda a uma variável de condição. O temporizador é criado durante

a fase de inicialização do módulo, mas apenas inicia a sua contagem no inicio da execução do primeiro ciclo. O temporizador é configurado para gerar um sinal periodicamente, com período idêntico ao período de execução do ciclo desejado. Aquando a ocorrência do sinal, uma rotina de resposta ao sinal limita-se a incrementar um contador de ciclos em atraso. Esta variável é verificada no inicio de cada ciclo para testar da ocorrência de um sinal indicando que o tempo de inicio do próximo ciclo já tenha expirado. Assim, e caso a variável seja superior a 0, esta é decrementada, sendo então permitido ao módulo continuar a sua execução (que neste caso será permitir à secção synch verificar as condições de inicio de um ciclo). Pelo contrário, caso o módulo se encontre adiantado, e assim ainda não ter passado o instante do inicio de novo ciclo quando o teste à variável de ciclos em atraso é efectuado (ou seja, esta mesma variável apresenta-se com o valor zero), o módulo fica suspenso numa variável de condição. O módulo é posteriormente libertado pela rotina de resposta ao sinal periódico, sendo que esta assinala a variável de condição caso o contador de ciclos em atraso estivesse em zero. Em todos os casos, o acesso à variável de ciclos em atraso encontra-se protegida por um mutex.

Esta implementação com recurso a temporizador, variável de condição, mutex e ainda a uma rotina de tratamento do sinal torna-se necessária para garantir uma execução periódica correcta. De facto, foi posta de parte a hipótese de simplesmente verificar as horas no inicio de cada ciclo para verificar se seria necessário esperar ou não. Com esta versão, se fosse necessário esperar o módulo seria suspenso por um intervalo de tempo idêntico ao tempo necessário até ao início do próximo ciclo. Infelizmente esta versão, embora muito mais simples de implementar, sofre de uma condição de corrida ('race condition'), entre a chamada ao sistema para verificar as horas actuais, e a chamada ao sistema para suspender por um intervalo de tempo definido. Se entre estas duas chamadas o processo fosse retirado do CPU pelo escalonador do sistema operativo, o módulo acabaria por chamar a função de suspensão muito mais tarde, mas já sem ter oportunidade de corrigir o intervalo de tempo durante o qual seria suspenso.

Puder-se-ia considerar este tempo de suspensão como fazendo parte do tempo de resposta (WCRT) do ciclo que se inicia logo após a suspensão temporizada, sendo por isso contabilizado como interferência de processos de prioridade mais elevada. Apesar disto, este raciocínio invalidaria os pressupostos nos quais se baseiam as equações regularmente utilizadas para o calculo do WCRT de tarefas com restrições temporais. De facto as equações são apenas válidas se a interferência ocorre a partir do instante em que o processo é libertado. Adicionalmente, o processo ao suspender a sua execução durante os primeiros instantes em que deveria estar a ocupar o CPU pode ser considerado como uma suspensão voluntária de execução do próprio processo, o que também não está contemplado no modelo de processos para as quais as equações são válidas.

Devido às razões expostas, foi decidido recorrer à implementação baseada em

temporizadores, mutexes e variáveis de condição para a secção period garantir a periodicidade de execução dos ciclos de cada módulo.

Por forma a validar a correcta periodicidade de execução, foram efectuados vários testes baseados sempre numa mesma configuração do MatPLC, no qual um módulo executa um ciclo infinito de forma periódica. Os testes consistem na leitura de um relógio de cada vez que o módulo inicia o seu ciclo periódico, e a posterior armazenagem desses instantes para memoria. Apenas no final do teste, e com o intuito de não introduzir atrasos extras na normal execução do módulo, os valores são escritos para ficheiro em disco para posterior tratamento e análise. O relógio utilizado para medir os tempos foi o contador de ciclos de relógio existente nos CPUs Pentium da Intel e em todos os outros CPUs compatíveis com estes. Este contador de 64 bits, implementado em hardware dentro do próprio CPU, é inicializado a zero no instante em que o computador é ligado, e a partir desse momento é incrementado em cada impulso da entrada de relógio do CPU. Este contador serve então de relógio tempo real, com uma resolução elevada, e que depende apenas da frequência de relógio do CPU. Para um CPU a 100 Mhz, este relógio apresenta uma resolução de 0,01 us. A leitura deste relógio faz-se com numa única instrução máquina, sendo ela por conseguinte atómica e indivisível.

O teste recorre ao exemplo mais básico fornecido com o MatPLC, que consiste num módulo de leitura de teclado, outro módulo de lógica que se limita a acender uma série de quatro 'luzes' em sequência, e um módulo de interface que copia o estado das luzes para o terminal de texto. Os três módulos são configurados para executar periodicamente com periodicidade de 50 ms, sem qualquer sincronização entre eles. O módulo de lógica é o único que não necessita de aceder a dispositivos de hardware, logo, ao contrário dos restantes deste exemplo, não irá sofrer atrasos devido ao acesso a dispositivos de entrada e saída (que neste caso são o teclado e o monitor). Por esta razão apresentam-se apenas as medidas da periodicidade do módulo de lógica.

Em todos os testes foram efectuadas 64k leituras, mas devido a limitações do software de geração de gráficos são apenas mostradas as primeiras 25 mil leituras. A cada leitura, uma por cada período, foi ainda subtraído o valor ideal em que se deveria ter iniciado a execução, pelo que os valores apresentados representam na realidade os atrasos do inicio de execução de cada período em relação ao ideal, mais conhecido por 'jitter'.

O primeiro teste foi efectuado no computador caravela, um computador Pentium III a 730Mhz, com 512Mbytes de RAM, a executar o núcleo Linux 2.4.22. Outros testes foram ainda efectuados na maquina macau, um computador Pentium II a 350MHz, com 320 MBytes de RAM, a executar o QNX 6.2.0. O QNX é um sistema operativo tempo real baseado numa arquitectura micro-núcleo, mas que dispõe também um ambiente gráfico próprio com programas para ler emails, navegar na Internet (browsers – protocolo http e formato html),

e claro editores de texto para programação.

Os primeiros testes no caravela foram efectuados com o MatPLC a executar em simultâneo com o ambiente gráfico (X e KDE), e com a maquina ligada à rede da faculdade de engenharia. No entanto a maquina não se encontrava a ser utilizada durante o teste por nenhum utilizador, embora outros processos (por exemplo editores de texto, browsers, leitores de email) se encontravam abertos. A maioria destes processos encontravam-se inactivos (i.e. ou seja nunca ocupavam o CPU), no entanto alguns deles (por exemplo leitura periódica de email do servidor SMTP) executavam periodicamente, tal como o eram os processos normais de gestão de um ambiente Linux (por exemplo 'cron').

Os testes seguintes seguiram a mesma configuração, mas agora com um outro processo a executar em paralelo a gerar acessos contínuos ao disco (um comando 'grep' do Unix a pesquisar todos os ficheiros no disco por uma determinada sequência de bytes), e ainda com a maquina a responder a pings contínuos ('ping flood') gerados por outra maquina ligada à mesma rede. Esta configuração permite testar o MatPLC a executar em simultâneo com outras aplicações que, para alem de ocuparem o CPU, provocam a ocorrência de interrupções ligadas à actividade do disco e da carta de rede. Durante estes testes a maquina apresentava um 'load average' de aproximadamente 1, com o CPU geralmente aproximadamente livre em 30% do tempo. Estas ultimas medidas foram efectuadas com recurso ao programa 'top' do Unix, com amostragem de valores periódicas cada segundo.

A terceira configuração utiliza o caravela fisicamente desligado da rede e em modo de utilizador único, no qual executa apenas o MatPLC e as threads indispensáveis ao funcionamento correcto do núcleo Linux.

Os testes no macau, com sistema operativo QNX, foram efectuados com o ambiente gráfico aberto e com algumas aplicações abertas, mas mais uma vez sem intervenção do utilizador durante o teste.

Para cada teste são apresentados dois gráficos, um com as primeiras 25000 iterações, e o segundo apenas com as primeiras 2500 iterações, sendo que o segundo permite utilizar uma escala menor na escala do tempo, pelo que se pode assim observar com mais pormenor os efeitos de cada teste. Adicionalmente, o teste com ambiente gráfico aberto mas sem utilização intensiva na maquina caravela foi repetido com 10 instâncias do módulo de lógica a executar em simultâneo, todos eles com o mesmo período de 50 ms.

Na escala vertical dos gráficos seguintes, ou seja a escala do tempo, é apresentado o jitter no inicio de execução de cada iteração do módulo, em relação ao instante ideal.

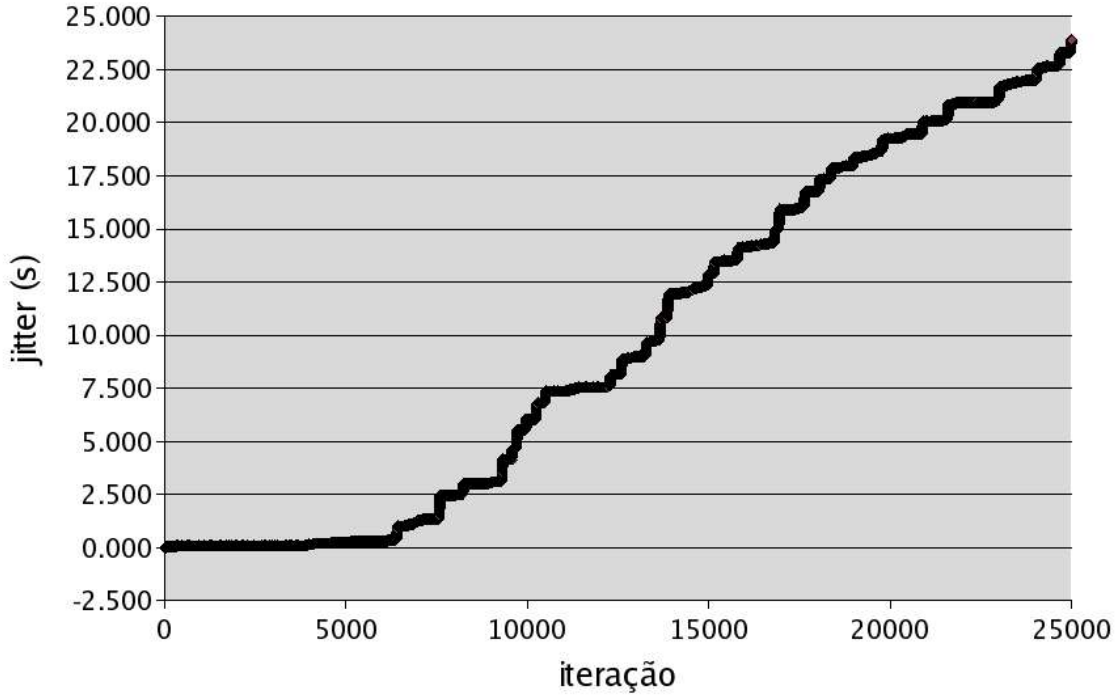

Figura 3.12 - Teste 1 - Período 50 ms, máquina caravela com ambiente gráfico em funcionamento.

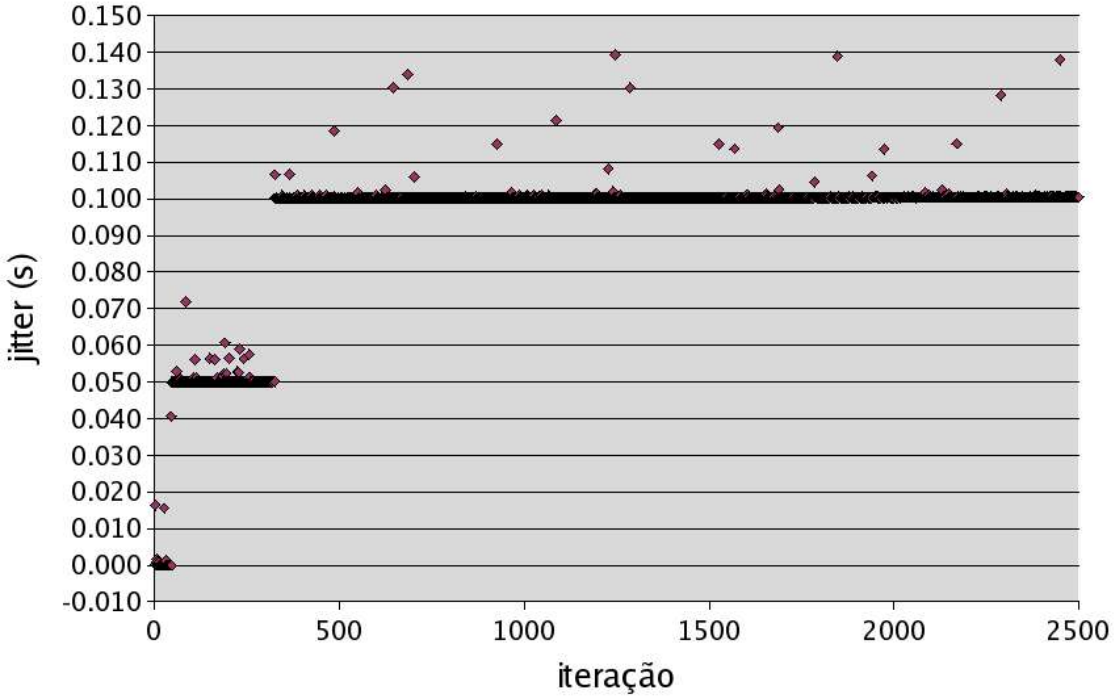

Figura 3.13 - Teste 1 - Período 50 ms, máquina caravela com ambiente gráfico em funcionamento.

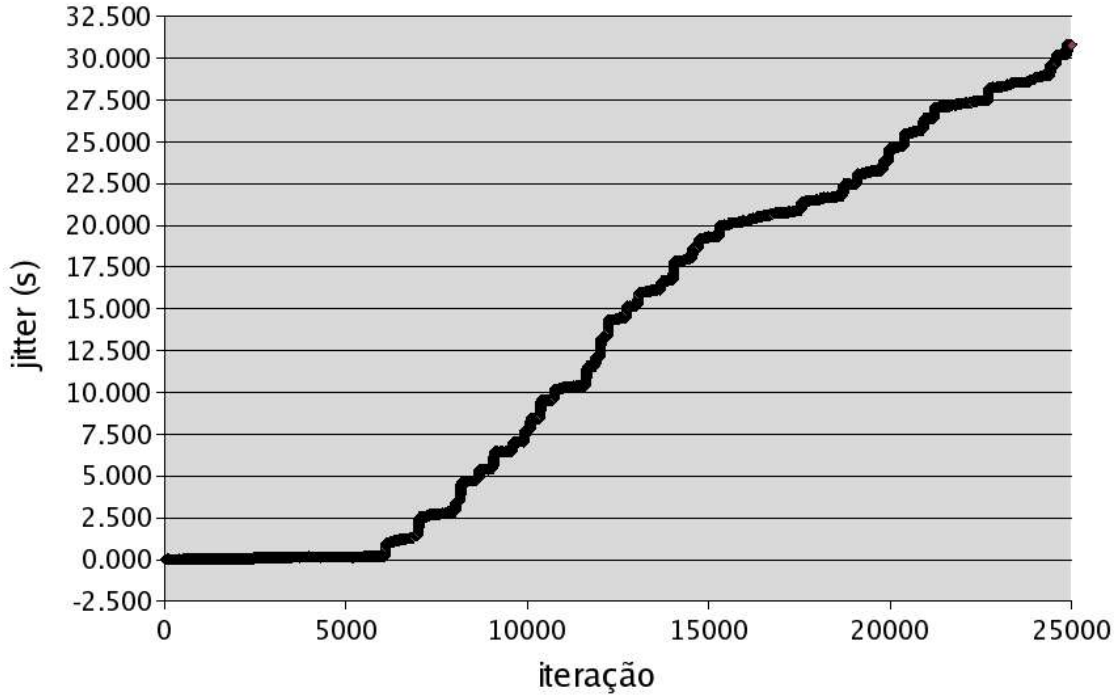

Figura 3.14 - Teste 2 - Período 50 ms (10 módulos), máquina caravela com ambiente gráfico em funcionamento.

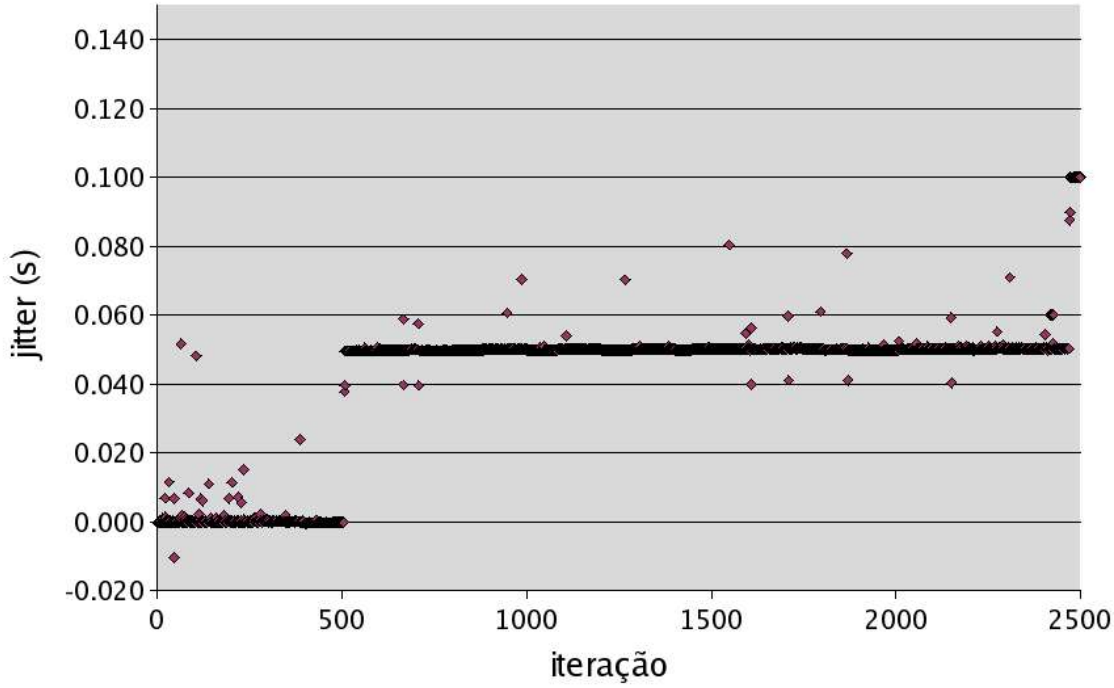

Figura 3.15 - Teste 2 - Período 50 ms (10 módulos), máquina caravela com ambiente gráfico em funcionamento.

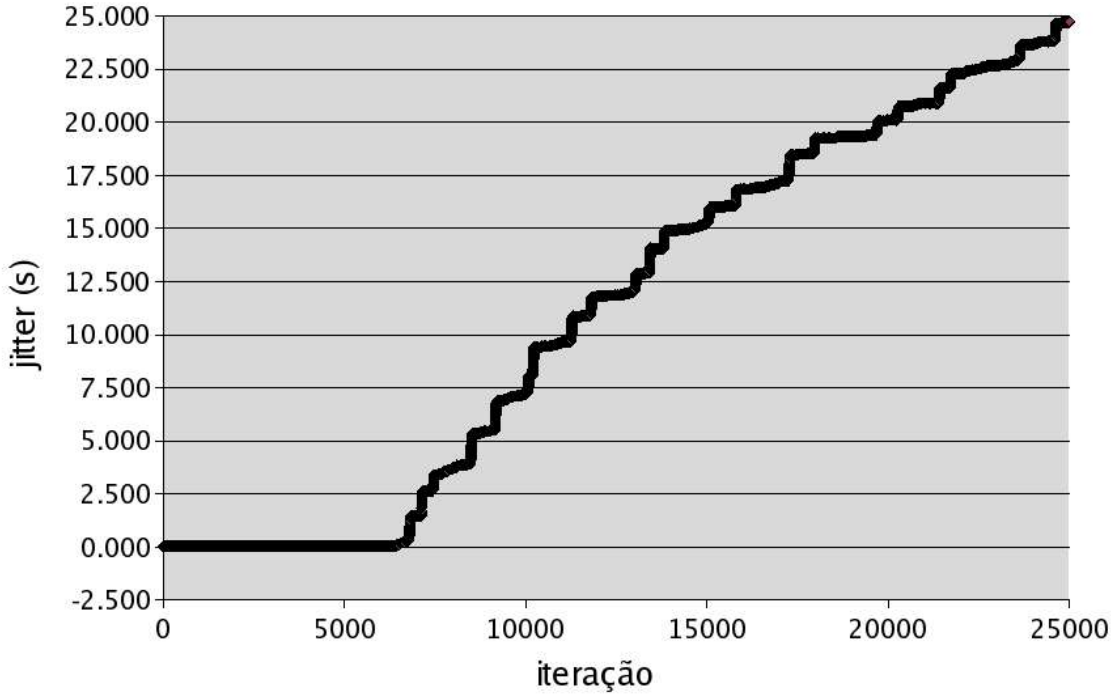

Figura 3.16 - Teste 3 - Período 50 ms, máquina caravela com actividade intensa de rede e disco.

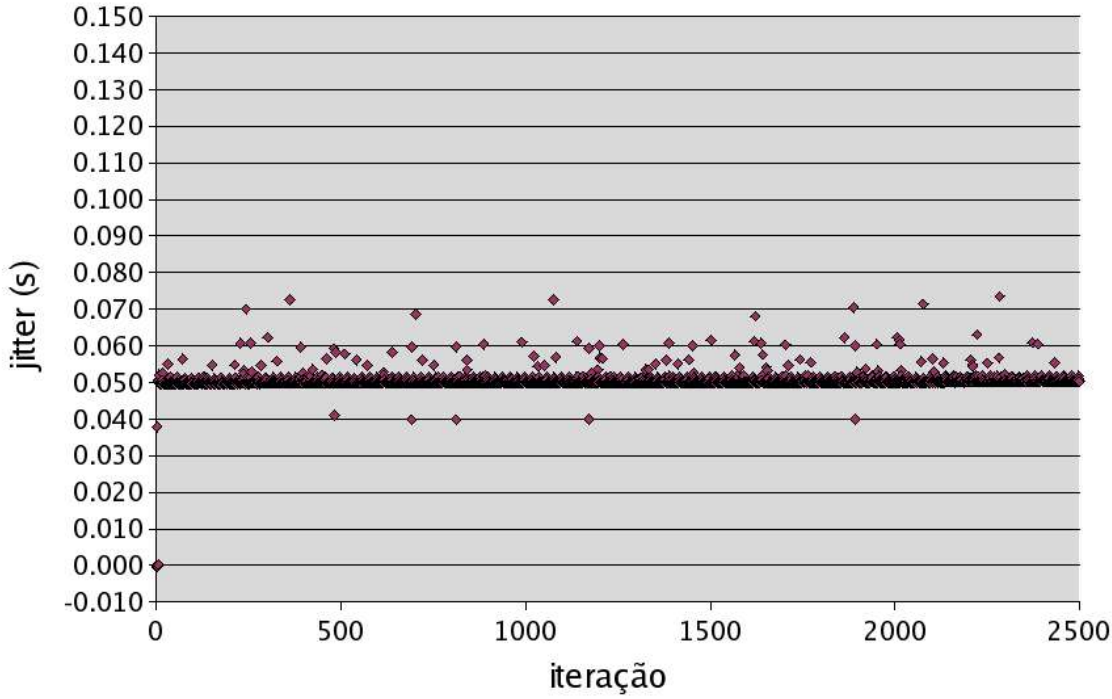

Figura 3.17 - Teste 3 - Período 50 ms, máquina caravela com actividade intensa de rede e disco.

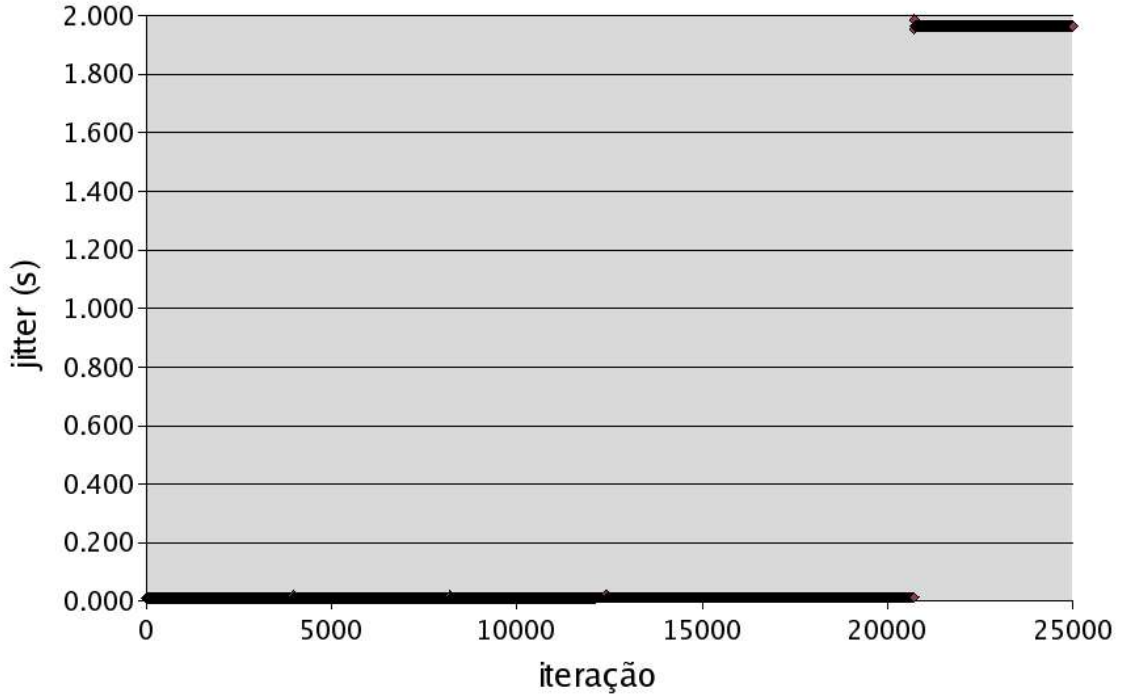

Figura 3.18 - Teste 4 - Período 50 ms, máquina caravela sem outras actividades (modo utilizador único).

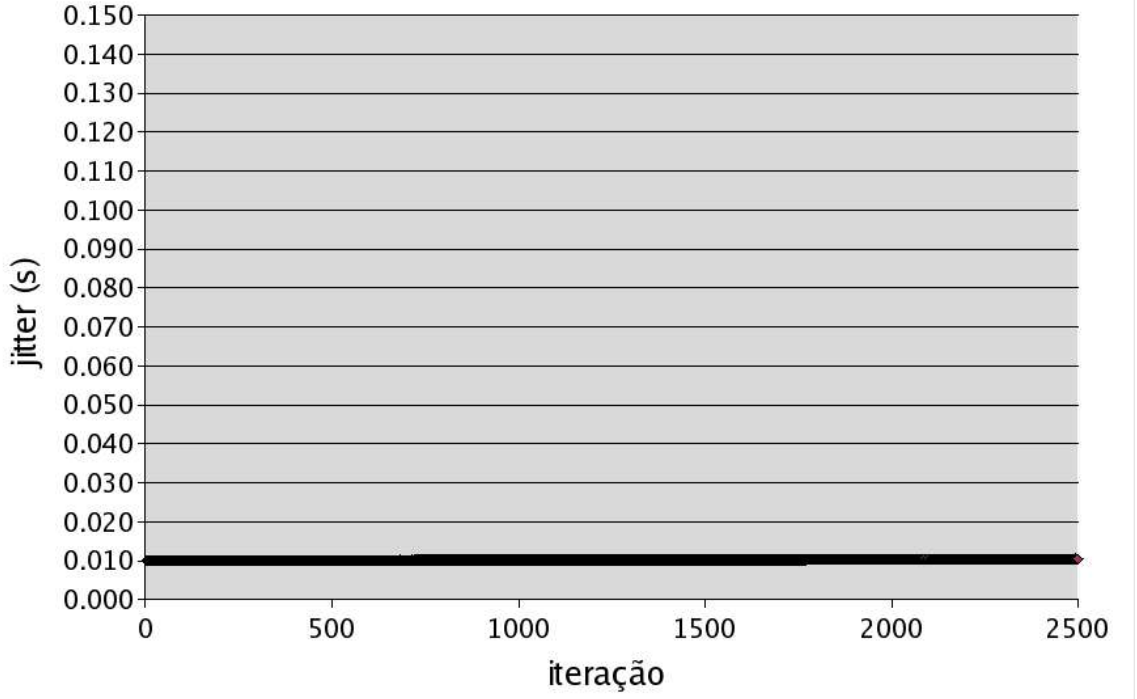

Figura 3.19 - Teste 4 - Período 50 ms, máquina caravela sem outras actividades (modo utilizador único).

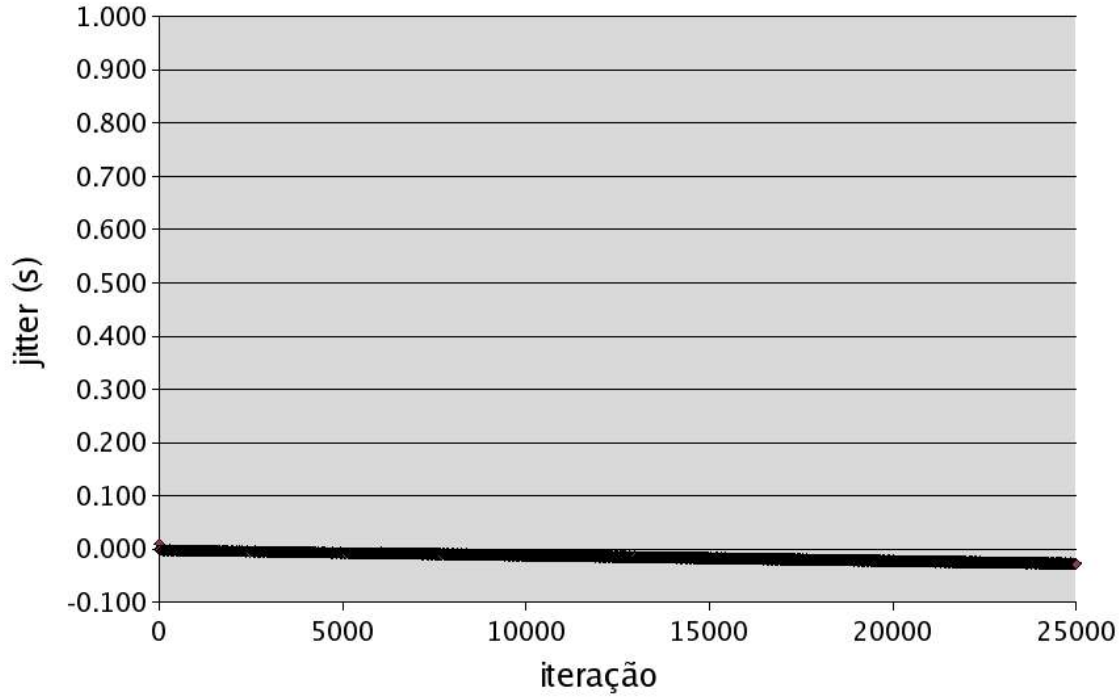

Figura 3.20 - Teste 5 - Período 50 ms, máquina macau com ambiente gráfico em funcionamento (sem prioridades Tempo-Real).

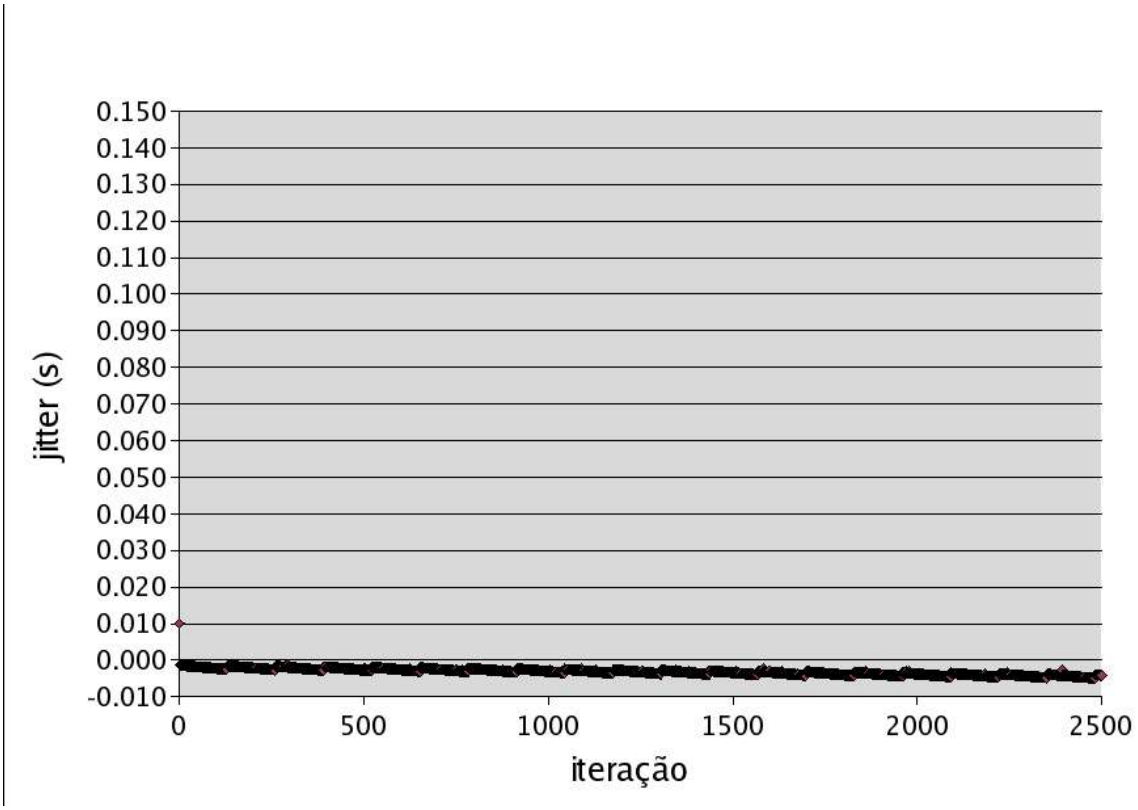

Figura 3.21 - Teste 5 - Período 50 ms, máquina macau com ambiente gráfico em funcionamento (sem prioridades Tempo-Real).

Como se pode observar, em todos os testes efectuados no caravela, o módulo acabou sempre por falhar a execução de algumas iterações. O numero de iterações falhadas variou, no entanto, com o ambiente de teste, e está resumido no quadro seguinte.

Teste	Iterações Totais	Iterações Falhadas	Percentagem de Falhas
1	32000	624	1.95%
2	32000	756	2.36%
3	32000	656	2.05%
4	32000	39	0.12%
5	32000	0	0.00%

Figura 3.22 - Resumo dos resultados dos testes 1-5.

O teste com o Linux 2.4 em modo utilizador único, embora muito melhor que os outros cenários com o Linux 2.4, ainda falha a execução de iterações. Embora se preveja que algumas aplicações de controlo possam falhar a execução de algumas iterações, este género de falhas não deverá ser tolerada pela maioria de aplicações de controlo, e por isso não deverá também ser tolerada no MatPLC. Verifica-se que o MatPLC em QNX não chega a falhar qualquer iteração, mesmo sendo executado com prioridade igual aos outros processos do ambiente gráfico que se encontravam abertos durante o teste. Daqui se conclui que a falha de execução de iterações se pode atribuir ao Linux 2.4 em si, ou então ao modo que o MatPLC utiliza os serviços do Linux.

Como já foi referido anteriormente, a secção de gestão de período é implementado com um contador de iterações em atraso. Tendo ainda em conta que o CPU em nenhum dos testes chegou próximo a uma utilização de 100%, pode-se concluir que a falha completa da execução de uma iteração só se pode dever à falha do incremento deste contador. Isto significa que ocasionalmente em Linux o processo não recebe o sinal despoletado periodicamente pelo temporizador, ou que o temporizador não está a gerar o referido sinal.

3.3.6 A Secção RT

A secção RT faz a gestão dos parâmetros necessários para garantir que um determinado módulo execute com garantias de tempo real [17]. Na realidade, esta secção limita-se a gerir dois parâmetros, a prioridade do módulo e a gestão da memoria utilizada pelo mesmo.

A prioridade sob a qual cada módulo deverá ser executado pelo sistema operativo

pode ser definido pelo utilizador que faz a configuração do MatPLC. Uma vez que cada módulo é executado por um processo independente de o sistema operativo, torna-se possível atribuir prioridades distintas a cada módulo. A secção RT limita-se a chamar as funções apropriadas do sistema operativo para atribuir a prioridade desejada ao processo em questão, sendo que de momento o MatPLC suporta apenas sistemas operativos com prioridades fixas. A prioridade de cada módulo é apenas activada aquando da execução da primeira iteração, uma vez que entre a inicialização do MatPLC e o inicio da primeira iteração o módulo poderá ainda necessitar de inicializar outros recursos do qual irá necessitar. Esta inicialização faz-se assim ainda sob a prioridade normal, e não já com a prioridade potencialmente mais elevada que iria utilizar para a execução das suas iterações, o que permite que outros módulos não sofram interferências indesejadas de um módulo com prioridade elevada mas que se encontra ainda a inicializar os seus recursos.

Deve ser salientado que os módulos não são totalmente independentes. Estes interagem através da partilha da informação armazenada no GMM, bem como através da sincronização das suas actividades recorrendo para isso aos serviços da secção SYNCH.

A partir do momento que os módulos executam sob prioridades elevadas, estas suas interacções podem introduzir bloqueios indesejados entre os módulos, havendo mesmo a possibilidade de haver bloqueios não limitados no tempo. De facto, e uma vez que o acesso à memoria partilhada gerida pelo GMM é gerida com recurso a um semáforo, existe então a possibilidade da ocorrência de inversão de prioridade. Para ultrapassar este inconveniente o MatPLC necessita pois de sistemas operativos que implementem algoritmos de escalonamento que limitam a inversão de prioridade, por exemplo o Priority Ceiling Protocol (PCP), o Priority Inheritance Protocol (PIP), ou as suas variantes.

De forma mais pormenorizada, o acesso à memoria partilhada e gerida pelo GMM é de facto protegida por um de quatro mecanismos de sincronização: semáforo com semântica de Unix SysV, semáforo POSIX, semáforo POSIX anónimo, e mutex POSIX. O mecanismo utilizado depende dos mecanismos fornecidos pelo sistema operativo a ser utilizado. No entanto, o funcionamento com garantias de tempo real exige que seja utilizado um mutex POSIX para o controlo de acesso à memória partilhada pois este é o único mecanismo de sincronização dos atrás mencionados que permite a utilização de protocolos como o PCP e PIP.

Como foi referido, a outra questão levantada pela utilização de prioridades distintas para cada módulo está relacionada com a sincronização dos módulos. A questão reside na possibilidade do utilizador configurar o MatPLC de tal forma que a execução de um módulo com prioridade elevada depender da conclusão de um outro módulo de prioridade mais baixa. Neste caso a interacção através da sincronização das actividades de cada módulo não pode ser tratada de forma

automática pelo MatPLC. De facto as consequências de utilizar uma configuração destas pode ser exactamente o que utilizador pretende, pelo que optou-se por não introduzir automatismos indesejados no MatPLC que poderiam interferir com os desejos do utilizador. Esta opção exige no entanto um maior conhecimento por parte do utilizador das consequenciais das suas opções de configuração.

Outro aspecto gerido pela secção RT é a utilização de memória por parte do módulo e do MatPLC em si. De facto torna-se necessário ter em conta que os sistemas operativos no qual pode ser executado o MatPLC utilizam uma arquitectura de memoria virtual. Estas arquitecturas recorrem por vezes a uma área de memoria em disco utilizada como reserva, para onde pode ser transferida a memoria utilizada por um determinado processo caso a memoria RAM disponível se esgotar. Esta troca provoca atrasos não determinísticos e indesejados relacionados com os tempos de acessos ao disco mais elevados. Uma aplicação com restrições de tempo real necessita obrigatoriamente de ser determinística, pelo que não pode tolerar estas trocas para memoria virtual.

Como forma de obviar estes atrasos a secção RT faz a gestão da memória utilizada pelos módulos do MatPLC, solicitando ao sistema operativo que a troca de memoria para disco (swap) seja desactivada. Embora seja gerida pela mesma secção da biblioteca de funções do MatPLC, esta opção é controlado pelo utilizador, sendo independente da prioridade escolhida para cada módulo. Ou seja, cabe ao utilizador escolher tanto a prioridade de execução de cada módulo, bem como o tipo de memoria a utilizar pelo MatPLC. É de salientar que a opção de desactivar a utilização da área de memoria em disco pelo MatPLC afecta todos os módulos do MatPLC em simultâneo. A razão de ser desta opção deve-se ao facto de que o bloqueio de memoria em RAM será principalmente utilizado quando pelo menos um módulo executa com uma prioridade elevada e com restrições de tempo real. Tendo em conta que o módulo de prioridade elevada, bem como todos os outros, necessitam obrigatoriamente de obter periodicamente acesso à memoria partilhada gerida pelo GMM, o módulo de prioridade elevada pode sofrer bloqueio por parte de qualquer outro módulo com prioridade inferior. Neste caso, e para que o módulo de prioridade elevada não sofra bloqueios por tempos elevados e indeterminados, torna-se necessário garantir que a memoria utilizada por parte de todos os módulos, mesmo os de baixa prioridade, seja bloqueada em RAM.

Os gráficos seguintes evidenciam os efeitos de solicitar uma prioridade elevada para um módulo quando o MatPLC executa sobre um sistema operativo tempo real. Neste caso, o teste foi efectuado na mesma maquina macau já mencionada em testes anteriores (Pentium II a 350 Mhz e 320 MBytes de RAM), a executar o sistema operativo QNX 6.2.1. A maquina encontrava-se durante a execução dos testes com o ambiente gráfico aberto e ligada à rede, mas sem actividade de monta a ser executada em paralelo com o MatPLC. O próprio MatPLC

encontrava-se a executar a demonstração básica, com os módulos a executar de forma assíncrona e sem período de execução definido, ou seja cada módulo a executar as iterações sucessivamente sem que nenhum período de execução tenha sido imposto.

Os gráficos mostram os tempos decorridos entre o inicio da execução de duas iterações sucessivas do módulo lógico, que gere a sequência de activação das luzes controladas pela demonstração básica. Os primeiros dois gráficos correspondem ao teste durante o qual todos os módulos foram executados com a mesma prioridade, sendo esta prioridade ainda igual à dos restantes processos abertos no sistema operativo, tais como editores e ambiente gráfico. Nos dois gráficos seguintes aparecem representados os resultados de um segundo teste durante o qual o módulo lógico foi configurado para executar com prioridade superior a todos os outros processos em execução naquele momento na maquina, incluindo os restantes módulos do MatPLC.

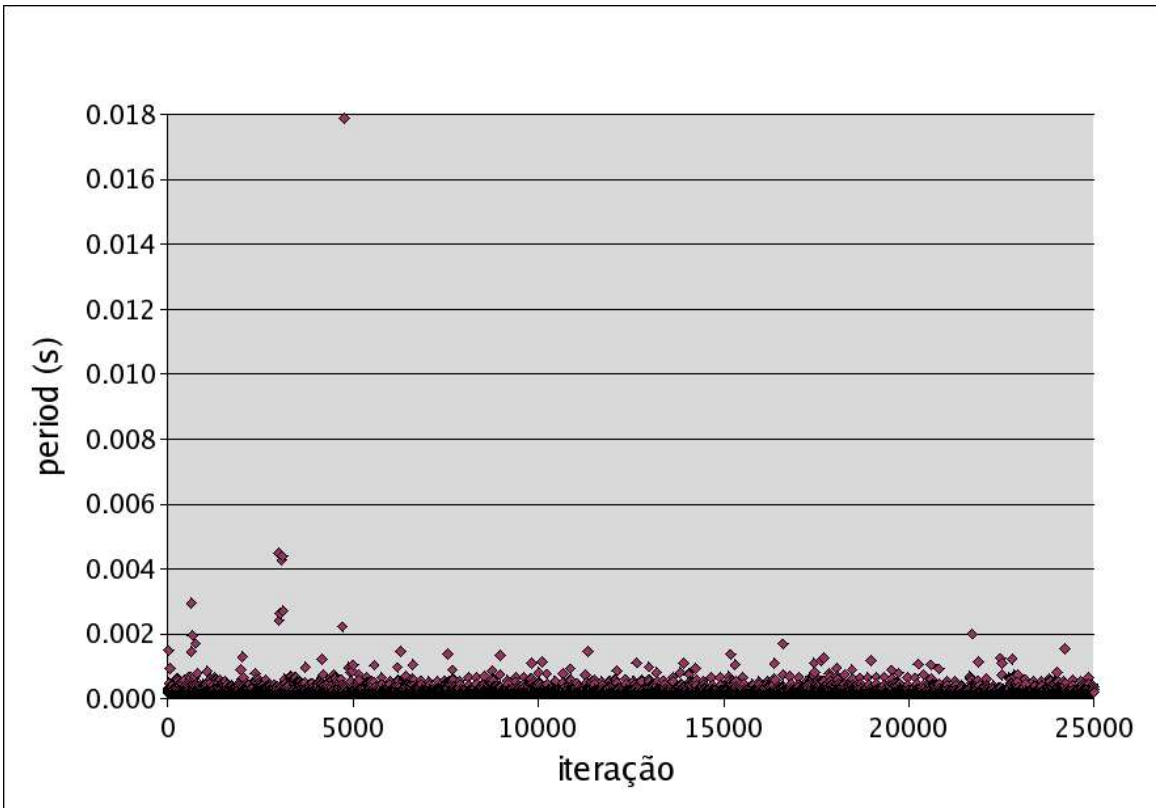

Figura 3.23 - Máquina macau com ambiente gráfico em funcionamento (não RT)

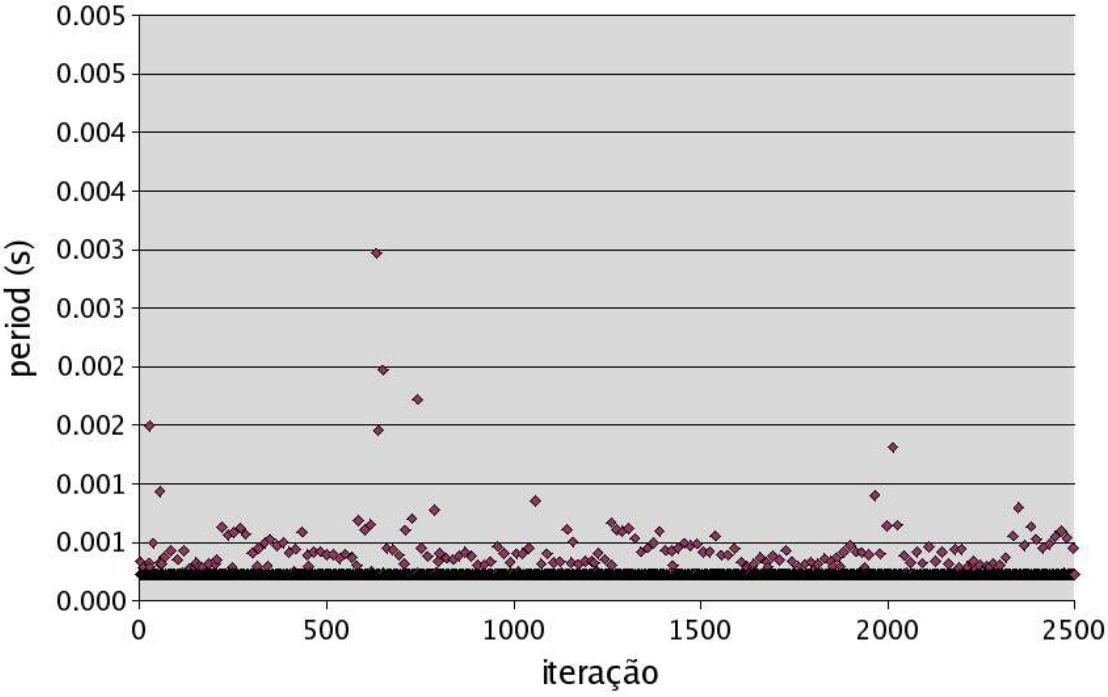

Figura 3.24 - Máquina macau com ambiente gráfico em funcionamento (não RT)

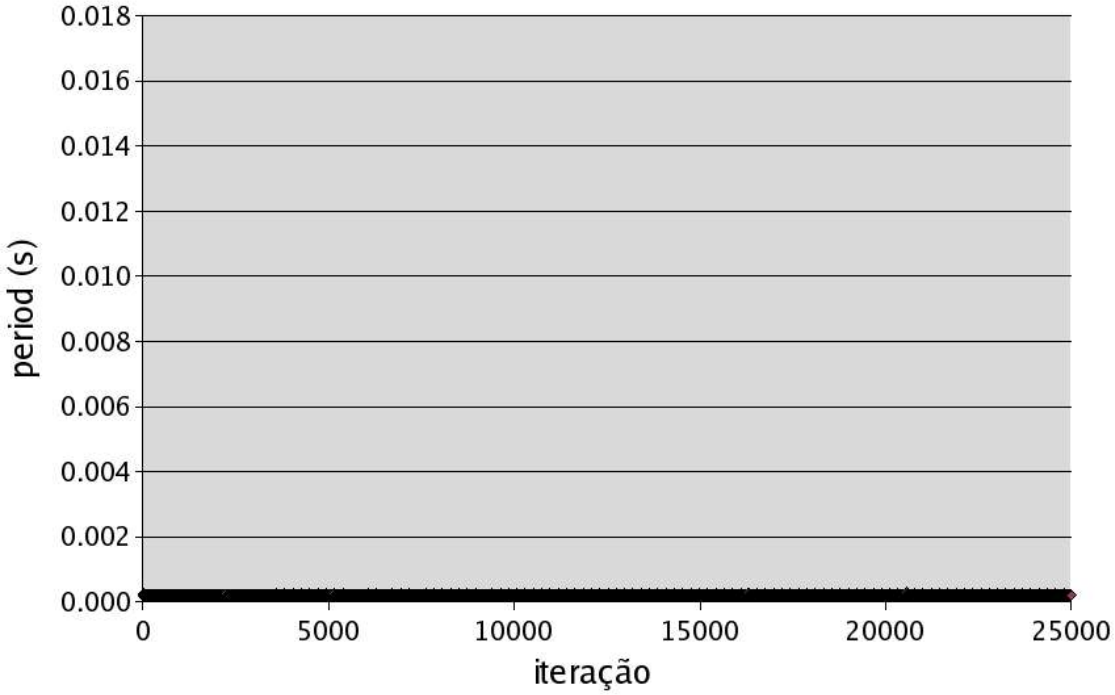

Figura 3.25 - Máquina macau com ambiente gráfico em funcionamento (RT)

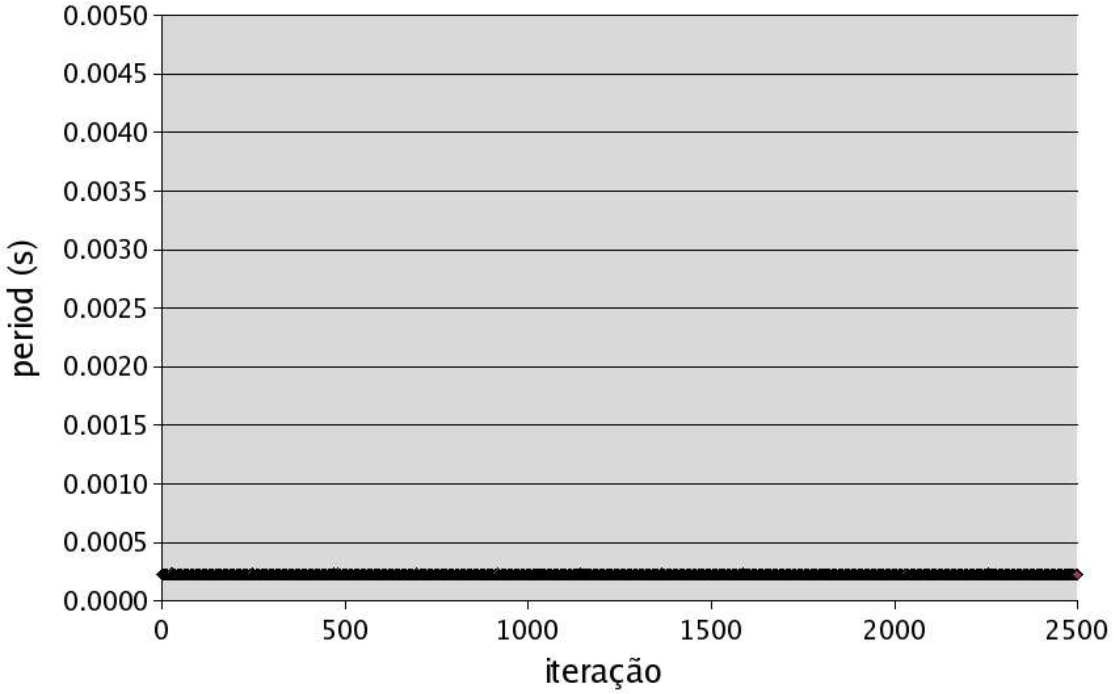

Figura 3.26 - Máquina macau com ambiente gráfico em funcionamento (RT)

3.3.7 A Secção LOG

A secção log não é mais do que um conjunto de funções para produzir mensagens que o utilizador poderá ler. Pretende-se que todos os módulos utilizem estas funções para produzir 'logs' (i.e. registos), em vez de utilizarem cada um os seus próprios métodos. Desta forma, consegue-se centralizar a localização dos registos, bem como normalizar o seu formato.

Cada mensagem deverá ser classificada como sendo de erro, aviso, ou simples informação. Adicionalmente, e de forma ortogonal a esta classificação, a cada mensagem produzida deverá ser atribuído uma prioridade ou nível de importância. Estes níveis seguem uma estrutura numerável de 1 a 9.

De momento a secção log limita-se a escrever os registos produzidos por cada módulo para ficheiros. Os ficheiros utilizados para este fim são configuráveis pelo instalador para cada módulo individualmente. Poderá ainda ser utilizado um único ficheiro, ou ficheiros distintos para cada classificação de mensagens (erro, aviso, informação). Assim, o instalador tanto pode configurar o sistema para o extremo de enviar todas as mensagens de todos os módulos para o mesmo ficheiro, até ao outro extremo de cada tipo de mensagem de cada módulo para um ficheiro distinto, num total de 3 ficheiros por módulo. Todos as permutações possíveis entre os dois extremos são também possíveis.

De forma a limitar o numero de mensagens guardadas e a poupar o espaço de disco ocupado, o instalador pode ainda definir qual o limiar de importância das mensagens a partir do qual estas deverão ser ignoradas. Mais uma vez, esta configuração é feita ao nível de cada módulo, mas por enquanto é válida para todas os três tipos de mensagens em simultâneo.

É de esperar que futuras iterações desta secção permitam o envio das mensagens para outros sistemas de tratamento de mensagens, como o syslog existente nos sistemas operativos UNIX e seus similares. O próprio sistema syslog pode já ser configurado para guardar as mensagens em ficheiro, ou até para as enviar via rede para uma estação remota. Para sistemas operativos que não tenham o serviço syslog, será ainda de considerar uma outra versão da secção log que ela própria envie as mensagens via rede para uma estação concentradora de mensagens. Isto permitirá a utilização do MatPLC em sistemas com recursos de disco limitados, mas mesmo assim continuar a guardar as mensagens enviadas pelo sistema.

Para ambientes tempo real, torna-se ainda necessário fornecer uma implementação da secção log que tenha tempos de execução determinísticos. Isto porque algumas das mensagens geradas serão a partir de módulos com prazos fixos e provavelmente apertados e que têm de ser respeitados. Nestes casos o próprio tratamento das mensagens e o seu arquivo para posterior referencia deverá ter tempos curtos e previsíveis. Para tal prevê-se ainda uma outra versão desta secção que se deverá limitar a enviar as mensagens para um 'pipe'. Um pipe

é um mecanismo de comunicação entre processos definido pelo POSIX, o qual permite a transferência de dados entre processos, sendo que um processo escreve os dados para o pipe, e outro lê esses mesmos dados do outro extremo do pipe. O pipe funciona como uma uma fila FIFO de dados entre os dois processos. É de referir que poderão haver vários processos a escrever e/ou a ler do mesmo pipe. Os sistemas operativos tempo real disponibilizam pipes com tempos de escrita e leitura determinísticos, pelo que a versão desta secção que envia as mensagens para um pipe pode ser considerada determinística.

Esta versão será acompanhada por um programa utilitário que se encarregará de ler todas as mensagens do pipe, e encaminha-las para o local final, seja ele um ficheiro, o sistema syslog ou via rede para um sistema remoto. O mais provável será a utilização deste programa utilitário como um deamon, ou seja um programa que esteja continuamente em execução. O ciclo de vida deste processo deamon poderá ser controlado de forma transparente para os utilizadores pela própria secção log. Assim, o processo será lançado aquando da inicialização do MatPLC, e terminado quando o MatPLC é desactivado.

3.3.8 A secção CONF

A secção conf é responsável pela leitura do ficheiro de configuração. À semelhança da secção log, o seu objectivo é basicamente o de centralizar todas as operações de análise sintáctica do ficheiro de configuração. Assim, todos os módulos poderão ler os seus parâmetros de configuração de um mesmo ficheiro, utilizando a mesma sintaxe. Mais tarde, se for necessário suportar uma sintaxe mais alargada (por exemplo, configuração baseada em XML), bastará actualizar esta secção para que todos os módulos e as restantes secções do MatPLC passem a utilizar a nova sintaxe de configuração.

A sintaxe actualmente utilizada permite atribuir valores simples a parâmetros de configuração, com uma sintaxe do tipo

```
<nome_do_parâmetro> '=' <valor>
```

nos quais tanto o parâmetro e o valor são simples cadeias de caracteres.

Em alternativa, é ainda possível atribuir valores em forma de matriz a um parâmetro de configuração. Neste caso a sintaxe, definida em formato BNF (Backus-Naur Format) utilizada é:

```
<matriz> ::= <linha> [<matrix>]
<linha> ::= <nome_do_parâmetro> <lista_de_valores> EOL
<lista_de_valores > ::= <valor> [' ' <lista_de_valores >]
```

Aqui o símbolo 'EOL' representa o caracter geralmente utilizado para demarcar um fim de linha em ficheiros de texto.

Cada parâmetro de configuração terá ou um valor simples ou uma matriz de valores, mas nunca os dois em simultâneo. A semântica do par parâmetro-valor não é analisada nem é conhecida pela secção conf. Esta secção limita-se a fazer a análise da sintaxe, cabendo ao módulo ou a secção do MatPLC a que diz respeito cada parâmetro a interpretação do valor em causa.

Os parâmetros são agrupados em secções. Neste caso, as secções do ficheiro de configuração não têm correspondência alguma com as secções da biblioteca de funções do MatPLC. As secções servem apenas para agrupar os parâmetros, sendo que, por omissão, cada módulo irá apenas utilizar os parâmetros pertencentes à secção cujo nome coincide com o nome do próprio módulo. O inicio de cada secção no ficheiro de configuração é marcada pela sintaxe

```
'[' <nome_da_secção> ']'
```

A secção especial '[PLC]' é utilizada para os parâmetros que são válidos para todos os módulos e que no fundo servem para configurar o funcionamento interno do MatPLC. Exemplos de parâmetros que pertencem a esta secção são os respeitantes à sincronização entre os módulos, os pontos do MatPLC, e a lista dos módulos que deverão ser inicializados quando o MatPLC é criado.

3.4 Exemplos de Módulos

3.4.1 O DSP

O módulo DSP está vocacionado para o tratamento digital de sinais (DSP – Digital Signal Processing). Este é um módulo que é fornecido com o MatPLC já pronto para executar, sendo que ao instalador cabe apenas a sua configuração. Trata-se de um módulo de lógica, pois não interage com o utilizador, nem com entradas e saídas físicas, limitando-se a processar os dados armazenados nos pontos do MatPLC, e a guardar o resultado do processamento em outros pontos.

O módulo DSP segue também ele uma arquitectura modular. É composto por blocos que fazem efectivamente o processamento dos sinais, tais como controladores PID, filtros digitais, somadores, multiplicadores e funções não lineares. Cada bloco tem parâmetros de entrada e de saída que deverão ser ligados a pontos do MatPLC. Cabe ao instalador definir uma lista dos blocos que pretende utilizar e quais os pontos que cada bloco irá usar como entradas e saídas.

O mesmo tipo de bloco poderá ser utilizado várias vezes, sendo provável que de cada vez terá parâmetros diferentes. Por exemplo, um controlo PID com dois anéis, um interior para controlo de velocidade e um exterior para controlo de posição, irá ser implementado com recurso a dois blocos PID, cada qual com os

seus próprios parâmetros de configuração (P, I e D).

Cada bloco implementa um algoritmo tipicamente utilizado em processamento digital de sinais. Embora os sinais a serem tratados poderão ser contínuos no tempo, estes são amostrados periodicamente para posterior tratamento pelo módulo que se encontra a fazer a interface com o meio físico. Por este motivo cada bloco deverá ser executado ciclicamente, sendo que em cada ciclo será considerado uma nova amostra dos sinais de entrada, e produzida uma nova amostra dos sinais de saída. Os blocos são executados pela mesma ordem em que são declarados no ficheiro de configuração.

Os blocos podem ser ligados em linha ao atribuir o mesmo ponto simultaneamente à saída de um bloco anterior e a uma entrada de um bloco posterior. Estas linhas podem ainda ser fechadas, formando um anel. Cabe ao instalador definir o local onde o anel será quebrado para a execução de cada iteração do ciclo de processamento de amostras. Isto consegue-se pela ordem pelo qual os blocos são configurados uma vez que estes executam cada iteração pela mesma ordem em que foram declarados no ficheiro de configuração.

O correcto funcionamento dos algoritmos de processamento de sinal implementados em vários blocos pressupõe que este módulo seja executado com período fixo. Cabe assim ao instalador a configuração do módulo para que este execute periodicamente, bastando para isso recorrer aos serviços do MatPLC e da sua secção period. Convém ainda garantir que os sinais de entrada sejam amostrados periodicamente, e os sinais de saída actualizados com o mesmo período. Para tal o instalador poderá utilizar os serviços de sincronização de módulos para garantir que os módulos que tratam de mapear as entradas e saídas físicas nos pontos internos sejam também eles executados síncronamente com o próprio módulo DSP.

De momento este módulo disponibiliza blocos somador, multiplicador, potenciação, PID, filtro, rampa, alarm, não-linear e multiplexador. O bloco somador encarrega-se de somar ou subtrair vários sinais de entrada, gerando um único sinal de saída. De forma semelhante, o bloco multiplicador encarrega-se de multiplicar vários sinais de entrada, e produzir um único sinal de saída. O bloco potenciação faz a potenciação de um sinal de entrada por um valor definido noutro ponto do MatPLC, produzindo assim um único sinal de saída.

O bloco PID implementa um controlador PID paralelo em malha aberta. Tem como parâmetros as constantes Proporcional, Integral e Derivativa, um único sinal de entrada e um único sinal de saída. Um controlador PID típico em malha fechada poderá ser configurado recorrendo a um bloco somador e um bloco PID. O bloco somador encarrega-se de determinar a diferença entre o sinal de referencia e o sinal na planta, tendo assim na sua saída o sinal de 'erro' que servirá como entrada ao bloco PID.

Para alem do modo automático, o bloco PID permite ainda um funcionamento

em modo manual. Para tal tem uma entrada digital através da qual se pode controlar o seu estado em cada momento. Uma segunda entrada é utilizada para obter o sinal que deverá ser copiado para a saída do bloco PID quando este se encontra em modo manual. Se nada for feito para o evitar, a passagem entre do modo manual para o modo automático pode resultar numa descontinuidade brusca no valor na saída do bloco PID. Para evitar esta descontinuidade o bloco PID mantém a sua saída interna do modo automático igual ao valor de saída em modo manual, ajustando para tal o valor do integral. Devido ao modo em que é implementada, esta transferência sem descontinuidade do modo manual para o modo automático só é possível se o parâmetro Integrador for diferente de zero.

A mesma descontinuidade poderá ocorrer durante a passagem de modo automático para modo manual. Neste caso, e se esta descontinuidade não for desejável, caberá a blocos externos garantir que o valor do modo manual não difere do modo automático no instante em que a passagem de modo é efectuada.

O bloco PID implementa ainda um limitador do valor de saída, que só é aplicado quando em modo automático. Mais uma vez, este limite é conseguido através da alteração do valor do integral do sinal de entrada, de forma a garantir que este integral não fique com valores demasiado elevados durante os intervalos de tempo em que o limitador de saída entra em acção.

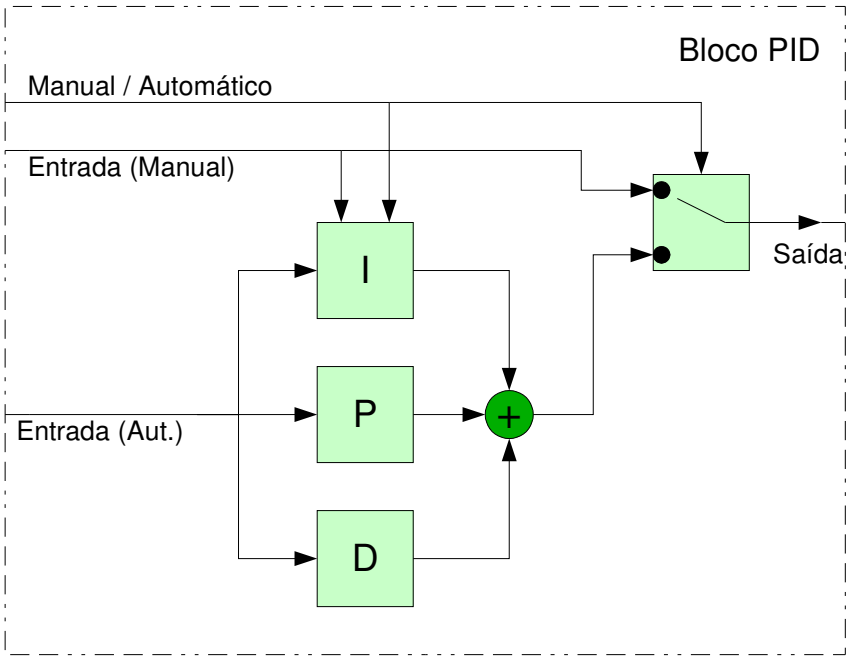

Figura 3.27 - Construção interna de um bloco PID.

O bloco filtro implementa um filtro digital IIR (Infinite Impulse Response) de segunda ordem. Cabe ao instalador configurar os cinco parâmetros deste bloco. Estes filtros poderão ser ligados em série para conseguir filtros de ordem superior.

Em alternativa, o instalador poderá configurar o filtro através das suas características de frequência, sendo o filtro IIR equivalente determinado internamente. Poderão ser definidos filtros com formatos passa baixo, passa alto,

passa banda ou rejeição de banda. Igualmente, poderão ser escolhidas aproximações de Butterworth, Chebyschef ou Elíptica para cada tipo de filtro. Os parâmetros necessários para a determinação dos parâmetros internos incluem as frequências limite, o ganho na(s) banda(s) de passagem, e a atenuação mínima na(s) banda(s) de rejeição.

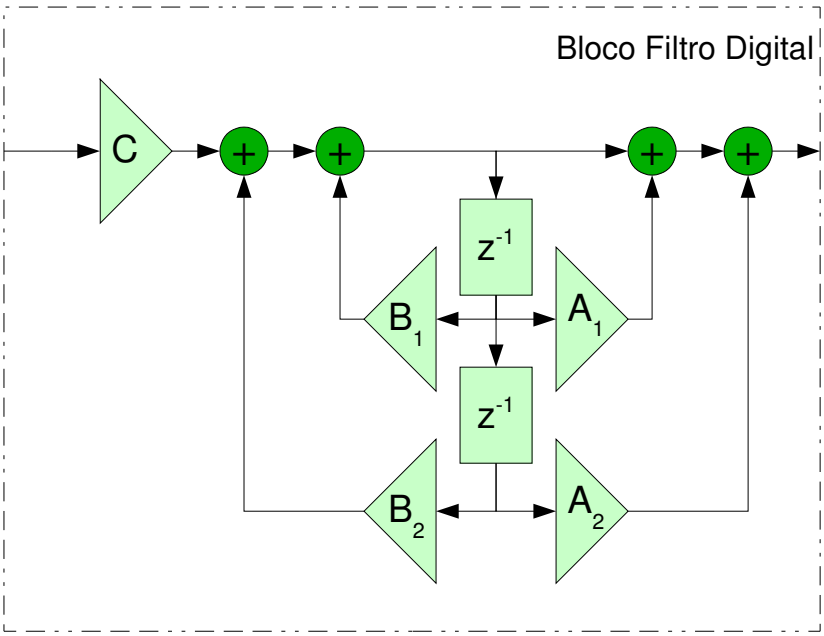

Figura 3.28 - Composição interna do filtro digital.

O bloco rampa implementa um limitador de derivada de primeira e segunda ordem ao único sinal de entrada. Ou seja, o sinal de saída segue o sinal de entrada desde que a primeira e segunda derivadas deste não ultrapassem os limites estabelecidos pelo instalador. No caso destes limites serem ultrapassados, o sinal de saída deixa de seguir o sinal de entrada, mas continua a tentar aproximar-se deste o mais rapidamente possível continuando a respeitar os limites para a primeira e segunda derivadas. É possível atribuir limites distintos para as derivadas positivas e as derivadas negativas.

Este bloco destina-se principalmente a limitar sinais de entrada que se sabe poderem ter variações bruscas, e produzir assim um sinal contínuo e sem saltos.

O bloco alarm permite verificar se um sinal de entrada se encontra dentro de determinados parâmetros, tendo como saída um sinal booleano que é activado quando o sinal de entrada ultrapassa os limites.

O bloco não-linear implementa uma função não linear entre o sinal de entrada e o sinal de saída. Permite introduzir uma zona morta ao sinal de entrada, bem como limites inferior e superior ao seu valor.

O módulo multiplexador copia o valor presente em uma das suas várias entradas para a sua única saída. A entrada utilizada depende do valor na entrada de controlo do bloco.

3.4.2 O ModBus

O módulo ModBus é um módulo de entrada e saída que implementa o protocolo de comunicação ModBus. Este módulo permite a interacção do MatPLC com qualquer equipamento que saiba falar este protocolo, tais como variadores de velocidade, entradas e/ou saídas digitais e/ou analógicas remotas, outros PLCs comerciais, etc.

O protocolo permite a leitura e escrita de variáveis nos equipamentos remotos. De facto, as variáveis são referenciadas não pelo seu nome, mas sim pelo endereço de memoria em que se encontram armazenadas, pelo que cada fornecedor de equipamento que suporta este protocolo deverá informar os endereços em que se encontram as variáveis de interesse.

Para utilizar este módulo em conjunto com o MatPLC cabe apenas ao instalador a sua configuração. Esta envolve a configuração dos parâmetros da rede (endereços, velocidade de transmissão, bits de paridade, etc.), bem como os dados que deverão ser transferidos entre o MatPLC e os equipamentos remotos. Para que seja efectuada a transferência de dados é necessário definir a correspondência ou mapeamento desejado entre pontos internos do MatPLC e as zonas de memoria em cada equipamento remoto. É ainda necessário definir em que direcção será feita a transferência da informação, se do MatPLC para o equipamento remoto, ou vice-versa.

Este módulo executa um ciclo infinito. Em cada iteração do ciclo é efectuada uma transferência de todos os dados configurados no mapa de correspondências. O algoritmo seguido pelo módulo começa por ler todos os dados dos equipamentos remotos, e guardar estes na sua copia local do mapa global. É depois efectuada a sincronização do mapa da copia local com o mapa global. Segue-se o envio de todos os dados configurados para os equipamentos remotos.

O protocolo ModBus foi criado pela Modicon para a transferência de dados entre vários PLCs, e ainda entre PLCs e entradas/saídas remotas. Embora não seja especificado como tal, este protocolo pode ser considerado como tendo três níveis distintos: o nível físico, o nível de ligação de dados, e o nível lógico. Para o nível físico o standard escolhe o RS 485, embora também seja possível encontrar versões sobre RS232. Para o nível de ligação de dados o standard define três alternativas, uma baseada no envio dos dados em binário (conhecida por RTU), outra baseada na transferência dos dados codificados em ASCII (conhecida por ASCII), e a ultima baseada no envio das tramas sobre conexões TCP/IP. As versões RTU e ASCII pressupõem a utilização de RS485 no nível físico, enquanto que o TCP/IP recorre a qualquer nível físico sobre o qual estes protocolos podem ser transferidos. Cada uma destas versões do nível de ligação lógica utiliza o seu próprio algoritmo de detecção de erros. O ultimo nível lógico define o protocolo utilizado para a transferência de dados, incluindo o formato das tramas. Este protocolo é baseado numa arquitectura mestre/escravo.

Devido ao atrás exposto, este módulo foi implementado em dois níveis. O nível inferior é composto por três versões, uma para cada opção do protocolo de ligação lógica: RTU, ASCII e TCP/IP. O nível superior é composto por duas versões, uma que implementa o protocolo do lado do mestre, e outra, ainda não implementada, que contempla o protocolo do lado do escravo.

A componente RTU foi a que se mostrou mais problemática pois o standard que define o protocolo é ambíguo em alguns detalhes, em especial no que concerne às temporizações máximas entre a transmissão de cada byte da mesma trama, e à recuperação de erros após a violação destas temporizações. Adicionalmente, o funcionamento correcto desta versão baseia-se em garantir intervalos máximos de tempo entre a transmissão de bytes consecutivos pela porta série correspondente ao tempo que demoraria a enviar alguns bits, o que não é de todo possível garantir com o acesso à porta série através dos 'device drivers' habitualmente fornecidos com o Linux. De facto, o protocolo prevê que as tramas enviadas sejam separadas por um intervalo de tempo mínimo equivalente à transmissão de 3,5 bits. Adicionalmente, todos os bytes da mesma trama não devem ter separações superiores a 1,5 bits. Para além de não ser possível garantir com toda a certeza a transmissão dos bytes da mesma trama com separações inferiores a 1,5 bits, o que dificulta a transmissão, também não é possível contabilizar com a precisão necessária o tempo que medeia duas tramas consecutivas, o que dificulta a recepção.

A probabilidade de ultrapassar o limite máximo de 1,5 bits entre bytes na transmissão foi reduzida ao máximo recorrendo à técnica de entregar sempre uma trama completa ao device driver da porta série, numa única chamada ao sistema, sempre que se deseja enviar uma trama ModBus. Por outro lado, a única forma encontrada de contornar correctamente a impossibilidade de detectar as delimitação entre as tramas aquando da sua recepção foi a de considerar todos os bytes recebidos como uma sequência contínua sem interrupções, e recorrer a um algoritmo de pesquisa de tramas válidas entre essa sequência de bytes. O algoritmo de pesquisa de tramas válidas necessita de informação referente à constituição das tramas. Esta informação não deveria ser incorporada no nível de protocolo correspondente ao RTU, mas apenas no nível imediatamente superior, uma vez que se deseja uma separação rigorosa de responsabilidades entre níveis de protocolo. No entanto, a correcta implementação do nível RTU assim o obrigou. Por outro lado, o algoritmo de detecção de tramas baseia-se nos códigos de detecção de erros (CRC – Cyclic Redundancy Check) que fazem parte do próprio protocolo.

Seria possível simplificar bastante a implementação do protocolo, em especial a componente de recepção de tramas RTU, se este fosse implementado dentro de um módulo do núcleo do Linux, e assim com capacidade de medir os tempos que medeiam a recepção de cada byte. No entanto esta opção foi posta de lado pois corresponderia a introduzir mais complexidade na utilização do módulo do

MatPLC, que assim só funcionaria se estivesse presente o módulo do núcleo que implementaria o protocolo.

3.4.3 A Biblioteca de Funções para E/S

Como já foi referido, cabe ao instalador definir o mapeamento entre os pontos internos do MatPLC e as zonas de memória dos equipamentos remotos, bem como a direcção de transferência de informação entre os equipamentos. O instalador pode ainda escolher que os dados sejam invertidos bit-a-bit antes da sua transferência de/para (consoante o ponto esteja a ser lido ou escrito) o MatPLC, suportando assim tanto equipamentos cujas entradas e/ou saídas são activas ao nível baixo, bem como os equipamentos com entradas/saídas activas ao nível alto. Por outro lado, prevê-se que futuramente poderá ainda ser possível fazer adaptações de pontos que representam valores analógicos, tal como multiplicação por constantes e/ou linearização de curvas. Estas opções poderão ser utilizadas para a adaptação de grandezas de transdutores/actuadores não lineares, isolando assim o algoritmo de controlo do actuador de facto utilizado para medir uma grandeza. O módulo de entrada/saída apresenta-se assim com uma interface standard ao algoritmo de controlo, o que significa que mais tarde poderão ser substituídos os sensores e actuadores, sem que isso implique alteração no módulo que implementa o algoritmo de controlo, bastando assim a re-configuração do módulo de entrada/saída para as novas características dos sensores/actuadores utilizados.

Como é obvio, torna-se desejável que todos os módulos com funções de entrada/saída de valores apresentem as mesmas potencialidades de linearização e adaptação dos valores de entrada/saída, o que potencia que a substituição de um sensor/actuador por outro possa implicar não só uma nova curva de linearização, bem como um novo hardware de interface ou protocolo de rede para fazer a interface com o novo dispositivo. Para tal, a parte do código referente à linearização, inversão e multiplicação de valores de entrada, bem como ao mapeamento dos pontos internos do MatPLC nas entradas/saídas físicas, foi todo colocado numa biblioteca de funções comuns a que todos os módulos de entrada/saída deverão recorrer.

De facto, tendo ainda em conta que a maioria dos módulos de entrada/saída terão uma arquitectura comum, i.e. o de executar ciclicamente o mapeamento entre os pontos internos e as entradas/saídas externas, a biblioteca foi desenvolvida de tal forma que de facto as suas funções compreendessem um módulo de entrada/saída praticamente completo. Isto significa que esta biblioteca, para alem das funções de mapeamento e de linearização, compreende ainda a função principal pela qual se inicia a execução do processo (em C, isto corresponde à função chamada main(int, argv **)). Para produzir um novo módulo de entrada e saída faltam apenas as funções de acesso às entradas/saídas físicas, o que poderá corresponder à implementação de um protocolo de

comunicação (como o modulo ModBus.), ou ao acesso ao hardware em si, tanto directamente ou através de 'device drivers' ao nível do núcleo.

Capítulo 4

Compilador IEC 61131-3

Com o intuito de aumentar a aceitação do MatPLC tornou-se filosofia do projecto de, sempre que possível, recorrer a standards existentes em alternativa a criar novas interfaces e métodos pela simples razão de serem diferentes. Adicionalmente, e com o mesmo objectivo, foi feito um esforço de implementar o suporte por parte do MatPLC de standards com grande aceitação e divulgação no mercado. Neste sentido, e de forma a complementar o projecto do MatPLC, foi implementado um compilador para as linguagens textuais do standard IEC 61131-3.

No decorrer deste trabalho foram identificadas falhas no próprio standard, que tiveram de ser ultrapassadas. Embora alguns dos erros identificados se possam explicar como falhas não intencionais e por lapsos na definição das linguagens, outras questões relacionadas com as próprias opções tomadas para as linguagens produzem resultados indesejados e que possivelmente são inesperados e não intencionais.

Neste capitulo será portanto descrito o trabalho desenvolvido durante a implementação do compilador já referido, sendo ainda identificados e explicadas as falhas identificadas no standard. Irá ainda ser detalhado as formas escolhidas para contornar algumas das questões identificadas, bem como irão ser efectuadas

sugestões de como outras poderiam ser eliminadas. No entanto, o capitulo começa com uma introdução às linguagens em si, bem como ao standard dentro do qual estas se encontram definidas.

4.1 O Standard IEC 61131

Com a crescente aceitação dos PLCs no ambiente industrial surgiram, como é natural, um grande numero de fabricantes dos mesmos a concorrer neste mercado. Muito embora os PLCs destes fabricantes tivessem muitas semelhanças, está claro que cada um apresentou as suas próprias soluções no que toca muitos aspectos tanta da sua construção física, dos protocolos utilizados nas redes de comunicação, bem como nas linguagens utilizadas para a sua programação. De facto, embora a programação dos PLCs fosse por motivos históricos efectuada com base em linguagens que se assemelham à concepção de circuitos eléctricos ('Ladder Logic'), a sua extensão para abarcar capacidades não disponíveis em circuitos eléctricos (tais como a chamada a subrotinas e a utilização de contadores) fez com que as linguagens de programação de cada fabricante se tenham começado a diferenciar. No inicio estas diferenças mantinham-se pequenas, sendo ainda possível que um programador transitasse rapidamente da programação de um PLC para outro sem grande dificuldade. No entanto, e à medida que as capacidades dos PLCs foram aumentando, com eles foi crescendo também a complexidade das linguagens de programação utilizadas para tirar proveito dessas capacidades acrescidas. Assim a passagem de um programador de um PLC para outro começou a envolver um cada vez maior tempo de aprendizagem e habituação novo equipamento. Paralelamente, a passagem de um programa inicialmente desenvolvido para uma marca de PLC para outra também foi aumentando.

Para contrariar esta crescente disparidade de linguagens de programação dos PLCs, o IEC (International Electrotechnical Commission) iniciou trabalhos com o intuito de desenvolver um standard que definisse o comportamento que todos os PLCs deveriam ter. Foi assim constituída, em 1979, uma comissão para produzir um standard que definisse o modo como um PLC interagia com o exterior, sem no entanto definir a sua construção interna. Em 1982 esta comissão produziu uma proposta que se mostrou tão complexa que se tornou necessário reparti-la em cinco partes.

A primeira parte é composta por informação geral, e inclui definições básicas. A segunda abarca o equipamento em si. Define os aspectos físicos do PLC no que concerne à interface com o exterior, bem como os modos de testar os equipamentos de forma a garantir que os mesmos cumprem o disposto no

standard. A terceira parte define a visão que um utilizador e programador tem de um PLC. Define o funcionamento do PLC em termos lógicos, uma arquitectura de programação do mesmo, bem como diversas linguagens de programação, que se encaixam na arquitectura referida, e que poderão ser utilizadas para a programação dos PLCs. A quarta parte inclui um guia para os utilizadores dos PLCs, enquanto que a quinta parte foca os aspectos de comunicação entre os PLCs. Esta quinta parte de facto limita-se a definir a visão que um programador tem quando utiliza uma rede de dados para transferir informação, sem no entanto definir quaisquer protocolos de comunicação que deverão ser utilizados pelos PLCs.

4.1.1 O Standard IEC 61131-3

Como foi referido, a terceira parte do standard IEC 61131, geralmente referenciada como IEC 61131-3 [4], engloba as linguagens de programação suportadas pelos PLCs. Uma primeira versão desta parte do standard foi aprovada em Março de 1993. Esta versão foi mais tarde revista e modificada, resultando numa segunda versão que foi oficialmente aprovada em 2003. Devido ao elevado preço destes standards, bem como à impossibilidade de os puder divulgar livremente, o compilador desenvolvido para o MatPLC foi baseado na última versão de trabalho, datada de 10 de Dezembro de 2001. Foi mais tarde confirmado pelo próprio autor que esta versão não difere de forma significativa da versão final, tendo sido apenas corrigidas algumas gralhas, e alterada a formatação de algumas listas.

O standard IEC 61131-3, que por motivos de brevidade será a partir deste momento apenas referenciado simplesmente pelo 'standard', define quatro linguagens de programação distintas. Adicionalmente, define ainda uma linguagem de especificação de máquinas de estado, que é considerado por muitos como uma quinta linguagem de programação.

No entanto, e ainda mais importante do que as linguagens de programação, o standard define ainda uma arquitectura de programação dentro da qual se enquadram as referidas linguagens. De facto, o standard define blocos de programação, referenciados como POU - Program Organization Units (Unidades de Organização de Programas) no standard, como a unidade mais básica de composição dos programas. Um bloco é geralmente definido e construído com base em uma das linguagens de programação definidas pelo próprio standard. No entanto, vários blocos podem ser agrupados por forma a formar um outro bloco mais complexo, sendo que o programa final irá ser uma simples composição de vários blocos. Os blocos podem ser de um de três tipos: funções (functions), funções-bloco (function blocks), e programas (programs).

Com um ambiente de programação baseado no IEC 61131-3 são fornecidos vários blocos de programação (geralmente referenciados como blocos básicos),

que implementam funcionalidades utilizadas com frequência em vários programas. Alguns deste blocos (por exemplo, contadores, temporizadores, somadores, etc.) são definidos pelo standard, e deverão obrigatóriamente estar disponíveis para serem utilizados por qualquer programador. Outros blocos (tais como controladores PID, interfaces com módulos de comunicação em rede, etc…) são geralmente fornecidos pelo fabricante com o intuito de facilitar a vida ao programador, ou para permitir a utilização de equipamento fornecido pelo mesmo de forma mais simples e eficaz.

Todas as linguagens de programação suportam a utilização de variáveis com os mesmos tipos de dados (por exemplo inteiros, intervalos de tempo, booleanos, etc.). Ou seja, as variáveis utilizadas em cada linguagem serão sempre de um tipo de entre o mesmo conjunto de tipos. Uma vez que todas as linguagens recorrem ao mesmo conjunto de tipos, a utilização de uma determinada linguagem de programação para definir o funcionamento de um determinado bloco permite que esse mesmo bloco seja invocado por outro bloco programado em uma das restantes linguagens de programação sem se tornar necessário qualquer conversão de dados.

Deverá ser salientado que as linguagens de programação não são totalmente intercambiáveis. Ou seja, embora seja possível re-escrever o mesmo comportamento e lógica de funcionamento em qualquer uma das linguagens de programação, não existe no entanto um mapeamento directo entre as mesmas. Este facto adevem da existência de características específicas em cada linguagem de programação que não encontram paralelo nas restantes. Por exemplo, a linguagem IL (Instruction List) contem uma instrução de salto incondicional (GOTO), a qual não tem correspondente directo na linguagem ST (Structured Text). Como foi referido, a falta desta instrução na linguagem de programação ST não implica que seja impossível de programar um determinado comportamento ou lógica de funcionamento nessa linguagem, no entanto faz com que não seja possível fazer uma conversão directa de um programa escrito numa linguagem para outra linguagem. A falta de conversão directa complica bastante qualquer algoritmo de conversão entre linguagens de programação, pelo que torna-se difícil automatizar este processo de conversão. A conversão, no entanto, é sempre passível de ser efectuada manualmente.

O atrás exposto implica que as linguagens não são idênticas nas características e facilidades de programação oferecidas, o que torna natural que algumas linguagens sejam mais apropriadas para certas tarefas do que outras. Por exemplo, a programação de uma determinada estrutura de controlo contínuo com blocos de controlo PID, limitadores, conversores, etc, será mais facilmente programado com a linguagem gráfica FBD (Function Block Diagram), enquanto que um programa que exija cálculos complexos que não se encontram já programados em blocos básicos será mais facilmente programada recorrendo à linguagem ST.

4.1.2 Blocos de Programação

Como já foi referido, cada bloco de programação será sempre de um de três tipos distintos, nomeadamente funções, funções-bloco, e programas. Todos os três tipos consistem em duas partes distintas: uma primeira parte dedicada à declaração de variáveis, e uma segunda parte com o código que define o algoritmo implementado pelo mesmo bloco.

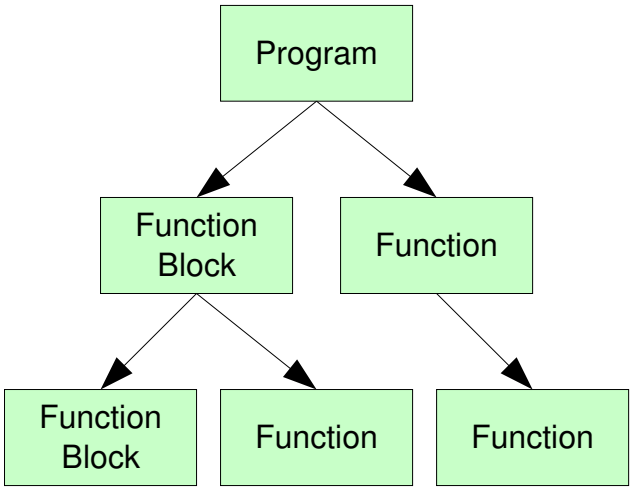

Figura 4.1 - Invocação de funções e funções-bloco a partir de funções, funções-bloco, e programas.

As variáveis poderão ser locais, ou ser utilizadas para passar dados entre o bloco e os restantes blocos que invocam este bloco. As variáveis locais são apenas visíveis dentro do próprio bloco, ou seja apenas poderão ser acedidas pelo código do próprio bloco. Pelo contrário, as utilizadas para passagem de parâmetros são visíveis tanto pelo código do próprio bloco bem como pelo código do bloco que efectua a invocação. Estas variáveis de passagem de parâmetros podem ser apenas de entrada, de saída, ou simultaneamente de entrada e saída. As variáveis de entrada podem apenas ser escritas pelo código que pretende invocar o bloco na mesma altura em que o bloco é efectivamente invocado. Pelo contrário, o código que invoca o bloco apenas poderá ler as variáveis de saída. No entanto, e no caso de estas serem parâmetros de saída de funções bloco, estas variáveis poderão ainda ser lidas em qualquer altura, não estando a sua leitura restrita apenas ao instante após o retorno do bloco invocado.

Já as variáveis ou parâmetros de entrada/saída em simultâneo herdam as características tanto das variáveis de entrada como das de saída. Ou seja, poderão ser escritas pelo código que invoca o bloco no instante em que este é invocado, sendo ainda possível a sua leitura logo após a invocação.

A segunda parte de um bloco de programação consiste no algoritmo especificado recorrendo a uma das linguagens definidas no standard. Este código pode conter invocações a outros blocos, no entanto com a restrição de que funções apenas podem invocar outras funções, enquanto que programas e funções-bloco podem invocar outras funções ou funções-bloco. Em qualquer dos casos enunciados, o

código nunca poderá conter chamadas recursivas ao próprio bloco, seja esta chamada recursiva feita de forma directa, ou de forma indirecta através de outros blocos.

As funções têm semântica semelhante às funções de linguagens procedimentais tradicionais (tais como C, Pascal, Java, etc…). Estas retornam um único valor como resultado da invocação da função. No entanto, e a par de uma ou mais variáveis de entrada, podem ainda retornar mais valores através das variáveis de saída ou de entrada/saída. As funções podem ainda ter variáveis locais. Isto representa uma mudança em relação à primeira versão do standard que apenas permitia que as funções tivessem variáveis de entrada e variáveis locais, estando assim limitadas a retornar um único valor como resultado, sem ter a oportunidade de retornar mais valores através de variáveis de saída.

A semântica das funções exige que estas sejam idem-potentes. Ou seja, uma função deverá retornar sempre os mesmos valores sempre que seja invocada com os mesmos parâmetros, qualquer que seja o estado do restante sistema e ambiente em que se encontra, bem como do instante em que é efectuada a invocação. De facto esta restrição limita-se a definir, de forma indirecta, que uma função não poderá manter variáveis de estado entre duas invocações, nem tão pouco poderá aceder a variáveis globais, sendo assim que o resultado de cada invocação deverá depender apenas dos parâmetros passados durante a invocação, e nunca de o conteúdo de outras variáveis de estado. No entanto, cada vez que é efectuada uma invocação a uma função não é obrigatório especificar o valor de todos os parâmetros de entrada dessa função, sendo apenas obrigatório especificar todos os parâmetros que sejam de entrada/saída em simultâneo. Os parâmetros que não sejam explicitamente especificados tomarão um valor especificado especialmente para estes casos omissos.

O tipo de cada função bloco é de alguma forma semelhante a uma classe de qualquer linguagem de programação orientada a objectos, no sentido de que uma instância de uma função-bloco deverá ser declarada como uma variável. Cada instância de uma função bloco manterá a sua própria cópia das variáveis locais do tipo de função-bloco correspondente, ou seja, cada instância mantém as suas próprias variáveis de estado.

Os tipos de função-bloco são no entanto muito mais restritas do que as classes habituais noutras linguagens de programação. De facto, a herança entre tipos de função bloco não é suportada, e estas têm sempre exactamente uma função membro. Isto significa que todas as funções membro especiais, tais como construtores e destrutores, não são suportadas. A única função membro não retorna qualquer valor ao contrário das funções, podendo no entanto ter variáveis de entrada, saída e entrada/saída à semelhança das funções. Uma vez que cada função bloco tem apenas uma única função membro, a invocação desta função membro é geralmente dita como sendo uma invocação à função bloco em si, pois não existe qualquer dúvida no código cuja execução resultará dessa

invocação. De forma semelhante, os parâmetros desta única função membro são geralmente referidos como sendo os parâmetros da função bloco em si.

Mais uma vez, não é necessário especificar todos os parâmetros de entrada aquando da invocação de uma função-bloco, com a excepção dos parâmetros de entrada/saída em simultâneo que, esses sim, deverão ser sempre especificados em todas as invocações. Mais uma vez, e à semelhança das funções, os parâmetros cujos valores não sejam explicitamente especificados numa determinada invocação tomarão um valor especificado especialmente para estes casos omissos. No entanto, aqui surge uma situação dúbia, na qual alguns locais do standard especificam que será sempre utilizado este valor de inicialização, enquanto noutras secções é referido que, caso a função bloco já ter sido invocada anteriormente, as variáveis de entrada mantêm o mesmo valor que tinham na última invocação anterior (consultar secção 4.3.7. desta tese para mais pormenores).

Já para as variáveis de saída não há duvida que estas deverão manter os seus valores entre duas invocações, pelo que poderão ser lidas a qualquer momento por qualquer código que tenha acesso a essa função bloco. No entanto, apenas o código da própria função-bloco pode alterar os valores mantidos nas variáveis de saída. Pode-se por isso afirmar que os parâmetros de saída das funções bloco são persistentes entra as suas invocações.

As funções blocos poderão ainda ter variáveis locais, às quais apenas o próprio código da função bloco tem acesso (seja ele para escrita ou leitura). Estas variáveis internas podem ser persistentes ou voláteis. As persistentes manterão o seu valor entre duas invocações da função bloco, enquanto que as voláteis são inicializadas com o seu valor a utilizar nos casos omissos, no inicio de cada invocação da função bloco.

Uma vez que as funções têm de ser idem-potentes, estas não podem conter quaisquer variáveis persistentes entre chamadas. Por esta razão, as funções não poderão nem criar instâncias de qualquer tipo de função bloco, nem tão pouco invocar uma instância de uma função bloco que lhe seja passada como parâmetro. Pode, no entanto, aceder às variáveis de saída (que são persistentes) das funções bloco que lhe sejam passadas como parâmetro.

Os tipos de programas são quase em tudo idênticos aos tipos de funções blocos. À semelhança das funções bloco, as instâncias de programas são criadas a partir de um tipo de programa, e têm também uma única função membro. Os programas distinguem-se apenas das funções bloco através dos locais onde instâncias de programas podem ser declarados. Enquanto que as instâncias de funções bloco podem ser declaradas (e definidas simultaneamente) dentro de outras funções bloco ou dentro de programas, as instâncias de programas apenas podem ser declaradas dentro de configurações.

As configurações serão descritas com mais pormenor mais adiante, no entanto é

de salientar desde já que é na especificação destas que se encontra implícito que a execução de os programas por parte dos PLCs se deve efectuar de forma cíclica, ou seja, cada programa executará o seu código ciclicamente desde o inicio até ao fim do mesmo.

```
FUNCTION WEIGH : WORD
  (* Interface *)
  VAR_INPUT
    w_command: BOOL;
    gross_w: WORD;
    tare_w: INT;
  END_VAR
  (* Local Vars. *)
  VAR
    w1, w2, w3: INT;
  END_VAR
  (* Body *)
  (*
      ...
  *)
END_FUNCTION
```

```
FUNCTION_BLOCK FILTER
  (* Interface *)
  VAR_INPUT
    signal_in: INT;
  END_VAR
  VAR_OUTPUT
    signal_out: INT;
  END_VAR
  (* Persistent Vars. *)
  VAR
    s1, s2, s3: INT;
  END_VAR
  (* Temporary Vars. *)
  VAR_TEMP
    t1: REAL;
  END_VAR
  (* Body *)
  (*
      ...
  *)
END_FUNCTION_BLOCK
```

```
PROGRAM AGV
  (* Interface *)
  VAR_INPUT
    fut_dest: loc_type;
    add_dest: BOOL;
  END_VAR
  VAR_OUTPUT
    cur_dest: loc_type;
    dest_reach: BOOL;
  END_VAR
  VAR_IN_OUT
    emg_stop: BOOL;
  END_VAR
  (* Local Vars. *)
  VAR
    (* ... *)
  END_VAR
  (* Body *)
  (*
      ...
  *)
END_PROGRAM
```

Figura 4.2 - Exemplos da declaração de uma função, uma função-bloco, e de um programa.

A figura 4.2 contem exemplos de declarações de uma função, uma função bloco, e de um programa. O texto delimitado por '(*' e '*)' são comentários, e é ignorado pelos compiladores. É de salientar que as declarações de variáveis podem aparecer em qualquer ordem, com variáveis locais, temporárias e de passagem de parâmetros interpostas. Nos exemplos estas aparecem no entanto bem delimitadas por forma a facilitar a sua identificação e compreensão. Embora não apareça explicitado nos exemplos, as linguagens têm no entanto algumas particularidades difíceis de explicar. Por exemplo, não pode ser efectuada a declaração de múltiplas variáveis do tipo 'strings', i.e. cadeias de caracteres, na mesma linha. Ou seja, para declarar várias variáveis do tipo string ter-se-á de declarar cada variável uma a uma, repetindo o tipo dessa variável para cada uma das variáveis. Outra particularidade reside na declaração de variáveis localizadas, i.e. variáveis cuja localização na memória é explicitamente definido pelo programador. Tais como as variáveis do tipo string, estas também apenas poderão ser declaradas uma por linha.

4.1.3 As Linguagens de Programação

O standard especifica quatro linguagens de programação e uma de estruturação de código (a SFC):

ST – Structured Text – Texto Estruturado

IL – Instruction List – Lista de Instruções

LD – Ladder

FBD – Function Block Diagram – Diagrama de Funções Bloco

SFC – Sequential Function Chart – Diagrama Sequencial de Funções

Das linguagens referidas, LD e FBD são linguagens gráficas, enquanto que IL e ST são textuais. A programação de SFC poderá ser efectuada com recurso a uma sintaxe textual ou um diagrama gráfico, embora a primeira opção seja raramente utilizada. É de referir que a linguagem IL é muitas vezes referenciada como STL (Statement List – Lista de 'Statements'). Uma vez que este nome não se encontra definido no standard, a sua utilização não deverá ser encorajada.

A programação em LD é semelhante à concepção de um circuito eléctrico baseado em relés. De facto, esta linguagem foi criada exactamente com o propósito de conseguir esta semelhança, e assim aumentar a sua aceitação por parte de electricistas que nunca tenham escrito programas, mas que no entanto tenham experiência em desenvolver circuitos de controlo com memória baseados apenas em relés. Pode-se assim dizer que esta linguagem existe principalmente por motivos históricos, pois a aceitação desta linguagem por parte dos electricistas, embora ainda mantenha uma importância relativa, foi de fundamental importância aquando da introdução dos PLCs no mercado, uma altura em que o controlo das maquinas era efectuado por circuitos baseados em relés.

Um programa LD é assim composto por circuitos horizontais que estabelecem a ligação entre dois barramentos verticais que conduzem a 'electricidade' e a distribuem por todos os circuitos. Um programa LD fica assim com o aspecto de um escadote, com vários 'degraus' horizontais todos eles ligados a dois barramentos verticais. Por este motivo é comum dar o nome de degrau a cada ligação horizontal do circuito completo.

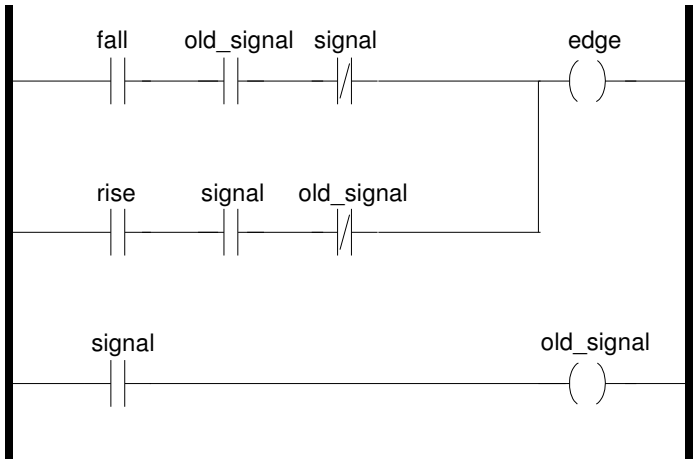

Figura 4.3 - Exemplo de um programa em LD.

Cada degrau consiste então em um ou mais contactos de relés, ligados em série e/ou paralelo, consoante a expressão lógica que se queira implementar, e ainda de pelo menos uma bobine de excitação de relé, representando a variável na qual será armazenada o resultado da expressão lógica expressa pelo degrau. A semântica dos contactos assemelha-se ao funcionamento dos relés, nos quais a electricidade é permitida fluir de um lado ao outro do contacto consoante a bobine (neste caso a variável booleana associada ao contacto) esteja activa ou inactiva. De forma semelhante, a semântica de uma bobine assemelha-se ao funcionamento das bobines de um relé, sendo que se esta recebe electricidade do seu conector à sua esquerda, a electricidade continua a fluir até ao seu conector à direita, e em simultâneo a variável booleana associada ao relé é activada. Através de uma ligação apropriada dos contactos e das bobines, e pela escolha acertada das variáveis booleanas associadas a cada contacto e bobine, torna-se assim possível implementar qualquer circuito lógico.

O standard extendeu ainda a linguagem por forma a permitir que sejam efectuadas chamadas a subrotinas, ou mais precisamente, sejam efectuadas chamadas a outros blocos, sejam eles funções ou funções-bloco. O standard prevê ainda o tratamento de variáveis mais complexas do que as simples variáveis booleanas tradicionalmente utilizadas nos circuitos eléctricos, tais como inteiros ou mesmo intervalos de tempo ou tempo absoluto.

O standard define várias alternativas de contactos e de bobines. Existem contactos normalmente abertos, normalmente fechados, e activos aos flancos. Quanto às bobines, para alem das normais, são ainda aceites bobines negadas, de activação e desactivação (semelhantes às entradas de set e reset de um flip-flop), e ainda activadas apenas aos flancos.

Os FBD gráficos estão no extremo oposto, sendo uma linguagem de alto nível na qual o programador se limita a colocar caixas que representam funções e funções bloco, e a definir a troca de dados entre estas através de linhas a ligar as entradas e saídas das respectivas caixas. Os dados são transferidos das saídas para as entradas sem nenhuma retenção associada. Adicionalmente o standard especifica que um determinado bloco só pode ser avaliado após todas as suas entradas terem sido actualizadas com os seus valores.

As entradas são representadas à esquerda das caixas, enquanto que as saídas aparecem á direita. Os parâmetros que são simultaneamente entrada e saída aparecem nos dois lados, sendo que as duas conexões se encontram ligadas por um traço no interior da própria caixa. Uma vez que o standard define blocos básicos tais como contadores e temporizadores, programar nesta linguagem assemelha-se ao desenvolvimento de um circuito digital baseado em integrados com operações lógicas, contadores, temporizadores, flip-flops, etc…

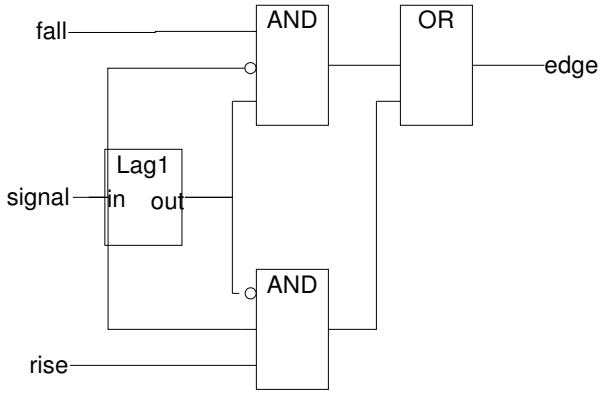

Figura 4.4 - Exemplo de um programa em FBD.

A parte do código de uma função bloco poderá conter várias redes completas, em que cada rede é composta por vários blocos a trocar dados entre si, e sem que haja troca de dados entre quaisquer blocos de redes distintas. Nestes casos, a ordem pela qual as redes são avaliadas é ou aleatória, ou dependente da implementação. O standard aceita ainda que a ordem de avaliação seja especificada pelo programador, sendo isto efectuado por algum mecanismo que é mais uma vez dependente da implementação, e que por isso resulta em código não portável.

Cada rede é avaliada de acordo com as seguintes regras:

- Um bloco é apenas avaliado após todas as suas entradas terem sido actualizadas;
- A avaliação de uma rede só termina após todas as suas saídas terem sido actualizadas.

Redes com ciclos fechados são permitidas. Isto é, redes nas quais a saída de um bloco esteja ligado a uma entrada do mesmo bloco, seja esta ligação efectuada de forma directa ou através de outras entradas e saídas de outros blocos da mesma rede. Nestes casos a sequência de avaliação é mais uma vez aleatória, sendo que a primeira regra atrás enunciada terá de ser quebrada para o primeiro bloco com ciclo a ser avaliado. No entanto, o standard prevê ainda um expediente para especificar qual o bloco a ser avaliado primeiro. O expediente baseia-se na quebra do ciclo por parte do programador no local imediatamente antes do bloco que o programador pretende que seja avaliado inicialmente. A ligação deverá depois ser reposta através de uma variável, sendo que a saída irá armazenar o seu valor nessa variável, e a entrada irá obter o valor à mesma variável. A introdução explícita da variável permite assim introduzir uma retenção numa ligação que não tinha nenhuma retenção associada por omissão.

Infelizmente o standard também permite que o programador especifique qual o bloco a ser avaliado no inicio recorrendo mais uma vez a algum mecanismo que seja dependente da implementação, tornando assim o código não portável entre fabricantes de PLCs.

A cada rede poderá ser associado um nome terminado por dois pontos ‘:’. Isto permite que a sequência de avaliação seja transferida de forma explícita de uma rede para outra. A transferência de controlo (um salto condicional ou incondicional) é indicado por uma linha que toma valores booleanos, sendo esta linha terminada por uma dupla seta apontando para o nome da rede para qual deverá ser transferido o controlo de execução. O salto é apenas executado se a linha estiver com o valor lógico Verdadeiro.

O fim da execução de uma função bloco cuja parte de código se encontra definida através de uma FBD é declarada através de forma semelhante à transferência de controlo. Neste caso usa-se também a dupla seta, mais uma vez ligada a uma linha que toma valores booleanos, mas deverá agora apontar para o símbolo '<RETURN>'. O retorno é também ele apenas efectuado se o valor lógico na linha for Verdadeiro.

A linguagem SFC é utilizada para especificar máquinas de estado, sendo baseada no Grafcet (também ele estandardizado pelo IEC no standard 69848). O IEC 61131-3 prevê duas sintaxes distintas para a programação com esta linguagem. Uma sintaxe baseada em texto, e ainda outra baseada em gráficos. A primeira é raramente, se alguma vez, utilizada, pelo que nas referencias à linguagem SFC é geralmente implícita a utilização da sintaxe gráfica.

No SFC, tal como no Grafcet, os estados da máquina de estados são conhecidos por etapas. Ligações direccionadas entre as etapas especificam a sequência de activação destas ultimas. Entre quaisquer duas etapas que se encontram conectadas, deverá sempre existir uma, e uma só, transição. De forma semelhante, entre quaisquer duas transições deverá existir sempre uma etapa. Uma transição é utilizada para controlar o instante em que se efectua a transição de activação entre duas etapas.

Diz-se que uma transição está sensibilizada quando todas as suas etapas precedentes estão simultaneamente activas. Uma transição só poderá disparar quando se encontra sensibilizada, e simultaneamente quando uma condição lógica a ela associada se satisfaz (i.e. apresenta um valor lógico verdadeiro). O disparo de uma transição resulta na desactivação de todas as etapas precedentes, e a activação de todas as suas etapas sucessoras.

Uma ou mais acções podem ser associadas a cada etapa. As acções são geralmente apenas executadas enquanto a etapa está activa. Algumas acções retardadas podem no entanto vir a ser activadas após a etapa já estar desactivada, sendo no entanto despoletadas apenas pela activação dessa mesma etapa.

As condições associadas às transições, bem como as acções associadas às etapas, deverão ser definidas recorrendo a uma das restantes quatro linguagens de programação definidas no standard (ie. ST, IL, LD ou FBD). As acções poderão ainda ser definidas recorrendo a um outro SFC. Conclui-se assim que não é

possível escrever um programa completo utilizando apenas o SFC, sendo sempre necessário recorrer a pelo menos mais uma das outras quatro linguagens de programação.

A linguagem IL é uma linguagem de programação de baixo nível, assemelhando-se bastante ás linguagens máquina utilizadas para a programação de micro-processadores ou micro-controladores. De facto, um programa IL consiste numa lista de instruções, sendo que cada instrução é geralmente acompanhada por um único parâmetro. Salvo alguns casos especiais, a instrução especifica uma operação a ser efectuada sobre o valor armazenado num registo comum, e ainda sobre o parâmetro anexo à instrução. O resultado da operação é novamente armazenado no registo comum, ficando assim disponível a ser utilizado pela instrução seguinte. À semelhança de LD, esta linguagem tem as suas raízes em linguagens semelhantes suportadas pelos primeiros PLCs, tendo sido no entanto melhorada ao longos dos tempo até que à versão estandardizada que suporta já chamadas a subrotinas, e o acesso a variáveis mais complexas que os simples valores booleanos.

```
LD fall
AND old_signal
ANDN signal
OR (
LD rise
AND signal
ANDN old_signal
)
ST edge
LD signal
ST old_signal
```

Figura 4.5 - Exemplo de um programa em IL.

Por ultimo, a linguagem ST está vocacionada para ser utilizada por programadores com maiores conhecimentos de informática. É uma linguagem procedimental, consistindo numa sequência de declarações. As declarações mais simples são as de atribuição, compostas por um lado esquerdo, um sinal de atribuição, e um lado direito que pode conter uma expressão. As próprias expressões, a par das operações lógicas e aritméticas, podem ainda conter chamadas a funções. Entretanto, chamadas a funções bloco, as quais não retornam qualquer valor, constituem elas próprias uma declaração completa. São ainda suportadas declarações de controlo de fluxo tais como IF THEN, ELSE e CASE, e ainda as declarações de iteração FOR, WHILE e REPEAT.

Estas duas últimas linguagens textuais, ST e IL, não são de momento discriminadas com mais pormenor pois serão novamente abordadas em secções posteriores, uma vez que foram estas as linguagens para as quais foi desenvolvido um compilador, e consequentemente para as as quais foi efectuada uma análise mais crítica das mesmas.

```
fedge := fall AND (old_signal AND NOT signal);
redge := rise AND (signal AND NOT old_signal);
edge := fedge OR redge;
old_signal := signal;
```

Figura 4.6 - Exemplo de um programa em ST.

4.1.4 Tipos de Dados

Como já foi referido, todas as linguagens de programação definidas no standard suportam os mesmos tipos de dados. Os tipos de dados elementares suportados, em conjunto com o numero de bytes ocupados por cada variável desse tipo, encontram-se discriminados na tabela da fig. 4.7. É de salientar que o standard prevê tipos de dados para armazenar intervalos de tempo, bem como tempos absolutos. A resolução suportada por estes tipos de dados é dependente da implementação, pelo que o espaço ocupado em memória será também ele dependente da implementação. O tipo TIME_OF_DAY é utilizado para valores absolutos do tempo de um qualquer dia, tal como a hora de fazer soar a sirene do almoço. Entretanto, o tipo de dado TIME é utilizado parta armazenar intervalos de tempo, tais como períodos de execução, tempo que demora uma qualquer operação, ou diferenças entre dois TIME_OF_DAY.

Tipo de Dado	*Dimensão (bytes)*	*Valor Inicial*
Boolean		FALSE
Byte	1	0
Word	2	0
DWord	4	0
LWord	8	0
SINT / USINT	1	0
INT / UINT	2	0
DINT / UDINT	4	0
LINT / ULINT	8	0
Real	4	0
LReal	8	0
String	1 (por caracter)	'' (empty string)
WString	2 (por caracter)	“” (empty string)
Time	(depende da implementação)	T#0S
Time_of_Day	(depende da implementação)	TOD#00:00:00
Date	(depende da implementação)	D#0001-01-01
Date_and_Time	(depende da implementação)	DT#0001-01-01-00:00:00

Figura 4.7 - Tipos de dados elementares.

Todos os tipos de dados têm um valor com o qual as variáveis são inicializadas quando estas são criadas. Este valor é apenas utilizado em casos omissos, ou seja

é apenas utilizado quando a declaração da variável não contém ela própria o valor com o qual essa variável deverá ser inicializada. O valor inicial associado a cada tipo de dado é ainda utilizado nas chamadas a funções, nos casos em que um parâmetro não é atribuído nenhum valor de forma explícita. De forma semelhante, este valor inicial é ainda utilizado nas chamadas a funções-bloco nos parâmetros não especificados, mas agora apenas na primeira vez em que a função bloco é chamada, uma vez que chamadas posteriores irão utilizar os valores das chamadas anteriores para os parâmetros não especificados de forma explícita.

São ainda suportados tipos de dados derivados. O mais simples tipo de dado derivado é uma simples cópia de um tipo de dado previamente definido (podendo ser um tipo de dado elementar), sendo apenas re-definido um novo valor para o valor com o qual deverão ser inicializadas (nos casos omissos) as variáveis deste novo tipo.

O programador pode ainda definir tipos de dados derivados baseados em sub-gamas (subranges), enumerações, arrays e estruturas. O standard especifica apenas uma sintaxe textual através da qual o programador poderá definir tipos de dados derivados. Isto implica que a programação de blocos recorrendo a linguagens gráficas pressupõe uma prévia declaração, em linguagem textual, dos tipos de dados derivados que esse código possa vir a utilizar. Na realidade os ambientes de desenvolvimento existentes permitem ao programador declarar os tipos de dados derivados recorrendo a uma interface gráfica (por exemplo, recorrendo a menus e listas). Esta prática não está, como é obvio, estandardizada, pelo que dificulta mais uma vez a portabilidade dos programas.

Tipos de dados derivados baseados em sub-gamas apenas podem ser definidos com base em tipos de dados inteiros elementares. Isto significa que não é possível definir uma sub-gama tendo como tipo de dado base uma outra sub-gama, nem tão pouco usando como base um tipo de dado derivado de um tipo de dado inteiro elementar. Uma sub-gama restringe a gama de valores que uma variável deste tipo derivado pode tomar, sendo úteis para a detecção rápida de erros. Este tipo de dados pressupõe que o programador defina os novos limites, sendo assim que a gama de valores aceitáveis será sempre um intervalo continuo de inteiros numeráveis, e nunca uma conjunção de vários intervalos contínuos. O valor utilizado para a inicialização de variáveis nos casos omissos será o do limite inferior da sub-gama, a não ser que na própria definição do novo tipo de dado seja especificado um outro valor para utilizar nesses casos omissos.

Enumerações são tipos de dados derivados que consistem na definição explícita, em forma de lista, de todos os valores que as variáveis deste novo tipo poderão tomar. Caso não seja especificado de forma explícita um outro valor, o valor que deverá ser dado às variáveis nos casos omissos será sempre o do primeiro valor da lista na enumeração.

Os valores serão sempre identificadores (ou seja, sequências de caracteres que

sejam também válidas para usar como nome de variáveis, de funções, etc…). Os identificadores utilizados como valores possíveis dentro de uma enumeração são ainda passíveis de serem re-utilizados para identificar outras entidades tais como variáveis, funções, funções bloco e programas. Nestes casos poderão existir situações em que se crie ambiguidade em relação à qual das duas entidades está a ser referenciada, o valor da enumeração, ou a variável ou função, função-bloco, ...). Com o intuito de permitir ultrapassar estas situações de ambiguidade, o standard prevê uma sintaxe através da qual se poderá qualificar de forma explícita que um identificador referencia um valor de uma enumeração. Pressupõe-se assim que, por omissão, e em casos de ambiguidade, o compilador deverá tomar como sendo referenciada a variável (ou função, função-bloco, …), e não o valor da enumeração.

As estruturas são tipos de dados derivados que consistem numa agregação de várias variáveis, podendo cada uma destas variáveis ser de tipos de dados distintos. Pelo contrário, um array consiste numa série de variáveis, todas do mesmo tipo. Cada variável da série pode ser referenciada de forma independente através de um índice numérico. Por estranho que possa parecer, o standard não prevê no entanto a possibilidade de declarar arrays de funções bloco, uma vez que uma função bloco tipo não é considerado um tipo de dado derivado.

4.1.5 Localização de Variáveis

Antes da existência do standard IEC 61131-3, os PLCs tinham geralmente a sua memória dividida em blocos, sendo ainda as entradas e as saídas físicas mapeadas num determinado bloco de memória. Esta arquitectura permitia o acesso às entradas e saídas físicas do PLC através de um simples acesso a uma zona especifica de memória, não sendo por isso necessário introduzir instruções e operações especificas para o acesso ás interfaces com o exterior. Adicionalmente, quando o programador necessitava de uma variável, este teria de escolher uma posição de memória na qual esta seria armazenada, e partir desse momento era esperado que acedesse directamente a essa zona de memória sempre que quisesse aceder a essa variável. Na realidade, não existia o conceito de variável, sendo que era esperado que o programador acedesse sempre de forma directa à memória, cabendo-lhe a ele a tarefa de organizar a memória estabelecendo a finalidade de cada byte.

O standard IEC 61131-3 substitui este funcionamento com um modelo no qual o programador especifica apenas as variáveis de que irá necessitar, cabendo a partir daí ao compilador a responsabilidade de organizar a memória. Simplifica-se assim a tarefa do programador quanto à organização de memória, dificultando no entanto a tarefa de aceder ás entradas e saídas físicas que se encontravam mapeadas em determinadas zonas de memória.

Para ultrapassar este ultimo obstáculo, o standard permite que o programador

possa definir de forma explícita a localização na memoria na qual deverá ser armazenada uma determinada variável. Bastará assim ao programador colocar uma variável numa zona de memória na qual está mapeada uma entrada para que possa ler a entrada física simplesmente através da leitura da variável que lá está armazenada.

Este mapeamento directo de uma variável numa determinada zona de memória não é no entanto permitido para as variáveis locais de funções, isto porque o seu acesso iria interferir com a idem-potencia da função. No entanto, variáveis que tenham sido mapeadas directamente na memória podem ainda assim ser passadas como parâmetro à função, sendo que agora o seu valor é considerado um valor de entrada explícito da função, o que não destrói a sua propriedade de idem-potencia.

As variáveis podem ainda ser declaradas como sendo ou não de retenção. Variáveis de retenção deverão manter o seu valor intacto, mesmo que o PLC venha a ser desligado e ligado novamente. Pelo contrário, as variáveis sem retenção estão garantidas a serem inicializadas de cada vez que o PLC é ligado ou re-iniciado.

4.1.6 As Configurações

Com os blocos até agora descritos (ou seja as funções, funções bloco e os programas), já é possível escrever um programa completo para um PLC. No entanto, e embora isto seja suficiente para muitas aplicações simples que necessitam de apenas um único programa, aplicações mais complexas que requerem vários programas a executar em paralelo não são possíveis de implementar. Adicionalmente, não será possível tirar proveito de PLCs com mais do que um CPU, uma vez que com só um programa apenas um desses CPUs ficaria ocupado.

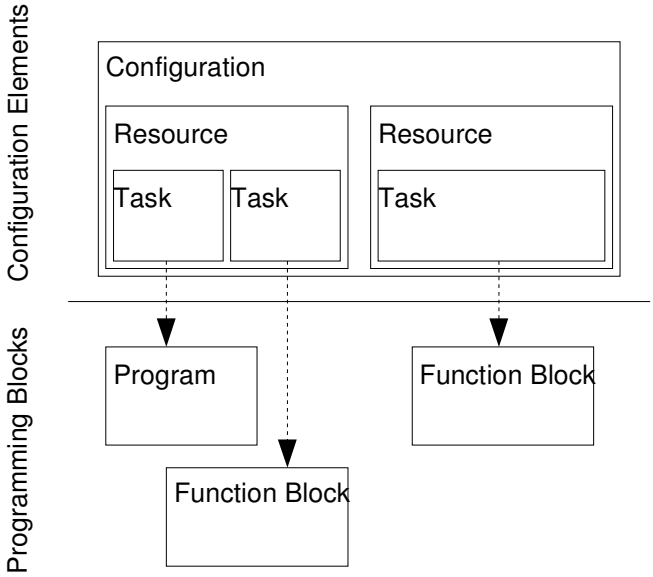

Figura 4.8 - Os componentes de uma configuração.

O standard tem estas restrições em conta ao definir uma nova abstracção de modelação a um nível superior ao dos programas. Esta nova abstracção, a configuração, permite ao programador especificar a existência de vários programas a executar em paralelo, e assim tirar total proveito dos PLC com múltiplos CPUs. Uma configuração contem informação suficiente para especificar de forma completa quais os programas que deverão executar em quais CPUs, bem como novas variáveis globais. Não contem código executável, mas antes limita-se a instanciar os programas e/ou funções bloco que sejam necessários para a aplicação, criar tarefas, e a fazer a atribuição de cada programa a uma das tarefas.

De forma mais específica, uma configuração contem a definição de todos os recursos necessários para a aplicação, e o tipo de cada recurso. Na realidade, os recursos representam os CPUs de um PLC, sendo que os tipos deste recursos variam com o modelo do CPU fornecido pelo fabricante. É então suposto que cada fabricante forneça uma lista dos tipos de recursos que os seus PLCs suportam.

Em cada recurso podem ser criadas uma ou mais tarefas. As tarefas são de alguma forma semelhantes a processos de um sistema operativo, sendo no entanto intencionalmente muito mal definidos pelo standard que se limita a especificar o comportamento das tarefas, sem definir como estas deverão ser implementadas. As tarefas podem ser configuradas de tal forma que sejam executadas periodicamente, ou então no flanco ascendente de uma qualquer variável booleana.

O número de tarefas que um determinado tipo de recurso permite, bem como os parâmetros que podem ser atribuídos as essas tarefas (por exemplo, o período de execução), é dependente da implementação. Por outras palavras, o número máximo de tarefas e as características que estas podem tomar depende das capacidades do CPU que as irá executar. Muitos CPUs simples limitam-se a suportar uma única tarefa fixa e pré definida, enquanto que outros poderão suportar um numero fixo de 3 ou 4 tarefas. Actualmente poucos CPUs permitem que o programador defina um elevado numero de tarefas, o que é no entanto permitido pelo standard.

Cada tarefa é configurada para executar um ou mais programas/funções bloco que tenham já sido instanciados dentro da configuração. Para além dos programas e funções blocos, é ainda possível instanciar variáveis visíveis por toda a configuração, ou então dentro de um determinado recurso (i.e. CPU) dessa configuração, sendo que assim esses variáveis apenas poderão ser utilizadas pelo recurso em que foram instanciadas.

O standard refere-se às variáveis instanciadas dentro de uma configuração (ou de um recurso de uma configuração), como sendo de variáveis globais. Estas podem ser acedidas por programas ou funções bloco instanciadas na mesma

configuração (ou no mesmo recurso). No entanto, do ponto de vista dos programas ou funções bloco, estas variáveis são externas, e devem por isso ser re-declaradas dentro desses programas ou funções bloco como sendo variáveis externas. As variáveis externas são assim apenas referências a variáveis globais, não sendo na realidade variáveis no sentido de conter uma zona de memória no qual se possa armazenar valores. O mapeamento entre uma variável externa e uma variável global é efectuado aquando da instanciação do programa ou função-bloco dentro da configuração.

4.1.7 A Linguagem ST

Um bloco de código ST consiste numa sequência de declarações, cada uma terminada por um ponto e virgula ';'. As declarações mais simples são as chamadas a funções bloco, e as declarações de atribuição (i.e. a atribuição do valor de uma expressão a uma variável).

As variáveis podem ser substituídas por expressões na maioria dos casos, com excepção ao lado esquerdo de uma declaração de atribuição. Por exemplo, uma expressão pode aparecer como um parâmetro a ser passado a uma função ou função bloco aquando da sua invocação, ou mesmo dentro de o índice de um array. As expressões podem conter operações lógicas (incluindo o ou exclusivo), comparações, operações aritméticas (incluindo a operação de exponenciação), e ainda invocações a funções. A avaliação das expressões é efectuada por ordem decrescente de precedência das operações, e dentro destas da esquerda para a direita. A avaliação de expressões lógicas termina logo que o resultado da expressão tenha sido determinado. Isto significa que, por exemplo, numa expressão com uma operação lógica, o segundo operando dessa operação lógica não chegará a ser avaliado se a avaliação do primeiro operando resultar no elemento absorvente da operação lógica em causa.

A invocação de funções ou funções bloco pode ser declarado com uma de duas sintaxes: uma sintaxe de invocação formal, e outra não formal. Na sintaxe não formal, os parâmetros são atribuídos valores através de uma lista de expressões, separadas por virgulas, pela mesma ordem pela qual os parâmetros se encontram declarados na função ou função bloco que se encontra a ser invocada. Neste caso, é obrigatório atribuir valores a todos os parâmetros da função, sejam eles parâmetros de entrada, de saída, ou de entrada e saída em simultâneo.

Na invocação formal os parâmetros são atribuídos valores através de uma lista de declarações de atribuição, sendo estas declarações mais uma vez separadas por virgulas. Cada declaração de atribuição dessa lista começa com o nome do parâmetro em causa, seguido do símbolo de atribuição ('=>' para parâmetros de saída, e ':=' para parâmetros de entrada e entrada/saída), e terminado pelo valor, expressão ou nome da variável a atribuir a esse parâmetro da função que se encontra a ser invocada.

Nestas invocações com sintaxe formal, não é necessário atribuir valores a todos os parâmetros da função a ser invocada, nem tão pouco que a sequência de atribuições siga a mesma ordem pelo qual os parâmetros foram declarados. É, no entanto, obrigatório especificar variáveis para todos os parâmetros de entrada/saída em simultâneo. Para os parâmetros de saída para aos quais não é atribuída nenhuma variável, os valores resultantes da invocação serão simplesmente descartados. No caso dos parâmetros de entrada aos quais não são atribuídos valores de forma explícita, o funcionamento depende se está a ser invocada uma função ou uma função bloco. No caso das funções os parâmetros de entrada aos quais não é atribuído nenhum valor tomarão os seus valores de inicialização em casos omissos. Já no caso das funções bloco, o standard é algo ambíguo, uma vez que numas secções é especificado que os parâmetros de entrada tomarão o valor que mantinham da invocação anterior, ou o valor de inicialização em casos omissos caso ainda não tivesse havido nenhuma invocação anterior, enquanto que noutras secções é especificado que estas tomarão sempre o valor de inicialização nos casos omissos.

As invocações a funções e a funções bloco distinguem-se ainda pelo facto de que as invocações a funções têm forçosamente de ter pelo menos um parâmetro, enquanto que as invocações a funções bloco podem ser efectuadas com uma lista de parâmetros vazia.

A linguagem inclui duas declarações de selecção: IF e CASE. São ainda suportadas três declarações de iteração: FOR, WHILE e REPEAT. A semântica de todas as declarações de iteração e selecção são semelhantes às declarações equivalentes de outras linguagens procedimentais tais como C, Java e Pascal. As suas sintaxes poderão ser descritas em formato BNF como indicado na figura 4.9.

As restantes duas declarações da linguagem ST são utilizadas para transferir o fluxo de controlo. A declaração RETURN transfere o fluxo de controlo para o código que invocou o bloco no qual se encontra esta declaração. A declaração EXIT é utilizada dentro de uma declaração de iteração, sendo que transfere o fluxo de controlo para o exterior da declaração de iteração dentro da qual se encontra.

Uma consideração importante é o facto de que os programas dos PLCs executam de forma cíclica, e deverão por isso executar até ao fim o mais rapidamente possível para que o novo ciclo possa iniciar. Por esta razão, as declarações de iteração WHILE e REPEAT não devem ser utilizadas para sincronizar com um qualquer evento. Ou seja, a condição que determina o fim destes ciclos de iteração não deverá depender de factores exteriores ao próprio ciclo de iteração. Por exemplo, não será permitido que estas declarações de iteração apenas terminem quando uma qualquer entrada física do PLC mude de estado. A sincronização do programa com eventos exteriores deverá ser, em alternativa, especificado com recurso a um SFC (com as condições associadas ao disparo de transições). Isto pois a implementação de um SFC por parte do PLC se baseia

tipicamente no teste das condições associadas às transições em cada ciclo, em vez de interromper a execução do ciclo à espera que o evento ocorra.

```
IF <expression> THEN <statement_list>
{ELSEIF <expression> THEN <statement_list>}
[ELSE <statement_list>]
END_IF

CASE <expression> OF
<case_list> ':' <statement_list>
{<case_list> ':' <statement_list>}
[ELSE <statement_list>]
END_CASE

case_list := <value> {, <value>}
FOR <control_variable> ':=' <expression> TO <expression>
[BY <expression>]
DO <statement_list>
END_FOR

WHILE <expression> DO <statement_list> END_WHILE

REPEAT <statement_list> UNTIL <expression> END_REPEAT
```

Figura 4.9 - Sintaxe das declarações de iteração.

4.1.8 A linguagem IL

Um programa em IL consiste numa lista de instruções, uma instrução por linha, sendo estas executadas pela mesma ordem em que são declaradas.

Opcionalmente, cada instrução pode ainda ser precedida por um identificador que identifica a instrução, e que será utilizada pelas instruções de transferência de fluxo de controlo para identificar a instrução para a qual este deverá ser transferido.

As instruções podem ainda ser seguidas de um modificador de instrução e/ou um operando. Nem todas as instruções podem ser objecto dos modificadores de instruções. De forma semelhante, a existência de operando pode ser obrigatória para algumas instruções, enquanto que para outras o operando pode ser opcional ou mesmo até ilegal.

Tipicamente, cada instrução executa uma operação sobre o valor armazenado num registo fixo, e ainda potencialmente sobre o valor no operando, sendo o resultado armazenado no mesmo registo fixo, pronto para ser utilizado pela instrução seguinte. Embora dependendo da instrução em causa, os operandos podem ser um nome de uma variável, uma constante, ou outras entidades.

As instruções mais básicas limitam-se a copiar o valor no operando para o registo fixo (LD), ou o valor no registo fixo para o operando (ST). Entretanto, as instruções set (S) e reset (R) colocam o operando com o valor lógico Verdadeiro

e Falso respectivamente, caso o registo fixo contenha o valor Verdadeiro, caso contrário estas instruções não têm qualquer efeito, deixando o operando com o mesmo valor que tinha antes da instrução.

Instruções que executam operações lógicas (AND, OR, XOR, ANDN, ORN e XORN) operam sobre o valor no registo fixo e no operando, sendo o resultado armazenado no registo fixo. O mesmo ocorre com as instruções aritméticas (ADD, SUB, MUL, DIV, e MOD) e de comparação (EQ, NE, GT, GE, LT, LE). No entanto, e para estas instruções lógicas, aritméticas e de comparação, o tipo de dado armazenado no registo, para além de ser compatível com a instrução em si, terá também de ser compatível com o tipo de dado especificado no operando. Por exemplo, não é permitido efectuar operações lógicas quando o registo fixo não tem nele armazenado um valor lógico. Adicionalmente, será também um erro efectuar uma operação aritmética se os tipos de dados armazenado no registo e no operando não forem do mesmo tipo de dados, e simultaneamente um tipo de dados numérico para o qual a operação aritmética faça sentido.

As instruções lógicas, aritméticas e de comparação podem ser seguidas do modificador '(', sendo que assim o operando da instrução deixa de ser necessário. Neste caso o valor no registo fixo é temporariamente transferido para uma pilha FILO (first-in-last-out) e a operação suspensa, até que a instrução ')' correspondente seja atingida. Entre o modificador '(' e a instrução ')' poderão ser efectuadas quaisquer conjunto de instruções IL, sendo que quando a instrução ')' é encontrada a instrução inicialmente suspensa é então efectuada tendo como parâmetros o valor no registo fixo, e ainda o valor que foi inicialmente transferido para a pilha FILO.

A sequência de execução das instruções pode ser modificada através da instrução JMP ou RET. A primeira destas deverá ter como operando o identificador (i.e. conhecido em Inglês por 'label') que identifica a instrução para a qual deverá ser transferida a sequência de execução. Por sua vez, a instrução RET não necessita de operando, e limita-se a transferir a sequência de execução para o código que invocou o bloco na qual se encontra a instrução RET. Em ambos estes casos, as instruções podem vir a ser seguidas pelo modificador 'C' ou CN, resultando nas instruções JMPC, JMPCN, RETC, ou RETCN. Nestes casos as instruções apenas transferem a sequência de controlo caso o registo fixo tenha o valor apropriado (Verdadeiro para o caso de C, e Falso para o caso de CN), caso contrário são simplesmente ignoradas.

A chamada de funções em IL utiliza uma sintaxe na qual o nome da função ocupa o lugar da instrução, sendo depois seguida de uma lista dos parâmetros. Tal como em ST, são aceites no entanto qualquer uma de duas sintaxes alternativas: a sintaxe de invocação formal, e a de invocação não formal. A invocação formal e não formal seguem as mesmas regras já enunciadas para ST, diferindo apenas no facto de que para a invocação não formal, o primeiro parâmetro é obtido a partir do valor armazenado no registo fixo, e não da lista de

parâmetros. Em qualquer uma das sintaxes de invocação de funções, o resultado dessa invocação será sempre armazenado no registo fixo.

```
(* Calling function LIMIT *)
(* assume LIMIT requires 3
 * parameters...
 *)

(* Using formal argument list *)
LIMIT (
  in  := 1
  min := 2
  max := 3,
)
ST result

(* Using non-formal argument list *)
LD 1
LIMIT 2, 3
ST result
```

Figura 4.10 - A chamada de funções em IL.

Por outro lado, a invocação de funções bloco pode tomar uma de quatro sintaxes alternativas. Três destas sintaxes requerem o uso de uma instrução específica – CAL. Tal como as instruções RET e JMP, também a instrução CAL pode ser modificada pelos modificadores 'C' ou 'CN', sendo que os efeitos são em tudo idênticos aos efeitos introduzidos nas instruções RET e JMP.

Usando a instrução CAL, os parâmetros utilizadas para a invocação da função bloco podem ser declarados recorrendo a: (1) uma lista de parâmetros formais, (2) uma lista de parâmetros não formais, e (3) inicialização prévia dos parâmetros de entrada com os valores apropriados (recorrendo á instrução ST), seguida da leitura explícita dos parâmetros de saída após a invocação (recorrendo à instrução LD).

Como seria de esperar, a utilização da listas de parâmetros formais e não formais segue as mesmas regras de invocação de funções bloco em ST, usando também as sintaxes de invocação formais e não formais. Deve ainda ser salientado que a opção (3) é uma excepção à regra atrás mencionada de que a alteração dos parâmetros de entrada de uma função bloco só se pode efectuar aquando da sua invocação.

A quarta e última opção de invocar funções bloco recorre a instruções especiais, e só é válida para funções bloco dos seguintes tipos de função bloco: R_TRIG e F_TRIG (detectores de flancos ascendente e descendente, respectivamente), CTU, CTD, CTUD (contadores crescente, decrescente, e bidireccional respectivamente), TP, TON, TOFF (temporizadores de impulso, atraso à ligação, e atraso ao desligar, respectivamente), SR, RS (flip-flops bi-estáveis, com prioridade á activação e desactivação respectivamente). Todos estes tipos de

funções bloco encontram-se definidos no próprio standard, fazendo assim parte de uma biblioteca de funções bloco mais vasta, que deverá ser parte integrante de todos os ambientes de trabalho e PLCs compatíveis com o standard. Como foi referido, esta quarta opção para invocar funções bloco recorre a instruções especiais, que coincidem com o nome de parâmetros das funções bloco atrás descritas. Assim, as instruções S, S1, R e R1 são usadas na invocação dos flip-flops, CLK na invocação dos detectores de flanco, CU, CD, R e PV na invocação dos contadores, e IN e PT na invocação dos temporizadores. Estas instruções especiais substituem a instrução CAL, e têm como único operando o nome da função bloco a ser invocada. O parâmetro cujo nome coincide com a instrução toma o valor que se encontra no registo fixo, e a função bloco é invocada. Assim sendo, esta quarta opção apenas permite especificar o valor de um único parâmetro aquando da invocação da função bloco, com os restantes parâmetros a manterem os mesmos valores da invocação anterior. Isto implica que com este método apenas um único parâmetro pode ter o seu valor alterado entre duas invocações consecutivas.

4.2 O Compilador de IL e ST

No âmbito do projecto MatPLC, e com o intuito de facilitar a sua utilização e programação por parte dos seus utilizadores, foi desenvolvido um compilador [18] das linguagens textuais IL e ST definidas no standard IEC 61131-3. Este compilador executa em 3 passagens: mapeamento do programa textual para uma representação interna, análise semântica do programa, e geração do código. Em cada uma das passagens o programa a ser compilado é analisado de uma ponta à outra, e os dados gerados por essa passagem é utilizada como entrada da passagem seguinte.

Na primeira passagem o programa IL ou ST, gerado pelo programador e expresso em formato textual de acordo com as normas de sintaxe definidas no standard, é analisado e transformado num formato interno. Este formato interno é mais fácil de tratar e analisar pelas fases posteriores, e embora contenha informação suficiente para que o programa a ser compilado esteja completamente definido, não contém informação referente à formatação do programa em formato textual (por exemplo, alinhamento das declarações ST ou instruções IL). De facto, o formato interno não é mais do que uma árvore na qual cada nó é composto por um objecto que representa um elemento sintáctico do programa de entrada. Durante esta primeira passagem serão detectados eventuais erros de sintaxe no programa a ser compilado.

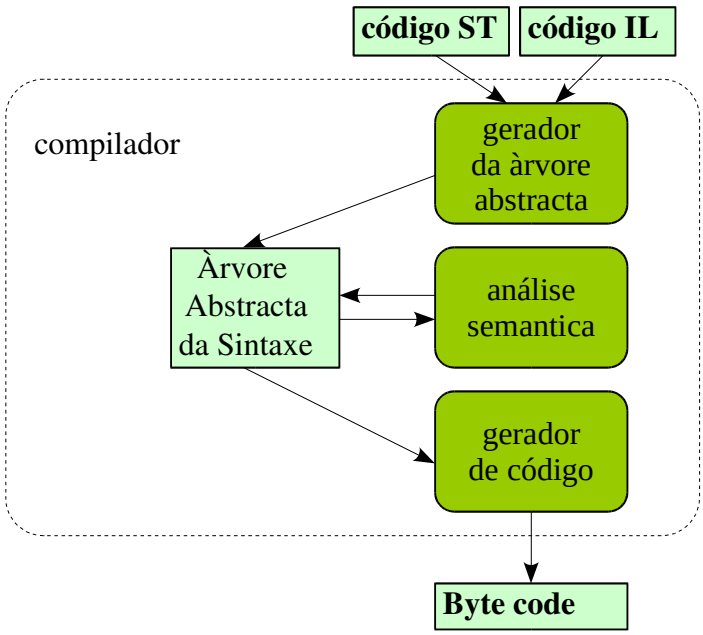

Figura 4.11 - Estrtura do compilador IL e ST.

A segunda passagem tem como único objectivo detectar eventuais erros de semântica no programa IL ou ST. Assume desde logo que a representação interna está isenta de erros de sintaxe, simplificando assim o seu trabalho. Caso detecte algum erro, o compilador aborta desde logo, e não chega a executar a terceira passagem. No entanto, caso não sejam detectados quaisquer erros, esta passagem não gera quaisquer dados, deixando a representação interna no estado em que a encontrou no inicio da sua execução.

Esta passagem permite assim alertar o programador de que o seu programa IL e/ou ST contem erros de semântica, e facilita ainda o desenvolvimento da passagem seguinte, que pode assim assumir que a representação interna está livre de quaisquer erros, sejam eles semânticos ou sintácticos.

A terceira e ultima passagem é responsável por produzir o código que irá ser executado. Até ao momento foi apenas implementado um gerador de código C++, ou seja, os programas IL e ST são compilados, sendo o resultado dessa compilação um programa em C++. Este programa é posteriormente compilado por outro compilado C++, após o qual fica disponível o código executável que implementa o código IL e/ou ST inicial.

A arquitectura do compilador IL e ST permite que seja mais tarde introduzido um outro gerador de código, em substituição do gerador de código C++, sem que seja necessário re-implementar a análise léxica, sintáctica e semântica das linguagens IL e ST. Uma alternativa possível seria um gerador de código que resulte num programa em Java, que poderá depois ser interpretado por uma máquina virtual Java (JVM – Java Virtual Machine).

Ainda outra hipótese é o de integrar o compilador IL e ST na família de compiladores da GNU. Esta família de compiladores suporta já várias linguagens,

incluindo o C (gcc), o C++ (g++), o Ada (gnat), Java, Fortran, objective C e até Cobol. É ainda capaz de gerar código executável para diversas famílias de processadores e micro-processadores, incluindo os processadores da família x86 da Intel e seus compatíveis (incluindo as novas versões de 64 bits da AMD), o IA-64 (Itanium, arquitectura de 64 bits da Intel), o Intel 960, o PowerPC originalmente da Motorola, ARM, Alpha (da DEC), SPARC (da SUN), MIPS, e até o vetusto PDP-11, entre muitas outras arquitecturas de CPUs.

A família de processadores gcc (GNU Compiler Collection) utiliza também uma arquitectura baseada em duas fases principais: um pré-processador (front-end) que gera uma árvore de sintaxe abstracta (estrutura de dados intermédia), e um pos-processador (back-end) que gera o código executável. Existe um pré-processador por cada linguagem suportada pela família de processadores, e um pos-processador por cada arquitectura de CPU. Assim, para que seja integrado o compilador IL e ST na família de compiladores da GNU, basta que este gere uma arvore de sintaxe abstracta com a mesma arquitectura de dados que é já utilizada pela família de compiladores da GNU. Uma opção seria a de utilizar apenas a estrutura de arvore de sintaxe abstracta utilizada já pelo gcc. Assim, a primeira passagem do compilador IL e/ou ST iria gerar directamente a arvore de sintaxe abstracta recorrendo a uma estrutura de dados compatível com a utilizada pelo gcc. A segunda passagem, de analise semântica, manter-se-ia, sendo que a terceira passagem passaria a ser composta por um pos-processador da família gcc. Infelizmente esta opção seria bastante mais complexa de implementar do que a alternativa que veio a ser utilizada. De facto, a estrutura de dados utilizada para a arvore de sintaxe abstracta pelo gcc é suficientemente poderosa para representar quaisquer das linguagens suportadas pela família de compiladores gcc. Por outro lado, esta estrutura está bastante mal documentada, tanto do ponto de vista de comentários no próprio código, como de documentação externa. Estes dois aspectos em conjunto, a complexidade e a falta de documentação, faz com que a utilização desta estrutura de dados se torne num exercício desnecessariamente penoso.

Por esta razão, a opção tomada foi a mesma que foi já utilizada pelos compiladores de ADA (gnat) e Cobol. Isto é, o de utilizar uma estrutura de dados própria para a arvore de sintaxe abstracta que se adeqúe às necessidades da linguagem em causa. Esta opção resulta que, para que o compilador seja integrado na família gcc, a sua terceira passagem (a da geração de código), seja responsável pela transformação da estrutura de dados privada utilizada para a arvore de sintaxe abstracta, na estrutura de dados utilizada pela família de compiladores da GNU.

A representação interna do programa, ou seja a árvore abstracta de sintaxe, é uma estrutura de dados em formato de árvore, composta por nós cada um dos quais representa um elemento sintáctico do programa que se encontra a ser compilado. Cada nó é de facto uma instância de uma classe de objectos

apropriada para representar o elemento sintáctico em causa. Este objecto contem ele próprio as referências aos nós inferiores da arvore de sintaxe abstracta. O numero de nós inferiores depende do números de elementos que compõem esse elemento sintáctico. Por exemplo, uma declaração de atribuição em ST tem dois sub-elementos sintácticos, a variável à esquerda do sinal de atribuição, e a expressão à direita do mesmo. A expressão em si será um novo elemento sintáctico que poderá ser composto por uma operação aritmética, a chamada a uma função, etc...

Alguns elementos sintácticos são compostos por um numero variável de sub-elementos. Nestes casos o objecto que representa esse elemento sintáctico mantém uma lista com a referencia aos objectos que representam esses seus sub-elementos.

A lista completa dos elementos sintácticos encontra-se no anexo A. É de salientar que a referida lista inclui muitos elementos auxiliares que se tornaram necessários devido à complexidade desnecessária da linguagem introduzida pelo standard (complexidade que será discutida em secções seguintes).

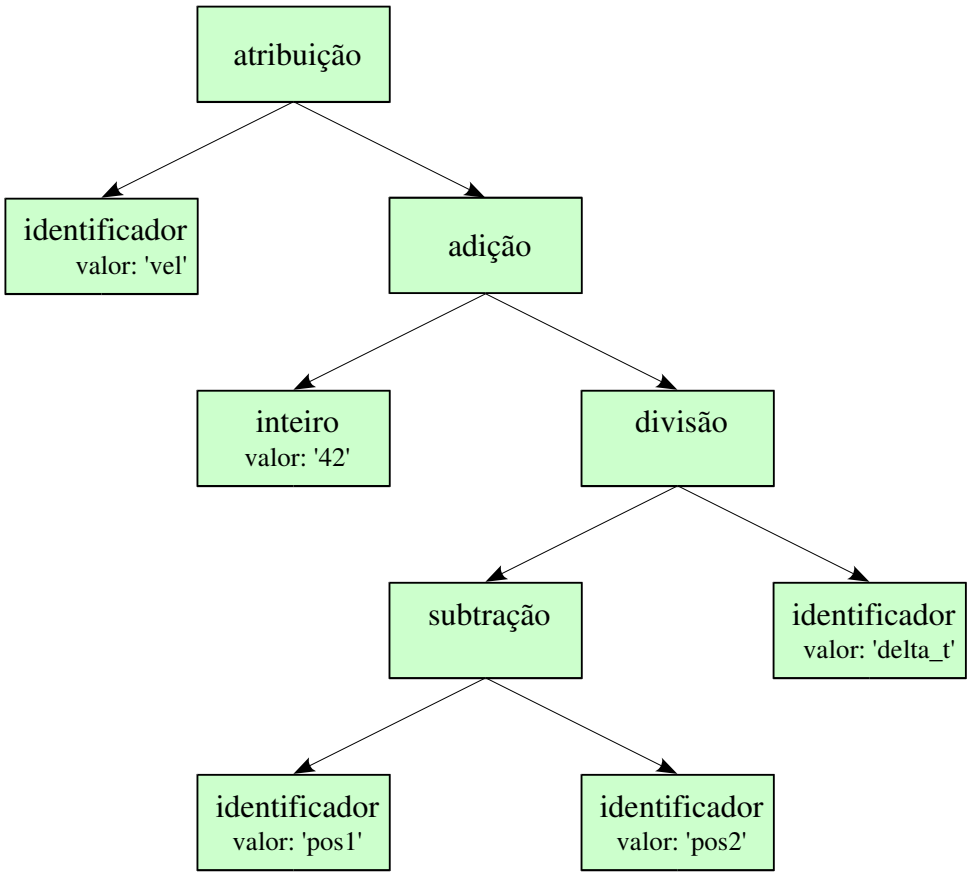

Figura 4.12 - Representação da Arvore de Sintaxe Abstracta resultante da análise da declaração em ST 'vel := 42 + (pos1 – pos2) / delta_t;'

No exemplo da fig. 4.12, todos os elementos sintácticos não terminais são compostos por dois sub-elementos. No entanto, isto não é regra, existindo elementos sintácticos que referenciam mais elementos. Exemplos destes últimos

serão a declaração de iteração FOR (com referencia a cinco sub-elementos), a declaração de controlo IF (que referencia quatro elementos), ou mesmo a declaração de uma função bloco (com referencia a três sub-elementos: o nome da função bloco, as declarações de variáveis, e o bloco de código).

Todas as classes que representam um elemento sintáctico herdam de uma classe básica 'symbol_c'. Certos elementos sintácticos são terminais, isto é não são eles próprios compostos por sub-elementos sintácticos. Muitos destes elementos terminais necessitam de armazenar o valor de uma constante ou o nome de uma variável, pelo que foi criado uma classe básica 'token_c', que herda de 'symbol_c' e armazena uma sequência de caracteres, e da qual as classes para elementos sintácticos terminais podem herdar. O mesmo ocorre para classes que necessitam de armazenar uma lista com um comprimento variável de de referencias a outros objectos, para as quais foi criada a classe 'list_c' que gere a lista de referencia e herda também de 'symbol_c.

4.2.1 Gerador da Arvore de Sintaxe Abstracta

A primeira passagem durante a qual é gerada a arvore de sintaxe abstracta é composta por duas fases distintas: a análise léxica e a análise sintáctica. Estas duas fases executam na mesma passagem devido à forte inter-dependência entre as duas. Numa linguagem ideal e simples, deveria ser possível efectuar as duas fases em passagens distintas, no entanto a sintaxe das linguagens IL e ST não tem suficiente redundância para que isto seja possível, sendo assim necessário manter listas de identificadores anteriormente definidos, o que obriga a que estas duas fases sejam efectuadas na mesma passagem.

Durante a análise léxica o texto é partido nos seus elementos léxicos constituintes, tais como palavras chave (FUNCTION, VAR, FOR, RETURN, FUNCTION_BLOCK, etc.), sinais básicos (':=', '=>', ';', etc.), números (43, 23.87, 8#123, etc.) e identificadores (nomes de variáveis, de funções, de parâmetros de funções, etc.). Na análise sintáctica os elementos léxicos são agrupados por forma a constituírem elementos sintácticos, por exemplo uma expressão, a declaração de uma função, a chamada a uma função bloco, etc.

A análise léxica foi implementado com recurso ao utilitário 'flex', um programa que, a partir de um ficheiro de definição dos elementos léxicos possíveis na linguagem, gera um programa que analisa o texto e separa-o nesses mesmos elementos léxicos. Por outro lado, o analisador sintáctico foi implementado com recurso ao utilitário 'bison', sendo este utilitário uma re-implementação de um outro utilitário mais conhecido 'yacc'. Estes utilitários geram um programa com capacidade de fazer a analise sintáctica a partir de um ficheiro de definição dos elementos sintácticos da linguagem para a qual se deseja criar o compilador.

Uma consideração importante na implementação do analisador léxico para a linguagem IL tem a haver com o facto desta linguagem dar um significado

especial ao fim de uma linha de texto, ou seja, a sintaxe da linguagem IL é sensível à divisão do código por linhas. Por exemplo,

```
LD var1
ADD var2
```

é diferente de

```
LD var1 ADD var2
```

Pelo contrário, na linguagem ST o final de uma linha funciona como um caracter em branco, ou seja um espaço, pelo que

```
IF weigh THEN WEIGH := INT_TO_BCD (gross – tare); END_IF ;
```

é perfeitamente equivalente a

```
IF weigh
  THEN WEIGH := INT_TO_BCD (gross – tare);
END_IF ;
```

Por outro lado, a sintaxe utilizada para a declaração dos blocos (i.e. as funções, funções bloco, programas e configuração) funciona como a linguagem ST, interpretando o fim-de-linha como um espaço em branco. Uma vez que a linguagem IL é composta pela declaração dos blocos, os quais poderão conter código IL, a dificuldade de análise léxica aumenta. Ou seja, o elemento léxico que deve ser passado ao analisador sintáctico quando é encontrado um fim-de-linha depende do local onde se encontra este fim-de-linha. Se estiver no meio de um bloco de código IL, deve ser gerado um elemento léxico correspondente ao fim-de-linha ('EOL'), caso contrário deverá ser gerado um elemento léxico correspondente ao espaço em branco (' ').

Isto obriga a que o analisador léxico não possa ser totalmente desprovido de estado interno. Terá assim de manter algum estado que corresponde ao local de código que está a ser analisado em cada momento. Foram consideradas três hipóteses de implementação:

- introduzir na definição da sintaxe da linguagem o elemento léxico EOL em todos os locais em que este poderá ser interpretado como um caracter em branco, e deixar o analisador léxico funcionar sem estado interno, gerando sempre o elemento léxico EOL sempre que encontra um fim-de-linha;

- manter estado interno no analisador léxico, reflectindo o local do código que está a ser analisado em cada momento, sendo que assim poderá decidir se quais dos elementos léxicos (EOL ou caracter em branco) que deve ser entregue ao analisador sintáctico sempre que encontra um fim-de-linha. A responsabilidade de gerir a máquina de estados cabe ao analisador léxico;

- semelhante à hipótese anterior, no que refere à necessidade de manter uma máquina de estados. No entanto, cabe agora ao analisador sintáctico gerir a máquina de estados, e fazer esta progredir entre os estados em função do local do programa que se encontra a ser analisado.

A primeira opção é muito vantajosa do ponto de vista do analisador léxico, pois este reduz-se a emitir o elemento léxico EOL sempre que encontra um fim-de-linha. No entanto, o contrário já não é verdade para o analisador sintáctico, que deve ver a definição da sintaxe da linguagem alterada. A alteração é bastante simples de fazer, no entanto, e devido à reduzida redundância da sintaxe das linguagens IL e ST, a alteração da definição da sintaxe do ST e das declarações de ST e IL introduz demasiadas situações de conflito, em que se torna impossível distinguir qual de duas ou mais sintaxes possíveis é a correcta. Isto porque o analisador sintáctico gerado pelo bison está limitado a linguagens independentes de contexto, e que possam ser processadas considerando apenas um elemento léxico à frente do que se encontra a ser analisado (i.e., linguagens LR(1)).

A segunda opção tem a vantagem de não necessitar quaisquer alterações ao analisador sintáctico, em contrapartida o analisador léxico torna-se bastante mais complexo, sendo que passa a necessitar da tal máquina de estados já mencionada. No entanto, a maior desvantagem de todas reside no facto de que, para que o analisador léxico possa gerir a passagem de estados, este terá de ser capaz de analisar a linguagem ao nível da sua sintaxe. Ou seja, deixa-se de ter uma clara separação entre o código responsável pela análise sintáctica e a análise léxica, o que pode levar a que, se houver certas alterações à sintaxe da linguagem, tanto o analisador sintáctico como o analisador léxico terão de ser actualizados.

A terceira opção parece a mais prometedora, uma vez que se divide a complexidade entre os dois segmentos de código, e mantém-se a compartimentalização de responsabilidades de cada segmento. No entanto a sua implementação com as ferramentas disponíveis não é possível de todo. De facto, a ferramenta que gera o analisador léxico está já preparada para gerir máquinas de estado e de interpretar o código de entrada em função do estado que se encontra activo. Infelizmente, recorrendo a esta máquina de estados, a passagem entre estados tem de ser efectuada pelo analisador léxico, e nunca pelo analisador sintáctico, uma vez que recorre a macros que apenas se encontram disponíveis no código respeitante ao analisador léxico.

A segunda opção passou assim a ser a que mais simplificava a implementação, e foi assim a escolhida. A maquina de estados recorre a três estados, para além do estado inicial. Um estado para quando se encontra a processar um bloco de configuração, outro para quando processa a declaração de uma função, função bloco ou programa, e ainda outra quando passa para o segmento do código dentro do bloco que se encontra a ser declarado. Claro está, esta máquina de estados é apenas necessária para o compilador de IL, que necessita de distinguir as alturas em que deve ou não gerar os elementos léxicos EOL.

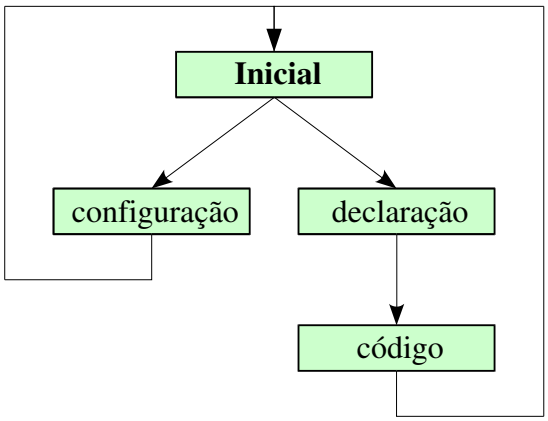

Figura 4.13 - Maquina de estados do interpretador léxico.

No entanto, e tendo em consideração que os elementos léxicos das linguagens IL e ST são comuns, com a única excepção do EOL, e ainda que o analisador léxico para o compilador de IL conseguia já determinar quando se encontrava a processar o segmento com código dentro da declaração de funções, funções bloco ou programas, tornou-se obvio que seria vantajoso utilizar o mesmo analisador léxico para as duas linguagens. Assim, quaisquer correcções que se viessem a mostrar necessárias teriam imediatamente efeito nos dois compiladores. No entanto, outra vantagem ainda maior seria que a potencialidade de gerar um único compilador que fosse capaz de compilar IL e ST em simultâneo, desde que estas linguagens não fossem misturadas dentro da mesma função, função bloco ou programa. Esta capacidade é ainda facilitada pelo facto da sintaxe correspondente à declaração das funções, funções bloco, programas e configurações ser idêntica para as duas linguagens, e que a sintaxe das duas linguagens é na sua maioria complementar, sem criar conflitos entre elas.

Uma primeira alternativa que se apresentou foi o de acrescentar à definição sintáctica da linguagem ST a possibilidade de esta ter o elemento léxico EOL em qualquer local onde pudesse aparecer um caracter em branco. No entanto, e á semelhança do sucedido anteriormente, a sintaxe da linguagem ST passa a ser demasiado complexa para que possa ser analisada considerando apenas um único elemento sintáctico à frente do elemento que se encontra a ser analisado.

Para realizar este novo analisador léxico, foi então necessário acrescentar mais dois estados à máquina de estados já mencionada. Um estado para quando o analisador léxico se encontra a processar a linguagem ST (durante a qual não emite o elemento léxico EOL), e outro para quando se encontra a processar código em IL. Estes dois estados são assim uma especialização do estado anteriormente utilizado para identificar o segmento de código da declaração de uma função, função bloco ou programa.

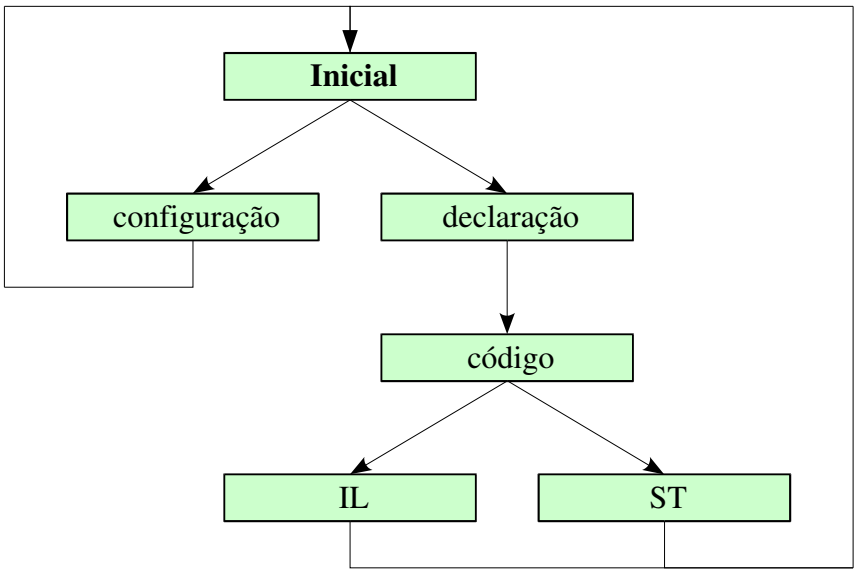

Figura 4.14 - Máquina de estados do intepretador léxico, com distinção entre IL e ST.

A distinção do código IL do ST ficou assim dentro do analisador léxico. Ou seja, o analisador léxico passou a necessitar de mais informação relativa à sintaxe das duas linguagens para que pudesse distinguir as duas. No entanto, e considerando que a divisão clara de conhecimentos entre os dois módulos (analisador léxico e sintáctico) já tinha sido quebrada anteriormente, esta opção não traz mais inconvenientes do que os que já tinham sido introduzidos na arquitectura do compilador.

Foi ainda decidido acrescentar a possibilidade de inclusão de um ficheiro com código IL e/ou ST, dentro de outro ficheiro que se encontre a ser compilado. Ou seja, permitir uma funcionalidade semelhante aos '#include “ficheiro.c”' de C e C++. Esta opção não está prevista no standard, pelo que foi decidido recorrer a uma sintaxe que fosse interpretada como sendo um comentário, e por isso ignorada, por outros compiladores. A sintaxe utilizada foi então semelhante à utilizada pelo C, mas introduzida dentro de um comentário '(*#include “ficheiro.st”*)'.

Espera-se que versões posteriores do compilador passem a utilizar uma sintaxe de 'pragmas', já prevista pelo standard exactamente para este fim. Ou seja, permite que cada compilador acrescente funcionalidades especificas desse compilador, e que não se encontram especificadas de forma explícita no standard. Neste caso, a sintaxe para a inclusão de ficheiros passará a ser '{include “ficheiro.st”}.

O analisador sintáctico, como já foi referido, foi implementado com recurso à ferramenta bison, que gera um analisador sintáctico de linguagens LALR(1) a partir da descrição da sintaxe da linguagem. No entanto, as linguagens ST e IL contêm pouca redundância, pelo que não é possível fazer a analise sintáctica sem recurso a tabelas de símbolos. Estas tabelas não são mais do que uma lista de todos os identificadores já encontrados no programa e cuja classificação já é

conhecida.

Nas linguagens IL e ST um identificador é uma sequência de letras e números (com uma letra no inicio), podendo ainda ter o caracter '_'. Estes identificadores são utilizados para atribuir nomes a variáveis, funções, funções bloco, etc. No entanto, nas linguagens IL e ST existem situações em que não é possível distinguir se um identificador se refere a uma variável ou a um elemento de uma enumeração (um tipo de dado derivado). Por exemplo, na declaração 'var1 := elem;', o analisador sintáctico não é capaz de distinguir se o identificador 'elem' se refere a um item de uma enumeração, ou se é uma outra variável. Isto, claro está, apenas pelo contexto da linha que se encontra a ser analisada, e sem recurso a outras informações.

Por este motivo torna-se necessário manter as tais tabelas de símbolos, ou identificadores, onde é introduzido o nome das variáveis e funções conhecidas. Estas tabelas são preenchidas pelo analisador sintáctico, pois é este módulo que reconhece as declarações das funções, variáveis, etc. As tabelas estão ainda disponíveis para serem consultadas pelo analisador léxico, o que permite que este faça uma pesquisa pelas tabelas de símbolos sempre que encontra um identificador no programa. Assim, o analisador léxico consegue distinguir os identificadores que identificam funções dos identificadores utilizados para nomear variáveis, e enviar ao analisador sintáctico um elemento léxico especifico para nomes de funções ou de variáveis. Assim, o analisador sintáctico deixa de necessitar de reconhecer o objecto do identificador apenas pelo contexto em que se encontra.

Foram utilizados cinco tabelas de símbolos, uma para cada uma das seguintes classes de elementos sintácticos: nomes de variáveis, de funções, de funções bloco, de programas, de configurações, e ainda os nomes atribuídos a tipos de dados derivados. À excepção da tabela de símbolos respeitante às variáveis, todas as tabelas de símbolos são preenchidas pelo analisador sintáctico à medida que forem encontradas as declarações das funções, funções bloco, programas, configurações e tipo de de dados derivados. Isto significa que uma função ou função bloco apenas poderá ser invocada após a sua declaração. De forma semelhante, só é permitido declarar variáveis de tipos de dados derivados após a declaração do tipo de dado derivado em causa.

As tabelas de variáveis são também preenchidas à medida que as suas declarações forem encontradas no código. No entanto, e ao contrário das restantes tabelas, esta é re-inicializada sempre que as variáveis deixem de estar visíveis. Ou seja, quando se chega ao fim de uma função, as variáveis declaradas dentro dessa função são retiradas da tabelas de símbolos respeitantes às variáveis. É de referir que as configurações têm dois níveis dentro do qual se podem declarar variáveis: dentro da configuração em si, sendo estas visíveis em toda a configuração, ou dentro de um recurso dessa configuração, sendo estas ultimas apenas visíveis dentro do recurso em que foram declaradas. Por este motivo a tabela de símbolos

das variáveis trabalha também com vários níveis, sendo que quando termina um nível interior (um recurso) são apenas retiradas as variáveis declaradas dentro desse recurso.

Adicionalmente, dentro de uma função o nome da própria função pode ser usada como uma variável (utilizado para atribuir o valor de retorno da função). Para que isto seja suportado, o analisador sintáctico insere o próprio nome da função na tabela de símbolos de variáveis no inicio da declaração da própria função. É possível introduzir o nome da função na tabela de símbolos de variáveis uma vez que uma função é idem-potente, pelo que nunca se poderá invocar a si mesma. Logo, qualquer utilização do identificador que identifica a função dentro da própria função será sempre como uma variável, e nunca como o identificador de uma função, pelo que nunca haverá conflitos. Tal como as restantes variáveis, o nome da função é retirado da tabela de símbolos de variáveis logo que a declaração da função termine.

4.2.2 O Analisador Semântico

A segunda passagem do compilador serve apenas para detectar erros de programação, sendo que o analisador semântico não altera de qualquer forma a arvore de sintaxe abstracta. Esta segunda passagem, ou seja o analisador semântico, não foi ainda implementado, principalmente por não ser essencial para o funcionamento do compilador.

Isto não significa, no entanto, que o compilador não detecta já alguns erros semânticos. De facto, a introdução das tabelas de símbolos no analisador sintáctico faz com que muitos erros de semântica sejam identificados durante a primeira passagem. Por exemplo, a utilização de uma variável que não tenha sido previamente declarada, um erro de semântica, é detectado na primeira passagem pois o analisador sintáctico e léxico irá gerar um elemento léxico genérico de 'identificador' quando for utilizada a variável não declarada, o que fará com que o analisador sintáctico produza um erro pois na grande maioria das situações (mas não todas) a utilização de um identificador genérico em substituição de uma variável é ilegal.

Tendo ainda em consideração que de momento existe apenas um gerador de código C++, na fase de compilação do código C++ irão ainda ser detectados muitos erros semânticos. Isto porque muitos dos erros de semântica irão ser reproduzidos no código C++, uma vez que o gerador de código não se dedica a verificar se o código que se encontra a ser compilado contem erros. Claro está que não é desejável que se encontrem erros nesta última fase do processamento do compilador, no entanto por enquanto vai servindo para identificar alguns erros de semântica. O exemplo de um erro que é identificado nesta fase é a atribuição de valores incompatíveis a variáveis. Por exemplo, a atribuição de um valor em vírgula flutuante a uma variável de inteiros.

Devido ao modo em que foi implementado o gerador de código, esta terceira fase consegue também ela identificar alguns erros de semântica nos programas IL e ST. De facto, a conversão de alguns elementos sintácticos necessitam de pesquisas que, na realidade, são uma verificação da semântica do código ST e IL. Por exemplo, a passagem para C++ da invocação de uma função com a sintaxe formal, exige que seja efectuada uma pesquisa na declaração da função a ser invocada de cada parâmetro ao qual é atribuído um valor durante a invocação. Esta pesquisa, se não encontrar o parâmetro em causa, está de facto a detectar um erro de semântica no código ST ou IL original.

4.2.3 Gerador de Código

A terceira passagem compreende o gerador de código. Como já foi referido, de momento existe apenas uma única versão para esta terceira passagem, que se baseia na geração de código C++, que é posteriormente compilado pelo compilador g++.

O gerador de código faz uma passagem iterativa por todos os elementos sintácticos na arvore de sintaxe abstracta. Para cada classe de elemento lá encontrado, é invocada uma função própria para a conversão desse elemento sintáctico em C++. Por vezes, durante a conversão de um elemento sintáctico, torna-se necessário fazer uma iteração recursiva ou embebida dentro da iteração geral da terceira passagem, de parte da arvore de sintaxe abstracta. Utilizando o exemplo da invocação de uma função com sintaxe não formal, por cada parâmetro da invocação torna-se necessário fazer uma iteração da arvore de sintaxe abstracta respeitante à declaração da função que se encontra a ser invocada. De forma mais precisa, é efectuada uma iteração pela declaração das variáveis de interface dessa função, para que seja encontrada a posição da lista de parâmetros em que deverá ser colocado o parâmetro original na invocação equivalente em C++.

É de salientar que, nas referidas iterações, o código que irá ser executado por cada elemento sintáctico na arvore de sintaxe abstracta irá depender do tipo desse elemento sintáctico, bem como da iteração em causa. Tendo em conta que cada tipo de elemento sintáctico tem a sua própria classe de objectos, sendo um objecto instanciado por cada elemento desse tipo na arvore de sintaxe abstracta, uma implementação possível destas iterações seria a de adicionar um método a cada classe de elemento sintáctico que contenha o código respeitante à iteração por essa classe. Como são necessárias diversas iterações, seria necessário acrescentar um método por cada iteração.

A referida implementação tem duas sérias desvantagens. Uma delas deve-se ao facto de que esta implementação resulta no código respeitante às iterações (sendo que cada iteração implementa um determinado algoritmo) ficar disperso pelas classes dos elemento sintácticos. O algoritmo que implementa em conjunto

fica também disperso por várias classes, ficando assim o código difícil de perceber e mal compartimentalizado.

A segunda desvantagem resulta no facto de que será necessário um método por cada algoritmo de iteração em cada classe dos elementos sintácticos. Ou seja, sempre que se deseje acrescentar uma nova iteração, que implemente um novo algoritmo, torna-se necessário acrescentar um método a cada classe de elemento sintáctico.

Para conseguir efectuar as iterações, sem ser necessário acrescentar uma função membro a cada classe dos elementos sintácticos, foi utilizada um padrão de visitante [19]. Este padrão consiste em introduzir um único método em cada classe de elementos sintáctico. A este método é passado como parâmetro um objecto que deseja visitar o elemento sintáctico (na realidade o objecto que representa o elemento sintáctico). Este método do elemento sintáctico limita-se a invocar o objecto que o deseja visitar, passando-se a ele próprio como parâmetro ao objecto visitante. Uma vez que o objecto visitante tem uma função membro para cada classe de elemento sintáctico, o objecto do elemento sintáctico irá na realidade invocar o método do objecto visitante especializado em iterar pelo tipo de elemento sintáctico em causa.

Todos as classes de objectos visitantes herdam de uma classe básica pura, que contem um método virtual para cada classe de elemento sintáctico. Cada iteração, que deseje implementar um determinado algoritmo de iterar pela arvore de elementos sintácticos, herda da classe básica pura do visitante (visitor_c), e define a implementação que o algoritmo de iteração necessita que seja executado para cada classe de elemento sintáctico. Desta forma, as funções respeitantes a um determinado algoritmo de iteração ficam todos na mesma classe visitante.

A conversão de IL e ST para C++ é na sua maioria efectuada por um único objecto da classe generate_cc_c. Esta classe herda das classes generate_cc_base_c e generate_cc_typedecl_c. A primeira destas faz a conversão de elementos sintácticos básicos, tais como identificadores (ex. nomes de variáveis e funções), constantes numéricas e outros tipos de de dados básicos (tempos, datas, cadeias de caracteres). Por seu lado, a classe generate_cc_typedecl_c é responsável pela conversão da declaração de tipos de dados derivados (enumerações, arrays, estruturas, etc.). Os restantes membros da classe generate_cc_c deveriam então ser responsáveis pela conversão dos restantes elementos sintácticos. No entanto, e devido à complexidade de alguns destes elementos, a conversão destes foi transferida para uma outra classe dedicada a esse elemento. Assim, a classe generate_cc_c fica apenas responsável pela conversão da componente da interface das declarações de funções, funções bloco, programas e configurações. Mesmo assim, a conversão das declarações de variáveis é passada para um objecto da classe generate_cc_vardecl_c. A classe generate_cc_tempvar_decl_c é uma classe auxiliar utilizada para declarar variáveis que, embora não tenham correspondente directa no código original em IL e/ou ST, são necessárias no código resultante em

C++. Estas são variáveis temporárias, pelo que são declaradas com nomes que, embora válidos em C, não são permitidos pelo standard em IL e ST, o que permite garantir que nunca haverá conflito entre as variáveis declaradas na linguagem IL e ST e as temporárias declaradas por esta classe.

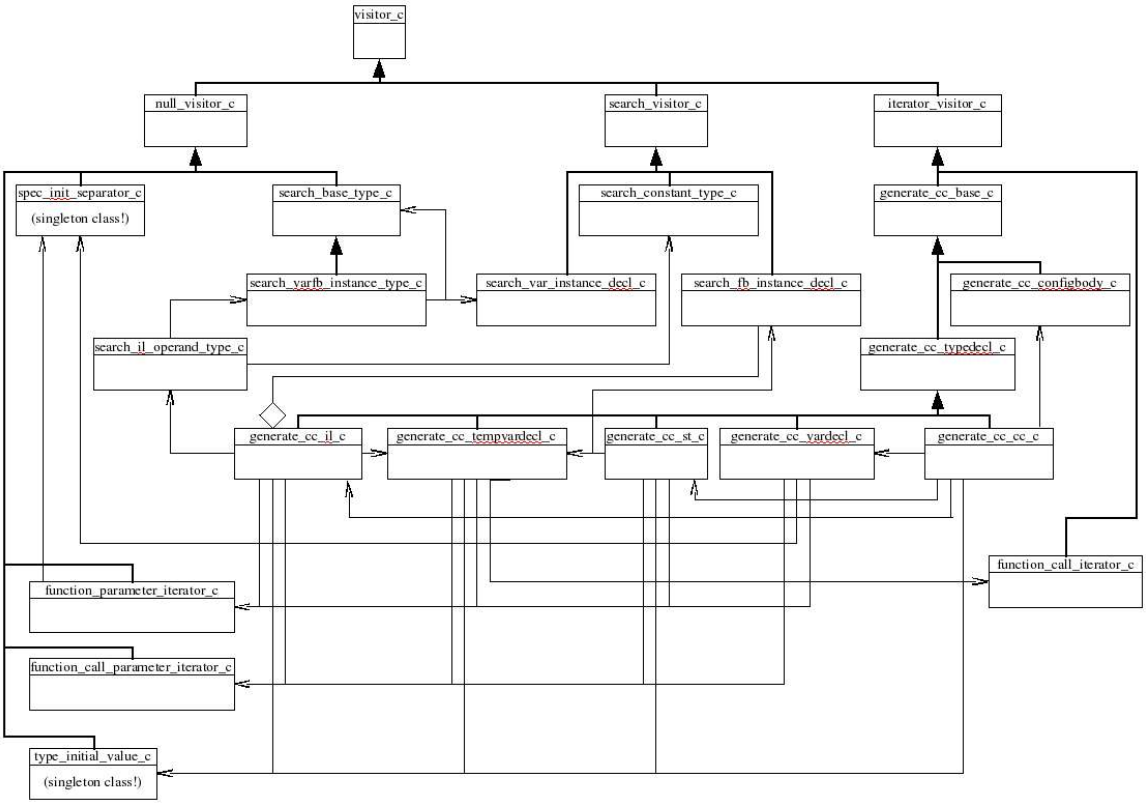

Figura 4.15 - Diagrama de classes do gerador de código C++.

Para a conversão das partes correspondentes ao código em si, a classe generate_cc_c cria uma instância de um objecto da classe generate_cc_il_c para código IL, e outro da classe generate_cc_st_c para código ST, a transfere para estes a responsabilidade de conversão dos elementos sintácticos exclusivos dessas linguagens. Utiliza ainda um objecto da classe generate_cc_configbody_c para o qual transfere a conversão para C++ das configurações, dos seus recursos, instâncias de programas e tarefas.

Ainda assim, e com o intuito de facilitar a tarefa de algumas das classes anteriores, foram ainda implementadas classes responsáveis pela pesquisa do tipo de um determinado operando, constante ou função bloco (search_varfb_instance_type_c, search_il_operand_type_c, search_constant_type_c). Existem ainda classes para encontrar o local em que foi declarada uma variável ou uma função bloco (search_var_instance_decl_c, e search_fb_instance_decl_c).

A classe function_call_iterator_c é utilizada para percorrer todas as chamadas a funções e funções bloco dentro do código dos programas, funções bloco e funções. Objectos da classe function_param_iterator_c percorrem a declaração de

uma classe de modo a retornar, em chamadas sucessivas a um dos seus métodos, os parâmetros de entrada, saída e entrada/saída da função, função bloco, ou programa em causa. Por seu lado, objectos da classe function_call_param_iterator_c são utilizados para iterar pelos parâmetros utilizados numa determinada invocação a uma função ou função bloco.

4.2.4 Mapeamento dos Blocos para C++

A conversão das funções, funções bloco e programas para C++, embora semelhante, é efectuada de modo distinto para cada um destes tipos de blocos.

As funções são mapeadas em funções de C++. Mantém-se o nome da função, e todos os parâmetros da função são também eles mapeados como parâmetros da função C++. Parâmetros de entrada são implementados como parâmetros normais (passagem de valor), enquanto que os parâmetros de saída e entrada/saída são implementados por parâmetros referenciais, ou seja, com passagem da referencia da variável passada como parâmetro durante uma invocação. Variáveis internas da função são declaradas no interior da função C++.

Por exemplo, a função na fig. 4.16, é convertida no código da fig. 4.17.

```
FUNCTION WEIGH : WORD     (* BCD encoded *)
  VAR_INPUT
    weigh_command : BOOL;
  END_VAR

  VAR_IN_OUT
    gross_weight : WORD ; (* BCD encoded *)
  END_VAR

  VAR_OUTPUT
    tare_weight : INT ;
  END_VAR

  VAR
    sensor1, sensor2: WORD;
  END_VAR

(* Function Body *)
(* codigo respeitante ao corpo da função... *)
END_FUNCTION
```

Figura 4.16 - Exemplo de declaração de função.

```
// FUNCTION
WORD WEIGH(
  BOOL weigh_command,
  WORD &gross_weight,
  INT &tare_weight)
{
  WORD sensor1;
  WORD sensor2;
  WORD WEIGH = 0;
// codigo respeitante ao corpo da função...
  __end:
  return WEIGH;
}
```

Figura 4.17 - Conversão para C++ da função na fig. 4.16.

É de salientar que a função em C++ contem uma variável adicional, com nome idêntico ao da própria função, para permitir que o valor de retorno possa ser lá armazenado, tal como o previsto nas linguagens do standard IEC 61131-3. Assim, sempre que a função retorne este deverá retornar esse valor, resultando no código C++ 'return weigh;'. No entanto, as funções bloco não retornam qualquer valor, sendo que para estas a conversão da declaração 'RETURN' de ST deverá ser convertida para C++ sem qualquer parâmetro: 'return;'. No entanto, o código que faz a conversão da declaração 'RETURN' em ST será sempre o mesmo esteja o código ST inserido numa função ou numa função bloco. Para ultrapassar esta divergência, todas as instruções 'RETURN' em ST são convertidas para um 'goto __end;' em C++, sendo que no final da função ou função bloco será inserido a instrução de retorno em C++ apropriada para o caso, ou seja 'return weigh;' para uma função, e 'return;' para uma função bloco.

Quanto às funções bloco, essas são convertidas em classes, enquanto que cada instância de uma função bloco é mapeada num objecto C++. Uma vez que no standard IEC 61131-3 as funções bloco têm apenas uma única função membro que, por não haver hipótese de más interpretações, não tem qualquer nome, na conversão para C++ é criada uma função membro da classe sempre com o mesmo nome 'f'.

No entanto, para funções bloco os parâmetros de entrada, saída e de entrada/saída não são convertidos em parâmetros da função 'f' como seria de esperar, sendo em alternativa convertidos em variáveis locais da classe. A razão de converter as variáveis de saída em variáveis locais prende-se com o facto de que é permitido aceder aos parâmetros de saída de uma função bloco em qualquer altura. Por outro lado, um dos métodos de invocar funções bloco em IL prevê que os valores dos parâmetros de entrada sejam previamente carregados nas respectivas variáveis de entrada, razão pela qual também as variáveis de entrada são convertidas para variáveis internas da classe. Neste caso, como uma invocação a uma função bloco não retorna qualquer valor, deixa de ser necessário declarar uma variável com o nome da própria classe. Por exemplo, a função bloco

da fig. 4.18 é convertida para a classe C++ da fig. 4.19.

```
FUNCTION_BLOCK INTEGRAL
   VAR_INPUT
     RUN : BOOL ;       (* 1 = integrate, 0 = hold *)
     XIN : REAL ;       (* Input variable          *)
     X0  : REAL ;       (* Initial value           *)
     CYCLE : REAL ;     (* Sampling period         *)
   END_VAR
   VAR_IN_OUT
     R1 : BOOL ;        (* Overriding reset        *)
   END_VAR
   VAR_OUTPUT
     XOUT : REAL ;      (* Integrated output       *)
   END_VAR
   VAR
     local_var : LINT;
   END_VAR
   VAR_TEMP
     local_temp_var : ULINT;
   END_VAR

(* Function Body *)
(* codigo respeitante ao corpo da função bloco... *)
 END_FUNCTION_BLOCK
```

Figura 4.18 - Exemplo de declaração de função bloco.

```
// FUNCTION_BLOCK
class INTEGRAL {
  public:
    BOOL RUN;
    REAL XIN;
    REAL X0;
    REAL CYCLE;
    BOOL R1;
    REAL XOUT;

  private:
    LINT local_var;

  public:
    INTEGRAL(void)
      {}

  public:
    void f(void) {
      ULINT local_temp_var;
      // código respeitante ao corpo da função bloco
      // aparece aqui...
      } /* f() */

}; /* class INTEGRAL */
```

Figura 4.19 - Classe em C++ correspondente à conversão da função bloco da fig. 4.18.

Com este mapeamento torna-se necessário transformar cada invocação a uma função bloco numa sequência de acções que começa por atribuir às variáveis correspondentes aos parâmetros de entrada e entrada/saída os valores que se desejam passar na invocação, transferir o controlo de execução para a função bloco (ou seja, invocar a função bloco), seguido da leitura e todas as variáveis de saída e entrada/saída desejáveis.

Neste caso é ainda de salientar o mapeamento das variáveis internas da função bloco. As variáveis normais, i.e. as variáveis que mantêm o seu estado ou valor interno entre duas invocações da função bloco, são mapeadas para C++ como variáveis internas à classe. Estas variáveis têm agora classe de acesso 'private', ou seja, são variáveis privadas que apenas as funções membro das própria classe poderão aceder.As variáveis temporárias, ou seja, as variáveis que deverão ser re-inicializadas de cada vez que a função bloco é invocada, são declaradas no interior da função f, pelo que serão inicializadas, tal como requerido pelo standard, em cada invocação.

A conversão dos programas é em tudo idêntica á da conversão das funções bloco. De facto, o standard especifica a mesma semântica para estas duas unidades de organização, divergindo apenas no local em que podem ser criadas instâncias das mesmas.

De momento, a versão actual existente do compilador não suporta ainda as capacidades completas das configurações. Por enquanto, às configurações é apenas permitida uma única tarefa e uma única instância de programa. Esta instância é posteriormente transformada num módulo do MatPLC, configurada para que seja executado com um período idêntico ao da tarefa à qual foi atribuído o programa na configuração.

Está previsto que versões posteriores possam suportar a semântica completa das configurações, o que poderá ser conseguido recorrendo a uma de várias alternativas.

Existem duas escolhas a fazer. A primeira reside em decidir qual dos elementos da arquitectura do IEC 61131 será mapeado em módulos do MatPLC. Neste caso surge desde logo três alternativas: (a) mapear uma configuração inteira no mesmo módulo, (b) utilizar um módulo por cada recurso dentro da mesma configuração, ou (c) mapear cada tarefa dentro da configuração num módulo do MatPLC. A primeira alternativa (a) tem como consequência que será possível ter várias configurações a executar em simultâneo no mesmo MatPLC, enquanto que as restantes duas assumem o máximo de uma configuração por MatPLC.

A segunda escolha reside em como implementar as várias tarefas dentro do mesmo módulo do MatPLC. Uma vez que o standard prevê que uma mesma configuração, ou até o mesmo recurso dentro de uma configuração, possa ter diversas tarefas, esta escolha faz sentido tanto para a hipótese (a) como para a (b). Também aqui poderão ser utilizadas uma de duas opções: (i) recorrer aos

threads POSIX, sendo que a cada tarefa será atribuído um thread, e (ii) implementar no próprio MatPLC um mini escalonador que decide em que instante poderá ser executada cada tarefa dentro do módulo do MatPLC.

Tendo em conta que o standard IEC 61131-3 prevê que os recursos sejam mapeados cada um no seu CPU, faz mais sentido recorrer à opção (b) uma vez que nestas situações o MatPLC irá executar sobre um sistema operativo que gere os vários CPUs em simultâneo, devendo ser possível, caso se venha a revelar necessário, solicitar ao sistema operativo que cada um dos processos nos quais foram mapeados os recursos sejam alocados de forma permanente ao seu CPU. A restrição de poder suportar múltiplos CPUs não invalida no entanto as opções (a) e (c), continuando estas a ser possíveis. Para o caso do (a), cada recurso deverá ser executado no seu próprio thread, sendo ainda possível bloquear a execução deste último a um CPU específico.

Outra consideração resulta da utilização de variáveis globais dentro das configurações. De facto, todas as tarefas deverão ter acesso a estas variáveis, pelo que se forem utilizadas as alternativas (b) ou (c), tonar-se-á necessário que todos os processos ou threads tenham acesso a uma zona de memória comum onde essas variáveis globais possam ser armazenadas. Ora isto implica que ou estes processos criem eles próprios uma zona de memória partilhada, ou as variáveis sejam mapeadas em variáveis do MatPLC (i.e. pontos do MatPLC), sendo que neste caso será utilizada a zona de memória partilhada que é gerida pela secção GMM para armazenar as variáveis globais. Infelizmente qualquer destas duas soluções apresenta inconvenientes. A primeira solução, de utilizar uma zona de memória partilhada dedicada às variáveis globais tem o inconveniente de não se integrar na filosofia do MatPLC, na qual cabe ao núcleo do MatPLC a co-ordenação de todos os recursos que sejam partilhados pelos módulos. Já a segunda solução tem o inconveniente de ser difícil, se não impossível, de implementar a semântica da partilha das variáveis globais recorrendo apenas aos pontos do MatPLC. De facto, no caso de dois programas distintos (cada qual atribuído à sua tarefa) acederem os dois à mesma variável global, acontece que a informação deverá estar disponível para os dois programas com prontidão, sendo que mal um programa altere o valor armazenado na variável global esse valor deverá estar disponível para o segundo programa. Já no caso das variáveis (ou pontos) do MatPLC, a actualização da cópia que cada módulo guarda da variável será efectuada apenas periodicamente.

É esta segunda consideração que sugere que a opção (a) será a mais adequada. Neste caso, e como referido anteriormente, cabe agora ainda decidir se as tarefas serão implementadas com recurso aos threads do sistema operativo, ou se deverão ser implementadas pelo próprio código gerado pelo compilador para o MatPLC. Para esta segunda escolha são já válidas as duas opções, pelo que a escolha deverá ser adiada para a altura que se vier a implementar este mapeamento das configurações no modelo do MatPLC, altura em que

possívelmente outras considerações práticas poderão indicar que uma das duas opções ser mais adequada.

Assim, e por enquanto, cada configuração é convertida numa classe em C++. Esta classe contem uma única função membro 'void run(void)' (para alem dos construtores e destrutores), que será chamada para executar a configuração. Esta função de momento limita-se a chamar a função 'f' do programa que a configuração irá executar.

O standard IEC 61131-3 prevê ainda que as configurações possam declarar variáveis globais. Apenas os programas instanciados dentro dessa mesma configuração poderão aceder as essas variáveis. No entanto, o mecanismo de aceder às variáveis globais exige que o programa em si declare uma referencia à variável global dentro da sua própria declaração. O acesso às variáveis globais por parte do programa será sempre efectuada através dessa referencia. Cabe á configuração, na altura em que cria a instância do programa, definir o mapeamento entre as variáveis globais e as referencias do programa, ou seja, de definir qual a variável global que irá ser afectada quando for acedida cada variável de referencia. As variáveis de referencia são declaradas como VAR_EXTERN dentro do programa, ou seja, variáveis externas (ao programa), enquanto que as variáveis globais são declaradas como VAR_GLOBAL dentro da configuração.

Nestes casos as variáveis globais são convertidas em variáveis internas à classe que representa a configuração. Por seu lado, as variáveis externas dos programas são declaradas como referencias a variáveis, isto dentro das classes nas quais foram convertidos esses mesmos programas. Estas variáveis de referencia são inicializadas pelo construtor da classe que representa o programa. O construtor é invocado pela classe que representa a configuração aquando da instanciação do programa, sendo nesta altura especificadas as variáveis globais às quais farão referencia cada uma das variáveis de referencia do programa instanciado.

Por exemplo, o programa e configuração da fig. 4.21 são convertidos para as classes C++ da figura 4.20.

```
// PROGRAM
class SCROLL {
  public:
  private:
    BOOL &left_bt, &right_bt;
    INT &bits;
  public:
    SCROLL(BOOL *left_bt, BOOL *right_bt, INT *bits)
     :left_bt(*left_bt), right_bt(*right_bt), bits(*bits)
     {}

  public:
    void f(void) {
      // código respeitante ao corpo do programa
    } /* f() */

}; /* class SCROLL */
```

Figura 4.20 - Classe C++ correspondentes à conversão da configuração da fig. 4.21.

```
PROGRAM SCROLL
  VAR_EXTERNAL
    left_bt, right_bt : BOOL;
    bits : INT;
  END_VAR
  (* Program Body *)
  (* codigo respeitante ao corpo do programa... *)
END_PROGRAM

CONFIGURATION config1
  VAR_GLOBAL
    bit_count : INT := 1;
    lbit, rbit : BOOL;
   END_VAR

  PROGRAM main: SCROLL(left_bt := lbit, right_bt := rbit,
bits := bit_count);
END_CONFIGURATION
```

Figura 4.21 - Exemplo de declaração de um programa e de uma configuração.

Conversão de Variáveis Localizadas

O standard prevê ainda a existência de variáveis localizadas, utilizadas para aceder às entradas e saídas físicas dos PLCs. No caso do MatPLC, torna-se necessário aceder aos pontos armazenados no mapa global do MatPLC a partir dos programas IL e ST. Uma vez que, do ponto de vista dos módulos, estes pontos tomam a vez de as entradas e saídas físicas dos PLCs, torna-se natural que se aceda a estes pontos através de variáveis localizadas. Para tal o compilador que foi desenvolvido foi implementado de tal forma que sejam aceites variáveis localizadas com uma sintaxe para a descrição do local da mesma distinta da que é definida no standard. Como já foi referido, o standard permite que uma variável seja localizada em uma de três zonas de memória: Q para as saídas, I

para as entradas, e M para memória interna. O local de armazenagem dentro de cada uma destas zonas é especificado com números separados por pontos, e a dimensão da variável em bits é dada por uma segunda letra opcional. Assim, são permitidas as especificações de locais tais como '%I3.45', '%MW2.87.5', ou '% QB3'. No entanto, e uma vez que os pontos do MatPLC não seguem a sintaxe descrita, podendo ser identificadas por uma sequência de caracteres, o compilador foi desenvolvido de forma a aceitar variáveis localizadas com localizações determinadas por um identificador, como por exemplo '%var1', '% lâmpada', '%motor', etc.

Estas variáveis são mapeadas em objectos C++, os quais se responsabilizam por invocar as funções do MatPLC necessárias para ler ou escrever os dados armazenados nestas variáveis dentro da zona de memoria partilhada do MatPLC. O facto de serem convertidas para objectos permite que o restante código possa aceder a estas variáveis localizadas como a quaisquer outras, não sendo assim necessário ter em conta o tipo de variável quando se faz a conversão do código IL e ST para C++.

Os objectos para as quais são mapeadas estas variáveis são todos de uma mesma classe modelo (template class), sendo o parâmetro da classe modelo o tipo de dado armazenado no ponto do MatPLC, a partir do qual se poderá determinar o numero de bits desse mesmo ponto.

```
template<typename value_type, int size = 8 * sizeof
(value_type)> class __plc_pt_c {
  private:
    plc_pt_t plc_pt;
  // ...
  public:
    virtual void operator= (value_type value) {
      plc_set(plc_pt, *((u32 *)&value));
    }

    virtual operator value_type(void) {
      u32 tmp_val = plc_get(plc_pt);
      return *((value_type *)&tmp_val);
    }
  // ...
};
```

Figura 4.22 - Classe modelo utilizada no mapeamento de variáveis com mapeamento directas.

Esta classe mantêm uma referencia ao ponto do MatPLC, a qual é criada pelo construtor da classe utilizando o nome do ponto do MatPLC na qual é armazenada a variável localizada. A classe re-define ainda o operador de atribuição '=', de forma a que a função plc_set() seja invocada de cada vez que é atribuído um valor diferente à variável localizada, e assim o novo valor seja transferido para o mapa gerido pela secção GMM do núcleo do MatPLC. De forma semelhante, é definido a conversão da classe para o tipo utilizado como

parâmetro da classe modelo, o que permite que um objecto desta classe seja trabalhado como se fosse uma variável do tipo de dado da qual pertence a variável global.

Assim, e como exemplo, uma variável declarada em ST ou IL da seguinte forma

```
VAR temperature AT %oiltemp: INT; END_VAR
```

é convertida para C++ como

```
__plc_pt_c<INT, 8*sizeof(INT)>  temperature;
```

sendo depois inicializada no construtor da função bloco ou programa na qual se encontra declarada com 'temperature(“oiltemp”);'.

A partir deste momento, é possível fazer atribuições directas ao objecto, ou aceder ao mesmo, como se fosse um INT. Ou seja, passa a ser possível fazer

```
temperature = 35;
var1 = temperature + 5;
```

dentro do código C++, o que facilita a conversão para C++ da restante sintaxe das linguagens IL e ST.

Conversão dos Tipos de Dados

O standard IEC 61131-3 define um conjunto de tipos de dados elementares próprio, sendo portanto necessário mapear estes tipos nos tipos de dados disponíveis em C++. Para facilitar a conversão dos programas, todos os programas C++ começam por uma definição dos tipos de dados definidos no standard IEC 61131-3 para os seus equivalentes em C++.

Para os CPUs com uma arquitectura compatível com o x86 da Intel, o mapeamento dos tipos de dados mais simples segue o que se encontra definido na tabela da fig. 4.23. Alguns dos restantes tipos de dados mais complexos recorrem já a dados pré-definidos pelo POSIX. O standard suporta ainda a definição de tipos de dados derivados. Estes são convertidos para C++ de forma relativamente directa, uma vez que o C++ suporta também tipos de dados derivados baseados em estruturas, enumerações e arrays. Os tipos de dados derivados baseados em sub-gamas de tipos de dados elementares numeráveis são convertidos como o tipo de dado elementar numerável no qual é baseada a sub-gama.

C++ (na arquitectura x86)	MatPLC	IEC 61131-3
`bool`		`BOOL`
`signed char`	`i8`	`SINT`
`short int`	`i16`	`INT`
`int`	`i32`	`DINT`
`long long int`	`i64`	`LINT`
`unsigned signed char`	`u8`	`USINT`
`unsigned short int`	`u16`	`UINT`
`unsigned int`	`u32`	`UDINT`
`unsigned long long int`	`u64`	`ULINT`
`unsigned signed char`	`u8`	`BYTE`
`unsigned short int`	`u16`	`WORD`
`unsigned int`	`u32`	`DWORD`
`unsigned long long int`	`u64`	`LWORD`
`float`	`f32`	`REAL`
`double`	`f64`	`LREAL`

Figura 4.23 - Conversão dos tipos de dados elementares para C++.

C++	IEC 61131-3

`struct timespec {`
`  /* Seconds. */`
`  time_t tv_sec;`
`  /* Nanoseconds. */`
`  long int tv_nsec;`
`};`
	`TIME`
	`TIME_OF_DAY`

`struct date {`
`  u8 day;`
`  u8 month;`
`  u16 year;`
`}`
	`DATE`

`struct date_time {`
`  struct date d;`
`  struct timespec t;`
`}`
	`DATE_AND_TIME`

`u8 *`	`STRING`
`u16 *`	`WSTRING`

Figura 4.24 - Conversão dos tipos de dados derivados.

Conversão de Código ST

A conversão do código ST é relativamente simples, visto tratar-se de uma linguagem com uma sintaxe bastante próxima do C++. No entanto, as invocações de funções e de funções bloco, devido à semântica imposta pelo standard, implica um trabalho extra, sendo assim relativamente trabalhoso de implementar. Nesta descrição do mapeamento da linguagem ST para C++ será abordado em primeiro lugar as expressões, uma vez que estas poderão ser utilizadas dentro de várias das declarações da linguagem ST.

As expressões em ST são então convertidas para expressões em C++, não sendo necessário alterar significativamente a sintaxe. Apesar disto, tornou-se ainda necessário transformar a operação de exponenciação de ST para uma chamada a uma função em C++, pois esta última linguagem não contem uma operação de exponenciação. A semântica das expressões é também a mesma no que concerne à sua avaliação, i.e. da esquerda para a direita, e ainda uma avaliação que termina logo no ponto em que fica a conhecer o resultado de expressões booleanas.

Já a conversão da maioria das declarações é também relativamente directa. De facto, as declarações de atribuição (':='), de selecção (IF THEN ELSE END_IF e CASE END_CASE), de iteração (FOR DO END_FOR, WHILE DO END_WHILE e REPEAT UNYIL END_REPEAT), e de controlo de fluxo (EXIT) têm correspondente directo em C++ nas declarações de atribuição ('='), de selecção (if then else, e switch), de iteração (for, while, e repeat until) e de controlo de fluxo (exit) respectivamente, sendo que a sintaxe sofre pequenas alterações.

As invocações de funções e de funções bloco sofrem já alterações substancias, devido principalmente à semântica que difere bastante entre as duas linguagens. Como já foi referido, na conversão de funções bloco para C++ são retirados os parâmetros da função bloco, sendo estes declarados como variáveis da classe que representa a função bloco. Isto permite que os parâmetros que não sejam especificados numa determinada invocação possam manter os seus valores da invocação anterior.

Por outro lado, a invocação de funções já não sofre da questão de semânticas distintas, mas sim da sintaxe que é difícil de converter de forma linear. De facto, na invocação de funções em ST, os parâmetros que não sejam especificados são inicializados com um valor especifico desse parâmetro. O C++ também permite este mecanismo na declaração de funções como no exemplo seguinte:

```
int mcdenominator(int a = 1; int b = 1, int c = 1);
```

em que a função pode até ser invocada sem qualquer parâmetro. No entanto, esta sintaxe do C++ obriga a que os parâmetros que se deixam por especificar sejam sempre os últimos da lista de parâmetros. Ou seja, no exemplo anterior, não seria possível invocar a função se se quiser passar os valores 2 e 3 paras os

parâmetros b e c respectivamente, e deixar o parâmetro a no seu valor de inicialização. Pelo contrário, a sintaxe das invocações formais em ST permite que qualquer conjunto de parâmetros seja especificada de forma explícita, enquanto que os restantes são deixados para tomar os seus valores de inicialização.

O método utilizado para contornar esta questão foi o de não recorrer à sintaxe do C++ que permite especificar os valores a utilizar por omissão para os parâmetros de invocação de funções, e especificar sempre todos os parâmetros, mesmo que estes tenham sido deixados por especificar no programa em ST. Neste caso o compilador responsabiliza-se por determinar qual o valor de inicialização do referido parâmetro e de o passar de forma correcta. Infelizmente, o mesmo já não se pode fazer com tanta facilidade para os parâmetros de saída de dados, para os quais se torna necessário especificar uma variável na qual se irá colocar o resultado da invocação da função. Para tal são declaradas variáveis temporárias que são utilizadas apenas para este fim, e cujos conteúdos são simplesmente ignorados.

```
FUNCTION f: INT
  VAR_IN_OUT
    inout1, inout2: INT;
  END_VAR
  VAR_INPUT
    in1, in2: INT;
  END_VAR
  VAR_OUTPUT
    out1, out2: INT;
  END_VAR

  (* ... *)
END_FUNCTION
```

Figura 4.25 - Declaração de função utilizada nos exemplos de conversão de invocações.

Por exemplo, quando a função da fig. 4.25 é invocada da seguinte forma:

```
i3 := f(in1 := 100, inout1 := i1, out1 => i2);
```

resulta o código C++

```
  INT __TMP_0 = 0;
  INT __TMP_1 = 0;

  i3 = f(i1, __TMP_0, 100, 0, i2, __TMP_1);
```

Caso a função f seja declarada como uma função bloco fb_t, com os mesmos parâmetros, então a invocação

```
VAR fb: fb_t; END_VAR
fb(i1, __TMP_0, 100, 0, i2, __TMP_1);
```

resulta no seguinte código C++

```
fb_t fb;
fb.inout1 = i1;
fb.in1 = 100;
fb.f();
i1 = fb.inout1;
i2 = fb.out1;
```

A Conversão da Linguagem IL

A linguagem IL assemelha-se à linguagem máquina no sentido que é composta por instruções simples que geralmente utilizam um registo comum para obter parâmetros e guardar o resultado da instrução.

A conversão desta linguagem para C++ passou por, desde logo, começar por definir e declarar a variável que seria utilizada para o registo comum. Uma vez que esta variável deverá ser capaz de armazenar diversos tipos de dados, e para não desperdiçar memória, todos os blocos de código em IL começam por declarar a variável comum. Esta variável é do tipo __IL_DEFVAR_T, uma união dos tipos de dados elementares.

```
typedef union __IL_DEFVAR_T {
        BOOL    BOOLvar;

        SINT    SINTvar;
        INT     INTvar;
        DINT    DINTvar;
        LINT    LINTvar;
        USINT   USINTvar;
        UINT    UINTvar;
        UDINT   UDINTvar;
        ULINT   ULINTvar;

        BYTE    BYTEvar;
        WORD    WORDvar;
        DWORD   DWORDvar;
        LWORD   LWORDvar;

        REAL    REALvar;
        LREAL   LREALvar;

        DATE    DATEvar;
        TIME    TIMEvar;
        TOD     TODvar;
        DT      Dtvar;
} __IL_DEFVAR_T;
```

Figura 4.26 - União de tipos de dados elementares utilizada para a declaração da variável comum da linguagem IL.

A partir deste momento, a conversão do programa para C++ passa a ser relativamente directo. Por exemplo, o código IL da fig. 4.27, é convertido para o código C++ da fig. 4.28.

```
LD left_bt
R  direction
LD right_bt
S  direction

LD direction
JMPC sleft

sright: LD bits
        SCROLL_RIGHT 16
        ST bits
        RET

sleft: LD bits
       SCROLL_LEFT 16
       ST bits
       RET
```

Figura 4.27 - Exemplo de código IL.

```
    __IL_DEFVAR_T __IL_DEFVAR;

     __IL_DEFVAR.BOOLvar = left_bt;
     if (__IL_DEFVAR.BOOLvar) direction = false;
     __IL_DEFVAR.BOOLvar = right_bt;
     if (__IL_DEFVAR.BOOLvar) direction = true;
     __IL_DEFVAR.BOOLvar = direction;
     if (__IL_DEFVAR.BOOLvar) goto sleft;
     sright:
     __IL_DEFVAR.INTvar = bits;
     __IL_DEFVAR.INTvar =
                SCROLL_RIGHT(__IL_DEFVAR.INTvar, 16);
     bits = __IL_DEFVAR.INTvar;
     goto __end;
     sleft:
     __IL_DEFVAR.INTvar = bits;
     __IL_DEFVAR.INTvar =
                SCROLL_LEFT(__IL_DEFVAR.INTvar, 16);
     bits = __IL_DEFVAR.INTvar;
     goto __end;

     __end:
```

Figura 4.28 - Código C++ correspondente à conversão do código IL da fig. 4.27.

A linguagem IL permite ainda a utilização de modificadores tais como o '(' e ')'. Nestes casos torna-se necessário armazenar o valor na variável comum para que esta possa ser utilizada para um novo conjunto de instruções entre os parenteses. A acção de armazenar o valor da variável comum é geralmente implementado por uma pilha FILO (First In, Last Out). No entanto, esta pilha seria de dimensão fixa, limitando assim o numero de parenteses imbricados que seriam suportados pelo compilador. Como alternativa foi utilizado o esquema de definir um novo bloco de código em C++ para cada parenteses, sendo assim possível definir uma

nova variável comum sem destruir a anterior. Passa-se assim o ónus de gerir a pilha para o compilador gcc, ficando assim limitados apenas pela quantidade de memória disponível no computador. Por exemplo, o código IL da fig. 4.29, é convertido para o código C++ da fig. 4.30.

```
LD fall
AND old_signal
ANDN signal
OR (
LD rise
AND signal
ANDN old_signal
)
ST edge
LD signal
ST old_signal
```

Figura 4.29 - Exemplo de código IL.

```
__IL_DEFVAR.BOOLvar = fall;
__IL_DEFVAR.BOOLvar &= old_signal;
__IL_DEFVAR.BOOLvar &= !signal;
{
  __IL_DEFVAR_T __IL_DEFVAR;

  __IL_DEFVAR.BOOLvar = rise;
  __IL_DEFVAR.BOOLvar &= signal;
  __IL_DEFVAR.BOOLvar &= !old_signal;

  __IL_DEFVAR_BACK.BOOLvar = __IL_DEFVAR.BOOLvar;
}
__IL_DEFVAR.BOOLvar |= __IL_DEFVAR_BACK.BOOLvar;
edge = __IL_DEFVAR.BOOLvar;
__IL_DEFVAR.BOOLvar = signal;
old_signal = __IL_DEFVAR.BOOLvar;
```

Figura 4.30 - Código C++ correspondente à conversão do código ST da fig. 4.29.

Tendo em conta que a semântica da invocação de funções e de funções bloco não difere da linguagem ST, não é de admirar que a conversão destas operações segue de perto o que é utilizado para a linguagem ST. De facto, também para IL torna-se necessário recorrer a variáveis temporárias para a invocação de funções quando algum parâmetro de saída não tenha sido explicitamente especificado. Por outro lado, a invocação de funções bloco é também efectuada com uma pré inicialização dos parâmetros de entrada, e seguida da leitura dos parâmetros de saída. De facto, a única diferença reside na sintaxe utilizada para expressar as invocações na linguagem IL, que exige assim uma classe própria (generate_cc_il_c) para implementar a conversão.

Deve no entanto ser salientado que por enquanto a conversão ainda não permite a invocação de funções e funções bloco com a sintaxe de invocação formal, e em

que adicionalmente é utilizada uma lista de operações IL para estabelecer o valor a passar a pelo menos um dos parâmetros. Ou seja, invocações com a sintaxe exemplificada na fig. 4.31 não são ainda suportadas.

```
LIMIT(
  IN := var1,
  MIN := (
    LD var2
    ADD var3
  ),
  MAX := var4
)
```

Figura 4.31 - Exemplo de sintaxe de invocação de funções em IL ainda não suportadas.

A conversão de invocações como esta necessita que a lista de instruções dentro da invocação seja previamente avaliada, e o seu resultado armazenado numa variável temporária, para ser posteriormente utilizada na invocação. Ou seja, para o exemplo da fig. 4.31 anterior, a invocação da função seria convertida para o código C++ da fig. 4.32.

```
{
   INT __TMP_0 = 0;
    __IL_DEFVAR_T __IL_DEFVAR_BACK;
    __IL_DEFVAR_T __IL_DEFVAR;

    // código anterior à invocação da função...
    {
      __IL_DEFVAR_T __IL_DEFVAR;

      __IL_DEFVAR.INTvar = var2;
      __IL_DEFVAR.INTvar += var3;
      __TMP_0=__IL_DEFVAR.INTvar;
    }
    LIMIT(var1, __TMP_0, var4);
    // código posterior à invocação da função...
}
```

Figura 4.32 - Possivel conversão para C++ da invocação de função expressa na fig. 4.31.

Devido ao facto de que implementar esta conversão é relativamente trabalhosa, e de que a mais valia que traria ao compilador não seria da maior importância, optou-se por adiar a sua implementação para outra oportunidade.

Uma outra limitação surge devido à utilização da união __IL_DEFVAR_T para a declaração do registo comum. De facto, deve ser possível a atribuição de valores de tipos de dados derivados ao registo comum. Esta funcionalidade é utilizada para, por exemplo, copiar o conteúdo de uma variável para outra. Por exemplo, o código da fig. 4.33 não seria suportado.

Para a correcção desta falta do compilador, e para o exemplo acima, torna-se necessário acrescentar uma variável à união __IL_DEFVAR_T para os tipos de

dados derivados 'point_t'. Uma vez que o programador do programa IL é livre de declarar os tipos de dados derivados que achar necessários, e com os nomes que entender, a declaração do tipo __IL_DEFVAR_T tem de ser efectuada de forma dinâmica pelo compilador em si, após uma análise do código em causa. Assim, o compilador necessita de ser corrigido de forma a efectuar uma primeira iteração por todas as instruções de uma lista IL para determinar o conjunto de tipos de dados que serão armazenados no registo comum. Com esta informação, deverá então declarar o tipo __IL_DEFVAR_T que será utilizado apenas para essa lista de instruções. Assim, o código IL da fig. 4.33, deveria resultar no código C++ da fig. 4.34.

```
TYPE
  point_t : STRUCT
    x, y : INT;
    END_STRUCT;
END_TYPE

FUNCTION f : INT
VAR
  i1, i2: INT;
  p1, p2 : point_t;
END_VAR
(* ... *)
LD p1
ST p2
LD i1
ST i2
END_FUNCTION
```

Figura 4.33 - Exemplo de código IL por enquanto ainda não suportado.

```
INT f(...) {
  typedef union {
    point_tvar: point_t;
    INTvar: INT;
  } __IL_DEFVAR_T;
  __IL_DEFVAR: __IL_DEFVAR_T;

  INT f;
  point_t p1, p2;
  INT i1, i2;

  __IL_DEFVAR.point_tvar = p1;
  p2 = __IL_DEFVAR.point_tvar;
  __IL_DEFVAR.INTvar = i1;
  i2 = __IL_DEFVAR.INTvar;
  return f;
}
```

Figura 4.34 - Possivel conversão para C++ do código IL da fig. 4.33.

Deve ser salientado que esta questão não pode ser tratada com um apontador genérico na união __IL_DEFVAR_T, e usando esse apontador para armazenar o endereço de memória em que se encontra armazenado o valor que foi colocado no registo comum quando esse valor é de um tipo de dado não elementar. Neste caso, as duas instruções da função anterior seriam convertidas no seguinte código C++:

```
__IL_DEFVAR.generic_ptr = (void *)&p1;
p2 = *((point_t *)__IL_DEFVAR.generic_ptr);
__IL_DEFVAR.INTvar = i1;
i2 = __IL_DEFVAR.INTvar;
```

Esta solução não é correcta pois apenas funcionaria se o valor da variável 'p1' nunca pudesse sofrer alterações enquanto que o registo comum não fosse re-carregado com outro valor. De facto, existe um cenário em que tal pode acontecer, que corresponde à invocação de uma função bloco. Por exemplo, no código IL seguinte, à variável 'p1' sofre uma alteração do seu valor ainda antes do

registo comum ser recarregado.

```
LD p1
CAL funcblock(
      param1 => p1
    )
ST p2
```

Mais uma vez, e devido à morosidade de implementar a iteração prévia de todas as instruções da lista para determinar todos os tipos de dados que irão ser armazenados no registo comum, esta correcção ao compilador foi adiada para oportunidades futuras.

4.3 Questões Relacionadas com o IEC 61131-3

No decorrer da implementação do compilador para as linguagens IL e ST tornou-se necessário fazer um estudo pormenorizado destas linguagens, tanto a nível sintáctico como semântico. Ao fazer esta análise foram encontrados no standard pequenas incongruências na definição da sintaxe destas linguagens que se podem atribuir a gralhas, gralhas essas que se esperam tenham sido corrigidas na versão final do standard. No entanto, foram ainda desvendadas pormenores das linguagens que dificilmente se podem atribuir a gralhas. De facto, através de troca de emails pessoais com algumas pessoas envolvidas no desenvolvimento do standard (Heinz-Dieter da Modicon, Jim Christensen da Rockwell, e Eelco van der Wal, director executivo do PLCopen) ficou claro que as opções eram intencionais muito embora por vezes as consequências destas opções não tenham sido completamente compreendidas.

Passa-se então a detalhar as questões encontradas no standard, e a sugerir alterações de forma a obviar os inconvenientes encontrados.

4.3.1 Nomes de Variáveis

As linguagens IL e ST, bem como a arquitectura de programação nas quais se enquadram, incluem diversas entidades às quais o programador poderá atribuir um identificador. Por exemplo, os tipos de dados derivados especificados pelo programador, as funções, funções bloco, programas e configurações que o programador define para atingir os fins pretendidos, as variáveis utilizadas nestes blocos de programação, bem como os itens de uma enumeração. No entanto, e em alguns casos, o standard é ambíguo quanto à possibilidade ou não de re-utilizar para um segundo elemento sintáctico o identificador já utilizado para identificar um outro elemento.

Em primeiro lugar, pode-se admitir que os identificadores não deverão ser re-

utilizados para elementos sintácticos do mesmo tipo pela simples razão de que, caso contrário, esses dois elementos sintácticos nunca poderiam ser distinguidos entre si. Por exemplo, não faz sentido permitir que duas funções tenham o mesmo nome, nem que dois tipos de dados derivados partilhem o mesmo identificador. No entanto, a partilha de identificadores por parte de elementos sintácticos de classes distintas já poderá fazer algum sentido. O nome atribuído a um programa poderia ser re-utilizado para identificar uma variável dentro desse mesmo programa, ou até dentro de uma qualquer outra função bloco ou função. Em caso de nada ser dito, deverá então ser assumido que os identificadores podem ser re-utilizados para elementos sintácticos de classes distintas.

No entanto, o standard limita de certa forma esta re-utilização de identificadores para alguns casos. Isto é efectuado de uma forma não linear. De facto, a principal afirmação pela qual se pode tirar conclusões acerca da re-utilização dos identificadores encontra-se na secção 2.1.3 do standard, a qual poderá ser traduzida para "as palavras chave definidas no anexo C não deverão ser utilizadas para qualquer outro motivo, por exemplo nomes de variáveis...". Tal como previsto, no anexo C do standard IEC 61131-3 encontram-se discriminadas as palavras chave que seriam de esperar, e que aqui se encontram repetidas na fig. 4.35.

No entanto, e de forma inesperada, a esta lista foram ainda acrescentados quaisquer nomes que tenham sido utilizados para identificar tipos de dados, funções e funções bloco. Daqui se pode concluir que os identificadores utilizados para identificar uma função, por exemplo, deixa de estar disponível para identificar uma variável, ou mesmo um valor de uma enumeração. Assume-se então que a re-utilização de identificadores é perfeitamente legal nos restantes casos. Por exemplo, uma variável poderá re-utilizar o nome de um programa, de uma configuração, ou mesmo um valor de uma enumeração. De facto, a existência de uma sintaxe utilizada para resolver situações de ambiguidade na distinção de uma variável de um valor de uma enumeração reforça esta conclusão. Caso contrário esta sintaxe seria perfeitamente desnecessária. Considere-se, por exemplo, a seguinte declaração de um tipo de dado derivado:

```
TYPE
  cores_t: (branco, azul, vermelho, roxo, verde);
END_TYPE
```

Considere-se agora o seguinte extracto de código ST

```
VAR
  var1, branco: cores_t;
END_VAR
branco := azul;
var1 := branco;
```

```
ACTION, END_ACTION
ARRAY, OF
AT
CASE, OF, ELSE, END_CASE
CONFIGURATION, END_CONFIGURATION
CONSTANT
EN, ENO
EXIT
FALSE
F_EDGE
FOR, TO, BY, DO, END_FOR
FUNCTION, END_FUNCTION
FUNCTION_BLOCK, END_FUNCTION_BLOCK
IF, THEN, ELSIF, ELSE, END_IF
INITIAL_STEP, END_STEP
NOT, MOD, AND, XOR, OR
PROGRAM, WITH
PROGRAM, END_PROGRAM
R_EDGE
READ_ONLY, READ_WRITE
REPEAT, UNTIL, END_REPEAT
RESOURCE, ON, END_RESOURCE
RETAIN, NON_RETAIN
RETURN
STEP, END_STEP
STRUCT, END_STRUCT
TASK
TRANSITION, FROM, TO, END_TRANSITION
TRUE
TYPE, END_TYPE
VAR, END_VAR
VAR_INPUT, END_VAR
VAR_OUTPUT, END_VAR
VAR_IN_OUT, END_VAR
VAR_TEMP, END_VAR
VAR_EXTERNAL, END_VAR
VAR_ACCESS, END_VAR
VAR_CONFIG, END_VAR
VAR_GLOBAL, END_VAR
WHILE, DO, END_WHILE
WITH
```

Figura 4.35 - As palavras chave definidas no anexo C do standard IEC 61131-3.

A linha 'var1 := branco' contem uma ambiguidade, uma vez que o identificador 'branco' poderá estar a referir-se tanto à variável 'branco', bem como a um dos valores possíveis que os dados do tipo 'cores_t' poderão tomar. Foi para resolver estas situações de ambiguidade que o standard inclui a sintaxe '<enumerated_type_name>#identifier' para referenciar um valor de um tipo de dado baseado numa enumeração.

Ou seja, uma vez que o valor 'branco' do tipo 'cores_t' poderá ser identificado por 'cores_t#branco', na linha anterior 'var1 := branco', 'branco' deverá ser entendida como estando a referenciar a variável, e não a cor. Assim, e neste caso, a variável

'var1' passará a conter a cor 'azul', e não a cor 'branco'. Se fosse pretendido atribuir a cor branco, então o código correcto seria 'var1 := cores_t#branco'. É de salientar que, caso a variável 'branco' não tivesse sido declarada, a possível ambiguidade não existiria, pelo que o código 'var1 := branco' estaria correcto e seria entendida como estando a atribuir a cor 'branco' à variável 'var1'.

No entanto, e voltando aos identificadores utilizados para identificar funções, tipos de dados e funções bloco que são considerados palavras chave, uma análise mais cuidada a esta classificação revela que poderá resultar em situações confusas para um programador. De facto, é comum que um programa tenha várias versões, sendo este desenvolvido de forma incremental, em que cada versão, embora funcione correctamente e como o pretendido, seja seguida de uma versão posterior que adiciona uma qualquer funcionalidade ao programa. Não é de estranhar até que o programador que acrescenta a funcionalidade nem seja o programador original da versão anterior, e por isso não tenha um conhecimento detalhado dessa primeira versão, mas perceba apenas o suficiente para que seja possível acrescentar a funcionalidade extra pretendida. Ora, nestes casos a classificação dos identificadores das funções, funções bloco e tipos de dados derivados como palavras chave poderá trazer sérios embaraços, pois pode ocorrer que o programa original utilize uma variável com um determinado nome (por exemplo, 'foo'), e que o segundo programador utilize este mesmo nome para identificar uma nova função que tenha acrescentado ao programa. Assim, a partir do momento que a função 'foo' é acrescentada, o identificador 'foo' passa a ser considerado uma palavra chave, e a variável 'foo' deixa de ser legal. Do ponto de vista do segundo programador o programa inicial, que estava correcto em termos de sintaxe e que funcionava correctamente, de repente passa a conter erros de sintaxe e a não compilar. De facto, e mais uma vez do ponto de vista deste segundo programador, se este pretende manter o primeiro programa em funcionamento sem que o queira editar, então terá de considerar todos os identificadores utilizados para todas as variáveis como sendo palavras chaves efectivas, pois não poderá re-utilizar esses identificadores para identificar uma função, função bloco ou tipo de dado derivado.

Este comportamento não está limitado às variáveis, o mesmo ocorrendo para todos os locais em que sejam utilizados identificadores, ou seja: nomes de variáveis, programas, configurações, tarefas (TASK), recursos (RESOURCE), tipo de um recurso, identificador de linha ('label'), elemento de uma estrutura, etapa, acção e transições. Estes últimos três são referentes a SFCs (Sequential Function Charts), os quais também podem ser utilizados para definir o corpo das funções bloco e programas.

Este comportamento que é potencialmente muito confuso para um programador parece ser no entanto totalmente desnecessário. Verifica-se que as linguagens IL e ST contêm informação contextual suficiente para poder distinguir entre o nome de uma função ou função bloco e o nome de uma variável. O nome de um tipo

de dado derivado também não pode ser confundido com o de uma variável, visto estes identificadores serem utilizados em locais muitos distintos dos programas.

Sendo assim, uma possibilidade de melhorar o standard e corrigir a desvantagem que foi apontada é o de recorrer ao conceito de espaço de nomes ('name space'), cuja utilização é comum na definição de linguagens de programação. Um espaço de nomes é no fundo uma base de dados, ou uma lista, que contem identificadores que já tenham sido utilizados para certo fim. Cada identificador não pode aparecer repetido no mesmo espaço de nomes.

Com este conceito, poder-se-ia ter definido um espaço de nomes para os elementos de alto nível (funções, funções bloco e tipos de dados derivados), e ainda outro espaço de nomes para variáveis. Assim, o nome de uma função seria inserida no espaço de nomes de alto nível, passando esse nome a estar indisponível para ser re-utilizado para identificar outra função, função bloco ou tipo de dado derivado. No entanto, e uma vez que os nomes das variáveis são colocados num outro espaço de nomes, o nome que foi atribuído a essa função continuaria a estar disponível para identificar uma variável. A utilização de espaços de nomes permite um controlo mais fino da utilização dos identificadores do que simplesmente considerar o identificador como uma palavra chave, passando a estar indisponível para qualquer outra utilização.

Para que esta sugestão seja válida, deverá ser possível distinguir a entidade a que se refere um identificador que aparece nos dois espaços de nomes. Ou seja, quando aparece no código um identificador que tenha sido utilizado para identificar uma variável e uma função em simultâneo, deverá ser possível distinguir pelo contexto em que aparece esse identificador a qual dos elementos sintácticos se refere. Como já foi referido, verifica-se que as linguagens IL e ST contêm informação contextual suficiente para que tal seja possível. Para confirmar esta afirmação será necessário verificar todos os locais em que poderão ser utilizados os identificadores dentro das linguagens, o que se passa a fazer.

A utilização dos identificadores de tipos de dados derivados resume-se a:

```
declaração de variáveis ::= <lista_de_variàveis> ':'
<tipo_de_dado>
identificação de valores de enumerações ::=
<tipo_de_dado_enumerado>#<identificador>
```

Na declaração de variáveis não existe possibilidade de confusão, uma vez que à esquerda de ':' todos os identificadores se referem a variáveis, enquanto que à direita os identificadores se referem a um tipo de dados. O mesmo ocorre com a identificação de valores de enumerações, que pode aparecer dentro de expressões em ST, ou ser utilizado como parâmetro de uma instrução IL, tal como os identificadores de variáveis. No entanto, neste caso será sempre possível distinguir uma variável de um tipo de dado uma vez que o identificador de um tipo de dado vem sempre seguido de um '#', o que nunca poderá ocorrer para uma variável.

A utilização de identificadores de funções, para além da declaração da própria função, resume-se à invocação da mesma. A sintaxe de invocação de funções em ST é

```
<nome_de_função> '(' <lista_de_parametros> ')'
```

o que poderá aparecer nos mesmos locais de uma variável, ou seja dentro de uma expressão. Também aqui se verifica que a referencia a uma função não se poderá confundir com a referencia a uma variável, uma vez que todas as referencias a uma função vêm seguidas de '(', o que nunca ocorre com uma variável.

Já a sintaxe de invocação de funções em IL é

```
<nome_de_função> '(' <lista_de_parametros_il> ')'
```

ou

```
<nome_de_função> [ <lista_de_operandos_il> ]
```

as quais são utilizadas como uma instrução IL, ou seja, aparecem cada uma numa linha só por si, tal como as restantes instruções da linguagem IL. Assim, também aqui o identificador que referencia o nome de uma função nunca poderá ser confundido com uma referencia a uma variável, uma vez que as varáveis nunca são utilizadas no início de uma instrução IL, sendo antes utilizadas como operandos de uma instrução IL.

A utilização dos nomes de um tipo de função bloco é ainda mais restrita do que as funções, uma vez que são utilizados apenas na instanciação de funções bloco, ao lado das declarações de variáveis. Assim, os nomes de tipos de função bloco aparecem em

```
<lista_de_funções_bloco> : <tipo_de_função_bloco>
```

os quais nunca poderão ser confundidos por identificadores de variáveis visto serem precedidos de ':'.

Verifica-se assim que será possível a partilha de identificadores entre variáveis e as funções, tipos de funções bloco, e tipos de dados derivados. No entanto, e como foi referido acima, a utilização de identificadores não se resume a estes elementos sintácticos, sendo ainda utilizados para identificar programas, configurações, tarefas (TASK), recursos (RESOURCE), tipos de recursos, identificadores de linha ('label'), elementos de estruturas, etapas, acções e transições.

Os nomes de tipos de programas, para além de serem utilizados na declaração do próprio programa, são usados apenas na instanciação de programas dentro de configurações, com a sintaxe

```
PROGRAM <nome_de_programa> [WITH <nome_de_tarefa>] ':'
<nome_de_tipo_de_programa> [ '('
<parametros_de_configuração_do_programa> ')' ]
```

que utiliza a palavra chave 'PROGRAM', sendo por isso impossível de confundir com uma referencia a uma função, tipo de função bloco, ou tipo de dado

derivado.

O mesmo ocorre para as tarefas, cujos identificadores aparecem sempre precedidos da palavra chave 'WITH', seja ele na declaração de programas como na sintaxe acabada de ser mencionada acima, ou na atribuição de uma tarefa a uma instância de uma função bloco (dentro dos parâmetros de configuração de um programa), com a sintaxe

```
<nome_de_instancia_de_função_bloco> WITH <nome_de_tarefa>
```

Os nomes de configurações utilizam-se apenas na sua própria declaração, não estando prevista a sua utilização pelo standard em noutro local da sintaxe, pelo que, mais uma vez, não poderá ser confundida com uma referencia a um elemento cujo nome seja colocado no espaço de nomes de alto nível.

Os recursos e tipos de recursos são utilizados com a sintaxe

```
RESOURCE <nome_do_recurso> ON <nome_do_tipo_de_recurso>
```

Adicionalmente, o nome de um recurso aparece ainda em elementos sintácticos que têm o objectivo de qualificar uma determinada variável,

```
<nome_do_recurso> '.' <nome_de_variavel>
```

isto sempre dentro da declaração de uma configuração. Mais uma vez se verifica que tanto o nome do recurso como o nome do tipo de recurso nunca se poderão confundir com referencias a funções, tipo de função bloco ou tipo de dados derivados, pela simples razão de serem sempre precedidos da palavra chave RESOURCE e ON, ou de ser seguido por um '.'.

Razões semelhantes ocorrem para os elementos de estruturas (sempre precedidos por um '.', ou então utilizados dentro de um STRUCT ... END_STRUCT), e de referencias de linhas (labels), sempre seguidos de um ':', ou precedidos de uma instrução de salto IL.

4.3.2 Operadores de Expressões

As sequências de caracteres 'NOT' e 'MOD' são simultaneamente considerados pelo standard como sendo o nome de funções pré-definidas (secções 2.5.1.5.2 e 2.5.1.5.3 do standard), delimitadores e palavras chave (tabela C.1 e C.2, respectivamente, do anexo C do standard).

São considerados delimitadores devido à sua utilização como operadores em expressões ST (secção 3.3.1 do standard), como por exemplo 'var1 := NOT ((var2 MOD 3) > 2)'. São ainda consideradas palavras chave pois são utilizadas como instruções IL (secção 3.2.2 do standard), por exemplo

```
LD var2
MOD 3
GT 2
NOT
ST var1
```

Esta duplicação de definições resulta numa ambiguidade na interpretação de alguns elementos sintácticos. No exemplo 'var1 := NOT ((var2 MOD 3) > 2)', o identificador 'NOT' poderá ser interpretado tanto como sendo um operador de uma expressão, ou como uma referencia à função pré-definida, sendo que neste último caso a expressão conteria uma invocação a essa função.

O mesmo ocorre com as instruções IL, em que as linhas 'NOT' e 'MOD 3' podem ser interpretadas como sendo as instruções definidas na secção 3.2.2, ou como uma invocação às funções pré-definidas.

Em ambos estes casos esta ambiguidade, embora existente, não traz consequências de maior, uma vez que a semântica das operações, instruções IL e das funções pré-definidas serem coincidentes, sendo que qualquer interpretação deverá resultar nos mesmo resultados lógicos e numéricos.

O mesmo, no entanto, não poderá ser dito da questão seguinte, que é de certa forma algo semelhante a esta.

4.3.3 Operadores IL

Ao contrário da primeira versão do standard, a segunda versão deixou de considerar as instruções IL como sendo palavras reservadas. Assim sendo, poder-se-á considerar que as sequências de caracteres utilizadas para identificar as instruções IL também poderão ser utilizadas para identificar variáveis, funções, tipos de funções bloco, etc. Por exemplo, deverá ser permitido declarar funções com os nomes 'LD', 'LDN', 'ST', etc., o mesmo ocorrendo com as variáveis, etc., como no exemplo da fig. 4.36.

```
FUNCTION ST : BYTE
  VAR_INPUT
    a, b : BYTE;
  END_VAR

  ST := a * b;
END_FUNCTION

FUNCTION FOO : BYTE
  VAR_INPUT
    c, LD : BYTE;
  END_VAR

  LD 1
  ST LD
  ST c
END_FUNCTION
```

Figura 4.36 - Exemplo de utilização de nomes de instruções IL para identificar funções declaradas pelo utilizador.

Infelizmente as invocações de funções na linguagem IL têm uma sintaxe que facilmente se confunde com as próprias instruções IL. De facto, as linhas 'ST LD' e 'ST c' da função 'foo' tanto podem ser interpretadas como sendo a instrução IL 'ST' seguidas de um parâmetro, a qual copia o conteúdo do registo comum para as variáveis 'LD' e 'c' respectivamente, ou como sendo uma invocação da função 'ST', que tinha acabado de ser declarada acima, sendo as variáveis 'LD' e 'c' passadas como parâmetro à referida função.

Esta ambiguidade terá de ser levantada através de uma de duas hipóteses. A primeira será a de definir todas as instruções IL como sendo palavras chave, sendo que assim as sequências de caracteres 'ST', 'STN', 'LD' etc. deixam de estar disponíveis para serem utilizadas para referenciar outros elementos sintácticos tais como as funções e variáveis no exemplo acima.

A segunda hipótese prende-se com a re-definição da sintaxe de invocação de funções em IL. Está claro que esta segunda hipótese é mais indesejável do que a primeira, pois assim deixa de ser possível invocar funções, tais como as funções pré-definidas 'SHL', 'SHR', ROL' e 'ROR' (que efectuam deslocações ou rotações à esquerda ou à direita dos bits da variável passada como parâmetro) como se estas funções fossem instruções da própria linguagem IL.

Por outro lado, a única desvantagem da primeira hipótese que se consegue descortinar seria a de reduzir o número de possíveis identificadores disponíveis para utilizar como nomes de funções, tipos de funções bloco, etc., o que não parece ser demasiado grave tendo em conta o enorme numero de identificadores que permanecem disponíveis.

4.3.4 Sequências de Funções Bloco

As linguagens definidas no standard IEC 61131-3 suportam todas elas a declaração de sequências ('ARRAY') de diversos tipos de dados, incluindo de quaisquer tipos de dados derivados tais como estruturas e enumerações, ou mesmo sequências de sequências.

No entanto, o standard não permite a definição de sequências de funções bloco. E isto apesar de ser perfeitamente normal instanciar funções bloco a partir de um tipo de função bloco, assumindo assim o tipo de função bloco quase a qualidade de um tipo de dado, a as instâncias de funções bloco a qualidade de variáveis. Sendo mesmo possível as próprias instâncias de funções bloco ser utilizadas como se de um variável do tipo estrutura se tratasse, acedendo de forma directa às variáveis internas da instância de função bloco com a sintaxe

<nome_de_função_bloco> '.' <nome_de_variável>

Não é justo afirmar que a falta desta funcionalidade nas linguagens do standard IEC 61131-3 é muito prejudicial à sua capacidade de exprimir algoritmos e dados, uma vez que o programador poderá facilmente emular uma sequência de

funções bloco através de uma sequência de um tipo de dado estruturado. Este tipo de dado estruturado deverá ter os mesmos elementos constituintes (i.e. as variáveis da estrutura) aos que teria a função bloco que pretende substituir. O código da função bloco seria depois substituído por uma função, que deveria ter entre os seus parâmetros a variável estruturada que contem as variáveis internas da função bloco que se pretende emular.

No entanto, o mesmo argumento se poderia fazer em relação às próprias tipos de funções bloco, sendo estas substituídas por um tipo de dado estruturado e uma função. Apesar disto, tal como as funções bloco são úteis para os programadores, pois permitem uma melhor compartimentalização e organização do código e dos dados, também as sequências de funções bloco o seriam.

Por este motivo torna-se difícil compreender a razão pela qual a capacidade de definir sequências de funções bloco não foi incluída no standard, especialmente tendo em conta que não introduziria complexidade adicional aos compiladores em si, ou aos interpretadores que executam o código. Adicionalmente, a sintaxe não necessita de sofrer grandes extensões e/ou alterações. Os compiladores e interpretadores já têm de ser capazes de lidar com as funções bloco, cuja implementação interna é muito provavelmente o de uma variável estruturada com todas as variáveis internas da função bloco, e/ou uma ligação gerada no instante da compilação, ou um apontador para o endereço da função com o código da função bloco.

Os compiladores e interpretadores também já têm de suportar sequências de tipos de dados estruturados. Acrescentar o suporte de sequências de funções bloco necessitaria apenas o suporte de sequências das estruturas de dados internas utilizadas para representar as funções bloco que, como foi afirmado, não diferem de forma significativa das sequências de tipos de dados estruturados que já são suportados.

A extensão à sintaxe da linguagem seria limitado a permitir a invocação de funções bloco que fizessem parte de uma sequência. Isto poderia ser efectuada com invocações de acordo com o exemplo da fig. 4.37.

A sintaxe atrás exemplificada, bem como a sintaxe necessária para referenciar as funções blocos inseridas em arrays e dentro de configurações, pode ser definida de forma formal como o que se encontra na fig. 4.38.

As construções que não se encontram aqui definidas (como por exemplo multi_element_variable, il_param_list, param_assignment, etc.) fazem parte do standard IEC 61131-3, e as suas definições podem ser encontradas no anexo B do mesmo.

```
FUNCTION_BLOCK foo_fb
  VAR_INPUT
    a, b, c : BYTE
  END_VAR
  (* ... *)
END_FUNCTION_BLOCK

PROGRAM bar
  VAR
    foo : ARRAY [1 .. 10] OF foo_fb;
  END_VAR
  (* ... *)
  (* sintaxe sugerida para a linguagem ST *)
  foo[8](1, 2, 3);
  (* sintaxe sugerida para a linguagem IL *)
  CAL foo[8](1, 2, 3)
END_PROGRAM
```

Figura 4.37 - Extensão à sintaxe sugerida para a invocação de funções bloco armazenadas em um array.

```
il_fb_call = il_call_operator fb_reference[ ‘(‘ (EOL
{EOL} [il_param_list]) | [il_operand_list] ‘)’ ]

fb_invocation = fb_reference ‘(‘ [param_assignment {‘,’
param_assignment}] ‘)’

fb_reference = fb_name | multi_element_variable

fb_task = fb_reference ‘WITH’ task_name

access_path = [resource_name ‘.’] direct_variable |
[resource_name ‘.’] [program_name ‘.’] [fb_reference
‘.’] symbolic_variable

instance_specific_init = resource_name '.' program_name
'.' [fb_reference '.'] ((variable_name [location] ':'
located_var_spec_init) | (fb_reference ':'
function_block_type_name ':=' structure_initialization))
```

Figura 4.38 - Sugestão de alterações à definição formal da sintaxe necessária para a invocação de funções bloco em arrays.

4.3.5 Identificadores em Configurações

A declaração de uma configuração pode conter vários sub-elementos sintácticos, tais como variáveis globais, recursos, tarefas e programas. No entanto, e mais uma vez, o standard é completamente omisso quanto ao tratamento que se deverá dar aos identificadores utilizados para cada um destes elementos sintácticos. Os mesmos identificadores podem ser re-utilizados para elementos sintácticos de classes distintas, ou será que todos os identificadores têm de ser diferentes por forma a evitar conflitos?

Tal como na primeira questão, talvez também aqui se deva presumir que tudo deverá ser permitido a não ser que seja explicitamente proibido pelo standard. Recorrendo a este principio conclui-se que se deve optar por permitir que os identificadores sejam re-utilizados para elementos sintácticos de classes distintas. Infelizmente, neste caso esta interpretação leva a que a sintaxe definida pelo standard permita situações de ambiguidade na interpretação de caminhos de acesso. O código seguinte exemplifica uma situação em que tal ambiguidade poderá ocorrer:

```
CONFIGURATION config1
  VAR_GLOBAL
    foo: fb_type;
  END_VAR
  TASK bar (INTERVAL := t#20ms, PRIORITY := 4);
  PROGRAM foo WITH bar : p_type(1, 2, 3, 4);
  VAR_ACCESS
    Var : foo.xyz : BOOL;
  END_VAR
END_CONFIGURATION
```

Devera ser reparado como a única variável global, e o único programa da configuração, partilham o mesmo identificador 'foo'. Caso tanto o tipo de função bloco 'fb_type' como o tipo de programa 'p_type' tenham um parâmetro de entrada, saída ou entrada/saída com o nome de 'xyz', então acontece que a referencia a uma variável 'foo.xyz' passa a ser ambígua, podendo referenciar tanto a variável do programa como a da função bloco. Esta ambiguidade de interpretação semântica surge muito embora o exemplo exposto cumpre todas as regras de sintaxe definidas no standard.

Poder-se-á ser tentado a concluir que esteja errado o principio inicial que foi adoptado de assumir que a partilha de identificadores deverá ser permitido a não ser que seja explicitamente proibido pelo standard. No entanto, este mesmo principio é o que nos permitiu concluir, numa das questões já analisadas, que a partilha de identificadores é válida para as variáveis e valores de enumerações, o que justifica a existência da sintaxe

```
enum_type '#' enum_value
```

que permite eliminar ambiguidades resultantes desta partilha de identificadores. Se agora for decidido que o principio de permitir a partilha de identificadores está errada, então deverá também ser concluído que a sintaxe que permite eliminar a ambiguidade seja desnecessária, e deva portanto ser retirada do standard.

4.3.6 Inicialização de Tarefas

O standard IEC 61131 restringe-se a definir a semântica das tarefas, não dando indicação de como estas deverão ser implementadas em cada caso. Embora esta atitude se apresente como correcta, acontece que ao não debruçar-se sobre como

poderão ser implementadas as tarefas, deixa alguns pormenores da semântica das tarefas por especificar, nomeadamente, no que refere á inicialização das mesmas e o tratamento que se deverá dar aos parâmetros de inicialização variáveis.

De facto, o standard permite que aos parâmetros de uma tarefa, tal como o seu período ou a sua periodicidade de execução, possam ser atribuídos os valores armazenados numa variável de saída de um programa. Acontece que o standard não especifica se esta atribuição deverá ser efectuada por referencia à variável, ou por valor. Ou seja, o que deverá ocorrer quando o valor armazenado na variável utilizada para estabelecer o valor de um qualquer parâmetro de uma tarefa sofrer alterações devido à própria execução do programa? Será que a atribuição do valor se deverá efectuar apenas durante a inicialização da tarefa, o que significa que a atribuição é efectuada por valor, ou deverá o parâmetro da tarefa sofrer alterações ao seu valor à medida que a variável for sofrendo mudanças no seu valor com o decorrer da execução do programa? Qualquer uma destas interpretações faz sentido. Cabe no entanto ao standard escolher qual delas é que se deverá considerar como correcta.

Adicionalmente, fica ainda por considerar a possibilidade da variável utilizada para inicializar um parâmetro de uma determinada tarefa pertencer a um programa que até se encontra a ser executado por essa mesma tarefa. Nestes casos surge um problema de inicialização circular entre a tarefa e o programa.

Considere, por exemplo, o extracto seguinte referente à declaração de uma configuração:

```
CONFIGURATION config2
  TASK t (INTERVAL := p.foo);
  PROGRAM p WITH t : p_type(1,2,3,4);
END_CONFIGURATION
```

Aqui o período de execução da tarefa ‘t’ será dado pelo valor armazenado na variável ‘foo’, uma variável de saída do programa p. A duvida reside na interpretação semântica da inicialização. Será que se deverá deixar executar o programa uma vez, para permitir que a variável de saída tome um valor coerente, antes que seja inicializada a tarefa. Mas, neste caso, a tarefa tem de primeiro ser inicializada para que o programa possa ser executado.

Outra alternativa será a de assumir que a tarefa é inicializada primeiro. Neste caso a variável de saída ‘foo’ do programa ‘p’ teria o seu valor de inicialização por omissão, visto o programa ainda não ter tido oportunidade de executar. Esta alternativa não faz grande sentido se for considerado que, para a primeira duvida levantada, tenha sido escolhido a hipótese de atribuir por valor os parâmetros das tarefas, e não por referencia. Isto faria com que o referido parâmetro não pudesse sofrer mais alterações, sendo que assim o seu valor seria sempre o da inicialização por omissão da variável ‘foo’.

Ainda outra duvida relativa às configurações reside no instante a partir do qual

fica disponível para utilização posterior um elemento sintáctico declarado dentro da configuração. Por exemplo, e utilizando a mesma configuração do exemplo anterior, surge a duvida se o código em si deverá ser considerado correcto, uma vez que é efectuada uma referencia ao programa 'p' (ou melhor, a uma variável de saída deste programa) ainda antes deste mesmo programa ter sido declarado. Nada no standard explicita de forma clara se a referencia ao programa antes de este ter sido declarado é válida ou não. A melhor afirmação que pela qual se pode talvez tirar algumas inferencias encontra-se na secção 1.4.3, e refere que os tipos de dados derivados, as funções e tipos de funções bloco ficam disponíveis logo que sejam declarados. Embora esta afirmação não faça qualquer referencia às configurações, poderá eventualmente estender-se o mesmo principio a estes elementos sintácticos, sendo que assim o código do exemplo atrás estaria incorrecto. No entanto, mais uma vez seria necessária tirar conclusões por inferencia, o que não deveria ocorrer num standard. Adicionalmente, a questão relativa ao significado semântico da inicialização dos parâmetros de uma tarefa (por valor ou por referencia) mantêm-se.

De qualquer forma, poderá ainda surgir a dúvida se a inicialização das tarefas deverá ocorrer antes de que qualquer programa tenha hipótese de executar. Ou se por contrário, os programas atribuídos às tarefas entretanto criadas poderão iniciar a sua execução ainda antes de terem sido criadas todas as tarefas que tenham sido declaradas na configuração. Por exemplo, considere a seguinte configuração:

```
CONFIGURATION config3
  TASK t1 (INTERVAL := 30ms);
  PROGRAM p1 WITH t1 : p_type(1,2,3,4);
  TASK t2 (INTERVAL := p.foo);
  PROGRAM p2 WITH t2 : p_type(1,2,3,4);
END_CONFIGURATION
```

Neste caso, surge a duvida se deverão ser primeiro criadas todas as tarefas 't1 e 't2' antes de dar hipótese a qualquer um dos programas executar, ou se pelo contrário, se permite que seja primeiro criada a tarefa 't1', seguido de uma execução do programa p1, e só depois ser criada a tarefa t2 para que seja permitida a execução de p1. Esta sequência de inicialização pode parecer contra natura, mas de facto nada no standard a proíbe. Por outro lado, se for considerado que todas as tarefas devem ser criadas antes que qualquer programa tenha hipótese de executar, volta a fazer sentido a duvida da inicialização circular, pois se está a inicializar um parâmetro de uma tarefa com o valor de uma variável de saída de um programa sem que este tenha tido a hipótese de executar, e assim de determinar o valor que deverá ser armazenado na referida variável.

Em suma, o standard é completamente omisso quanto à semântica da inicialização das tarefas, dando azo a múltiplas interpretações, potencialmente conflituosas, o que não se deveria aceitar num standard internacional.

4.3.7 Persistência de Variáveis em Funções Bloco

Como já referido anteriormente, o standard é algo dúbio quanto aos parâmetros de entrada, ficando por esclarecer o tratamento a dar-lhes aquando de invocações de funções bloco que não especifiquem de forma explícita qual o valor a atribuir a um ou mais desses parâmetros de entrada. De facto, na secção 2.5.2.1 do standard, é referido que 'parâmetros de entrada para os quais não tenham sido especificados valores para uma determinada invocação, manterão os seus valores da invocação anterior, caso alguma invocação anterior tenha ocorrido'. Ou seja, estas variáveis de entrada deverão ser persistentes.

No entanto, na descrição das invocações de funções bloco na linguagem ST (secção 3.3.2.2 do standard), é referido que as mesmas regras da invocação de funções (e especificadas na secção 2.5.1.1) serão aplicadas, sendo que se deverá substituir a palavra 'função' por 'função bloco' nessa descrição. Ora, nesta secção 2..5.1.1, é referido que 'no modo de invocar 1 da tabela 19a, a todas as variáveis para as quais não tenham sido atribuídos quaisquer valores serão atribuídos os valores a utilizar nos casos omissos'. Isto significa que as variáveis de entrada não devem ser persistentes.

4.3.8 Declaração de Variáveis em Configurações

O standard prevê que sejam declaradas variáveis dentro de configurações, as quais poderão ser acedidas pelos programas instanciados na mesma configuração em que foram declaradas as variáveis. No entanto, a definição formal da sintaxe que deverá ser utilizada na declaração destas variáveis permite a existência de variáveis às quais não é atribuído um tipo de dado específico.

Considere-se pois a definição do standard que aqui se reproduz.

```
global_var_declaration :=
'VAR_GLOBAL' ['CONSTANT' | 'RETAIN']
global_var_decl ';'
{global_var_decl ';'}
'END_VAR

global_var_decl := global_var_spec ':' [located_var_spec_init
| function_block_type_name]
```

Nesta definição a produção 'global_var_spec' é essencialmente uma lista de identificadores que serão utilizados na identificação das variáveis que se encontram a ser declaradas. Por seu lado, a produção 'function_block_type_name', tal como o seu nome indica, referencia um tipo de função bloco que tenha sido previamente declarada, enquanto que a produção 'located_var_spec_init' é, de forma simplificada, a identificação de um tipo de dado (básico ou derivado) podendo ser acompanhado de forma opcional por um valor inicial a atribuir às variáveis.

Sendo assim, o exemplo seguinte de declaração de variáveis será considerado

legal

```
VAR_GLOBAL
  foo:;
END_VAR
```

No entanto, o standard não inclui qualquer descrição da semântica a atribuir a estes casos.

Nesta questão tudo indica que se trata de um simples erro tipográfico ou erro na definição formal da sintaxe, sendo provável que o pretendido seja a seguinte definição.

```
global_var_decl := global_var_spec ':' (located_var_spec_init
| function_block_type_name)
```

4.4 A Segurança das Linguagens IL e ST

A maioria dos PLCs é utilizado em ambientes industriais para o controlo das acções de equipamento mecânico, incluindo robots e máquinas ferramenta com controlo CNC (Computer Numerical Control). São ainda encontrados a controlar sistemas de tratamento de águas, sistemas domóticos, etc. Em muitas destas aplicações uma acção incorrecta por parte do PLC pode resultar em danos avultados, sejam eles económicos através de destruição de equipamentos ou mesmo paragens intempestivas da produção, danos pessoais pondo em risco a integridade física de pessoas, ou ainda danos ambientais, tais como poluição provocada por derramamentos de produtos químicos. Para estas aplicações, exige-se que os próprios PLCs, bem como as aplicações que estes executam, apresentem um elevado grau de integridade de modo a minimizar a possibilidade de ocorrência falhas.

Tendo ainda em consideração que muitos dos programas escritos para PLCs o são em uma ou várias das linguagens definidas e estandardizadas no IEC 61131-3, torna-se vital que seja efectuada uma análise a estas linguagens tendo em vista a sua adequação à programação de aplicações de controlo integradas em sistemas de integridade elevada. Nenhuma linguagem de programação, por si só, consegue garantir que um programa será seguro e livre de erros. No entanto, algumas linguagens de programação põem à disposição do programador construções que podem ser mais susceptíveis de abuso, aumentando assim a probabilidade de que seja introduzido um erro por parte do programador. Por outro lado, outras características das linguagens de programação poderão permitir que os compiladores efectuem uma análise mais exaustiva ao programa que se encontra a ser compilado, encontrando assim erros ainda antes que os programas tenham hipótese de serem executados.

Vários autores estudaram já as características a que deverão obedecer as linguagens usadas na programação das aplicações com integridade elevada, e compilaram listas tanto das características desejáveis bem como das indesejáveis. Os requisitos que estes autores impõem às linguagens de programação para sistemas de integridade elevada divergem bastante entre si, existindo no entanto alguma sobreposição. Jagun Kwon et. al. [20] ao efectuarem uma análise da linguagem de programação Java, fizeram também uma introdução abrangente de os requisitos impostos por outros autores, tendo eles próprios compilado uma lista dos requisitos que acharam mais conveniente. Será esta a lista que será adoptada na análise que será efectuada às linguagens IL e ST do IEC 61131-3.

Por forma a facilitar a compreensão e reduzir ao mínimo referencias a outras partes do texto, a explicação das características desejáveis e indesejáveis será efectuada em conjunto com a análise das linguagens.

4.4.1 Avaliação das Linguagens IL e ST

1- Requisitos Fundamentais

1.1 - Requisitos Sintácticos e Semânticos

1.1.1 - Segurança dos tipos de dados (Type Safety)

As linguagens de programação para sistemas de integridade elevada devem ter um sistema de tipos de dados rígido, no qual não deverá ser permitida a conversão implícita de uma variável de um tipo de dado para outro, permitindo desta forma que o compilador possa detectar erros nos dados e tipos de dados mais facilmente. Devido a estas restrições, linguagens com regras restritas para o tratamento de tipos de dados devem permitir ao programador declarar novos tipos de dados derivados por forma a facilitar a gestão dos dados.

O IEC 61131-3 define uma arquitectura na qual várias linguagens utilizam os mesmos tipos de dados, permitindo assim uma troca de informação entre programas escritos em linguagens distintas sem ser necessário efectuar quaisquer conversões nos dados. Para além disto, a arquitectura impõe também regras restritas no tratamento de tipos, não estando previstas quaisquer conversões implícitas de dados entre tipos de dados distintos. Adicionalmente, operações aritméticas sobre tipos de dados numéricos são apenas válidas se os tipos de dados dos operandos forem todos iguais, sendo o resultado produzido no mesmo tipo de dado dos operandos (ver secções 2.3.2 e 2.5.1.4 do standard IEC 61131-3). É de salientar que mesmo para estas operações as sub-gamas são considerados tipos de dados distintos da do tipo de dado elementar no qual se baseia a sub-gama, não sendo assim possível efectuar operações aritméticas entre dados de uma sub-gama com dados do tipo de dado elementar no qual se baseia essa sub-gama.

O standard prevê ainda um conjunto completo de funções de conversão explícita

entre tipos de dados elementares que podem ser utilizadas para efectuar operações aritméticas com tipos de dados distintos, desde que sejam elementares.

Adicionalmente, todos os tipos de dados derivados, tais como estruturas (STRUCT), sequências (ARRAY) e enumerações podem ser analisadas na fase de compilação. No entanto, e apesar do conceito de apontador ou referencia para zona de memória (tal como a existente na linguagem C) não exista no IEC 61131-3, é permitido o acesso a uma determinada zona de memoria de forma indirecta através de variáveis ditas directas (ou mapeadas). De facto, um programador poderá vir a contornar as verificações sobre tipos de dados efectuadas pelo compilador se declarar duas variáveis, de tipos de dados distintos, com localização directa na mesma zona de memória. Embora o compilador ainda possa detectar estas ocorrências se as duas variáveis são declaradas dentro do mesmo bloco (ou POU – programa organization unit), torna-se mais difícil quando as variáveis pertencem a blocos distintos sendo estes compilados separadamente. Por outro lado, restringir a linguagem por forma a não permitir as variáveis directas ou mapeadas não é uma opção, pois este é o mecanismo principal através do qual o programa interage com o mundo físico exterior, i.e. através da leitura e escrita das entradas e saídas físicas do PLC.

Classificações possíveis:

1. Aplica regras restritas aos de tipos de dados, sendo estes passíveis de análise estática.

2. Aplica regras restritas aos de tipos de dados, mas alguns dados só poderão ser analisados dinamicamente (durante a execução), principalmente devido à existência de polimorfismo.

3. Sem regras restritas aos de tipos de dados, e permissão de conversões implícitas entre tipos de dados.

Classificação: 1

1.1.2 - Efeitos secundários de expressões / Precedência de operadores / Valores Iniciais

Uma linguagem que se pretenda segura para programar aplicações de elevada fiabilidade deverá estar livre de efeitos secundários durante a avaliação de expressões. De igual modo, a precedência de todos os operadores que poderão ser usados em expressões deverá estar claramente definida e não poderá resultar em ambiguidades. Não deverão existir valores inicias implícitos para as variáveis.

O standard IEC 61131-3 é bastante cauteloso e restrito quanto ao aspecto dos efeitos secundários em expressões uma vez que todas as funções são obrigatóriamente idem-potentes, e funções bloco e programas não poderão ser inseridos em expressões. Adicionalmente, a única forma de uma função aceder a variáveis estáticas (i.e. variáveis que mantêm o seu valor para alem da invocação da própria função) é se estas forem passadas por referencia a um parâmetro dessa

mesma função, pelo que a invocação de funções nunca poderá produzir efeitos secundários sem que esses efeitos se façam sentir através dos próprios parâmetros da função. Todas as mudanças de estado do sistema produzem-se assim através dos parâmetros da função. Deverá no entanto ser recordado que nem todas as invocações de funções inseridas numa expressão lógica poderão chegar a ser efectuadas, uma vez que a expressão é apenas avaliada até que o seu resultado final seja conhecido. Assim existe ainda o perigo de funções que produzam resultados secundários através dos seus parâmetros não tenham oportunidade de efectuar as mudanças de estado esperadas.

A precedência da avaliação dos operadores em expressões encontra-se definida sem quaisquer ambiguidades na própria definição formal da sintaxe da linguagem ST. A linguagem IL não suporta a utilização de expressões, pelo que a questão não se coloca para esta linguagem.

Todas as variáveis declaradas são inicializadas, seja com um valor explicitamente definido na própria declaração da variável, ou com um valor a utilizar por omissão, e definido para cada tipo de dado. Existe assim o perigo que um algoritmo assuma um valor inicial para uma variável de um determinado tipo de dado derivado, que no entanto vem a ser alterado mais tarde por outro programador ao alterar a definição do referido tipo de dado derivado. Poder-se-á no entanto definir um sub-conjunto das linguagens do IEC 61131-3 de tal forma que esse sub-conjunto satisfaça os requisitos que se desejam para as linguagens de programação de aplicações de elevada integridade. Sugere-se para este caso que a sintaxe da declaração de variáveis inclua obrigatóriamente a definição do valor inicial a atribuir à variável, de tal forma que o valor inicial do tipo de dado a atribuir por omissão nunca chegue a ser aplicado. Neste caso, sugere-se assim as alterações à sintaxe descritas no anexo B.

As invocações de funções bloco são por elas próprias uma declaração da linguagem ST, pelo que não poderão fazer parte de uma expressão. No entanto, e de alguma forma relacionada com a produção de efeitos secundários em expressões surge o pormenor de que nem todas as invocações de funções bloco ocorrerem no instante definido pela sequência de declarações da linguagem ST, ou mesmo das instruções IL. De facto, e utilizando as capacidades de configurar um sistema numa 'CONFIGURATION', o programador é livre de atribuir um programa a uma tarefa e as funções bloco criadas por esse mesmo programa a uma segunda tarefa. Para estes casos o standard declara simplesmente que 'a execução da função bloco fica sob o controlo exclusivo da tarefa à qual foi atribuído, qualquer que sejam as regras de avaliação do programa no qual foi declarada a função bloco'. Esta afirmação parece significar que qualquer invocação da função bloco dentro do código do programa deverá ser ignorada, mas de facto outras interpretações também são possíveis, ficando a duvida sobre qual a semântica que os autores do standard pretendem para a execução de uma função bloco nestas circunstâncias.

Assim, a execução das funções bloco poderá caber apenas à segunda tarefa, pelo que todas as invocações das funções bloco no programa não serão efectuadas na sequência da execução do programa pela tarefa que executa esse programa. Ou seja, ao contrário do que é habitual, aqui surge o perigo de que os efeitos secundários que se esperavam ocorrer pela invocação da função bloco não sejam de facto executados. Também para este caso será possível restringir a linguagem para que tal não seja permitido. Assim, sugere-se que as linguagens não aceitem que a execução de uma instância de uma função bloco declarada dentro de um programa seja atribuída de forma explícita a uma tarefa que não a que executa o próprio programa. Para tal deverão ser aplicadas as alterações à sintaxe definidas no anexo C.

Classificações Possíveis:

1. Todos os requisitos atrás mencionados são cumpridos.

2. Nem todos os requisitos atrás mencionados são cumpridos, mas existe um sub-conjunto da linguagem que os satisfaz.

3. Nem todos os requisitos atrás mencionados são cumpridos, nem existe nenhum método razoável de melhorar a linguagem.

Classificação: 2

1.1.3 - Modularidade / Estruturas

Para melhor organizar os programas é fundamental que as linguagens utilizadas na programação de aplicações de elevada fiabilidade suportem a divisão dessa aplicação em módulos estanques com interfaces de inter-acção entre esses módulos bem definidas.

A arquitectura definida pelo standard IEC 61131-3 permite a realização de programas altamente estruturados e organizados, ficando divididos em funções, funções bloco e programas. Saltos absolutos na linha de execução entre estes blocos não são permitidos, sendo apenas permitido a transferência temporária através de invocações. Adicionalmente, as variáveis internas declaradas dentro de cada bloco são privadas e apenas poderão ser acedidas pelo próprio bloco, enquanto que a troca de informação com o exterior do bloco se faz sempre através das variáveis de interface, ficando assim garantida a capacidade de esconder informação (information hiding).

A possibilidade de definir variáveis globais que poderão ser acedidas dentro de qualquer bloco (com a excepção das funções) é conseguida através de um mecanismo que mantém a modularidade dos programas. De facto, para que um bloco possa aceder a uma variável global deverá declarar primeiro uma variável interna ao bloco que servirá de referencia à variável global. Todos os acessos à variável global processam-se através desta referencia interna ao próprio bloco, e que será inicializada uma única vez quando o bloco é instanciado.

Infelizmente o standard é algo ambíguo no que concerne à possibilidade da compilação separada dos diversos módulos, seguido da posterior ligação para produzir a aplicação final. O standard permite a divisão dos blocos (funções, funções bloco e programas) em bibliotecas. No entanto não prevê qualquer sintaxe de declaração de funções definidas noutras bibliotecas, pelo que o compilador ou desiste de fazer a análise da correcção de invocação a funções na biblioteca externa, ou simplesmente proíbe essa invocações, ou então deverá prever um modo não estandardizado de obter a declaração das funções da biblioteca externa e assim efectuar a verificação da correcção das invocações a essas funções externas.

Tal como referido em secções anteriores, o standard apresenta também sérias anomalias no que concerne ao tratamento dado aos identificadores utilizados para identificar as funções, funções bloco e tipos de dados derivados. Em alguns casos a re-utilização do mesmo identificador para referenciar uma variável local e uma função bloco (por exemplo) poderá destruir a modularidade da aplicação, pois que a função na qual se encontra declarada a variável e que inicialmente se encontrava correcta em termos de sintaxe, pode passar a ficar errada pela simples inclusão da função bloco com o mesmo identificador.

Classificações possíveis:

1. A linguagem fornece mecanismos ricos e precisos de estruturar um programa, e estes podem ser divididos em termos de módulos ou objectos.

2. Estes mecanismos estão presentes, mas não são eficientes e a sua utilização não é vantajosa.

3. Os mecanismos mencionados não estão presentes na linguagem

Classificação: 1

1.1.4 - Semântica Formal / Standards Internacionais

Por forma a permitir que uma aplicação escrita numa determinada linguagem de programação seja interpretado de igual forma por todos os compiladores dessa linguagem é importante que exista um standard que defina essa linguagem, preferencialmente de forma formal e não ambígua.

De facto as linguagens IL e ST foram definidas num standard. Este standard inclui uma definição formal da sintaxe, não incluindo no entanto qualquer definição formal da sua semântica. Apesar disto, a semântica de um sub-conjunto da linguagem já foi formalizada [21].

Classificações possíveis:

1. Existe um standard formal internacional.

2. A linguagem poderá ser formalmente especificada, embora não exista um standard internacional.

3. Não existe nem standard internacional, nem a linguagem poderá ser formalmente especificada, ou esta encontra-se em situação desconhecida.

Classificação: 2

1.1.5 - Bem compreendida

Uma linguagem que seja bem compreendida, tanto do ponto de vista sintáctico como semântico, por parte dos programadores ajudará na construção de programas correctos de forma eficiente. A linguagem não deverá por isso ser desnecessariamente complicada, simplificando a sua adopção e a implementação de ferramentas (compiladores, optimizadores, analisadores de programas, etc.)

As linguagens definidas no standard IEC 61131-3 são relativamente simples, sendo a opinião do autor que estas são bem compreendidas pelos profissionais dedicados a desenvolver aplicações de controlo em ambiente industrial.

Classificações possíveis:

1. A linguagem é bem compreendida, e existem muitos programadores com experiência.

2. A linguagem é bem compreendida apenas por um numero limitado de pessoas.

3. A linguagem encontra-se pouco divulgada.

Classificação: 1

1.1.6 - Suporte para aplicações embebidas e para domínios específicos

As aplicações de integridade elevada são habitualmente executadas por sistemas embebidos, pelo que a linguagem deverá facilitar o acesso directo ao hardware necessário para que a aplicação controle o seu ambiente.

Tendo sido as linguagens do standard IEC 61131-3 desenvolvidas com o objectivo de serem utilizadas na programação de PLCs, era de esperar e compreensível que estas estejam bem adaptadas ao domínio industrial, e que incluam mecanismos de acesso directo a dispositivos de entrada/saída, bem como de acesso directo à memória.

Classificações possíveis:

1. A linguagem suporta aplicações embebidas de forma natural.

2. A linguagem permite uma utilização limitada em ambientes embebidos, mas poderão ser utilizadas bibliotecas e/ou outros mecanismos que o permitem fazer de forma eficaz.

3. A linguagem não pode ser utilizada em ambientes embebidos, nem tão pouco existem mecanismos que possam obviar esta característica.

Classificação: 1

1.1.7 - Concorrência / Processamento Paralelo

A realidade física que uma aplicação de elevada integridade pretende muitas vezes controlar é pela sua natureza um sistema com múltiplas acções a ocorrerem em simultâneo. Compreende-se por isso que uma aplicação de controlo será mais fácil de estruturar se esta puder ser dividida em tarefas a executarem de forma concorrente. A linguagem deverá por isso suportar programas multi-tarefa, métodos de controlar os algoritmos e atribuir valores aos parâmetros de escalonamento, e ainda mecanismos de sincronização entre tarefas, incluindo métodos de limitar o bloqueio não limitado.

O standard IEC 61131-3 inclui suporte ao nível das linguagens de programação para a execução concorrente de programas, o que é conseguido ao permitir que o programador declare um número de tarefas, seguida da posterior atribuição dos programas a uma das tarefas. É ainda possível atribuir a execução da instância de uma função bloco a uma tarefa, mesmo que essa instância tenha sido declarada dentro de um programa que se encontra a ser executado por uma outra tarefa.

Para este aspecto da linguagem é ainda relevante a dúvida relacionada com a semântica de inicialização das tarefas, cujos parâmetros podem ser considerados atribuídos por valor (um único valor durante o ciclo de vida da tarefa) ou por referencia (permite mudanças de valor no parâmetro durante o ciclo de vida da tarefa).

O controlo do escalonamento poderá ser apenas efectuado através da atribuição de uma prioridade a cada tarefa. Não é possível escolher o algoritmo de escalonamento, embora o standard preveja que seja possível que as implementações das linguagens possam suportar o escalonamento de prioridade fixa preemptivo ou não preemptivo. A escolha entre estes dois algoritmos, se escolha houver a fazer, terá de ser efectuada por recursos a mecanismos não estandardizados, e por isso dependentes da implementação.

O standard também não prevê quaisquer mecanismos de sincronização entre tarefas, sendo no entanto que estes mecanismos seriam de reduzida aplicabilidade em programas cuja execução se espera que seja efectuada de forma assíncrona (i.e. a executar um ciclo infinito). Para estes casos fará mais sentido que a linguagem deva permitir a sincronização do estado interno de cada tarefa, em vez da sincronização da execução das mesmas.

De facto, a arquitectura de programação especificada pelo IEC 61131-3 prevê que a sincronização seja efectuada com recurso a SFCs, que estão preparados para gerir a sincronização do estado interno dos blocos, blocos estes que poderão estar a ser executados por tarefas distintas. A sincronização será ainda possível de forma básica através da leitura/escrita de variáveis globais, às quais mais do que uma tarefa poderá ter acesso. No entanto, não é especificado se a linguagem em si garante que os acessos a estas variáveis estejam livres de conflitos, em particular em PLCs com mais do que um CPU (ou recurso) [22][23]. O acesso a

estas variáveis através de conjuntos de operações não atómicas é também problemática, uma vez que não são considerados quaisquer mecanismos de sincronização para alem dos SFCs mencionados.

Classificações possíveis:

1. Todos os requisitos mencionados são cumpridos.

2. Apenas alguns requisitos são suportados ao nível da linguagem, mas poderão ser utilizadas bibliotecas que implementam os requisitos.

3. Não existe forma razoável de suportar os requisitos mencionados.

Classificação: 1

1.2 - Técnicas de Verificação / Previsibilidade

1.2.1 - Previsibilidade Funcional

Deverá ser possível provar que o comportamento funcional de uma aplicação de elevada integridade é previsível. Para isso é comum recorrer a formalismos tais como a análise de fluxo de controlo e de dados, execução simbólica, e verificação formal do código.

A relativamente reduzida complexidade das linguagens em causa facilita a aplicação de técnicas de análise formal do código [24]. De facto, existem já alguns resultados publicados relativos a métodos de verificação formal das linguagens definidas no IEC 61131-3, incluindo o IL e ST. Resumos das técnicas até agora aplicadas podem ser encontradas em [25] e [26].

As técnicas de análise podem ser classificadas em técnicas de verificação de modelos (model checking) e técnicas de provar teoremas (theorem proving).

Dos trabalhos relacionados com verificação de modelos poder-se-á destacar Cannet *et. al.* [27] [28], que modeliza o comportamento de um programa de um PLC como um sistema de transições sincronizadas pela troca de mensagens, sendo este modelo posteriormente analisado com recurso à ferramenta Cadence-SMV (Symbolic Model Verifier) [29] desenvolvido pela Universidade de Carnegie Mellon. São apresentadas pelos autores mecanismos de modelização das linguagens LD, SFC, IL e ST, tendo no entanto sido necessário uma prévia definição da semântica destas tendo como base o standard IEC 61131-3. De forma semelhante, em [30] os FBDs são também eles convertidos para o modelo SMV, enquanto que R. Huuck *et.* al. [31], [32], [33], [34], [35] e [36] converte para SMV programas em SFC e IL. Já em [37] os SFCs são convertidos para Redes de Petri, sendo as propriedades destes posteriormente analisados, enquanto em [38] o mesmo é efectuado para a linguagem IL. Por outro lado, Willems [39] converte os programas IL em autómatos temporizados (timed automata), sendo estes posteriormente analisados pela ferramenta Uppaal [40]. Susta [41] efectua também a conversão de programas para autómatos temporizados, suportando já várias linguagens com a excepção dos SFCs.

Em [42] é apresentado um método de representação das linguagens IL e LD em lógica proposicional (propositional logic) extendida com aritmética de inteiros, a partir da qual torna-se possível avaliar o programa recorrendo a ferramentas comerciais de provar teoremas. No entanto esta abordagem apresenta lacunas tendo em conta que é apenas considerada a aritmética com inteiros, e a modelização de intervalos de tempo não é totalmente suportada. Adicionalmente, foi utilizada uma versão comercial das linguagens LD e IL, e não a versão normalizada. Por sua vez, Halang, Kramer *et. al.* [43], [44], [45], [46] e [47] recorrem a HOL (Higher Order Logic) para representar programas em (subconjuntos de) ST, FBD e SFC, considerando no entanto apenas uma única tarefa. Utilizam ainda um método composicional para provar programas que recorrem a funções já previamente provadas correctas.

Como conclusão poderá ser referido que a falta de normalização formal da semântica da linguagem dificulta a verificação formal do código expresso nestas linguagens. No entanto, e como já anteriormente referido, a semântica de um sub-conjunto da linguagem já foi efectuada por Egger *et. al.* [21].

Classificações possíveis:

1. Todas as técnicas de análise poderão ser utilizadas.

2. Nem todas as técnicas poderão ser utilizadas, mas considerando apenas um sub-conjunto da linguagem poderá facilitar a sua aplicação.

3. Desconhecido, ou não existem alternativas viáveis de aplicação dos referidos métodos de análise e verificação.

Classificação: 2

1.2.2 - Previsibilidade Temporal / Análise Temporal

A par da previsibilidade do comportamento funcional, deverá ainda ser possível determinar o comportamento temporal das aplicações de elevada integridade. Em particular, deverá ser possível determinar o tempo de execução de cada tarefa no seu pior caso (WCET – Worst Case Execution Time), permitindo assim a aplicação das técnicas de análise de escalonabilidade para sistemas de prioridade fixa ou dinâmica.

A determinação dos WCET de cada tarefa é altamente dependente do hardware que a irá executar, no entanto, e assumindo que o fabricante do processador (ou PLC) e do compilador que o acompanha disponibiliza informação suficiente, nada da linguagem impede que as referidas análises sejam efectuadas. Pelo contrário, algumas características das linguagens até a facilitam, em especial a proibição da utilização de funções recursivas, e a determinação que as declarações de iteração (FOR, WHILE, REPEAT) não sejam utilizadas para sincronização com estados exteriores ao próprio ciclo.

Classificações possíveis:

1. Tempos de WCET apertados podem ser calculados ou determinados.

2. Valores algo pessimistas para os tempos de WCET podem ser obtidos.

3. Os tempos de execução não são previsíveis, ou não é viável a sua determinação.

Classificação: 1

1.2.3 - Análise de Utilização de Recursos

Torna-se importante identificar à priori quais os recursos que poderão vir a ser necessários no decorrer da execução da aplicação de elevada integridade, permitindo assim prever se algum destes recursos poderá vir a esgotar-se.

Uma análise detalhada da utilização de recursos é possível tendo em conta que o standard proíbe de forma explicita qualquer tipo de chamadas recursivas (secção 2.5 do standard), o que permite uma determinação prévia da memória necessária para a alocação de variáveis temporárias para todas as invocações de funções e de funções bloco. Uma vez que não está previsto qualquer mecanismo de alocação dinâmica de memória, não é necessária qualquer análise a este tipo de recurso cuja utilização é habitualmente difícil de prever.

Classificações possíveis:

1. A previsão exacta da utilização dos recursos é possível.

2. Uma previsão pessimista dos recursos utilizados, ou a utilização efectuada dos recursos no pior caso é possível mas não pratica.

3. A utilização dos recursos é imprevisível.

Classificação: 1

1.3 - Processadores da Linguagem / Ambientes de execução / Ferramentas

1.3.1 - Compiladores Certificados / Ambientes de Execução

Para que haja uma elevada confiança no programa que é de facto executado pelo micro-processador, é também indispensável que haja confiança no compilador que efectuou a transformação do programa desde a sua descrição textual até ao código maquina final. O mesmo se aplica a todas as ferramentas eventualmente utilizadas para o processamento e/ou análise do código inicial. Preferencialmente deverá então existir um compilador que tenha sido formalmente comprovado como correcto, o mesmo ocorrendo para o ambiente no qual é executado o programa final. Deverá ainda ser possível efectuar um mapeamento inverso desde o código maquina até ao programa textual que deu origem a esse código maquina.

Embora não se conheçam quaisquer compiladores formalmente certificados por um organismo com autoridade nesta matéria, existe sim um compilador gerado

de forma automática, tendo como base a definição formal da semântica de um sub-conjunto das linguagens textuais já referida. Foi para isso utilizada a ferramenta de geração de compiladores OPAL. Não se conhece no entanto nenhum ambiente de execução formalmente verificado para acompanhar esta ferramenta.

De igual forma, não são conhecidos ambientes de execução formalmente certificados para a execução de código produzido pelos compiladores comerciais.

Classificações possíveis:

1. Existe pelo menos um compilador certificado que foi provado estar livre de erros.

2. Compiladores existem que contem alguns erros bem conhecidos e documentados, mas estes não inibem a produção de programas de elevada fiabilidade.

3. Desconhecido.

Classificação: 2

1.3.2 - Suporte de Execução / Questões Ambientais

A utilização de bibliotecas de funções pode tornar o código final que é executado mais complexo de analisar, como por exemplo na determinação de tempos de execução, pelo que todo este código adicional deverá ter um comportamento previsível.

O standard IEC 61131-3 de facto especifica um conjunto relativamente vasto de funções pré-definidas que se supõe que estejam presentes no ambiente de execução dos programas. Nada no standard impede que estas bibliotecas sejam de facto previsíveis, nem tão pouco que possam ser acompanhadas de documentação a explicitar os tempos de execução de cada função.

Classificações possíveis:

1. O comportamento funcional e temporal de todas as bibliotecas encontra-se disponível.

2. Apenas análise de tempos de execução no pior caso/pessimistas é possível.

3. Situação desconhecida.

Classificação: 2

2 - Requisitos Desejáveis

2.1 - Requisitos sintácticos e Semânticos

2.1.1 - Tratamento de Excepções / Comportamento face a falhas

A ocorrência de falhas ou erros num programa que se pretende ser de elevada fiabilidade nunca será desejável, no entanto se qualquer tipo de erro pode

ocorrer, o seu tratamento deverá ser previsível em termos de tempo de execução e código a ser executado, permitindo que o sistema sofra uma degradação suave, ou possa recuperar do erro.

O standard não prevê mecanismos de programar rotinas de tratamentos de erros, nem tão pouco se encontra completamente especificado o comportamento que deverá ter o programa na ocorrência de erros (secção 1.5.1 do standard), existindo mesmo a possibilidade de o comportamento a ter face aos erros ser especificado de forma dependente da implementação.

2.1.2 - Modelo da Matemática

A matemática implementada pela linguagem deverá ser definida pelo standard de forma a evitar ambiguidades, e deverão existir procedimentos de detectar a ocorrência de erros matemáticos durante a execução (por exemplo, divisão por zero, situações de overflow e underflow).

O modelo da matemática não está completamente definido de forma não ambígua, por exemplo ao permitir que cada implementação da linguagem possa utilizar a sua própria resolução em funções matemáticas (secção 2.5.1.5.2, parágrafo 3 do standard). Adicionalmente, embora o standard preveja um mecanismo de detectar a ocorrência de eventos como o 'overflow' e 'underflow' baseado em parâmetros de entrada e saída lógicos representando a ocorrência destes erros, não é obrigatório que este mecanismo esteja presente em todas as implementações do standard (secção 2.5.1.2, item 3). De facto, a ocorrência de erros poderá ser simplesmente ignorada sem que o programa seja avisado da sua ocorrência (secção 1.5.1, item d) 1)).

2.1.3 - Suporte de Documentação

A possibilidade de introduzir comentários embebidos no próprio programa permite aumentar a legibilidade e compreensão futura do código ao permitir que o programador original possa explicar a racionalidade dos algoritmos utilizados, bem como o modo de modularizar o programa. Algumas ferramentas de verificação automática do código poderão mesmo fazer uso de alguns comentários para permitir uma análise mais profunda do código.

O standard permite a inserção de comentários no código de linguagens textuais, no entanto de momento não são conhecidas quaisquer ferramentas de verificação de código que utilizem estes comentários.

2.1.4 - Suporte de Sub-gamas e Enumerações

Torna-se mais fácil efectuar verificações ao código se este utiliza tipos de dados estáticos, com uma gama de valores que coincide com a gama de valores dos dados que serão armazenados em variáveis desses tipos.

Para tal o standard suporta a declaração de tipos de dados derivados, tais como estruturas, sub-gamas de tipos de dados elementares numéricos numeráveis, e

ainda enumerações nas quais se enumera todos os valores possíveis para esses tipos de dados.

2.1.5 - Guias de Estilo de Programação

Guias de estilo poderão auxiliar no desenvolvimento de programas que seguem boas praticas de engenharia de software. No são no entanto conhecidos quaisquer guias de estilo para as linguagens em causa, nem estas são mencionadas no standard (como seria de esperar).

2.1.6 - Suporte de Abstracção e o Esconder de Informação

A utilização de técnicas de abstracção e de esconder informação, tal como presentes em linguagens orientadas a objectos, permite uma melhor organização do código, facilitando assim a manutenção do código, e reduzindo a sua complexidade.

O standard suporta que informação seja escondida através da utilização de blocos que mantêm estado interno (funções bloco e programas). Não são no entanto suportadas quaisquer técnicas de abstracção uma vez que não é permitida a herança entre funções bloco ou programas.

2.1.7 - Verificação de Asserções

O standard IEC 61131-3 não prevê quaisquer mecanismos de teste de asserções em quaisquer das linguagens

2.2 - Processadores das Linguagens / Ambientes de Execução / Ferramentas de Apoio

2.2.1 - Ferramentas de análise certificadas

Por forma a aumentar a confiança em programas que se pretendem que sejam de elevada fiabilidade, será vantajoso que existam ferramentas que analisem o código para possível detecção de erros (tais como race conditions). Estas ferramentas deverão, elas próprias, estar certificadas, para que se possa confiar nos resultados produzidos pelas ferramentas.

De momento não são conhecidas quaisquer ferramentas certificadas para a verificação de código expresso nas linguagens definidas no standard.

2.2.2 - Interface com Outras Linguagens

Existem situações em que se torna necessário escrever um programa numa outra linguagem que se mostra mais apropriada para a tarefa em questão. Para tal deverá ser possível integrar de forma fácil e segura os programas escritos nas diferentes linguagens.

O standard IEC 61131-3 define uma arquitectura de programação que ela própria permite já a integração de programas e rotinas escritas em linguagens distintas. O standard prevê mesmo que possam ser utilizadas outras linguagens para além daquelas que se encontram definidas no próprio standard. É possível

por isso ter um compilador que permite a programação de blocos (i.e., POU – Program Organization Units) em Java, C ou Pascal, embora de momento não são conhecidas ferramentas que o permitam.

2.2.3 - Optimização do Código

É sempre vantajoso que os programas possam ser optimizados por forma a melhorar o seu desempenho, seja ele temporal ou de ocupação de espaço de memória. No entanto, estas optimizações nunca deverão resultar em alterações da semântica do código optimizado, nem dificultar a análise do código produzido.

A optimização do código é certamente possível, tal como a extensão de ciclos com limites bem definidos, não sendo no entanto conhecidos quaisquer compiladores que o façam. Isto não significa que não existem, pois os fabricantes comerciais de PLCs não costumam divulgar muitas informações relativamente ao funcionamento interno dos seus compiladores.

2.2.4 - Portabilidade do Código

A portabilidade do código foi um dos objectivos principais da existência do standard IEC 61131-3, pelo que se torna compreensível que os programas escritos nas linguagens definidas neste standard possam ser facilmente portadas entre ambientes de execução distintos. No entanto, torna-se necessário ter em atenção muitos detalhes que o standard permite que sejam definidos pela implementação.

4.4.2 Resumo da Classificação

O itens da classificação aos quais foram atribuídos níveis encontram-se resumidos na tabela seguinte. Para comparação, são também apresentadas as classificações atribuídas à linguagem Java em [20]. Constata-se que, no geral, as linguagens IL e ST são apropriadas à implementação de aplicações de elevada integridade. Embora uma comparação directa com outras linguagens não faça muito sentido uma vez que algumas das classificações têm um pouco de subjectividade, é possível ainda assim observar que as linguagens IL e ST são mais adequadas do que o Java para as aplicações de elevada integridade. De facto, era esta a conclusão que se esperava obter, uma vez que a concepção das linguagens IL e ST teve logo à partida o objectivo de suportar aplicações de controlo que se esperavam ser embebidas, enquanto que o objectivo fundamental da linguagem Java era o de ser executável em qualquer sistema operativo e micro-processador.

As vantagens das linguagens IL e ST prende-se com a verificação forte dos tipos de dados, a possibilidade de modularizar as aplicações, o suporte de tarefas concorrentes e de aplicações embebidas.

Também nos requisitos de nível 2 as linguagens IL e ST saem bastante bem, sendo altamente previsíveis na utilização de recursos e de resposta temporal,

falhando apenas em pequenos pormenores relacionados com a previsibilidade do tratamento de situações de erro.

Item	*Classificação (1..3)*	
	ST e IL	***Java***
1 - Requisitos Fundamentais	1.42	1.83
1.1 - Requisitos Sintácticos e Semânticos	1.29	1.57
1.1.1 - Segurança dos tipos de dados (Type Safety)	1	2
1.1.2 - Efeitos secundários de expressões / Precedência de operadores / Valores Iniciais	2	2
1.1.3 - Modularidade / Estruturas	1	1
1.1.4 - Semântica Formal / Standards Internacionais	2	2
1.1.5 - Bem compreendida	1	1
1.1.6 - Suporte para aplicações embebidas e para domínios específicos	1	2
1.1.7 - Concorrência / Processamento Paralelo	1	1
1.2 - Técnicas de Verificação / Previsibilidade	1.33	2
1.2.1 - Previsibilidade Funcional	2	2
1.2.2 - Previsibilidade Temporal / Análise Temporal	1	2
1.2.3 - Análise de Utilização de Recursos	1	2
1.3 - Processadores da Linguagem / Ambientes de execução / Ferramentas	2	2.5
1.3.1 - Compiladores Certificados / Ambientes de Execução	2	2
1.3.2 - Suporte de Execução / Questões Ambientais	2	3

Figura 4.39 - Resumo de classificação das linguagens IL e ST

As principais falhas ocorrem nos requisitos de nível 3, os quais estão mais relacionados com as ferramentas que se encontram disponíveis do que com as linguagens em si. No entanto, e apesar desta falta de ferramentas comprovadas formalmente se prender maioritáriamente com razões económicas e de procura de mercado, poder-se-á ainda atribuir ao facto de não existir uma estandardização da definição formal da semântica destas linguagens.

Capítulo 5

Aplicações Distribuídas

5.1 O IEC 61499

5.1.1 Porquê um Novo Standard?

O standard IEC 61131-3 e as suas linguagens de programação encontram-se já bastante divulgadas no ambiente industrial, sendo estas bem compreendidas. No entanto este standard trata apenas os casos dos PLCs que funcionam de modo isolado, ou seja, que não trocam informação com outros equipamentos através de redes de dados. Ora a crescente divulgação das redes de dados, e a sua utilização para interligar os equipamentos de controlo industrial, bem como ligar estes a centros de supervisão, tornou necessário um standard adicional para tratar da troca de informação entre equipamentos. Para tal o IEC desenvolveu um capitulo adicional ao conjunto das normas IEC 61131 que tem como objectivo estandardizar o modo como os programas de controlo executados pelos PLCs lidam com a troca de informação através de redes de dados.

Surgiu assim um quinto capitulo do IEC 61131, conhecido por IEC 61131-5, que define serviços de comunicação através dos quais se poderá enviar e receber dados por uma rede de dados. Por outro lado, o modo de acesso a estes serviços

encontra-se especificado no IEC 61131-3, o qual define a sintaxe necessária para mapear variáveis nos canais de comunicação definidos na capítulo 5. Este mapeamento é efectuado dentro das configurações, sendo possível colocar à disposição para acesso remoto variáveis declaradas dentro da configuração, variáveis directas (mapeadas nas entradas/saídas físicas do PLC), parâmetros de entrada e/ou saída de funções bloco ou programas instanciados dentro da configuração, ou ainda variáveis privadas dos mesmos programas ou funções bloco. O acesso às variáveis que são partilhadas através da rede poderá ainda ser limitada a leituras, ou permitir tanto a leitura como a escrita.

Infelizmente este mecanismo revelou-se algo limitativo. De facto, embora o mecanismo permita a troca de informação relativa ao estado interno de PLCs ligados em rede recorrendo à leitura/escrita de variáveis internas dos PLCs remotos, já a sincronização temporal dos PLCs torna-se mais difícil. Um PLC que deseje ficar à espera que outro PLC atinja um determinado estado interno terá de efectuar leituras consecutivas (polling) da variável que contem a activação do estado em causa. Ou seja, os standards IEC 61131 capítulos 3 e 5, definem um mecanismo que permite apenas a troca de informação assíncrona, não sendo possível o envio de mensagens síncronas entre os PLCs.

O IEC cedo reconheceu as limitações do standard IEC 61131-3 para equipamento ligado em rede, pelo que desenvolveu um outro standard, que veio a ser conhecido por IEC 61499[48], que tem como objectivo final a troca de informação síncrona e assíncrona entre PLCs. No entanto este standard vai mais longe, ao definir também ele um modelo de desenvolvimento na qual se enquadra a troca de dados através da rede. Este modelo, embora semelhante ao que é definido no standard IEC 61131-3, com os seus POUs (funções, funções bloco e programas), reduz o número de tipos de entidades envolvidas, aumentando as suas capacidades.

5.1.2 O Modelo do IEC 61499

O modelo definido no IEC 61499 define cada PLC como sendo um dispositivo. Cada dispositivo poderá conter um ou mais recursos, sendo estes mapeados em CPUs de um PLC, à semelhança dos recursos definidos no IEC 61131-3. No entanto, o modelo de programação do IEC 61499 diverge do modelo do standard anterior ao permitir que as aplicações sejam distribuídas por vários recursos, sejam eles do mesmo dispositivo ou não. Isto significa que uma aplicação poderá estar a ser executada em simultâneo por vários PLCs, mas também que um mesmo recurso de um PLC possa estar a executar várias aplicações (ou partes destas) em simultâneo.

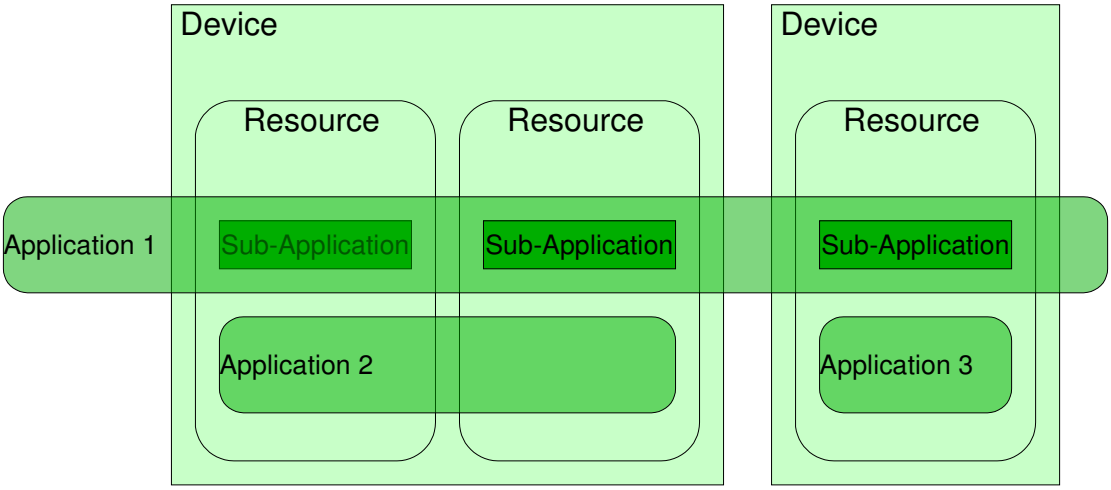

Figura 5.1 - Modelo do IEC 61499

As aplicações são elas próprias constituídas por um conjunto de sub-aplicações ou funções bloco que cooperam entre si. As sub-aplicações e as funções bloco do IEC 61499 podem ter interfaces de entrada e saída através dos quais partilham informação com as restantes sub-aplicações ou funções bloco com as quais cooperam. Estas ligações às quais correspondem a troca de informação entre as sub-aplicações e funções bloco podem ser expressas graficamente, resultando em gráficos que se assemelham de alguma forma aos gráficos FBD definidos no IEC 61131-3.

As sub-aplicações são constituídas por uma rede de funções bloco que cooperam entre si, enquanto que as funções bloco podem também elas ser compostas por um conjunto de outras funções bloco cooperando entre si. Ao contrário do IEC 61131-3, o modelo definido no IEC 61499 não contempla os POU que são os programas e as funções.

Cada função bloco consiste num bloco estanque com o qual apenas se interage através das suas interfaces. Todos os dados e algoritmos que a função bloco contem não são visíveis do exterior da mesma. Isto implica que uma função bloco é indivisível, não podendo ser distribuída por vários recursos. A função bloco consiste assim no elemento atómico de distribuição, que permite que as sub-aplicações e as aplicações sejam executadas em simultâneo por vários recursos.

Apesar da semelhança com as funções bloco definidas no IEC 61131-3, as funções bloco do IEC 61499 são mais complexas e completas. Para além da troca de dados através de parâmetros de entrada, saída ou entrada/saída já presentes nas funções bloco do IEC 61131-3, permitem ainda a troca de eventos entre elas. Cada função bloco, para além de definir os seus parâmetros de entrada, define ainda portos de entrada e saída através dos quais poderá ser notificado da ocorrência de um evento, ou então notificar a função bloco a jusante da ocorrência de um determinado evento. Assim, uma rede de funções bloco é

definida como um conjunto de funções bloco com os portos de recepção e envio de notificações de ocorrência de eventos ligados entre si, o mesmo ocorrendo para os parâmetros de entrada, saída e entrada/saída.

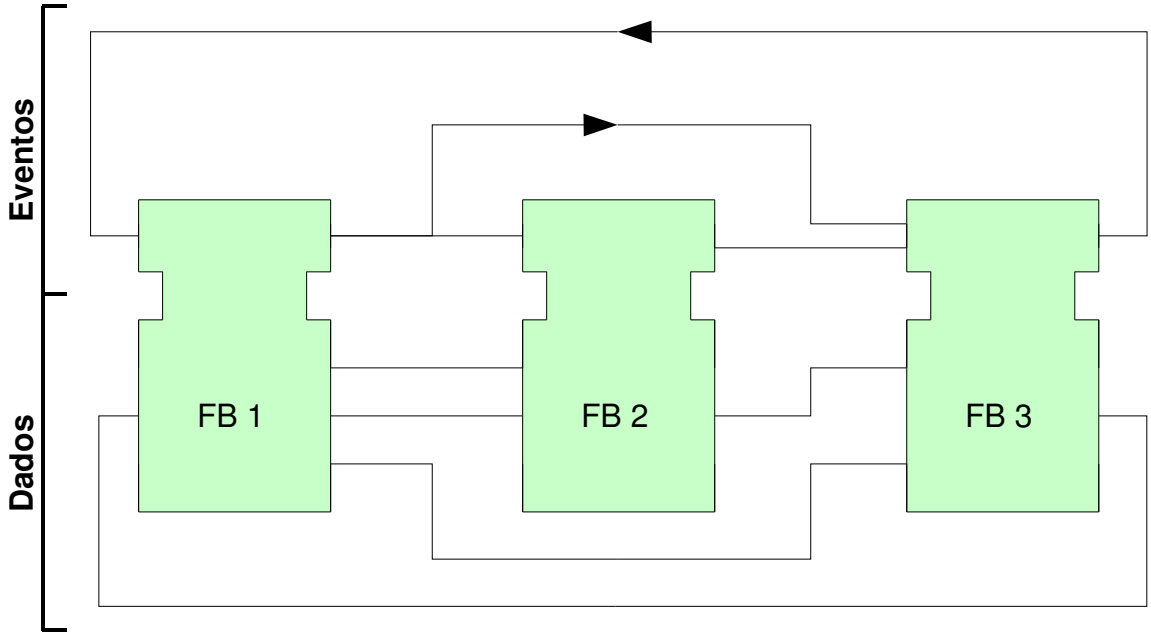

Figura 5.2 - Diagrama de Funções Bloco do IEC 61499

As funções bloco básicas são as funções bloco que não são constituídas por uma rede de outras funções bloco, sendo em alternativa expressas de forma explícita, tanto as variáveis necessárias para manter o seu estado interno, bem como os algoritmos que a função bloco implementa (ou seja o código que esta executa). Para além disto, as funções bloco básicas poderão ainda ter vários estados de execução, acompanhado de um gráfico que define a evolução da activação dos estados de execução. É de salientar que apenas um estado poderá estar activo em qualquer momento. Cada algoritmo (na realidade, uma secção ou trecho de código) é associado a um estado de execução, sendo executado apenas durante o período de tempo em que o estado ao qual se encontra associado permanece activo. Os algoritmos são especificados com o auxílio de uma das linguagens de programação definidas no IEC 61131-3, ou ainda de quaisquer outras linguagens para as quais venham a ser desenvolvidos compiladores.

A interacção com o mundo real, por exemplo a escrita e leitura das entradas e saídas físicas de um PLC, efectua-se através de interfaces especializadas para tal, existentes em cada recurso. Estas interfaces podem ser modelizadas como funções bloco especializadas, às quais o standard IEC 61499 dá o nome de funções bloco de serviços de interface ('service interface function blocks' no original). Para além das funções bloco de serviços de interface utilizadas na leitura e escrita de saídas e entradas físicas, este standard define ainda funções bloco de serviços de interface dedicadas ao acesso às redes de dados que interligam vários dispositivos (ou PLCs), e ainda para o controlo e configuração de o estado interno de cada dispositivo. Isto significa que com este standard deixam de existir as variáveis

directas (ou variáveis com mapeamento directo) definidas no IEC 61131-3, as quais eram utilizadas no acesso às entradas e saídas físicas.

O IEC 61499 especifica apenas a interface e a semântica implementada por cada função bloco de serviço de interface. Não especifica a sua implementação, deixando esta ao critério dos fabricantes de PLCs. Isto permite, por exemplo, que cada fabricante de PLC possa fornecer uma função bloco de serviço de interface com a rede de dados que utiliza os seus protocolos de rede proprietários, sem que seja necessário divulgar pormenores desses mesmos protocolos. No entanto, e uma vez que todos as funções bloco de serviço de interface com a rede de dados apresentam a mesma interface, bem como a mesma semântica, uma aplicação pode facilmente trocar de função bloco de serviço de interface de rede sem ser necessário introduzir qualquer alteração ao código dessa aplicação. Poder-se-á assim escrever uma aplicação distribuída que não depende do protocolo de rede de baixo nível, mas apenas dos serviços fornecidos pelas funções bloco de interface de rede.

5.1.3 Funções Bloco Básicas

As funções bloco básicas podem ser declaradas com uma sintaxe textual, ou através de uma representação gráfica. Na representação gráfica, exemplificada na fig. 3, a interface de eventos aparece na caixa em forma de 'T' no topo da representação, enquanto que a interface dos dados é colocada dentro da caixa inferior. As entradas, de dados ou eventos, aparecem à esquerda, enquanto que as saídas estão á direita. Cada evento pertence a um determinado tipo de evento, que fica implicitamente declarado pela simples utilização do referido tipo de evento. Eventos aos quais não é explicitamente atribuído um tipo pertencem por omissão ao tipo de evento genérico 'event'. Entradas (saídas) de eventos genéricos podem ser ligadas a quaisquer saídas (entradas) de eventos, sejam eles genéricos ou não. Pelo contrário, entradas (saídas) de eventos de um determinado tipo apenas podem ser ligadas a saídas (entradas) de eventos desse mesmo tipo, ou de eventos genéricos.

Embora o standard não explicite nenhuma restrição para as ligações de dados, presume-se que as ligações destas deverão respeitar os tipos de dados em questão.

É possível ainda especificar uma associação entre as entradas de dados e a entrada de eventos, o mesmo ocorrendo com as saídas de eventos e dados. Esta associação é representada graficamente com uma linha a unir o evento de entrada (saída) a cada entrada (saída) de dados à qual se encontra associada. Cada entrada (saída) de dados poderá estar ligada a várias entradas (saídas) de eventos, podendo mesmo não estar ligada a qualquer evento.

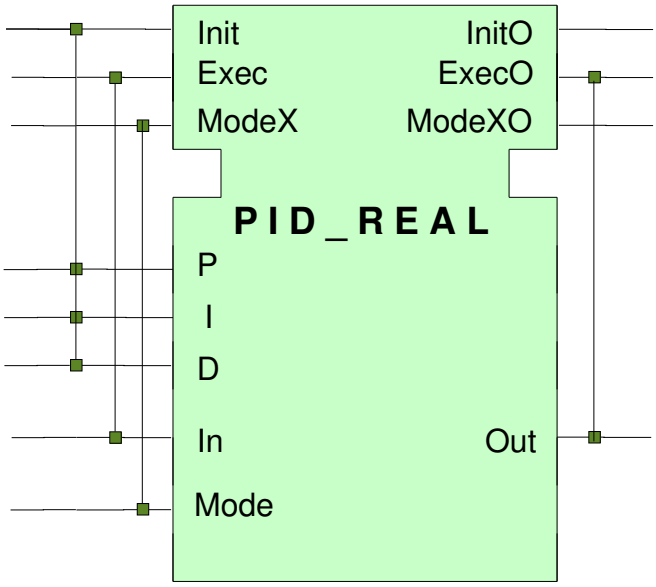

Figura 5.3 - Associações entre eventos e dados

O significado desta associação não é completamente definido pelo standard (secção 2.2.1.1), limitando-se este a limitar os tipos de ligações que poderão ser efectuadas aos dados de entrada que não se encontram associados a nenhum evento de entrada. Nestes casos, apenas se poderão ligar constantes ou variáveis que sejam apenas alteradas devido a acções de alteração de configuração do sistema. Já às variáveis de saída que não se encontram ligadas a nenhum evento de saída não é imposta qualquer restrição.

Apesar da semântica desta associação não ser claramente especificada pelo standard, pode-se no entanto inferir algum significado desta associação através da especificação do funcionamento interno que deverão observar as funções bloco (secção 1.4.5.3, acções relacionadas com os instantes t1 e t4). Desta secção conclui-se que sempre que uma função bloco gera um evento, então apenas poderá modificar os valores nas saídas de dados que se apresentam conectados ao evento que foi despoletado, ficando as restantes saídas com o mesmo valor que tinham anteriormente. De forma semelhante, sempre que uma função bloco recebe um evento, apenas serão lidos os dados de entrada que se encontram associados ao evento recebido.

Adicionalmente, o standard prevê mesmo que a semântica das associações entre as entradas (saídas) de dados e os eventos de entrada (saída) possa depender da implementação. No anexo I são enumerados dois exemplos de semânticas possíveis para a associação das entradas (saídas) de dados e a entradas (saídas) de eventos, isto sem que sejam proibidas outras semânticas não explicitamente referenciadas pelo standard.

O primeiro exemplo refere uma semântica de cópia, na qual a função bloco cria uma cópia dos valores das entradas de dados associados ao evento de entrada que acaba de receber. Esta semântica, em conjunto com as especificações que regulam o funcionamento interno das funções bloco, garante que a função bloco

irá ter acesso a valores que nunca sofrem alterações enquanto executa as acções despoletadas pelo evento de entrada.

Já o segundo exemplo refere uma semântica de restrições. Neste caso, uma associação entre dados de saída e um evento de saída estabelece a garantia por parte da função bloco que sempre que alguma dessas variáveis de saída sofrerem uma alteração do seu valor, será então gerado esse evento de saída. Já nos dados de entrada, a associação especifica uma necessidade da função bloco em que receba um determinado evento sempre que pelo menos uma das entradas de dados sofra uma alteração do seu valor.

Uma mesma função bloco pode conter vários algoritmos (secções de código), cabendo a um gráfico de controlo de execução (ECC – Execution Control Chart) o controlo de qual o algoritmo que deverá ser executado em cada instante. Os ECC são no fundo máquinas de estado compostas por estados, transições e acções.

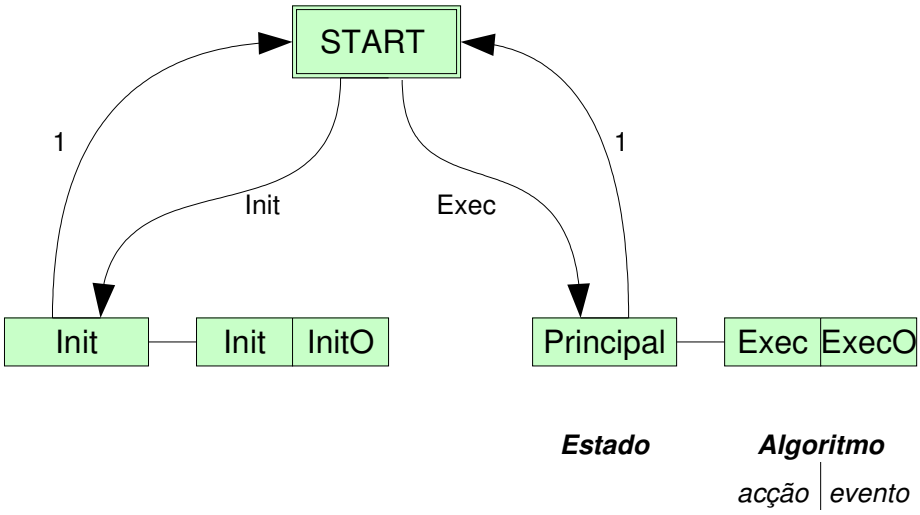

Figura 5.4 - Gráfico de controlo de Execução

Cada ECC contém um e apenas um estado inicial, o qual não tem associado qualquer acção. O estado inicial é representado graficamente por um elemento gráfico delimitado por uma linha dupla. Os restantes estados, representados com uma linha simples, podem ter acções associadas aos mesmos. Adicionalmente, a cada acção poderá ser associado um evento que será emitido após a conclusão da execução dessa acção. A cada acção do ECC poderá ainda estar associado um algoritmo da função bloco ao qual pertence o ECC. As transições que definem as condições de evolução do ECC têm associadas uma condição booleana, que poderá utilizar os valores das entradas e/ou saídas de dados, variáveis internas, ou ainda eventos de entrada. É de salientar que embora o ECC possa aceder ao estado das variáveis da função bloco, não poderá de modo algum alterar os seus valores.

A invocação de uma função bloco resulta na avaliação do ECC dessa função bloco. Caso a evolução do ECC obrigar à activação de algoritmos, estes serão

executados pela mesma ordem na qual se encontram declarados. Já a execução de cada algoritmo obriga a que este seja previamente colocado na lista de tarefas a executar pelo recurso em que se encontra a função bloco, sendo a sua execução escalonada pelo algoritmo de escalonamento que rege o CPU do recurso. Cada algoritmo deverá ser composto por uma sequência finita de instruções, sendo que a acção termina após a última instrução do algoritmo ser executada.

Após a conclusão de um estado, e durante a mesma invocação da função bloco, o ECC é re-avaliado, resultando na possível activação de um novo estado, e consequente execução das acções a ele associadas. A invocação da função bloco termina apenas quando o ECC não puder evoluir após a conclusão de execução das acções de um estado. Durante as possíveis re-avaliações do ECC devido a uma mesma invocação da função bloco, considerar-se-ão sempre os mesmos valores para as entradas de valores e de eventos (Secção 2.2.2.2, Tabela 1 da norma). Embora não esteja muito explícito no standard, através da exigência de que nenhum evento pode ser perdido conclui-se que os eventos que surjam durante a fase de execução iniciada por uma invocação da função bloco deverão ser armazenadas para posterior invocação. Surge então a dúvida de qual o tratamento a dar no caso de serem recebidos vários eventos durante esta fase em que os eventos de entrada são armazenados. Deverão os eventos de entrada ser processados todos em simultâneo, ou será que deverão ser processados pela mesma ordem na qual foram surgindo.

De facto, no IEC 61499, uma invocação de uma função bloco não é mais do que o envio de um evento para uma das entradas de eventos dessa mesma função bloco, pelo que o extravio de eventos iria corresponder a invocações que não se chegariam a realizar.

É de salientar que, ao contrário do que habitualmente se verifica nos PLCs, a norma ao especificar o modelo de execução das funções bloco não exige que os valores que se apresentam nas entradas de dados se mantenham inalterados enquanto a função bloco executa os algoritmos despoletadas pelo evento que acabou de receber (Secção 1.4.5.3, Nota 3). Adicionalmente, não são dadas quaisquer garantias quanto à sincronização da propagação de eventos e dados. Ou seja, não é garantido que quando uma função bloco recebe um evento, os valores das entradas correspondentes tenham sido já actualizadas com os seus novos valores.

O standard IEC 61499 define ainda formalmente uma sintaxe textual para a declaração destas funções bloco básicas. Esta sintaxe re-utiliza muitos dos elementos sintácticos definidos no standard IEC 61131-3, em especial os elementos utilizados na declaração de tipos de dados, tipos de dados derivados, valores constantes e variáveis. No entanto, e à semelhança do IEC 61131-3, o tratamento a dar aos identificadores é confuso por estar totalmente omisso do IEC 61499. De facto, não fica explícito se as palavras chave definidas no IEC 61131-3 são também consideradas palavras chave pela sintaxe definida no IEC

61499. Adicionalmente, nada é dito quanto ao tratamento a dar aos elementos léxicos introduzidos pela sintaxe do IEC 61499, como por exemplo 'ALGORITHM' e 'IN'. Ou seja, será que é aceitável a definição de variáveis de funções bloco com estes nomes?

5.1.4 Funções Bloco Compostas

Funções bloco compostas são funções bloco que agregam um conjunto de outras funções bloco cujas entradas e saídas de dados e eventos se encontram ligadas entre si, resultando numa rede. As ligações dos eventos para formar a rede de ligações seguem as regras que seriam de esperar:

- a cada saída de evento será ligada uma entrada de evento de uma das funções bloco da rede, ou a uma saída de eventos da função bloco composta

- a cada entrada de evento será ligada uma saída de evento de uma das funções bloco da rede, ou a uma saída de eventos da função bloco composta

- tanto as saídas de eventos bem como as entradas de eventos das funções bloco que compõe a rede podem ser deixadas por ligar

- já as entradas e saídas de evento da função bloco composta terão forçosamente de estar ligadas, podendo até existir uma ligação directa de uma entrada para uma saída.

Regras semelhantes aplicam-se às entradas de dados, com a excepção de que agora uma saída de dados poderá ser ligada a várias entradas de dados. Na realidade pode-se considerar que o mesmo ocorre com os eventos, uma vez que poderão ser utilizadas funções bloco pré-definidas que geram uma cópia do evento de entrada para dois ou mais eventos de saída, funcionando assim como multiplicadores de eventos.

A invocação, equivalente ao envio de um evento, a uma função bloco composta resulta no encaminhamento desse evento para a função bloco interna à qual se encontra ligada a entrada de evento, e correspondente invocação da função bloco interna. Os eventos gerados pela função bloco interna são encaminhados para as outras funções bloco internas apropriadas, ou para a saída de eventos da função bloco composta. Com o recurso de funções bloco que fazem a multiplicação de eventos poderão ser criadas funções bloco compostas cuja invocação resulta na execução de várias funções bloco internas em simultâneo. De facto, o encaminhamento de eventos dentro das funções bloco compostas introduzido pelo standard IEC 61499, permite solucionar uma das fraquezas do IEC 61131-3 para uso em programação de aplicações de elevada integridade. Com os eventos torna-se possível definir de forma conclusiva e sem ambiguidade a ordem de execução das funções bloco internas, o que não acontecia com o modelo de execução das funções bloco definidas no IEC 61131-3.

Apesar disto, a semântica associada às ligações entre as entradas e saídas de

dados não se encontra completamente definida pelo standard. De facto, embora não seja aparente qualquer afirmação categórica, parece implícito que a transferência de valores entre as entradas e saídas se efectua sem qualquer armazenamento do valor em causa. Deve ainda ser lembrado que nada é garantido quanto à sincronização da propagação de eventos e dados. Ou seja, não é garantido que quando uma função bloco interna receba um evento, os valores das entradas correspondentes tenham sido já actualizadas com os seus novos valores. No entanto, e caso seja utilizada uma semântica de restrições das associações entre eventos e dados, e ainda que cada dados esteja associado a nunca mais do que um evento, então torna-se possível efectuar uma verificação automática da correcção das ligações entre os dados e eventos das funções bloco internas que constituem a rede de funções bloco.

5.1.5 Aplicações e Sub-aplicações

As aplicações e sub-aplicações são praticamente idênticas às funções bloco compostas, regendo-se pelas mesmas regras de ligação das funções bloco internas, e pelas mesmas regras de execução e avaliação. No entanto, e uma vez que as sub-aplicações não podem ter variáveis internas próprias, poderão assim ser distribuídas por vários recursos. O mesmo já não acontece com as funções bloco compostas, uma vez que se estas fossem distribuídas por vários recursos a implementação das suas variáveis internas exigiria um mecanismo de partilha de memória entre esses mesmos recursos.

No entanto, e devido ao facto de as funções bloco internas puderem encontrar-se em dispositivos distintos, recorrendo por isso a uma rede de dados para transferir os dados e eventos, o tempo necessário para propagar os eventos e os novos valores das saídas de dados de uma função bloco para outra podem vir a sofrer aumentos consideráveis. Adicionalmente, a semântica da propagação dos dados poderá vir a sofrer grandes alterações tendo em conta que é relativamente pesado garantir a sincronização do estado global em sistemas distribuídos. Assim, será pouco provável que as implementações deste standard garantam que a chegada dos eventos se faça pela mesma ordem a todos os dispositivos.

5.1.6 Funções Bloco de Serviços de Interface

Como referido anteriormente, as funções bloco de serviços de interface modelizam as interfaces disponibilizadas por cada recurso, e através das quais as aplicações poderão ter acesso ao processo que se encontram a controlar, às rede de dados às quais se encontra ligado o dispositivo, e ainda a outros serviços auxiliares tais como serviços de gestão de estado, de configuração, ou mesmo de tratamento de erros e excepções.

Como seria de esperar, o standard especifica apenas a semântica que deverão implementar estas funções bloco de serviços de interface. Nada é referido quanto

à construção interna e detalhes de implementação. A definição da sua semântica recorre aos métodos já utilizados na definição das interfaces entre as sete camadas do modelo OSI (Open Systems Interconnect) para redes de comunicação. Desta forma, todas as funções bloco de serviços de interface deverão apresentar dois eventos de entrada (REQ – request, RSP – response), e dois de saída (CNF – confirmation, IND – indication). Em paralelo, apresentam ainda um evento de entrada utilizado para requerer a sua inicialização (INIT), e outro para confirmar a conclusão da inicialização (INITO).

Já para as entradas de dados apresentam uma entrada lógica (QI) utilizada em conjunto com os eventos para indicar se é requerido o serviço (valor lógico verdadeiro) ou o seu cancelamento (valor lógico falso). Apresentam ainda uma entrada para os parâmetros do serviço requisitado (PARAMS), e um número não especificado de entradas para os dados desse mesmo serviço (SD_1, SD_2, SD_3, ...).

As saídas de dados incluem uma saída lógica (QO) indicando o sucesso ou falha de um serviço que tenha sido completado, uma saída de estado (STATUS) que indica o estado final do serviço que foi requerido, e saídas de dados (RD_1, RD_2, RD_3, ...).

A especificação dos serviços fornecidos é efectuada, para cada tipo de função bloco de serviços de interface, com recurso a diagramas temporais que definem a sequência correcta de ocorrência dos eventos de entrada e saída.

Serviços de Comunicação

As funções bloco de serviços de comunicação permitem ás aplicações acederem às redes de dados aos quais se encontram ligados os dispositivos. Este acesso é efectuado de uma forma transparente da rede em questão, sendo que cada fornecedor deverá fornecer uma implementação das funções bloco de serviços de comunicação para cada rede de dados. Caso estas funções bloco fornecerem o mesmo serviço com a mesma semântica, será possível transferir uma aplicação para outros dispositivos que utilizem outra rede de dados sem se tornar necessário alterar o programa original.

Infelizmente o standard não especifica um conjunto de funções bloco de serviços de comunicação com semânticas bem definidas, definindo apenas quais as características que devem obedecer as funções bloco utilizadas nas interfaces com as redes de comunicação. De qualquer modo, são ainda sugeridas, sem força normativa, dois conjuntos de funções bloco que implementam as primitivas de publicação/subscrição (publish/subscribe) e de cliente/servidor (client/server). Embora seja ainda sugerida também uma sintaxe de transferência de dados baseada em ASN.1 para ser utilizada nas redes de dados, esta não é suficiente para por si só fomentar o aparecimento de sistemas compatíveis uma vez que as funções bloco deixam muitos outros pormenores à escolha do implementador. Por exemplo, o formato da entrada de dados PARAMS utilizada para especificar

endereços de rede, parâmetros de qualidade de serviço, etc. A própria semântica fornecida pelas funções não se encontra completamente especificada, ficando por exemplo ao critério do implementador a reacção a erros na transferência de dados.

Serviços de Gestão e Configuração

O standard especifica que o conjunto das funções bloco de serviços de gestão e configuração deverão permitir a gestão de um recurso de tal forma que seja possível criar e remover tipos de dados, tipos e instâncias de funções blocos, e conexões entre as instâncias de função bloco. Deverá ainda ser possível o arranque e paragem de instâncias de funções bloco e de aplicações, e obter o estado de qualquer entidade desse recurso. Para tal define o comportamento que deverão ter as funções bloco que são geridas por estes serviços. Em particular, define uma maquina de estados com quatro estados (IDLE, RUNNING, STOPPED, KILLED), as acções que a função bloco deverá executar em cada estado (inicializar, executar o ECC, completar a execução dos algoritmos, terminar a execução dos algoritmos, respectivamente), e os comandos que provocarão a transição entre estes estados (criar, arrancar, parar, re-inicializar, matar, e remover).

Já para as as funções bloco de serviços de gestão e configuração de dispositivos, estas deverão permitir a criação, remoção, inicialização, arranque, paragem e obtenção do estado de recursos dentro desse dispositivo.

O standard não define no entanto os tipos das funções bloco de serviços de gestão e configuração que deverão ser implementadas, ficando estas ao critério do implementador.

5.1.7 Interface de Adaptação

O modelo de programação definido no IEC 61499 é um modelo orientado a objectos, com as instâncias de funções bloco a servirem de objectos. Nestes casos, acontece por vezes que várias classes (que as funções bloco representam) inter-agem de uma forma comum. Por exemplo, a inter-acção entre um dispositivo físico (robot, AGV – automated guided vehicle) que transfere peças, e um outro dispositivo que aceita as peças para as trabalhar (torno, estação de pintura, armazém). Nestes casos poder-se-á definir uma forma de inter-agir comum entre estas duas classes de dispositivos, sendo que cada interacção será forçosamente entre um dispositivo de cada papel (de transferência ou de trabalho sobre a peça). No fundo, está-se a recorrer à abstracção para tratar todos os dispositivos de transferência de peças de modo comum, o mesmo ocorrendo para os dispositivos de trabalho de peças. No entanto, muitas vezes a inter-acção entre estes dispositivos é suficientemente complexa para necessitar a troca de vários eventos e de dados entre as funções bloco que os modelizam.

Por forma a facilitar a programação destas inter-acções mais complexas, o standard permite a definição de interfaces de adaptação, que não são mais do que a definição de um conjunto de eventos de entrada e saída, e parâmetros de entrada e saída. No exemplo referido, a interface de adaptação incluiria todos os eventos necessários para sincronizar as acções de transferência de peças, bem como os dados necessários para identificar as peças.

Assim, cada função bloco que representa um tipo de dispositivo físico (robot, torno, ...) poderá incluir todos os eventos e dados da interface de adaptação na sua própria interface (i.e. o conjunto de eventos e de dados que partilha com o exterior), ficando à partida possibilitado de inter-agir com as restantes funções bloco. No entanto, esta interface não é simétrica. Os dispositivos que agem no papel de transferência de objectos entram na interacção de um ponto de vista oposto aos dispositivos que operam sobre os objectos. Assim, também as funções bloco que representam os dispositivos que transferem as peças a serem trabalhados utilizam a interface de adaptação de forma simétrica aos que trabalham as peças. De um lado os eventos serão de saída ou gerados, enquanto do outro os eventos serão de entrada ou aceites, e vice-versa. O mesmo ocorre com os dados.

Por forma a facilitar a conexão de funções bloco que aceitam a interface de adaptação de lados opostos, esta representa-se graficamente como uma única saída para as funções bloco num dos lados da inter-acção, e como uma única entrada para as funções bloco do lado oposto. No entanto, esta única saída/entrada inclui implicitamente todos os eventos e dados necessários para completar a inter-acção com sucesso.

5.1.8 Relação com o IEC 61131-3

O IEC 61499 define funções bloco que são elementos sintácticos distintos das funções bloco do IEC 61131-3, muito embora partilhem o mesmo nome, o que poderá levar a algumas confusões. As funções bloco do IEC 61499 contêm algoritmos, os quais poderão ser especificados recorrendo às linguagens de programação definidas no IEC 61131-3, incluindo as funções e funções bloco desse standard. Assim, as funções bloco do 61131-3 serão utilizadas para definir o comportamento dos algoritmos das funções bloco do 61499.

No entanto, e para facilitar a re-utilização de código já desenvolvido no âmbito do IEC 61331-3, o IEC 61499 prevê duas formas de mapear as funções bloco e funções do IEC 61131-3 em funções bloco de serviço de interface do IEC 61499.

Na primeira das conversões, os nomes das funções e funções bloco do IEC 61131-3 serão precedidas de 'FB_', passando, por exemplo, a função 'LIMIT' a ser uma função bloco de serviço de interface básica chamada 'FB_LIMIT'. Os parâmetros de entrada e saída serão mapeados de forma directa em entradas e saídas de dados da nova função bloco, sendo ainda acrescentadas as entradas de

eventos 'INIT' e 'REQ', as saídas de eventos 'INITO' e 'CNF', e ainda as entrada de dado lógica 'QI' e saídas de dados 'QO' e 'STATUS'. A recepção de um evento 'REQ' positivo (ou 'REQ+' i.e. com entrada QI com valor lógico verdadeiro) resultará na execução do código da função ou função bloco do IEC 61131-3. Após a conclusão com sucesso da sua execução, será gerado um evento de saída 'CNF+' (i.e. com saída lógica QO com valor verdadeiro). Já um erro na execução do código deverá resultar na geração de um evento 'CNF-'.

A recepção de um evento 'INIT+' será ignorada pelas funções bloco de serviço de interface que resultam da conversão de funções do IEC 61131-3, provocando apenas a inicialização das variáveis internas e o estado dos SFC das funções bloco do IEC 61131-3 que se encontram a ser mapeadas. A conclusão da inicialização será indicada pela geração de um evento de saída 'INITO+'.

A recepção de eventos 'REQ-' e 'INIT-' deverão ser ignorados pela nova função bloco de serviço de interface.

O segundo método de conversão resulta em funções bloco de serviço de interface básicas com semântica praticamente idêntica à do primeiro método de conversão, sendo no entanto agora os novos nomes precedidos de 'E_', sendo que para o mesmo exemplo anterior a função 'LIMIT' resultaria numa função bloco 'E_LIMIT'. A única diferença na semântica para o primeiro método de conversão reside no facto de apenas ser gerado um evento de saída 'CNF+' após a conclusão da execução do código (iniciado pela recepção de em evento 'REQ+') se esta execução resultar numa alteração dos estados das saídas de dados.

5.2 Mapeamento na Arquitectura do MatPLC

Por forma a permitir que o MatPLC seja compatível com a norma IEC 61499, deverá ser inicialmente efectuado um mapeamento das abstracções definidas nesta norma para a arquitectura do MatPLC.

Desta forma, e embora a norma IEC 61499 assuma que os recursos representem processadores independentes de um mesmo PLC, no caso do MatPLC torna-se mais natural mapear cada recurso em um modulo do MatPLC. Caso o MatPLC esteja a ser executado numa plataforma SMP (Symmetrical Multi-Processing) com mais do que um CPU, poder-se-á permitir que cada modulo seja configurado de tal forma que execute unicamente num determinado CPU. Consegue-se assim atingir a flexibilidade pretendida pela norma, não só em PLCs com vários CPUS, mas também em PLCs com um único CPU.

Já as unidades de organização definidas na norma IEC 61499, ou seja as aplicações e as funções bloco, poderão ser implementadas directamente como tal

dentro do modulo do MatPLC. Na realidade, um compilador ou aplicação de configuração que segue este standard pode mapear estas abstracções em objectos da linguagem C++, que por sua vez irão interagir com os objectos C++ nos quais são transformados os blocos de código escritos nas linguagens definidas pela norma IEC 61131. Fica claro que a semântica especifica das funções bloco definidas na norma IEC 61499, em especial as maquinas de estado ECC, e a transferência de eventos entre funções bloco, terá de ser implementada em código que fará parte do modulo do MatPLC. Este código fica assim de fora do núcleo do MatPLC, tornando o MatPLC numa plataforma base sobre a qual se poderão implementar conjuntos de abstracções que implementam diversas normas (neste caso o IEC 61131 e o IEC 61499)[1].

Uma vez que o IEC 61499 pressupõe a possibilidade de execução em paralelo de vários algoritmos activos numa mesma função bloco composta, sendo a execução co-ordenada por um algoritmo escalonador não especificado, no caso do MatPLC a execução destes algoritmos poderá ser atribuída a uma thread, passando assim a responsabilidade de escalonamento dos algoritmos (transformados em threads) para o sistema operativo sobre o qual executa o MatPLC. Esta solução poderá no entanto resultar numa proliferação de threads (caso existam muitos algoritmos), que o sistema operativo pode não estar preparado para comportar. Por esta facto, a implementação do suporte do IEC 61499 provavelmente evoluirá para uma arquitectura na qual existirão um conjunto de threads trabalhadoras às quais serão atribuídas a execução de algoritmos à medida que estes vão sendo activados.

Como já foi referido, este mapeamento de abstracções resulta na implementação das funções bloco do IEC 61499 como classes de objecto em C++. Estas classes têm uma função-membro para cada entrada de evento da função bloco IEC 61499, que será invocada sempre que a função bloco recebe um evento de entrada. Esta função membro (que implementa a entrada de eventos) pode então fazer a leitura das entradas de dados associadas ao evento de entrada. Esta leitura é efectuada directamente de variáveis internas dos objectos em C++. Desta forma, as saídas de dados resultam na simples publicação das variáveis internas que contêm esses mesmos dados, sendo da responsabilidade da função bloco que recebe os dados a sua leitura sempre que recebe um evento de entrada. Como seria de esperar, os eventos de saída são implementados como a invocação das funções-membro que correspondem aos eventos de entrada ao qual se encontram ligados esses mesmos eventos de saída.

1 De forma semelhante a arquitectura do sistema operativo Windows NT, que é composto por um nucleo interno sobre o qual se podem desenvolver conjuntos de interfaces que fornecem servicos definidos numa determinada norma ou standard *de facto*. Efectivamente, sobre o nucleo do Windows NT existe já uma interface que fornece os servicos do Windows definidos pela própria Microsoft, e ainda uma outra interface que fornece os servicos definidos pela norma POSIX. A arquitectura permitiria ainda o desenvolvimento de outras interfaces que emulem outros sistemas operativos (p.ex. o VMS), no entanto, por razões comerciais, estas nunca chegaram a ser desenvolvidas.

As funções bloco pré-definidas no standard IEC 61499, bem como as funções bloco de serviços de comunicação, são disponibilizadas através de uma biblioteca de funções à qual os módulos do MatPLC são ligados (*linked*). Desta forma, a passagem de eventos e dados entre funções bloco residentes em recursos distintos será efectuada sempre através de uma função bloco de serviço de comunicação.

As funções bloco de serviços de comunicação acedem aos dispositivos de rede (por exemplo cartas de rede Ethernet ou de redes de campo) directamente, isto é sem recorrerem a outros módulos do MatPLC que implementam Entradas/Saídas remotas através dessas mesmas interfaces. A implementação do IEC 61499 deve ainda disponibilizar uma implementação de funções bloco de serviços de comunicação destinados a transferir dados e eventos entre módulos do MatPLC (nos quais são mapeados os recursos do IEC 61499). Esta transferência de informação pode recorrer a qualquer mecanismo de comunicação síncrono entre processos capaz de transferir dados, como o são os sockets UNIX.

5.3 Protocolos de Rede

Uma vez que o standard IEC 61499 não especifica ele próprio um protocolo de rede para a transferência de dados e eventos sobre redes Ethernet, cabe a cada implementação a escolha ou especificação de um protocolo adequado para tal. Tendo em conta que é filosofia do MatPLC re-utilizar standards livres e abertos e com aceitação no mercado sempre que estes existam, torna-se pois necessário efectuar uma avaliação das alternativas.

Do mesmo modo que os fabricantes adoptaram protocolos distintos para as redes de campo, o mesmo ocorre com os protocolos sobre Ethernet e eventualmente sobre TCP/IP. A Siemens utiliza o PROFInet, a Schneider o IDA, enquanto que a OMRON e a Rockwell recorrem ao Ethernet/IP (ODVA). Por outro lado, existem ainda outros protocolos ou 'middlewares' que poderiam eventualmente ainda ser utilizados para o suporte da distribuição das funções bloco do IEC 61499.

De forma resumida, os protocolos Ethernet/IP, e HSE (Fieldbus Foundation) não estão disponíveis para o publico em geral de forma gratuita e aberta, pelo que não podem ser considerados como possíveis soluções. Já o protocolo PROFInet encontra-se disponível de forma gratuita, mas obriga à aceitação de condições restritivas quanto à divulgação do mesmo por terceiros, bem como à própria implementação do protocolo. Por outro lado, o PROFInet baseia-se na infra-estrutura de comunicações DCOM da Microsoft, para a qual não existe de momento qualquer implementação aberta qualquer que seja o sistema operativo considerado. Por estes motivos o protocolo PROFInet também não pode ser

considerado como possível solução para o projecto MatPLC.

Pelo contrário, o protocolo do consórcio IDA [49][50]é distribuído sem quaisquer condições de utilização e implementação, pelo que será considerado e analisado em mais pormenor.

5.3.1 IDA – Interface para Automação Distribuída

O IDA é um consórcio, liderado pela Schneider, constituído em 2000 com o intuito de desenvolver um protocolo sobre Ethernet estandardizado para ambiente fabril. Tem a vantagem de ser baseado na arquitectura definida na norma IEC 61499, contendo desde logo o conceito de Funções Bloco (IDA Block), Sub-aplicações (IDA Subapplication), recursos (IDA Resource) e dispositivos (IDA Device).

Diverge no entanto da arquitectura do IEC 61499 no que refere às ligações entre blocos. Assim, as ligações entre blocos são conhecidas como conexões, sejam elas utilizadas para a transferência de eventos ou para o envio de dados. Por outro lado, o IDA permite três tipos de conexões: para a transferência de dados, para o envio de eventos, e para a invocação remota de funções. Tanto o envio de dados como o envio de eventos recorre ao mecanismo de *publish/subscribe* para a transferência dos mesmos.

As conexões são todas implementadas recorrendo ao protocolo RTPS (Real-Time Publish Subscribe)[51], que por sua vez utiliza o protocolo UDP para a transferência de mensagens. O RTPS é também ele um protocolo aberto, estando a sua especificação disponível sem restrições. Este fornece um serviço de publicação assíncrona ou síncrona de dados sem falhas. Uma vez que recorre para isso ao *multicast* UDP, e utiliza um modelo de objectos que age de forma determinística, tem condições de fornecer algumas garantias de tempo-real em condições específicas (nomeadamente se for utilizado hardware apropriado, uma implementação do UDP apropriada, etc.). O RTPS fornece ainda um serviço do tipo cliente/servidor com garantias sobre UDP, o qual é utilizado pelo IDA para a invocação remota de funções.

O protocolo RTPS trata ainda da descoberta dos fornecedores de dados por parte dos subscritores dos mesmos, baseando-se apenas no 'tópico' sob o qual é disponibilizado o dado em questão. Permite ainda que haja várias entidades a publicar o mesmo dado, cada uma com uma 'força' distinta, sendo que cada subscritor utiliza apenas o dado com a maior força que recebe. Com este mecanismo torna-se possível ter máquinas redundantes em *hot standby* e com *failover* automático e sem grandes atrasos. O mecanismo da 'força' está também disponível no serviço cliente/servidor, através do qual se pode ter vários servidores activos em simultâneo.

Tendo em conta que o protocolo ADI está logo à partida vocacionado para suportar a arquitectura definida no IEC 61499, e considerando ainda que existe

já uma implementação do protocolo RTPS sob a licença GPL (projecto ORTE [52]), torna-se claro que a adopção deste protocolo para o suporte das comunicações é não só adequado, como também indicado para o fim em vista.

Capítulo 6

Conclusões

6.1 O MatPLC

No decorrer do presente trabalho foi desenvolvido um PLC em software, ao qual foi dado o nome de MatPLC, capaz de ser executado em qualquer sistema operativo compatível com a norma POSIX. Este PLC recorre a uma arquitectura modular, o que permite um desenvolvimento mais organizado, e ainda a implementação simultânea de módulos de forma independente. Cada módulo executa no seu próprio processo (gerido pelo sistema operativo), sendo a comunicação e sincronização de dados e acções entre os módulos coordenada por um núcleo do MatPLC.

O núcleo do MatPLC recorre à uma arquitectura de exo-núcleo. Ou seja, as funções que o núcleo implementa são executadas pelas linhas de execução dos próprios processos que contêm cada modulo do MatPLC. Reduz-se assim ao mínimo as trocas de contexto entre processos, reduzindo também o tempo necessário para executar as tarefas do núcleo do MatPLC.

Por outro lado, o núcleo do MatPLC está também ele organizado de forma modular, consistindo em várias secções, cada uma com as suas responsabilidades.

Cada secção é composta por um conjunto de funções que são executadas no arranque do MatPLC, por um segundo conjunto que executam aquando da inicialização e término de cada modulo do MatPLC, e um último conjunto de funções que se destinam a ser invocadas por cada modulo durante a execução normal. As funções de inicialização encarregam-se de reservar todos os recursos de que necessitarão durante a execução do MatPLC e/ou modulo em questão, permitindo que a restante execução não sofra de atrasos intempestivos devido à reserva de recursos que se mostrem necessários. Consegue-se assim que a execução normal dos módulos seja mais determinística, facilitando a determinação de um majorante do tempo de execução máximo e no pior caso de cada modulo.

O núcleo disponibiliza serviços que permitem a partilha do estado das variáveis partilhadas por todos os módulos de forma sincronizada e sem perigo de 'race conditions'. Permite ainda a sincronização da execução dos módulos, sendo esta sincronização efectuada através de uma rede de petri pelo utilizador que se encontra a configurar o MatPLC. Permite ainda a configuração de parâmetros que possibilitam a execução dos módulos em ambientes de tempo real, incluindo a prioridade estática (para sistemas operativos tempo real com mecanismos de escalonamento baseados em prioridades estáticas), e a reserva de memória que será utilizada pelo modulo para a RAM física do computador (permitindo assim obviar a possibilidade da transferência de estes módulos para a memória mapeada para disco duro, e todos os atrasos de tempo que daí possam advir). O núcleo inclui ainda uma secção para a gestão do período de execução de cada modulo, e outra secção que gere o estado (RUN/IDLE) de cada modulo e do MatPLC em geral.

Foram ainda desenvolvidos alguns módulos que permitiram testar e avaliar a execução do núcleo do MatPLC. Foi com este intuito que foi implementado o modulo de processamento digital de sinais o qual inclui blocos para controladores PID, filtragem de sinais, e ainda blocos de condicionamento de valores com funções não lineares. Foram também implementados módulos de interface com dispositivos físicos de entrada e saída de dados, nomeadamente um modulo de interface com a porta paralela dos PC, bem como um modulo de interface com cartas de rede da Hilscher, através das quais o MatPLC pode comunicar com qualquer dispositivo ligado a redes de campo ASI, ModBUS, PROFIBUS, Devicenet, entre muitas outras.

Os testes efectuados à execução do MatPLC permitem concluir que a sua execução é altamente determinística quando executada sobre um sistema operativo tempo real. De facto, e para um computador baseado num CPU Pentium II a 350 Mhz, são possíveis ciclos de execução na ordem do 1 ms quando são executados dois módulos que executam de forma assíncrona, partilhando 32 Kbytes de dados entre eles. Tempos de ciclo inferiores são sempre possíveis caso se recorra à execução síncrona dos módulos.

6.2 O Compilador IEC 61131-3

A par do MatPLC, foi ainda desenvolvido um compilador para as linguagens IL e ST definidas no standard IEC 61131-3. Este compilador foi dividido em quatro secções, que executam em três fases distintas. Na primeira fase são executadas a secção responsável pela análise lexical e a secção de análise sintáctica do programa de entrada. Estas secções geram uma estrutura de dados em forma de arvore que representa o programa de entrada. Na segunda fase é executada a terceira secção que se responsabiliza em verificar a correcção semântica do programa de entrada. Esta secção, por motivos de tempo, não se encontra implementada desde já. Na terceira fase é executada a quarta secção, que se responsabiliza em gerar ficheiro executável que corresponde ao programa de entrada. De momento esta última fase é executada sob duas passagens: a primeira gera um programa em C++ equivalente ao programa de entrada, sendo este programa em C++ posteriormente compilado pelo compilador da gcc (da GNU) na segunda passagem, resultando daí o ficheiro com o código executável final.

Durante o desenvolvimento deste compilador foram identificadas diversas falhas no standard que define as linguagens, tanto a nível sintáctico bem como a nível semântico. Foram ainda sugeridas alterações ao standard que permitem ultrapassar as questões referidas.

Em termos sintácticos as falhas mais importantes relacionam-se com a deficiente gestão dos identificadores utilizados nos programas. De facto, o standard limita-se a definir um conjunto de palavras chave que não são permitidas ser re-utilizadas para muitos dos locais onde são necessários identificadores, criando situações potencialmente confusas para os programadores, e deixando mesmo assim algumas situações dúbias de interpretação do código. O standard permite ainda a declaração de varáveis sem ser necessário definir qual o tipo de dados que nela serão armazenadas, ficando por esclarecer qual a semântica a atribuir a estas declarações, sendo no entanto o mais provável que seja um erro na definição formal da sintaxe das linguagens.

Em relação à semântica das linguagens foram identificadas duas questões principais. A primeira relacionada com a falta de definição do modo como tratar os parâmetros de inicialização de tarefas declaradas dentro de configurações, a qual pode ser efectuada de uma de duas formas não compatíveis entre elas. A segunda questão semântica está relacionada com a persistência dos valores de algumas variáveis das funções bloco, sendo que neste caso o standard entra em contradição uma vez que são definidas semânticas distintas e incompatíveis entre elas em locais diferentes do standard.

Foi ainda efectuada uma análise à adequabilidade de utilizar as linguagens IL e ST para desenvolver aplicações que serão utilizadas no controlo de sistemas com

elevada fiabilidade. Através desta análise pode-se concluir que de facto as linguagens e a arquitectura nas quais se enquadram satisfaz a maioria dos requisitos mais importantes para as linguagens de desenvolvimento de aplicações de elevada integridade. De facto, não fossem as falhas identificadas na definição sintáctica das linguagens, bem como a falta de uma definição formal da semântica das mesmas, poder-se-ia concluir que estas linguagens seriam adequadas para o desenvolvimento de aplicações de elevada fiabilidade.

6.3 Desenvolvimentos Futuros

O presente trabalho teve como fio condutor o de proporcionar ferramentas para o desenvolvimento de aplicações de controlo de sistemas de produção em ambientes fabris, sendo estas ferramentas disponibilizadas sob a licença GPL. Este objectivo foi claramente atingido, no entanto fica ainda por saber se no futuro verá a ser realizada a grande vantagem das aplicações disponibilizadas sob esta licença, nomeadamente de que os utilizadores destas ferramentas venham eles próprios a participar no seu desenvolvimento, seja através de identificação de erros, do código para corrigir esses erros, ou mesmo através de código que acrescente novas capacidades às mesmas ferramentas. Tendo em conta que as ferramentas ainda não dispõe de ambientes gráficos de configuração, é compreensível que a comunidade ainda se encontra em fase de criação. Apesar disto, começaram já a aparecer de forma muito embrionária algumas solicitações para a introdução de novas funcionalidades, a disponibilização de integração de outros pequenos projectos com o MatPLC, e ainda a identificação de alguns erros, em especial no que se refere à portabilidade do código para diversas versões do núcleo do Linux e as suas bibliotecas (sendo que alguns dos erros identificados se encontram nas bibliotecas e não no MatPLC em si).

De qualquer modo, e de forma a fomentar o desenvolvimento da referida comunidade de utilizadores e programadores, é desejável a criação de referidas interfaces gráficas que permitam a configuração e controlo do estado do MatPLC. Em paralelo, é ainda importante que venham a ser disponibilizadas ferramentas que permitam o desenvolvimento de aplicações com base nas restantes linguagens de programação gráficas definidas no IEC 61131-3, nomeadamente as SFC, FBD, e LD. Tendo isto em consideração a intenção será a de criar estas ferramentas, muito provavelmente integradas no ambiente gráfico KDE (K Desktop Environment) que ele próprio se encontra já disponível para diversos sistemas operativos, incluindo o Linux, o FreeBSD, NetBSD, OpenBSD, AIX (da IBM) e Solaris (da SUN). A utilização do ambiente KDE permite tirar proveito da infra-estrutura de integração de aplicações desenvolvidas sobre esta mesma estrutura, que inclui as tecnologias Kparts (integração de uma aplicação

dentro de outra), DCOP (Desktop Communication Protocol) e KIO (acesso a ficheiros de forma transparente recorrendo a diversos protocolos de rede, ou mesmo ao disco local). Assim, o editor gráfico para cada uma das linguagens do IEC 61131-3 (SFC, FBD e LD) será um componente (Kpart), que será integrado numa aplicação genérica de configuração do MatPLC. Será provável que venham ainda a ser disponibilizados componentes que permitam a configuração através de uma interface gráfica para cada módulo do MatPLC (por exemplo, a configuração do modulo DSP), ou mesmo das opções globais do MatPLC (quais os módulos a incluir, variáveis a utilizar, sincronização entre módulos, etc). As opções escolhidas pelo utilizador poderão vir a ser gravadas num ficheiro de configuração a ser utilizado pelo MatPLC, o poderão mesmo vir a ser descarregadas para um outro dispositivo (que se dedica a executar o MatPLC, e não é utilizado para a configuração gráfica) através de algum protocolo de rede já suportado pelo KIO (FTP, HTTP, NFS, ...) ou um protocolo novo especificamente desenvolvido para o controlo do estado do MatPLC e dos seus módulos.

Em paralelo com as aplicações de configuração e programação gráficas, torna-se ainda importante o desenvolvimento de ferramentas que permitam a cooperação do MatPLC com outras aplicações já com forte aceitação no mercado. Em especial, será necessário permitir que o MatPLC interaja com aplicações que forneçam uma interface de programação compatível com a 'norma' OPC (OLE for Process Control). Várias alternativas se apresentam, incluindo o de desenvolver uma ponte entre o protocolo de rede do DCOM (Distributed Component Object Model), o mecanismo utilizado pelo OPC para a transferência de mensagens através de redes de dados, e o protocolo de rede do CORBA (Common Object Request Brocker Arquitecture), i.e. o IIOP (Internet Inter-Operable Protocol). Outra alternativa será o de desenvolver uma biblioteca de funções que permitam aceder a objectos COM remotos a partir de máquinas com Linux. Para tal poderá tirar-se partido de implementações do DCE (Distributed Computing Environment) para Linux, uma vez que o DCOM utiliza o mecanismo de RPC (Remote Procedure Calls) do DCE, e o mesmo protocolo de rede, para a comunicação entre objectos separados por uma rede de dados.

Ainda outra vertente que poderá ser explorada será a integração no MatPLC de mecanismos e protocolos de rede que permitem interações com tempos determinísticos entre processos separados por uma rede de dados. Mais específicamente, existem já vários protocolos em investigação para permitir a comunicação determinística sobre redes ethernet para os quais poderão ser desenvolvidos módulos de IO para interface com o MatPLC.

ANEXO A

Lista de Tipos de Elementos que Povoam a Árvore de Sintaxe Abstracta do Compilador IEC 61131-3.

```
/***************************/
/* B 0 - Programming Model */
/***************************/
SYM_LIST(library_c)

/************************/
/* B.1 - Common elements */
/************************/
/*****************************************/
/* B 1.1 - Letters, digits and identifiers */
/*****************************************/
SYM_TOKEN(identifier_c)

/*********************/
/* B 1.2 - Constants */
/*********************/
/* B 1.2.1 - Numeric Literals */
SYM_TOKEN(real_c)
SYM_TOKEN(integer_c)
SYM_TOKEN(binary_integer_c)
SYM_TOKEN(octal_integer_c)
SYM_TOKEN(hex_integer_c)
/* Not required:
SYM_TOKEN(signed_integer_c)
SYM_TOKEN(signed_real_c)
*/
SYM_REF2(numeric_literal_c, type, value)
SYM_REF2(integer_literal_c, type, value)
SYM_REF2(real_literal_c, type, value)
SYM_REF2(bit_string_literal_c, type, value)
SYM_REF2(boolean_literal_c, type, value)
/* helper class for boolean_literal_c */
```

```
SYM_REF0(boolean_true_c)
/* helper class for boolean_literal_c */
SYM_REF0(boolean_false_c)

/* B.1.2.2   Character Strings */
SYM_TOKEN(double_byte_character_string_c)
SYM_TOKEN(single_byte_character_string_c)

/* B 1.2.3 - Time Literals */
/* B 1.2.3.1 - Duration */
SYM_REF0(neg_time_c)
SYM_REF2(duration_c, neg, interval)
SYM_TOKEN(fixed_point_c)
SYM_REF2(days_c, days, hours)
SYM_REF2(hours_c, hours, minutes)
SYM_REF2(minutes_c, minutes, seconds)
SYM_REF2(seconds_c, seconds, milliseconds)
SYM_REF2(milliseconds_c, milliseconds, unused)

/* B 1.2.3.2 - Time of day and Date */
SYM_REF2(time_of_day_c, daytime, unused)
SYM_REF4(daytime_c, day_hour, day_minute, day_second, unused)
SYM_REF2(date_c, date_literal, unused)
SYM_REF4(date_literal_c, year, month, day, unused)
SYM_REF2(date_and_time_c, date_literal, daytime)

/**********************/
/* B.1.3 - Data types */
/**********************/
/* B 1.3.1 - Elementary Data Types */
SYM_REF0(time_type_name_c)
SYM_REF0(bool_type_name_c)
SYM_REF0(sint_type_name_c)
SYM_REF0(int_type_name_c);
SYM_REF0(dint_type_name_c)
SYM_REF0(lint_type_name_c)
SYM_REF0(usint_type_name_c)
SYM_REF0(uint_type_name_c)
SYM_REF0(udint_type_name_c)
SYM_REF0(ulint_type_name_c)
SYM_REF0(real_type_name_c)
SYM_REF0(lreal_type_name_c)
SYM_REF0(date_type_name_c)
SYM_REF0(tod_type_name_c)
SYM_REF0(dt_type_name_c)
SYM_REF0(byte_type_name_c)
SYM_REF0(word_type_name_c)
SYM_REF0(dword_type_name_c)
SYM_REF0(lword_type_name_c)
SYM_REF0(string_type_name_c)
SYM_REF0(wstring_type_name_c)

/* B.1.3.2 - Generic data types */
/* B 1.3.3 - Derived data types */
SYM_REF2(data_type_declaration_c, type_declaration_list, unused)
/* helper symbol for data_type_declaration */
SYM_LIST(type_declaration_list_c)
SYM_REF2(simple_type_declaration_c, simple_type_name, simple_spec_init)
SYM_REF2(simple_spec_init_c, simple_specification, constant)
```

```
SYM_REF2(subrange_type_declaration_c, subrange_type_name,
subrange_spec_init)
SYM_REF2(subrange_spec_init_c, subrange_specification, signed_integer)
SYM_REF2(subrange_specification_c, integer_type_name, subrange)
SYM_REF2(subrange_c, lower_limit, upper_limit)
SYM_REF2(enumerated_type_declaration_c, enumerated_type_name,
enumerated_spec_init)
SYM_REF2(enumerated_spec_init_c, enumerated_specification,
enumerated_value)
/* helper symbol for enumerated_specification_enumerated_spec_init */
SYM_LIST(enumerated_value_list_c)
SYM_REF2(enumerated_value_c, type, value)
SYM_REF2(array_type_declaration_c, identifier, array_spec_init)
SYM_REF2(array_spec_init_c, array_specification, array_initialization)
SYM_REF2(array_specification_c, array_subrange_list,
non_generic_type_name)
/* helper symbol for array_specification */
SYM_LIST(array_subrange_list_c)
/* helper symbol for array_initialization */
SYM_LIST(array_initial_elements_list_c)
SYM_REF2(array_initial_elements_c, integer, array_initial_element)
SYM_REF2(structure_type_declaration_c, structure_type_name,
structure_specification)
SYM_REF2(initialized_structure_c, structure_type_name,
structure_initialization)
/* helper symbol for structure_declaration */
SYM_LIST(structure_element_declaration_list_c)
SYM_REF2(structure_element_declaration_c, structure_element_name,
spec_init)
/* helper symbol for structure_initialization */
SYM_LIST(structure_element_initialization_list_c)
SYM_REF2(structure_element_initialization_c, structure_element_name,
value)
string_type_declaration_size string_type_declaration_init */
SYM_REF4(string_type_declaration_c,       string_type_name,
elementary_string_type_name, string_type_declaration_size,
string_type_declaration_init) /* may be == NULL! */

/**********************/
/* B 1.4 - Variables */
/**********************/
SYM_REF2(symbolic_variable_c, var_name, unused)

/* B.1.4.1   Directly Represented Variables */
SYM_TOKEN(direct_variable_c)

/* B.1.4.2   Multi-element Variables */
SYM_REF2(array_variable_c, subscripted_variable, subscript_list)
SYM_LIST(subscript_list_c)
SYM_REF2(structured_variable_c, record_variable, field_selector)

/* B 1.4.3 - Declaration & Initialisation */
SYM_REF0(constant_option_c)
SYM_REF0(retain_option_c)
SYM_REF0(non_retain_option_c)
SYM_REF2(input_declarations_c, option, input_declaration_list)
/* helper symbol for input_declarations */
SYM_LIST(input_declaration_list_c)
SYM_REF2(edge_declaration_c, edge, var1_list)
```

```
SYM_REF0(raising_edge_option_c)
SYM_REF0(falling_edge_option_c)
SYM_REF2(var1_init_decl_c, var1_list, spec_init)
SYM_LIST(var1_list_c)
SYM_REF2(array_var_init_decl_c, var1_list, array_spec_init)
SYM_REF2(structured_var_init_decl_c, var1_list, initialized_structure)
structure_initialization */
SYM_REF4(fb_name_decl_c, fb_name_list, function_block_type_name,
structure_initialization, unused)
SYM_LIST(fb_name_list_c)
SYM_REF2(output_declarations_c, option, var_init_decl_list)
SYM_REF2(input_output_declarations_c, var_declaration_list, unused)
/* helper symbol for input_output_declarations */
SYM_LIST(var_declaration_list_c)
SYM_REF2(array_var_declaration_c, var1_list, array_specification)
SYM_REF2(structured_var_declaration_c, var1_list, structure_type_name)
SYM_REF2(var_declarations_c, option, var_init_decl_list)
SYM_REF2(retentive_var_declarations_c, var_init_decl_list, unused)
SYM_REF2(located_var_declarations_c, option, located_var_decl_list)
/* helper symbol for located_var_declarations */
SYM_LIST(located_var_decl_list_c)
SYM_REF4(located_var_decl_c, variable_name, location,
located_var_spec_init, unused)
SYM_REF2(external_var_declarations_c, option, external_declaration_list)
SYM_LIST(external_declaration_list_c)
subrange_specification|enumerated_specification|array_specification|
prev_declared_structure_type_name|function_block_type_name */
SYM_REF2(external_declaration_c, global_var_name, specification)
SYM_REF2(global_var_declarations_c, option, global_var_decl_list)
SYM_LIST(global_var_decl_list_c)
SYM_REF2(global_var_decl_c, global_var_spec, type_specification)
SYM_REF2(global_var_spec_c, global_var_name, location)
SYM_REF2(location_c, direct_variable, unused)
SYM_LIST(global_var_list_c)
SYM_REF2(single_byte_string_var_declaration_c, var1_list,
single_byte_string_spec)
single_byte_character_string] */
SYM_REF2(single_byte_string_spec_c, integer,
single_byte_character_string)
SYM_REF2(double_byte_string_var_declaration_c, var1_list,
double_byte_string_spec)
double_byte_character_string] */
SYM_REF2(double_byte_string_spec_c, integer,
double_byte_character_string)
SYM_REF2(incompl_located_var_declarations_c, option,
incompl_located_var_decl_list)
/* helper symbol for incompl_located_var_declarations */
SYM_LIST(incompl_located_var_decl_list_c)
SYM_REF4(incompl_located_var_decl_c, variable_name, incompl_location,
var_spec, unused)
SYM_TOKEN(incompl_location_c)
SYM_LIST(var_init_decl_list_c)

/***********************************/
/* B.1.5 - Program organization units */
/***********************************/
/* B 1.5.1 - Functions */
SYM_REF4(function_declaration_c, derived_function_name, type_name,
var_declarations_list, function_body)
```

```
/* intermediate helper symbol for
 * - function_declaration
 * - function_block_declaration
 * - program_declaration
 */
SYM_LIST(var_declarations_list_c)
SYM_REF2(function_var_decls_c, option, decl_list)
/* intermediate helper symbol for function_var_decls */
SYM_LIST(var2_init_decl_list_c)

/* B 1.5.2 - Function Blocks */
SYM_REF4(function_block_declaration_c, fblock_name, var_declarations,
fblock_body, unused)
SYM_REF2(temp_var_decls_c, var_decl_list, unused)
/* intermediate helper symbol for temp_var_decls */
SYM_LIST(temp_var_decls_list_c)
SYM_REF2(non_retentive_var_decls_c, var_decl_list, unused)

/* B 1.5.3 - Programs */
SYM_REF4(program_declaration_c, program_type_name, var_declarations,
function_block_body, unused)

/*******************************************/
/* B.1.6  Sequential function chart elements */
/*******************************************/
/*********************************/
/* B 1.7 Configuration elements */
/*********************************/
SYM_REF6(configuration_declaration_c, configuration_name,
global_var_declarations, resource_declarations, access_declarations,
instance_specific_initializations, unused)
/* helper symbol for configuration_declaration */
SYM_LIST(resource_declaration_list_c)
SYM_REF4(resource_declaration_c, resource_name, resource_type_name,
global_var_declarations, resource_declaration)
SYM_REF2(single_resource_declaration_c, task_configuration_list,
program_configuration_list)
/* helper symbol for single_resource_declaration */
SYM_LIST(task_configuration_list_c)
/* helper symbol for single_resource_declaration */
SYM_LIST(program_configuration_list_c)
/* helper symbol for access_path,instance_specific_init */
SYM_LIST(any_fb_name_list_c)
structure_element_name] */
SYM_REF4(global_var_reference_c, resource_name, global_var_name,
structure_element_name, unused)
SYM_REF2(program_output_reference_c, program_name, symbolic_variable)
SYM_REF2(task_configuration_c, task_name, task_initialization)
SYM_REF4(task_initialization_c, single_data_source,
interval_data_source, priority_data_source, unused)
SYM_REF6(program_configuration_c, retain_option, program_name,
task_name, program_type_name, prog_conf_elements, unused)
SYM_LIST(prog_conf_elements_c)
SYM_REF2(fb_task_c, fb_name, task_name)
SYM_REF2(prog_cnxn_assign_c, symbolic_variable, prog_data_source)
SYM_REF2(prog_cnxn_sendto_c, symbolic_variable, prog_data_source)
SYM_REF2(instance_specific_initializations_c,
instance_specific_init_list, unused)
/* helper symbol for instance_specific_initializations */
```

```
SYM_LIST(instance_specific_init_list_c)
SYM_REF6(instance_specific_init_c, resource_name, program_name,
any_fb_name_list, variable_name, location, initialization)
/* helper symbol for instance_specific_init */
SYM_REF2(fb_initialization_c, function_block_type_name,
structure_initialization)

/***************************************/
/* B.2 - Language IL (Instruction List) */
/***************************************/
/*********************************/
/* B 2.1 Instructions and Operands */
/*********************************/
SYM_LIST(instruction_list_c)
SYM_REF2(il_instruction_c, label, il_instruction)
SYM_REF2(il_simple_operation_c, il_simple_operator, il_operand)
SYM_REF2(il_function_call_c, function_name, il_operand_list)
SYM_REF4(il_expression_c, il_expr_operator, il_operand,
simple_instr_list, unused)
SYM_REF2(il_jump_operation_c, il_jump_operator, label)
SYM_REF4(il_fb_call_c, il_call_operator, fb_name, il_operand_list,
il_param_list)
SYM_REF2(il_formal_funct_call_c, function_name, il_param_list)
SYM_LIST(il_operand_list_c)
SYM_LIST(simple_instr_list_c)
SYM_LIST(il_param_list_c)
SYM_REF4(il_param_assignment_c, il_assign_operator, il_operand,
simple_instr_list, unused)
SYM_REF2(il_param_out_assignment_c, il_assign_out_operator, variable);

/*******************/
/* B 2.2 Operators */
/*******************/
SYM_REF0(LD_operator_c)
SYM_REF0(LDN_operator_c)
SYM_REF0(ST_operator_c)
SYM_REF0(STN_operator_c)
SYM_REF0(NOT_operator_c)
SYM_REF0(S_operator_c)
SYM_REF0(R_operator_c)
SYM_REF0(S1_operator_c)
SYM_REF0(R1_operator_c)
SYM_REF0(CLK_operator_c)
SYM_REF0(CU_operator_c)
SYM_REF0(CD_operator_c)
SYM_REF0(PV_operator_c)
SYM_REF0(IN_operator_c)
SYM_REF0(PT_operator_c)
SYM_REF0(AND_operator_c)
SYM_REF0(OR_operator_c)
SYM_REF0(XOR_operator_c)
SYM_REF0(ANDN_operator_c)
SYM_REF0(ORN_operator_c)
SYM_REF0(XORN_operator_c)
SYM_REF0(ADD_operator_c)
SYM_REF0(SUB_operator_c)
SYM_REF0(MUL_operator_c)
SYM_REF0(DIV_operator_c)
SYM_REF0(MOD_operator_c)
```

```
SYM_REF0(GT_operator_c)
SYM_REF0(GE_operator_c)
SYM_REF0(EQ_operator_c)
SYM_REF0(LT_operator_c)
SYM_REF0(LE_operator_c)
SYM_REF0(NE_operator_c)
SYM_REF0(CAL_operator_c)
SYM_REF0(CALC_operator_c)
SYM_REF0(CALCN_operator_c)
SYM_REF0(RET_operator_c)
SYM_REF0(RETC_operator_c)
SYM_REF0(RETCN_operator_c)
SYM_REF0(JMP_operator_c)
SYM_REF0(JMPC_operator_c)
SYM_REF0(JMPCN_operator_c)
SYM_REF2(il_assign_out_operator_c, option, variable_name)

/*************************************/
/* B.3 - Language ST (Structured Text) */
/*************************************/
/* B 3.1 - Expressions */
SYM_REF2(or_expression_c, l_exp, r_exp);
SYM_REF2(xor_expression_c, l_exp, r_exp);
SYM_REF2(and_expression_c, l_exp, r_exp);
SYM_REF2(equ_expression_c, l_exp, r_exp);
SYM_REF2(notequ_expression_c, l_exp, r_exp);
SYM_REF2(lt_expression_c, l_exp, r_exp);
SYM_REF2(gt_expression_c, l_exp, r_exp);
SYM_REF2(le_expression_c, l_exp, r_exp);
SYM_REF2(ge_expression_c, l_exp, r_exp);
SYM_REF2(add_expression_c, l_exp, r_exp);
SYM_REF2(sub_expression_c, l_exp, r_exp);
SYM_REF2(mul_expression_c, l_exp, r_exp);
SYM_REF2(div_expression_c, l_exp, r_exp);
SYM_REF2(mod_expression_c, l_exp, r_exp);
SYM_REF2(power_expression_c, l_exp, r_exp);
SYM_REF2(neg_expression_c, exp, unused);
SYM_REF2(not_expression_c, exp, unused);
SYM_REF2(function_invocation_c, function_name,
parameter_assignment_list)

/********************/
/* B 3.2 Statements */
/********************/
SYM_LIST(statement_list_c)

/* B 3.2.1 Assignment Statements */
SYM_REF2(assignment_statement_c, l_exp, r_exp)

/* B 3.2.2 Subprogram Control Statements */
SYM_REF0(return_statement_c)
SYM_REF2(fb_invocation_c, fb_name, param_assignment_list)
/* helper symbol for fb_invocation */
SYM_LIST(param_assignment_list_c)
SYM_REF2(input_variable_param_assignment_c, variable_name, expression)
SYM_REF4(output_variable_param_assignment_c, not_param, variable_name,
variable, unused)
SYM_REF0(not_paramassign_c)
```

```
/* B 3.2.3 Selection Statements */
SYM_REF4(if_statement_c, expression, statement_list,
elseif_statement_list, else_statement_list)
SYM_LIST(elseif_statement_list_c)
/* helper symbol for elseif_statement_list */
SYM_REF2(elseif_statement_c, expression, statement_list)
SYM_REF4(case_statement_c, expression, case_element_list,
statement_list, unused)
/* helper symbol for case_statement */
SYM_LIST(case_element_list_c)
SYM_REF2(case_element_c, case_list, statement_list)
SYM_LIST(case_list_c)

/* B 3.2.4 Iteration Statements */
SYM_REF6(for_statement_c, control_variable, beg_expression,
end_expression, by_expression, statement_list, unused)
SYM_REF2(while_statement_c, expression, statement_list)
SYM_REF2(repeat_statement_c, statement_list, expression)
SYM_REF0(exit_statement_c)
```

ANEXO B

Alterações à sintaxe do IEC 61131-3 de modo a garantir que todas as variáveis sejam explicitamente inicializadas, e assim permitir que seja utilizada para a programação de sistema de elevada integridade. Os elementos retirados da sintaxe original encontram-se traçados, enquanto que os elementos acrescentados estão a negrito.

```
array_spec_init ::= array_specification ┼ ':=' array_initialization ┼

simple_spec_init ::= simple_specification ┼ ':=' constant ┼

subrange_spec_init ::= subrange_specification ┼ ':=' signed_integer ┼

enumerated_spec_init ::= enumerated_specification ┼ ':='
enumerated_value ┼

initialized_structure ::= structure_type_name ┼ ':='
structure_initialization ┼

fb_name_decl ::= fb_name_list ':' function_block_type_name ┼ ’:=’
structure_initialization ┼

single_byte_string_spec ::= 'STRING' ['[' integer ']'] ┼ ':='
single_byte_character_string ┼

double_byte_string_spec ::= 'WSTRING' ['[' integer ']'] ┼ ':='
double_byte_character_string ┼

global_var_decl ::= global_var_spec ':' ┼ ( located_var_spec_init |
function_block_type_name ’:=’ structure_initialization) ┼

var_spec ::= simple_specification
| subrange_specification | enumerated_specification
| array_specification | structure_type_name | 'STRING' ['[' integer ']']
```

':=' single_byte_character_string | 'WSTRING' ['[' integer ']'] **':='**
double_byte_character_string

prog_conf_element ::= ~~fb_task~~ | prog_cnxn

ANEXO C

Alterações à sintaxe do IEC 61131-3 de modo a garantir que não seja permitido que a execução de uma instância de uma função bloco declarada dentro de um programa seja atribuída de forma explícita a uma outra tarefa que não a que executa o próprio programa. Os elementos retirados da sintaxe original encontram-se traçados, enquanto que os elementos acrescentados estão a negrito.

```
program_configuration ::=
'PROGRAM' [RETAIN | NON_RETAIN]
  program_name ['WITH' task_name] ':' program_type_name
  ['(' prog_conf_elements ')']

prog_conf_elements ::= prog_conf_element {',' prog_conf_element}

prog_conf_element ::= ~~fb_task |~~ prog_cnxn

~~fb_task ::= fb_name 'WITH' task_name~~
```

Referências Bibliográficas

[1] Bradley M. Kuhn, "Software Freedom, the GNU Generation, and the GNU General Public License", Transrição de discurso dado no Porto em 12 de Outubro de 2001, (2001)

[2] General Motors Corp., "Manufacturing Automation Protocol Version 2.1", (1985)

[3] R.S. Matthews, K.H. Muralidhar, S. Sparks, "MAP 2.1 Conformance Testing Tools", , (March 1988)

[4] International Electrotechnical Comission, "61131-3 Programmable Controllers - Programming Languages, Ver. 2", (2003)

[5] Herman Bruyninckx, Peter Soetens, "Generic Real-Time Infrastructure For Signal Acquisition, Generation And Processing", Proceedings of the 4th Real-Time Linux Workshop, Boston, USA, (2002)

[6] D. Schleef, "Comedi: Linux Control and Measurement Device Interface", http://www.comedi.org/, disponivel em Setembro 2004

[7] Dan Falk, Ray Henry, John Sheahan, Don McLane, Will Shackleford,Tim Goldstein, Fred Proctor, Jon Elson, Henkka Palonen, "EMC Handbook", (2004)

[8] LinuxCNC, "HAL for Integrators", (2004)

[9] Ray Henry, Paul Corner, Steve Stallings, "EMC: a Free and Powerful PC-Based CNC Machinery Controller", http://www.linuxcnc.org, disponivel em Setembro 2004

[10] Herman Bruyninckx, "Open Robot Control Software: the OROCOS project", Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), (2001)

[11] Herman Bruyninckx, Peter Soetens, Bob Koninckx , "The Real-Time Motion Control Core of the Orocos Project", Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), (2003)
[12] Herman Bruyninckx, "Orocos: Open Robot Control Software, Open Realtime Control Services", http://www.orocos.org, disponivel em Setembro 2004
[13] Curt Wuollet, "MatPLC", http://mat.sf.net, disponivel em Setembro 2004
[14] Mário de Sousa, "MatPLC - the Truly Open Automation Controller", Proceedings of the 28th Annual Conference of IEEE Industrial Electronics Society (IECON), (2002)
[15] Dawson R. Engler, M. Frans Kaashoek, James O'Toole Jr., "Exokernel: An Operating System Architecture for Application-Level Resource Management", Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, (December 1995)
[16] D. Engler, Tese de Doutoramento "The Exokernel operating system architecture", Massachusetts Institute of Technology, (1999)
[17] Mario de Sousa, Jiri Baum, Andrey Romanenko, "MatPLC: Towards real-Time Performance", Proceedings of the 5th Real-Time Linux Workshop, (2003)
[18] Mario de Sousa, Adriano de Carvalho, "An IEC 61131-3 Compiler for the MatPLC", Proceedings of the 9th IEEE conference on Emerging Technologies and Factory Automation (ETFA), (2003)
[19] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, ISBN 0-201-63361-2, (1997)
[20] J. Kwon, A. Wellings and S. King, "Assessment of the Java Programming Language for Use in High Integrity Systems", ACM SIGPLAN Notices, (2003)
[21] Gottfried Egger, Andreas Fett, Peter Pepper, "Formal Specification of a Safe PLC Language and its Compiler", Proceedings of the 12th International Conference on Computer Safety Reliability and Security (SAFCOMP), (1994)
[22] Martin Ohman, Stefan Johansson., Karl-Erik Arzen, "Implementation Aspects of the PLC Standard IEC 1131-3", Proceedings of the IEEE Conference on Computer Aided Control Systems Design (CACSD), (April 1997)
[23] Stefan Johansson, Martin Ohman, "Prototype Implementation of the PLC Standard IEC 1131-3", Lund Institute of Technology, Sweden, (1995)
[24] Wolfgang A. Halang, Bernd J. Kramer, "Safety Assurance in Process Control", IEEE Software, pg 61-67, (January 1994)
[25] A. Mader, "A Classification of PLC Models and Applications", Proceedings

of the 5th Workshop on Discrete Event Systems (WODES), Gent, Belgium, (2000)

[26] O. Rossi, J. J. Lesage, J. M. Roussel, "Formal Validation of PLC Programs: A Survey", Proceedings of the European Control Conference (ECC), (1999)

[27] O. De Smet, S. Couffin, O. Rossi, G. Canet, J. J. Lesage, Ph. Schnoebelen, H. Papini, "Safe Programming of PLC Using Formal Verification Methods", Proceedings of the 4th International PLCOpen Conference on Industrial Control Programming (ICP), Utrecht, Netherlands, (October, 2000)

[28] G. Canet, S. Couffin, J. J. Lesage, A. Petit, Ph. Schnoebelen, "Towards the Automatic Verification of PLC Programs Written in Instruction List", Proceedings of the IEEE Conference on systems, Manufacturing and Cybernetics (SMC), Nashville, TN, USA, (Oct, 2000)

[29] K. L. McMillan, "Symbolic Model Checking", Kluwer Academic Publishers, ISBN 0792393805, (1993)

[30] M. J. Song, S. R. Koo, P. H. Seong, "Development of a Verification Method for Timed Function Blocks Using ESDT and SMV", Proceedings of the 8th IEEE International Symposium on High Assurance Systems Engineering (HASE), (2004)

[31] S. Bornot, R. Huuck, Y. Lakhnech, "Utilizing Static Analysis for Programmable Logic Controllers", Proceedings of the 4th Interntaional Conference on Automation of Mixed Processes: Hybrid Systems (APDM), Dortmund, Germany, (Setembro 18-20, 2000)

[32] R. Huuck, "Software Verification for Embedded Systems", Proceedings of the 8th IEEE International Conference on Methods and Models in Automation and Robotics (MMAR), Szczecin, Poland, (Setembro 2-5, 2002)

[33] R. Huuck, Tese de Doutoramento "Software Verification for Programmable Logic Controllers", Technischen Fakultat der Christian-Albrechts-Universitat zu Kiel, Deutschland, (2003)

[34] N. Bauer, R. Huuck, "Towards Automatic Verification of Embedded Control Software", Proceedings of the IEEE Asian Pacific Conference on Quality Software, (2001)

[35] R. Huuck, B. Lukoschus, N. Bauer, "A Model-Checking Approach to Safe PLCs", Proceedings od the IMACS Multiconference on Computational Engineering in Systems Applications (CESA), Lille, France, (2003)

[36] S. Bornot, R. Huuck, B. Lukoschus, "Verification of Sequential Function Charts Using SMV", Proceedings of the IEEE International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), Las Vegas, USA, (June 25-29, 2000)

[37] M. Bruggink, "Programming PLCs Using Sequential Function Chart", , (1999) (http://citeseer.ist.psu.edu/bruggink99programming.html)

[38] M. Heiner, T. Menzel, "A Petri Net Semantics for the PLC Language Instruction List", Proceedings of the IEE 4th Workshop on Discrete Event Systems (WODES), Cagliari, Italy, (1998)
[39] H. Willems, "Compact Timed Automata for PLC Programs", University of Nijmegen, The Netherlands, Computing Science Institute, (1999)
[40] W. Yi, K. G. Larsen, P. Pettersson, A. Skou, J. Bengtsson, G. Behrmann, A. David, T. Amnell, E. Fersman, L. Mokrushin, E. Fleury , "UPPAAL", http://www.docs.uu.se/docs/rtmv/uppaal/, disponivel em 10 Agosto 2004
[41] R. Susta, "Verification of PLC Programs", CTU (Prague) - Faculty of Electrotechnical Engineering, (2003)
[42] Arne Boralv, Herman Agren, "Formal Verification of Programmable Logic Controllers", Uppsala University, Sweden, (1995)
[43] B. Kramer, N. Volker, "A Highly Dependable Computing Architecture for Safety-Critical Control Applications", Real-Time Systems Journal - 13, pg 237-251, (1997)
[44] N. Volker, B. Kramer, "Modular Verification of Function Block Based Industrial Control Systems", Proceedings of Joint 24th IFAC/IFIP Workshop on Real-Time Programming and the 3rd International Workshop on Active and Real-Time Database Systems, Schloss Dagstuhl, Germany, (May 30 - June 2, 1999)
[45] W. Halang, B. Kramer, "Achieving High Integrity of Process Control Software by Graphical Design and Formal Verification", Software Engineering Journal, pg 53-64, (January 1992)
[46] W. Halang, "Automated Control Systems for the Safety Integrity Levels 3 and 4", Proceedings of the 9th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS), (October 1-3, 2003)
[47] W. Halang, S. Mostert, "Composing Dependable Real-Time Software of Function Blocks", Proceedings of the 6th International Workshop on Object-Oriented Real-Time Depdendable Systems (WORDS), (January 2001)
[48] International Electrotechnical Comission, "IEC 61499 Function Blocks, Part 1: Architecture (Publicly available Draft)", (2000)
[49] IDA Consortium, "IDA - Interface for Distributed Automation - White Paper, Rev. 1.0", (18 Abril 2001)
[50] IDA Consortium, "Interface for Distributed Automation - Arquitecture Description and Specification - Rev. 1.1", (Novembro 2002)
[51] Real-Time Innovations, "Real-Time Publish-Subscribe (RTPS) Wire Protocol Specification, Ver. 1.0", (Fevereiro 2002)
[52] Petr Smolik, "ORTE", http://www.ocera.org/download/components/WP7/ethdev-0.2.2.html, disponivel em 2004