View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Repositório da Universidade Nova de Lisboa

Universidade Nova de Lisboa

Faculdade de Ciências e Tecnologia

Departamento de Informática

Modelação e Integração em Sistemas Flexíveis de Produção

José António Barata de Oliveira

Lisboa, 1995

Universidade Nova de Lisboa

Faculdade de Ciências e Tecnologia

Departamento de Informática

Modelação e Integração em Sistemas Flexíveis de Produção

José António Barata de Oliveira

Dissertação apresentada para obtenção do Grau de Mestre em Engenharia Informática,
pela Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia

Lisboa, 1995

Sumário

Esta tese trata da modelação e integração em sistemas flexíveis de produção, com vista à construção de uma plataforma de suporte ao desenvolvimento de supervisores inteligentes. Faz-se uma apresentação sumária dos aspectos de modelação importantes para a construção de plataformas de integração, usando paradigmas de modelação com potencial para descreverem sistemas flexíveis de manufactura: programação orientada por objectos e "frames". Analisam-se as diferentes questões importantes relacionadas com a modelação estrutural e comportamental dos componentes que participam numa célula; propõem-se diversas taxonomias e uma metodologia para ligar os controladores locais à sua "imagem" (modelo).

De seguida, descreve-se um trabalho de concepção/implementação de dois sistemas flexíveis de manufactura, concebendo/desenvolvendo a plataforma integrada de um dos sistemas. Mostram-se os servidores utilizados para integrar os diversos controladores locais, recorrendo ao paradigma cliente-servidor e realçam-se as dificuldades de integração surgidas com as arquitecturas fechadas da maioria dos controladores locais. Construiu-se ainda um modelo do sistema, usando as técnicas apresentadas, suportando a ligação com os controladores locais.

Finalmente apresenta-se um método para a síntese de sistemas de supervisão, a partir de redes de Petri, descrevendo-se resultados experimentais resultantes da aplicação da metodologia a uma célula de montagem do sistema em estudo.

Abstract

The subject of this thesis is the modelling and integration of flexible manufacturing systems, in order to build a platform to support the development of intelligent supervisors. A discussion about the important aspects needed to build integrated platforms is made, using adequate modelling paradigms to describe flexible manufacturing systems: object oriented and frames. The structural and dynamic aspects of system's components are discussed; a taxonomy of components and a methodology to link local controllers to their "image" (model) is also presented.

The design and implementation work on two flexible manufacturing systems and the conception/development of the integrated platform for one of those systems is described. The servers used to integrate the several local controllers are shown, developed according to the client-server paradigm, stressing the local controllers integration difficulties due to their close architectures. Using the modelling techniques previously presented, a model of part of the system was built. This model includes/supports the link with the local controllers.

Finally a method to synthesize supervision systems from Petri nets is presented. Experimental results coming from the application of this method to an assembly cell of the mentioned system are included.

Índice de Matérias

SUMÁRIO 3

ABSTRACT 5

ÍNDICE DE MATÉRIAS 7

ÍNDICE DE FIGURAS 11

ÍNDICE DE QUADROS 13

1. INTRODUÇÃO 15

1.1. APRESENTAÇÃO GERAL 15
1.2. ORGANIZAÇÃO DA TESE 17

2. MODELAÇÃO DE INFORMAÇÃO 19

2.1. APRESENTAÇÃO 19
2.2. MODELAÇÃO ORIENTADA POR OBJECTOS 19
2.2.1. Introdução 19
2.2.2. Encapsulamento de Informação 21
2.2.2.1. Objectos 22
Atributos 22
Métodos 23
Identidade dos Objectos 24
Relações entre Objectos 25
Agregação/Composição 25
Utilização 25
2.2.2.2. Classes 26
Classes e Objectos 27
Relações entre Classes 29
Herança 29
Utilização 30
Instanciação 30
2.2.3. Reutilização de Software 31
2.2.3.1. Tipos de Dados Abstractos - ADTs 32
2.2.3.2. Tipos e Classes 33
2.2.3.3. Herança 34

2.2.3.3. Polimorfismo ... 37
2.3. "*FRAMES*" ... 39
2.3.1. Introdução ... 39
2.3.2. Noções Básicas ... 41
2.3.2.1. Aspectos Estruturais ... 41
"*Frame*" ... 41
Instâncias de "*Frames*" ... 41
Slots ... 42
Relações ... 44
2.3.2.2. Aspectos Comportamentais ... 45
Métodos ... 45
Demónios ... 47
2.3.3. Conclusão ... 48
2.4. REDES DE PETRI ... 49
2.4.1. Introdução ... 49
2.4.2. Noções Básicas ... 49
2.4.2.1. Exemplo 1 ... 52
2.4.2.2. Exemplo 2 ... 53
2.4.2.3. Exemplo 3 ... 54
2.4.3. Modelação Usando RdP ... 55
2.4.3.1. Autónomas ... 55
Abreviações ... 55
Rede de Petri Generalizada ... 55
Rede de Petri Colorida - RdPc ... 56
Rede de Petri com Predicados de Transição ... 58
Rede de Petri de Capacidade Finita ... 58
Extensões ... 59
Rede de Petri com Arco Inibidor ... 59
Rede de Petri Prioritárias ... 59
2.4.3.2. Não Autónomas ... 60
Redes de Petri Sincronizadas - RdPS ... 60
Redes de Petri Temporizadas - RdPT ... 61
Redes de Petri Temporizadas por Lugar ... 61
Redes de Petri Temporizadas por Transição ... 62
Redes de Petri Interpretada - RdPI ... 62
2.4.4. Análise das RdP ... 64
2.4.4.1. Análise Qualitativa ... 64
2.4.4.2. Análise Quantitativa ... 65

3. MODELAÇÃO CONCEPTUAL DE CÉLULAS ... 67

3.1. INTRODUÇÃO ... 67
3.2. FORMULAÇÃO DO PROBLEMA ... 68
3.3. CLASSES DE INFORMAÇÃO PRESENTES ... 75
3.3.1. Taxonomia de Componentes ... 76
3.3.2. Taxonomia de Papéis ... 79
3.3.3. Células ... 80
3.3.4. Materiais ... 80
3.3.5. Suportes ... 81
3.4. COMPORTAMENTO ESTÁTICO ... 82
3.4.1. Componentes ... 82

3.4.2. Papéis *90*
3.4.3. Células *95*
3.4.4. Materiais *99*
3.5. COMPORTAMENTO DINÂMICO 101

4. APLICAÇÃO A UM SISTEMA FLEXÍVEL DE MANUFACTURA 107

4.1. INTRODUÇÃO 107
4.1.1. Controladores Diferentes *107*
4.1.2. Dificuldades de Exprimir o Processo *108*
4.1.3. Necessidades de Sincronização *108*
4.1.4. Validação dos Resultados *109*
4.1.5. Conclusão *109*
4.2. SISTEMAS EM ESTUDO 110
4.2.1. Célula de Montagem *110*
4.2.2. Sistema NovaFlex *112*
4.2.2.1. Subsistema FMS 114
4.2.2.2. Subsistema FAS Multi-Robô 115
4.2.2.2.1. Célula de Montagem 1 115
4.2.2.2.2. Célula de Montagem 2 116
4.2.2.3. Subsistema de Transporte 116
4.2.2.4. Subsistema Armazém Automático 117
4.2.2.5. Subsistema Sensorial 118
4.2.2.6. Arquitectura de Controlo e Supervisão 118
4.3. MODELAÇÃO E INTEGRAÇÃO 119
4.3.1. Integração *120*
4.3.1.1. Steam 120
4.3.1.2. BOSCHSYS 123
O Controlador Local 123
A integração 123
4.3.1.3. GAA 125
4.3.1.4. Outros Subsistemas 129
4.3.2. Modelos *129*
4.4. REDE DE PETRI NA MODELAÇÃO DINÂMICA E SÍNTESE DO CONTROLO DE ALTO NÍVEL 134
4.4.1. Descrição global *134*
4.4.2. Síntese da RdP *135*
4.5. SUMÁRIO/EXPERIÊNCIA 139

5. CONCLUSÕES 141

6. APÊNDICE A - FOTOGRAFIAS DA NOVAFLEX E PRODUTO NOVACLOCK 143

7. APÊNDICE B - FOTOGRAFIA DA CÉLULA SONY E PÊNDULO DE CRANFIELD 145

8. AGRADECIMENTOS 147

9. BIBLIOGRAFIA 149

Índice de Figuras

FIGURA 2.1 - DESCRIÇÃO DO PROCESSO DE PROGRAMAÇÃO..20
FIGURA 2.2 - DESCRIÇÃO DE UM OBJECTO COMPOSTO - ROBÔ 25
FIGURA 2.3 - DESCRIÇÃO DE UM SISTEMA COM 2 TIPOS DE RELAÇÕES 26
FIGURA 2.4 - "SCHEMA-EVOLUTION\...35
FIGURA 2.5 - ROBÔ DEFINIDO POR ESPECIALIZAÇÃO..36
FIGURA 2.6 - ROBÔ POR COMPOSIÇÃO/AGREGAÇÃO..36
FIGURA 2.7 - EXEMPLOS DE MONOMORFISMO E POLIMORFISMO 37
FIGURA 2.9 - EXEMPLO DE REDE DE PETRI 50
FIGURA 2.10 - EXEMPLOS DE DISPARO DE RDPS...51
FIGURA 2.11 - CONTROLO DE TAPETE 52
FIGURA 2.12 - CONTROLO DE ARMAZÉM AUTOMÁTICO 53
FIGURA 2.13 - SISTEMA COM 2 AGVS 54
FIGURA 2.14 - EXEMPLO DE DISPARO PARA RDP GENERALIZADA 55
FIGURA 2.15- TRANSFORMAÇÃO DE RDP GENERALIZADA DA FIGURA 2.14 EM NORMAL 55
FIGURA 2.16 - EXEMPLO PARCIAL DE MONTAGEM DE UMA PEÇA 56
FIGURA 2.17 - EXEMPLO DE RDPC UTILIZADA COMO ATRIBUIÇÃO DE RECURSOS 57
FIGURA 2.18 - EXEMPLO DE RDP COM PREDICADOS NA TRANSIÇÃO - PTN 58
FIGURA 2.19- EXEMPLO DE RDP COM CAPACIDADE FINITA..59
FIGURA 2.20 - EXEMPLO DE RDP COM ARCO INIBIDOR..59
FIGURA 2.21 - EXEMPLO DE RDP COM ARCO INIBIDOR..59
FIGURA 2.22 - EXEMPLO DE RDP SINCRONIZADA 61
FIGURA 2.23 - EXEMPLO DE RDP TEMPORIZADA POR LUGAR..62
FIGURA 2.24 - EXEMPLO DE RDP INTERPRETADA 63
FIGURA 2.25 - EXEMPLO DE RDP E ÁRVORE DE ACESSIBILIDADE..65
FIGURA 3.1 - REPRESENTAÇÃO ABSTRACTA DE CÉLULA 68
FIGURA 3.2 - CONCATENAÇÃO POSSÍVEL DE CÉLULAS...69
FIGURA 3.3 - SISTEMA COM 2 CÉLULAS DE MONTAGEM LIGADAS POR UMA CÉLULA DE TRANSPORTE70
FIGURA 3.4 - ESTRUTURA DE CÉLULA ABSTRACTA 70
FIGURA 3.5 - EXEMPLO DE CÉLULA ABSTRACTA COM COMPONENTES REPRESENTANDO DIVERSOS PAPÉIS...72
FIGURA 3.6 - SISTEMA CONSTITUÍDO POR 5 CÉLULAS 72
FIGURA 3.7 - PAPÉIS DESEMPENHADOS PELOS COMPONENTES 73
FIGURA 3.8 - SISTEMA COMO UMA COMPOSIÇÃO DE CÉLULAS 74
FIGURA 3.9 - UM PRODUTO COMPLEXO: NOVACLOCK, CONSTITUÍDO POR PRODUTOS SIMPLES 74
FIGURA 3.10 - TAXONOMIA DE COMPONENTES - NÍVEL COMPONENTES...76
FIGURA 3.11 - TAXONOMIA DE COMPONENTES - NÍVEL TRANSPORTADORES 76

FIGURA 3.12 - TAXONOMIA DE COMPONENTES - NÍVEL FORNECIMENTO 76
FIGURA 3.13 - TAXONOMIA DE COMPONENTES - NÍVEL TRANSFORMAÇÃO 77
FIGURA 3.14 - TAXONOMIA DE COMPONENTES - NÍVEL FERRAMENTAS 77
FIGURA 3.15 - TAXONOMIA DE COMPONENTES - NÍVEL SUPORTE 77
FIGURA 3.16 - TAXONOMIA DE COMPONENTES - NÍVEL COMANDO 78
FIGURA 3.17 - TAXONOMIA DE COMPONENTES - NÍVEL SENSOR 78
FIGURA 3.18 - TAXONOMIA DE COMPONENTES - NÍVEL ARMAZENAMENTO 79
FIGURA 3.19 - TAXONOMIA DE PAPÉIS - NÍVEL PAPÉIS 79
FIGURA 3.20 - TAXONOMIA DE PAPÉIS - NÍVEL ENTRADA 79
FIGURA 3.21 - TAXONOMIA DE PAPÉIS - NÍVEL AGENTE 80
FIGURA 3.22 - TAXONOMIA DE PAPÉIS - NÍVEL SAÍDA 80
FIGURA 3.23 - TAXONOMIA DE CÉLULAS 81
FIGURA 3.24 - TAXONOMIA DE MATERIAIS 81
FIGURA 3.25 - TAXONOMIA DE SUPORTES 82
FIGURA 3.26 - MODELO DE UM AGENTE ROBÔ A DESEMPENHAR O PAPEL DE MONTADOR 90
FIGURA 3.27 - IMPLEMENTAÇÃO EM POO TRADICIONAL 91
FIGURA 3.28 - EXEMPLO DE RDP 96
FIGURA 3.29 - RELAÇÕES EXISTENTES NUMA CÉLULA 97
FIGURA 3.30 - RELAÇÕES EXISTENTES ENTRE PRODUTOS SIMPLES E COMPOSTOS 99
FIGURA 3.31 - MENSAGENS ENTRE OBJECTOS QUANDO DA OPERAÇÃO "MONTA\ 101
FIGURA 3.32 - DOIS MUNDOS DISTINTOS: CONTROLADOR DO ROBÔ E CONTROLADOR DE ALTO NÍVEL 102
FIGURA 3.33 - LIGAÇÃO DOS MODELOS AO CONTROLADOR LOCAL, ATRAVÉS DE RPC 104
FIGURA 4.1 - ARQUITECTURA HARDWARE DA CÉLULA DO SONY 111
FIGURA 4.2 - INFRAESTRUTURA DE CONTROLO DA CÉLULA DO SONY 112
FIGURA 4.4 - CÉLULA FMS 115
FIGURA 4.5 - CÉLULA FAS MULTI-ROBÔ 115
FIGURA 4.6 - PALETE COM MDT E SLS 117
FIGURA 4.7 - ARQUITECTURA DE CONTROLO DA NOVAFLEX 119
FIGURA 4.8 - ESQUEMA DAS SECÇÕES DO SUBSISTEMA DE TRANSPORTE 121
FIGURA 4.9 - ARQUITECTURA HARDWARE DO SUBSISTEMA DE TRANSPORTE 122
FIGURA 4.10 - ARQUITECTURA DE SOFTWARE DO SUBSISTEMA DE TRANSPORTE 122
FIGURA 4.11 - ARQUITECTURA DE HARDWARE DA INTEGRAÇÃO DO CONTROLADOR BOSCH RS82 . 124
FIGURA 4.12 - ARQUITECTURA DE SOFTWARE DO BOSCHSYS 125
FIGURA 4.13 - FORMATO DE MEMÓRIA DO MDT 126
FIGURA 4.14 - ARQUITECTURA HARDWARE DO ARMAZÉM AUTOMÁTICO 127
FIGURA 4.15 - ARQUITECTURA DE SOFTWARE DO ARMAZÉM AUTOMÁTICO 128
FIGURA 4.16 - ARQUITECTURA GERAL PROPOSTA 135
FIGURA 4.17 - RDP PARA DESCREVER FUNCIONAMENTO DO ALIMENTADOR GRAVÍTICO NA CÉLULA SONY 136
FIGURA 4.18 - RDP PARA DESCREVER FUNCIONAMENTO DE UM TAPETE 136
FIGURA 4.19 - RDP PARA DESCREVER FUNCIONAMENTO DE UM TAPETE - PTN 137
FIGURA 4.20 - RDP PARA DESCREVER CÉLULA 1 DO FAS 138
FIGURA A.1 - FOTOGRAFIA DA NOVAFLEX - SUBSISTEMA MULTI-ROBÔ FAS 143
FIGURA A.2 - FOTOGRAFIA DA NOVAFLEX - SUBSISTEMA FMS 144
FIGURA A.3 - PRODUTO NOVACLOCK 144
FIGURA B.1 - FOTOGRAFIA DA CÉLULA SONY 145
FIGURA B.2 - FOTOGRAFIA DO PÊNDULO DE CRANFIELD 146

Índice de Quadros

TABELA 3.1 - EXEMPLOS DE COMPONENTES E INDICAÇÃO DAS FUNÇÕES QUE PODEM DESEMPENHAR71

1. Introdução

1.1. Apresentação Geral

Como consequência da evolução dos mercados e globalização da economia, tem-se assistido, nos últimos anos, a um grande incremento do interesse pelos sistemas flexíveis de manufactura.

O reconhecimento deste interesse não está baseado numa análise meramente académica, mas sim na análise dos requisitos industriais actuais . Os sistemas flexíveis não são necessários apenas porque a investigação na área dos sistemas de produção o determinou, mas porque as necessidades "no terreno" (meio industrial e mercados) os impuseram.

Ao invés de um número limitado de produtos normalizados, caminha-se para uma situação de grande quantidade de variantes, levando a que a flexibilidade passe a ser um termo não exclusivo dos ambientes de investigação, para passar a ser, também, do industrial.

Este fenómeno não acontece apenas nas indústrias dos países mais desenvolvidos. Em Portugal que, como se sabe, possui um tecido industrial com um défice de desenvolvimento bastante grande, graças, não só à deficiente tecnologia mas, acima de tudo, a uma deficiente mentalidade empresarial (cultura de empresa), começam a surgir um número razoável de empresários com a preocupação de flexibilizar a sua estrutura produtiva, por forma a torná-la competitiva. Esta constatação não é baseada apenas em impressões pessoais, antes resultando de diversos contactos com alguns industriais do sul e norte do país.

A título de exemplo, refere-se o caso de uma indústria portuguesa de produção de componentes de madeira que, para continuar competitiva no seu mercado, teve de passar de 16 variedades de produtos para 164.

O desenvolvimento de sistemas flexíveis obriga a alguns requisitos importantes, quer do ponto de vista dos equipamentos, quer do ponto de vista de software. Como exemplo de restrições ao nível do hardware tem-se: uso de aparelhos de operações múltiplas, robôs com múltiplas ferramentas, fixadores e fornecedores de materiais flexíveis , ambiente sensorial rico, infra-estrutura de comunicação avançada, etc. A nível de software tem-se: aplicações distribuídas, linguagens com modelos ricos (programação orientada por objectos - POO, "frames", ...), paradigma cliente-servidor, protocolos de comunicação de alto nível (MMS, MAP-TOP), controladores locais desenvolvidos no sentido de poderem ser integrados em ambientes distribuídos, etc.

A possibilidade de instalação de dois sistemas flexíveis de montagem e manufactura pelo Grupo de Robótica e CIM, foi, se assim se pode dizer, o motor de arranque para todo este trabalho.

O primeiro sistema a ser montado foi uma célula flexível de montagem construída em torno de um robô do tipo SCARA, e destinando-se a suportar trabalhos de investigação (projectos B-LEARN, SARPIC, CIM-CASE) nas áreas de supervisão, monitorização, diagnóstico e recuperação de erros.

O segundo sistema é bastante mais complexo e foi construído no âmbito de um projecto PEDIP de criação de infra-estruturas, tendo sido instalado no Centro de Robótica Inteligente do UNINOVA. Consiste numa infra-estrutura piloto com vários subsistemas: armazém automático, célula de montagem com robô antropomórfico (6 eixos), célula de montagem com robô SCARA (4 eixos), célula de maquinação, transportadores e entrada e saída de materiais [1]. Esta unidade piloto destina-se a suportar os trabalhos de investigação em várias áreas ligadas a sistemas integrados de manufactura, a servir de unidade de demonstração e a ser um suporte para acções de formação.

A instalação de ambos os sistemas envolveu a resolução de um sem número de problemas ao longo das diversas fases de instalação (projecto, montagem e exploração). De todos esses problemas importa referir os relacionados com a definição da arquitectura de controlo e supervisão. Em ambos os casos, a diversidade de controladores e o tipo de arquitectura (fechada) dos controladores locais, foi sempre um problema para a definição de arquitecturas flexíveis.

A necessidade de arquitecturas flexíveis para o controlo dos sistemas de manufactura: supervisores de controlo, agentes de escalonamento dinâmico, etc, resultante dos requisitos previamente referidos, não se compadece com o tipo de arquitecturas fechadas disponibilizadas pela maioria dos controladores locais. As palavras-chave na construção de arquitecturas flexíveis são: integração [2,3] e modelação [4-6].

Desta forma, surgiu a necessidade de criar, para ambos os sistemas, uma infra-estrutura de integração que englobasse todos os controladores existentes (legados). Nesta infra-estrutura consideram-se os aspectos directamente relacionados com a integração dos controladores locais e os seus aspectos de modelação .

O trabalho relacionado com a integração consistiu no desenvolvimento de uma camada de software, baseada no paradigma servidor-cliente, para cada controlador local, que faz a ligação entre os controladores locais e o mundo dos modelos. O acesso a estes serviços, do lado dos modelos é realizado sempre da mesma forma: pedido a um servidor, enquanto que o acesso aos controladores locais já é dependente da sua arquitectura. A filosofia de integração consiste, então, em desenvolver, para cada controlador local que se pretenda inserir no sistema, um servidor.

O trabalho de modelação desenvolvido está relacionado com a preocupação em criar um conjunto de modelos que torne o sistema/componente físico que se está a modelar (células, robôs, tapetes, sistema de transporte, ...) mais "tratável" no ambiente informático. Na verdade, quanto maior for o nível de abstracção dos modelos criados menor será a "desadaptação semântica" entre as aplicações de alto nível e os controladores locais. Os modelos criados serão utilizados para se criarem supervisores inteligentes. A utilização de modelos com nível de abstracção elevado são, pode-se dizer, muito importantes, para a criação deste tipo de clientes, pelo facto de reduzirem a "desadaptação semântica" entre os controladores locais e o controlo de alto nível.

Para haver modelação é necessário escolher, entre os vários paradigmas existentes, aquele ou aqueles que melhor se adequam aos objectivos. O facto de haver já uma tradição no Grupo de Sistemas Robóticos e CIM na utilização de sistemas com Frames e uma convicção do autor sobre a sua adequabilidade para modelar sistemas/componentes, optou-se por apresentar uma proposta usando este modelo. Por outro lado, o facto do paradigma da programação orientada por objectos, estar a ter, cada vez mais, um grande incremento e novamente a convicção do autor sobre a sua adequabilidade, optou-se também por se apresentarem os modelos descritos neste paradigma.

O conjunto de modelos apresentados pretendem abranger as diferentes partes envolvidas num sistema de manufactura: componentes, células, produtos, quer do ponto de vista estrutural das suas propriedades, quer do ponto de vista dinâmico.

Apesar da importância dos aspectos estruturais dos modelos, a verdade é que o grande factor a realçar são os aspectos dinâmicos, já que são eles que governam, o modo como se estabelece a ligação entre os modelos e os controladores locais.

1.2. Organização da Tese

A tese foi organizada em cinco capítulos: Introdução, Modelação de Informação, Modelação Conceptual de Células, Aplicação a um Sistema Flexível de Manufactura e Conclusões.

No segundo capítulo faz-se uma descrição teórica, sem grandes preocupações formais, dos conceitos fundamentais para a modelação de sistemas flexíveis de manufactura. Apresentam-se três paradigmas: Objectos, "Frames" e Redes de Petri. Para cada um dos paradigmas faz-se uma descrição de forma a introduzir o leitor nos seus conceitos fundamentais.

O terceiro capítulo discute os conceitos envolvidos na modelação conceptual de sistemas flexíveis de manufactura. Faz-se uma abordagem aos conceitos estáticos e dinâmicos; nos conceitos estáticos equacionam-se os aspectos estruturais e taxonómicos dos componentes participantes; nos dinâmicos, equacionam-se os aspectos relacionados com o seu comportamento. Directamente relacionado com os aspectos dinâmicos está a ligação entre o(s) modelo(s) e o(s) controlador(es) local(ais). Finalmente, será apresentada uma proposta de modelação, envolvendo sistemas, células, componentes e materiais, que utiliza os dois modelos apresentados no capítulo anterior: objectos e frames.

O quarto capítulo discute com algum detalhe a implementação realizada no âmbito desta tese. O capítulo pretende descrever uma infra-estrutura de execução global que possa ser acedida por aplicações de alto nível (Supervisor). Começa-se por descrever as infra-estruturas criadas, tendo o autor participado no seu projecto e implementação, para de seguida se apresentarem algumas arquitecturas de integração de controladores locais existentes nos referidos sistemas. Depois, descreve-se a modelação de um subsistema da unidade piloto instalada no Centro de Robótica Inteligente - NovaFlex, que permite validar a proposta dos modelos apresentados no capítulo três e, ao mesmo tempo, validar também as arquitecturas de integração, previamente desenvolvidas. Termina-se o capítulo apresentando uma proposta de como se pode utilizar a infra-estrutura de execução, entretanto criada, a partir de uma rede de Petri. No quinto capítulo apresentam-se as conclusões sobre o trabalho realizado e apontam-se algumas ideias sobre trabalhos futuros na área

2. Modelação de Informação

2.1. Apresentação

Neste capítulo vão-se introduzir alguns conceitos importantes para a modelação de sistemas flexíveis de manufactura - SFM.

Não sendo possível, por não caber no âmbito da tese, fazer um estudo aprofundado sobre os diferentes processos de modelação que se podem aplicar nos SFM, optou-se por escolher aqueles que, na opinião do autor, são os mais importantes para a concepção de modelos informáticos dos diferentes sistemas/componentes dos SFM.

O capítulo deve ser lido como uma descrição, sumária, das técnicas de modelação que irão ser utilizadas nos capítulos seguintes da tese, durante a apresentação dos aspectos mais directamente relacionados com a implementação. Pretende-se assim fornecer ao leitor um conjunto de conceitos base que lhe possibilitarão uma melhor interpretação dos capítulos 3 e 4.

Assim vão-se apresentar três tipos de modelos: **modelo orientado por objectos**, **modelo de "frames"** e **redes de Petri**.

O modelo orientado por objectos é, para alguns investigadores, a panaceia que vem resolver todos os males da engenharia de software. Naturalmente que esta visão é demasiadamente optimista e, como seria de esperar, existem algumas limitações no modelo. De qualquer forma é bastante adequado para modelar SFM e uma proposta interessante para a produção de software de sistemas grandes e complexos.

O modelo por "frames", resultado de investigações na área da Inteligência Artificial, tem bastantes semelhanças com o modelo anterior, mas possui algumas características que o tornam bastante adequado e flexível para modelar SFM. Apesar de não ter a mesma divulgação que o modelo anterior, pode ser uma proposta interessante para a construção de protótipos.

As redes de Petri são adequadas para a representação dos aspectos dinâmicos dos sistemas. Dada a natureza intrinsecamente dinâmica dos SFM, a sua importância torna-se óbvia. Não se pode propriamente falar de um modelo rede de Petri. Os dois modelos anteriores são uma forma de estruturar os dados (modelos de dados), mas as redes distinguem-se neste aspecto, já que não estruturam informação, apenas descrevem comportamentos.

2.2. Modelação Orientada por Objectos

2.2.1. Introdução

A programação orientada por objectos é um dos paradigmas fundamentais na área da produção de Software nos anos 90. Os conceitos fundamentais da Engenharia de Software que se podem associar a este paradigma são o encapsulamento de informação e a reutilização de código.

Será talvez mais correcto começar por referir modelo de dados por objectos em vez de programação orientada por objectos que é uma definição mais restritiva. Quando se aborda o tema seguindo a definição mais abrangente de modelo, está-se a considerar as várias fases do processo de produção de software, que vai desde a análise e desenho à execução do programa.

Esta tese não pretende, de maneira nenhuma, analisar todas as fases da produção de software, antes pretendendo fazer uma descrição dos conceitos fundamentais em que está suportada a modelação de dados por objectos. Esta descrição será feita numa perspectiva de modelação de informação, circunscrita a uma área específica - **manufactura**.

De seguida apresentam-se alguns conceitos que são importantes para a percepção da necessidade do aparecimento deste paradigma e da sua relação com as linguagens de programação [7]:

- **Sistema Referente** - Corresponde ao fenómeno que irá ser descrito, que é sempre uma parte do mundo real. Quando em análise, é visto como um todo, estando durante este período "separado" do resto do mundo de forma a garantir a delimitação perfeita do fenómeno em análise. De notar que um sistema referente pode ser constituído por um conjunto de fenómenos. A montagem de um produto com um robô é um exemplo de um Sistema Referente.
- **Modelo** - Consiste na representação do sistema referente utilizando os mecanismos de abstracção do sistema que suporta o modelo. Engloba um conjunto de objectos com ciclos de vida dependentes do fenómeno que virtualizam no sistema referente. Estes objectos podem ter existência transitória ou persistente, consoante se mantêm ou não para além da execução do sistema para que foram criados.
- **Processo de Programação** - Construção de um modelo a partir de um sistema referente.

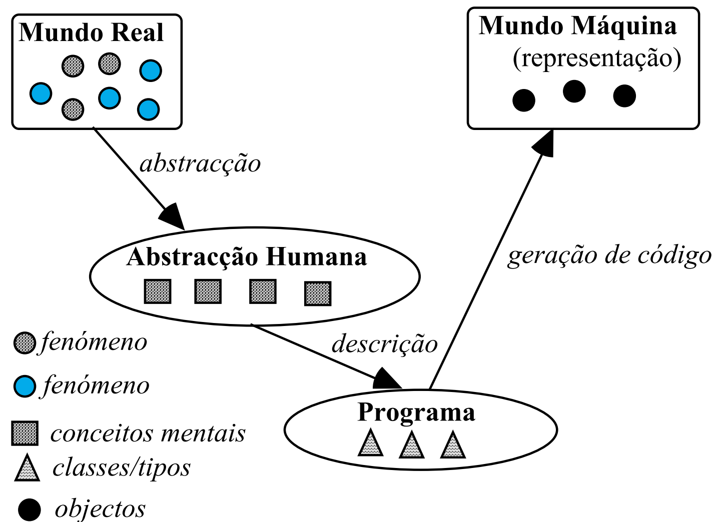


Figura 2.1 - Descrição do Processo de Programação

Na figura 2.1 observam-se os vários intervenientes no processo de programação. Colocado perante um determinado fenómeno que se pretende modelar através de um computador, o humano (programador/analista) define quais as fronteiras do fenómeno, bem como o conjunto de outros possíveis fenómenos que estejam relacionados com o processo.

Após a definição perfeita do fenómeno, o programador realiza um conjunto de operações mentais[1], em que a

[1] Considera-se este processo como sendo um processo cognitivo, em que o observador obtém uma percepção do problema.

mais importante será talvez a abstracção, no sentido de formar um conjunto de conceitos mentais que sejam a representação essencial do fenómeno a observar. A abstracção realizada sobre o fenómeno, leva a separar o essencial do fútil, para a percepção do problema. As sub funções mais importantes da abstracção são [7]: **classificação**, **composição** e **generalização**. Espera-se que, qualquer paradigma que pretenda compreender as várias fases da programação e fornecer mecanismos de abstracção fundamentais, como é o caso da programação orientada por objectos, suporte estes conceitos. Ir-se-á observar, com efeito, que as sub funções descritas fazem parte dos mecanismos fundamentais que caracterizam a programação orientada por objectos.

Nesta altura do problema ainda se está longe da representação noutro sistema (mundo), ainda se estão a utilizar conceitos ao nível de abstracção do humano. Existe a necessidade de transformar os conceitos abstractos humanos nos conceitos abstractos conhecidos do novo sistema. Este é o ponto fundamental do problema de programação, uma vez que será nesta altura que poderão surgir as desadaptações de conceitos.

A linguagem de modelação irá funcionar como a ferramenta que o humano tem para descrever os seus conceitos abstractos e será também a partir desta linguagem que serão geradas novas interpretações (código executável, ordens de produtos, código CNC, etc). Diz-se então, que a linguagem de modelação faz a ligação entre o mundo dos humanos e os restantes mundos. Quanto maiores forem as capacidades de abstracção da linguagem de modelação[2] menor irá ser a dificuldade do programador em exprimir o fenómeno que pretende modelar.

Pretende-se, assim, um paradigma de programação que tenha um nível semântico elevado em que os mecanismos de abstracção estejam próximos do humano, garantindo-lhe uma ferramenta de descrição, o mais confortável possível[3].

O paradigma da programação orientada por objectos afigura-se como capaz de responder aos requisitos apresentados e irá, assim, ser objecto de análise nesta tese.

O capítulo consistirá na apresentação dos conceitos fundamentais associados ao paradigma da modelação orientada por objectos, os quais serão agrupados em torno de 2 grandes temas da Engenharia de Software, Encapsulamento de Informação - ponto 2.2.2 - e Reutilização de Software - ponto 2.2.3.

2.2.2. Encapsulamento de Informação

O encapsulamento de informação é uma técnica de estruturação em que o sistema se constitui através de uma colecção de módulos acessíveis por uma *interface* muito bem definida [8].

Não se pode dizer que o encapsulamento de informação seja uma técnica que tenha surgido com a programação orientada por objectos. Desde o início, os implementadores e desenhadores de linguagens de programação procuraram integrá-las com mecanismos que suportassem esta técnica. Assim, vão-se encontrar, ao longo da evolução das linguagens de programação, mecanismos cada vez mais sofisticados de encapsular a informação.

Uma linguagem suporta encapsulamento de informação desde que forneça ao programador mecanismos que lhe permitam agrupar e estruturar a informação, segundo regras que permitem uma fácil análise e compreensão do problema. Os mecanismos de encapsulamento **escondem** do programador os detalhes de implementação.

Como exemplos de mecanismos de encapsulamento que se podem encontrar nas linguagens tradicionais têm-se: records (registo), vectores, tipos de dados, módulos, etc. Os registos, por exemplo, permitem que conceitos físicos sejam modelados e agrupados (encapsulados) numa entidade a que se atribui um nome.

---

[2] Maior capacidade de abstracção significa, neste contexto, uma maior capacidade de expressão de conceitos de nível abstracto elevado, como acontece, por exemplo, com algumas linguagens de programação ao suportarem a noção de tipo de dados abstracto - ADTs.

[3] Imagine-se como seria penoso descrever os fenómenos em linguagens cuja abstracção se situe apenas ao nível da existência do tipo Inteiro, Caracter e Real.

Novos mecanismos de encapsulamento foram criados e introduzidos nas linguagens de programação mais recentes, dos quais se realçam os **objectos** e as **classes**.

2.2.2.1. Objectos

O objecto representa a essência deste paradigma. Os objectos são as entidades existentes no modelo, virtualizam o sistema referente (mundo real) e são um dos mecanismos mais poderosos para encapsulamento de informação.

Os objectos caracterizam-se não só por modelarem a componente estrutural, mas também a componente funcional (comportamental) das entidades do mundo real. O estado de um objecto é representado pelo seu conjunto de propriedades estáticas mais o conjunto dos valores presentes nessas propriedades. Diz-se de um objecto que pode apresentar-se num determinado **estado**, ter um dado **comportamento** e ter uma **identidade** única. Evidentemente que existe uma relação entre o estado e a funcionalidade. Com efeito, determinado estado resulta da acção de uma dada funcionalidade. O modo como o estado do objecto evolui depende directamente da sua funcionalidade.

Como exemplo de objectos numa unidade produtiva, podem-se considerar os produtos, as paletes, os robôs, as máquinas ferramentas, etc.

O objecto robô, por exemplo, pode apresentar-se num estado caracterizado por estar em determinada posição e com determinado tipo de garra. Pode ser caracterizado pela sua funcionalidade, por exemplo alteração da velocidade, da aceleração, da sua posição, etc. Finalmente, durante a execução, este objecto é unicamente identificado. Os possíveis estados pelos quais o objecto robô passa dependem da sua lista de operações (funcionalidade) e dos valores dos atributos.

A possibilidade de exprimir o comportamento das entidades representa um grande avanço sobre o modelo meramente estático, representado pelo "velho" *record*. As entidades físicas passam a ser modeladas de uma forma mais completa, não amputadas da sua componente fundamental, como é o caso do comportamento.

Não se julgue no entanto que nos paradigmas anteriores não era possível modelar o comportamento das entidades físicas. Obviamente que esta possibilidade existia[4], mas não de uma forma clara e limpa. Privilegiava-se a funcionalidade em desfavor das estruturas que eram tratadas como meros auxiliares da funcionalidade, aparecendo por isso completamente diluídas nos procedimentos/funções que lhe davam corpo. Nestes paradigmas, não eram fornecidos os mecanismos de programação necessários para que o programador fosse conduzido a englobar, numa única entidade, o modelo estrutural e o modelo comportamental.

Da análise anterior infere-se que, na perspectiva clássica, privilegiava-se a descrição funcional do problema, adaptando-se as estruturas de dados a esta solução. Na perspectiva mais recente procuram-se detectar quais as entidades presentes, associando-lhes as suas características estáticas e funcionais mais relevantes. A utilização de um conceito matemático poderoso como é o tipo de dados abstractos - ADTs, conduz ao aparecimento de um melhor mecanismo de abstracção.

Atributos

Os atributos caracterizam as propriedades estáticas do objecto. De um ponto de vista de modelação têm a mesma função que os campos de um registo.

Os atributos podem conter valores simples ou outros objectos. O universo de valores que um determinado atributo pode conter dependerá directamente do facto da linguagem ser ou não "fortemente tipificada".

[4] Sem esta possibilidade seria impossível realizar qualquer programa.

No caso de uma linguagem não tipificada um atributo pode conter qualquer valor, que tanto poderá representar uma quantidade como a identificação de um objecto. Neste tipo de linguagens não é necessário declarar o universo a que pertence o atributo.

Nas linguagens tipificadas os atributos apenas podem conter valores pertencentes ao domínio do conjunto com que foram declarados. A noção de tipo abrange conceitos tão complexos como as classes, permitindo que os atributos possam conter valores que são eles próprios objectos.

Na descrição do objecto robô podemos encontrar os seguintes atributos e os seus respectivos domínios:

```
OBJECT robô
   ...
   aceleração: 1..10
   velocidade: 1..100
   garra: OBJECT garra
   ...
END robô
```

Ao atributo garra apenas poderá ser atribuído um valor que representa um outro objecto[5]. Os atributos poderão então ser utilizados para formar objectos compostos.

Uma questão premente que surge associada aos atributos relaciona-se com o seu acesso. No clássico registo, qualquer variável que se declare como sendo do tipo registo pode realizar ou ser usada na operação de afectação[6]. Num objecto, a manipulação dos atributos apenas se pode fazer quando se declara previamente essa intenção, através dos métodos. Os atributos são entidades encapsuladas no objecto; a forma visível do objecto encontra-se representada através dos seus métodos - funcionalidade. Não existe forma de um cliente aceder à representação interna do objecto[7]. Os atributos apenas podem ser acedidos nas funções (acções) declaradas no interior do próprio objecto.

Por cada atributo que represente uma propriedade importante do objecto terão de haver pelo menos 2 acções: modificação e leitura. Estas acções serão definidas através dos métodos, descritos no ponto seguinte.

Métodos

Os métodos representam a funcionalidade do objecto e são uma característica importante de um objecto. É através dos métodos que um objecto declara ao exterior qual o seu comportamento e é através deles que o estado do objecto é alterado.

Os métodos são operações que se podem realizar internamente ao objecto, mas podem existir operações que não são métodos[8]. Esta última situação acontece essencialmente por razões de eficiência, nomeadamente quando uma mesma operação necessita de ser partilhada por várias classes de objectos. De qualquer forma, convém sempre realçar que esta possibilidade traz associados alguns efeitos laterais perniciosos do ponto de vista da integridade do objecto, daí que os mais puristas defensores do paradigma desaconselhem a sua implementação nas linguagens de programação.

Os métodos e as operações exteriores não são a única forma de definir funcionalidade na POO. Internamente ao objecto podem ser declaradas funções ou procedimentos que estão completamente escondidas do mundo exterior do objecto, podendo ser utilizadas pelos métodos, como funcionalidade auxiliar.

---

[5] Iremos ver que esta é uma forma de formar objectos compostos e também uma forma de os objectos se relacionarem entre si.

[6] Isto acontece pelo facto de no tipo registo estar definida a operação de afectação, provocando que qualquer variável declarada neste domínio herde estas operações.

[7] Isto não é totalmente verdade representando apenas uma posição pessoal face ao problema. Com efeito existem algumas linguagens em que isto é possível.

[8] Por não estarem definidas internamente ao objecto nem fazerem parte do seu protocolo.

A distinção entre as funções e procedimentos que os objectos exteriores podem aceder das que não podem define-se no **protocolo** do objecto. Os métodos fazem, portanto, parte do protocolo do objecto.

As operações de modificação dos atributos - métodos transformadores, são implementadas através de funções ou procedimentos, enquanto que as operações de leitura de atributos - métodos acessores, são essencialmente implementadas à custa de funções. Os argumentos e o resultado destas operações podem ser, tal como no caso dos atributos, tipificados ou não, dependendo da linguagem de programação.

Apesar dos métodos transformadores e acessores serem os mais comuns[9], não são os únicos existentes. A necessidade de funcionalidade não se faz sentir apenas ao nível da consulta e actualização dos atributos, mas também ao nível de processamento da informação aí presente. Seja o seguinte exemplo:

```
OBJECT robô
   ...
   aceleração: 1..10
   velocidade: 1..100
   posição: ARRAY[1..3] OF REAL
   teta1, teta2, teta3, teta4: REAL
   garra: OBJECT garra
   method get_posição()
   ...
END robô
```

O método *get_posição* tanto pode ser acessor como, baseado na informação presente em *teta1*, *teta2*, *teta3* e *teta4*, calcular o valor da posição.

Qualquer análise à POO fica incompleta se não se referir que este paradigma é também conhecido como sendo um paradigma de troca de mensagens. Segundo esta perspectiva, a comunidade de objectos que interactuam no modelo do mundo constituído à custa deste paradigma faz-se através do envio de mensagens entre os objectos. Um objecto apenas pode enviar uma mensagem para outro desde que ele conheça essa mensagem. As mensagens a que um objecto é capaz de responder encontram-se declaradas no protocolo.

Segundo [9] um objecto é o encapsulamento de um conjunto de operações ou métodos que podem ser invocadas externamente e de um estado que representa o efeito dos métodos.

Identidade dos Objectos

Segundo [10] a identidade de um objecto é aquela propriedade que o torna distinguível de todos os outros.

A identificação de um objecto não pode ser feita através de uma chave baseada num atributo ou conjunto de atributos, uma vez que qualquer alteração do valor desse atributo provocava uma alteração na identificação do objecto. Esta alteração teria efeitos bastante perniciosos dado que o objecto em questão poderia ser membro de uma estrutura complexa que envolvesse vários outros objectos. A alteração da identidade obrigaria a ter de alterar todos os objectos com quem tivesse relações.

Não cabe nesta tese desenvolver considerações sobre a forma como é implementada a identificação dos objectos, apenas interessa referir o facto de que ela é única e que se manterá por toda a vida do objecto.

Associada a este tema aparece a questão da igualdade que pode ser tratada de 2 formas diferentes. Interessa muitas vezes não só detectar se 2 variáveis definem o mesmo objecto mas também se 2 variáveis nomeiam objectos diferentes mas com estados iguais. Algumas linguagens como o SMALLTALK e o C++ permitem a existência destes dois tipos de igualdade.

Dois objectos são diferentes desde que possuam identificadores diferentes e são estruturalmente idênticos desde que tenham o mesmo estado, isto é, para além de terem os mesmos atributos e métodos têm de ter valores idênticos nos atributos.

---

[9] Daí que algumas linguagens os implementem automaticamente.

A identidade é também relevante no ponto seguinte que diz respeito às relações entre objectos.

Relações entre Objectos

A apresentação deste ponto será inicializada com um aviso no sentido de não se confundir relações entre objectos com relações entre classes (hierarquia de tipos, etc) que será objecto de análise posterior.

Os tipos de dados abstractos - ADTs são um meio poderoso de fazer uma abstracção de dados, mas podem ser insuficientes, nomeadamente quando se pretendem organizar os objectos.

A relação entre objectos é extremamente importante na solução de um dado problema. Os objectos necessitam de se relacionar por 2 razões: (1) de ordem operativa e (2) de análise. Estas 2 diferenças conduzem à existência de 2 tipos de relações: Utilização e Agregação/Composição, respectivamente.

Agregação/Composição

A relação de Agregação/Composição está intimamente ligada ao modo como os humanos estruturam o meio que os rodeia. Este mecanismo é utilizado em quase todas as nossas análises. É uma forma de agrupar a informação. Um conceito como, por exemplo, um automóvel, não aparece como sendo um conjunto de entidades soltas constituídas pelo motor, rodas, portas, etc, mesmo que estejam de alguma forma relacionadas entre si, mas sim como uma entidade única que centraliza todas as entidades referidas. Existe a necessidade de composição para uma melhor apreensão.

Neste tipo de relação cada objecto é desenhado como sendo uma composição de objectos mais simples. Os objectos menores são utilizados para construir objectos de maior nível de abstracção. Pode-se dizer que os objectos menores fazem parte do estado dos objectos de nível mais elevado.

Este tipo de relação conduz a uma menor proliferação de objectos visíveis entre eles, mas por outro lado leva a relações mais fortes entre os objectos o que, em certos casos, pode ser prejudicial.

Alguns autores como [11] consideram que este é um mecanismo fundamental na resolução de problemas informáticos e que deveria ser bastante mais considerado nas discussões que se fazem em torno da evolução que as linguagens devem ter, no sentido de se tornarem orientadas por objectos[10].

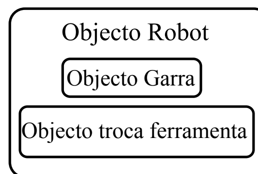


Figura 2.2 - Descrição de um Objecto Composto - Robô

Utilização

A relação de utilização aparece de uma forma natural já que é automaticamente estabelecida a partir do momento que um objecto faz uma chamada sobre outro. Obviamente que os objectos não têm uma relação de utilização com todos os objectos existentes no ambiente computacional.

Num sistema de objectos estabelecendo uma relação de utilização, cada objecto pode desempenhar os seguintes papéis:

- **Actor/Activo** - Enviam mensagens para os outros objectos mas nunca recebem nenhuma.
- **Servidor** - Objectos que apenas recebem mensagens.

[10] O autor apresenta em artigo quais as suas ideias sobre qual a evolução que os objectos devem ter na linguagem ADA.

• **Agente** - Enviam e recebem mensagens, interactuando fortemente com o sistema.

Pode-se dizer que com a relação de utilização se obtêm os mesmos efeitos do que com a relação de Agregação/Composição, mas de uma forma menos elegante. Neste tipo de relação proliferam os objectos visíveis, tornando difícil a abstracção. Mas, por outro lado, existe uma relação menos forte entre eles, o que, nalguns casos, poderá ser interessante.

A utilização de um ou outro tipo de relação deve ser cuidadosamente pesada em algumas situações, pois que ambas as filosofias podem apresentar vantagens interessantes. Nestas situações difíceis não existe nenhuma receita para aplicar, deixando-se a escolha da solução mais correcta ao implementador.

Num mesmo sistema podem coexistir os dois tipos de relações, basta pensar-se nas relações que os objectos de alto nível, resultantes da composição, estabeleceram entre si.

Figura 2.3 - Descrição de um Sistema com 2 Tipos de Relações

Um aspecto a levar em conta na utilização desta relação é o modo como se estabelece. O que acontece no momento em que um objecto "usa"[11] uma funcionalidade existente noutro depende do modo como os objectos são suportados pelo sistema. Torna-se evidente a necessidade de haver uma sincronização entre eles. Tem-se assim uma nova forma de classificar os objectos, sugerida por [12]:

• **Sequenciais** - Objectos executados num ambiente de mono programação. A sincronização é feita por chamada directa da função ou procedimento existente no objecto chamado.

• **Bloqueantes** - Objectos executados num ambiente de multi-tarefa, utilizando mecanismos de chamada remota como o "rendez-vous", que obriga o chamante a esperar que o chamado o atenda.

• **Concorrentes** - Objectos executados num ambiente de multi-tarefa, mas sem necessidade de haver encontro entre chamante e chamado.

As linguagens de programação mais divulgadas suportam ainda apenas o modelo sequencial. Os outros dois tipos apresentam-se como muito interessantes, especialmente para as aplicações de automação industrial.

2.2.2.2. Classes

Antes de se entrar na discussão do construtor classe vai-se abordar primeiramente o conceito mais genérico de classificação, que é um conceito mental de abstracção. As classes serão os mecanismos, utilizados pelas linguagens, para suportar a classificação.

A classificação surge como um mecanismo fundamental do encapsulamento de informação e é considerado por muitos autores o mecanismo fundamental da POO, especialmente quando associado à herança[12]. Pode também ser vista como uma outra forma de organizar os objectos[13]. Através dela, os objectos são agrupados em grupos e classificados segundo uma determinada forma. As classes são então o construtor mais utilizado pelas linguagens de programação orientadas por objectos para definir tipos de dados abstractos.

---

[11] Aparece no sentido de envio de mensagem.

[12] A herança é considerada por alguns autores como fazendo apenas sentido numa perspectiva de "*object orientation by classification*".

[13] Recorde-se que a outra forma importante referida é a composição/agregação.

A classificação é apenas mais uma forma de organizar a informação e pode eventualmente conduzir a estruturas mais simples. O modo com é realizada depende inteiramente da pessoa que faz a análise, isto é, não existe nenhuma receita que se possa aplicar para a criação de classes. Não há uma estrutura perfeita de classes nem o conjunto perfeito de objectos.

Na verdade, existem algumas técnicas que são aplicadas durante a fase de análise e que conduzem a soluções mais optimizadas mas é, de qualquer forma, uma tarefa muito pessoal. Uma das técnicas mais correntemente aplicadas é o agrupamento através da descoberta das características, quer de estrutura quer de comportamento, que os objectos têm em comum.

A classificação é mais do que um simples processo de identificação de classes. Através deste processo é possível identificar as hierarquias de **generalização**, **especialização** e **agregação** que existem entre as classes, isto é, descobrem-se também as relações entre as respectivas classes.

Segundo[12] têm havido ao longo da história da Ciência três aproximações gerais à classificação:

- **Categorização Clássica** - Todas as entidades que tenham em comum uma ou um conjunto de propriedades formam uma categoria.
- **Agrupamento Conceptual** - Deriva directamente dos trabalhos sobre a representação de conhecimento. As classes (agrupamentos) são criadas de acordo com determinadas descrições conceptuais e posteriormente as entidades (objectos) são agrupadas de acordo com essas descrições.
- **Teoria de Protótipos** - Uma classe de objectos é representada por um protótipo. Um objecto pertence a esta classe se e somente se for claramente semelhante.

As classes são então o mecanismo de suporte à classificação, permitindo a criação de objectos com o mesmo tipo de estrutura e comportamento. Segundo [9] a classe é um "template" a partir do qual os objectos são criados, contendo a definição do estado dos descritores e métodos para o objecto.

Uma analogia interessante para as classes é, por exemplo, um molde de plástico para a produção de uma dada garrafa. Com este molde garante-se que todas as garrafas produzidas terão o mesmo tamanho e formato.

É importante, nesta fase da descrição do problema, lançar uma outra frente de discussão que é a relação que existe entre os tipos e as classes. Este é um ponto extremamente importante e será objecto de análise mais detalhada. Pode-se dizer com alguma segurança que, por exemplo, numa linguagem tradicional tipificada como o PASCAL, ao se atribuírem tipos às variáveis se está a realizar uma forma primitiva de classificação[14]. De uma forma natural pode-se inferir que o mecanismo de tipos de dados abstractos, que devem ser considerados como uma extensão aos tipos suportados pelas linguagens mais tradicionais, pode ser facilmente implementado através de classes.

Durante esta apresentação, vai-se procurar fazer uma abordagem em que o conceito de tipo de dados abstracto aparece muito próximo do mecanismo classe e serão analisados alguns pontos importantes relacionados com os tipos.

Classes e Objectos

Até agora apenas de uma forma indirecta se relacionaram os objectos com as classes, não sendo tomada nenhuma posição forte sobre o assunto.

As classes são a forma pela qual os objectos criados no sistema são talhados. As classes podem conter a informação necessária, não só para criar os objectos, mas também o modo como eles irão ser gerados.

---

[14] Neste caso, a procura de características comuns centra-se apenas nos mecanismos estruturais.

Pode-se então introduzir uma definição para os objectos afirmando que um objecto não é mais do que uma instância de uma dada classe. Todas as instâncias partilharão a estrutura e comportamento da classe a que pertencem. De notar que, apesar dos objectos partilharem a estrutura das classes - atributos, são individuais ao nível dos valores que lhe estão atribuídos. A classe funciona como um repositório para o código da funcionalidade partilhado pelos objectos. Diz-se então que os objectos incorporam o estado de cada uma das instâncias da classe e, estas, armazenam o código de implementação dos métodos.

As classes têm capacidade para criar novas instâncias. Os métodos para a realização desta tarefa estão incorporados em meta classes - classes cujas instâncias são classes.

A confusão que se faz entre objectos e classes surge, essencialmente, pelo facto de serem interpretados de maneira diferente pelas diversas linguagens de programação orientadas por objectos.

Algumas linguagens - SMALLTALK - consideram as classes como sendo entidades com vida activa em memória e são, desse ponto de vista, idênticas aos objectos, que possuem também uma vida activa em memória. A diferença entre os dois conceitos surge no facto de um objecto corresponder a uma dada materialização de um conceito abstracto (classe) enquanto que a classe é representada por um único objecto.

Noutras Linguagens, - C, EIFFEL - as classes não têm existência real. Uma classe existe apenas no código, servindo como uma espécie de catálogo para a criação de objectos que, eles sim, têm existência material, existindo em memória. Os objectos são assim instâncias de classes, sendo os únicos componentes a que se pode aceder durante o programa. Nestas linguagens, as únicas entidades existentes no espaço memória são, então, os objectos.

A criação de objectos, quer num modelo, quer no outro, poderá ser realizada de uma forma dinâmica ou estática, consoante os objectos sejam criados em tempo de execução ou de compilação respectivamente. Para que a linguagem suporte objectos dinâmicos terá de ser associado à linguagem um mecanismo de gestão de memória que lhe fornecerá o espaço dinâmico. De notar que a noção de objecto dinâmico não está minimamente comprometida com o facto da classe poder ou não ser considerada um objecto de nível abstracto mais elevado. Nesta situação, a criação de um dado objecto é relativamente fácil, uma vez que implica apenas a sua duplicação a partir do objecto classe. Na situação dos objectos funcionarem como instâncias, para a sua criação, terá forçosamente de haver uma representação da classe em memória, para que o objecto possa ser criado à sua semelhança. A grande diferença entre os dois mundos está na funcionalidade, enquanto que na primeira situação é perfeitamente válido realizar uma operação sobre uma classe, por evocação de mensagem, na segunda situação isto é totalmente impossível.

Esta diferença implica também que os dois modelos tenham comportamentos diferentes ao nível da capacidade para fazer evoluir as suas classes. O primeiro considera-se como sendo dinâmico, dada a possibilidade de, em tempo de execução, se alterar a interpretação subjacente à execução, através por exemplo, da definição na classe da operação destruir, que destrói a classe. Neste caso, a interpretação do programa passava a contar com menos uma classe. Esta situação não é permitida no segundo modelo. Uma vez que apenas os objectos têm existência, as operações são realizadas apenas sobre estes, tornando-se impossível alterar a interpretação formal do programa.

Sendo a classe a forma a partir da qual os objectos são criados, muitas das questões levantadas durante a apresentação dos objectos têm de ser revistas e integradas nas classes, como é o caso da declaração das mensagens a que os objectos são capazes de responder. Conforme já foi dito, todos os objectos pertencentes a uma mesma classe têm o mesmo tipo de comportamento[15]. Assim, é natural que a declaração de quais as mensagens a que o objecto é capaz de receber, seja uma característica da classe e não do objecto individual.

Ter-se-á na definição da classe um local que será denominado a "interface", onde estará declarado tudo aquilo que a classe deseja dar a conhecer ao mundo exterior, enquanto que os "segredos" da sua funcionalidade

[15] Atenção que nada obriga a que o estado seja o mesmo.

e estruturas estarão escondidos na sua implementação. A "interface" será a visão externa enquanto que a implementação será a visão interna da classe.

Nada do que foi afirmado entra em contradição com o que já foi dito para os objectos. Os objectos apenas podem receber mensagens declaradas na "interface" da classe a que pertencem. O encapsulamento de dados é garantido ao nível da classe.

Relações entre Classes

As classes, tal como os objectos, não existem isoladamente daí que seja necessário estabelecer relações entre si. O modo como as classes se relacionam determina o tipo de problema e o modo como irá ser solucionado.

De alguma forma, o tipo de relações entre as classes, estará relacionado com o tipo de relações entre objectos nomeadamente a Agregação/Composição.

Na descrição de um dado domínio de um problema pode-se sentir a necessidade de relacionar uma classe com outra segundo duas vertentes: (1) necessidade de partilhar algum tipo de informação[16] e (2) por necessidade de algum tipo de ligação semântica[17]. Como exemplo da primeira vertente observe-se o caso de um robô scara e um robô antropomórfico que são ambos robôs, partilhando, por isso, algumas características comuns. Como exemplo da segunda vertente observe-se o caso de um robô e de um alimentador gravítico, que se relacionam através da necessidade de interactuarem um com o outro.

Segundo [12] existem 3 tipos básicos de relações que suportam as 2 vertentes apresentadas: (1) **generalização** (*is-a*), (2) **agregação** (*part-of*) e (3) **associação**.

A **generalização** está implicitamente ligada à partilha e aparece associada a conceitos como especialização e subclasse. Este tipo de relação, vai-se ver, está também intimamente relacionado com o conceito de herança que será apresentado num ponto à parte. No exemplo dos 2 robôs, os conceitos robô antropomórfico e robô scara podem-se considerar como sendo especializações do conceito mais geral robô. Esta noção de especialização versus generalização representa um papel muito importante também na reutilização de software, uma vez que, na criação de novas classes, não se tem de começar a partir do zero.

A **agregação** está directamente ligada à construção de objectos complexos e implicitamente relacionada com a vertente semântica referida. O objecto garra é uma parte do objecto robô. Por forma a que os objectos tenham esse relacionamento é normal que as classes a que pertencem se relacionem.

A **associação** está também intimamente ligada à vertente semântica, mas, ao contrário da agregação, relaciona classes que não têm nada a ver umas com as outras. Seja, por exemplo, o caso do relacionamento entre a classe alimentador gravítico e a classe robô. Os objectos destas classes não são de forma alguma parte um do outro ou uma qualquer especialização/generalização. Mas existe um relacionamento explícito, neste caso, dependente da aplicação que leva à necessidade de estabelecer um relacionamento entre as duas classes, dado que o robô terá de se movimentar para apanhar peças disponibilizadas pelo alimentador gravítico.

A forma como as linguagens de programação implementam os tipos de relações apresentadas, determina a sua expressividade. Várias aproximações têm sido seguidas pelas diversas linguagens, destacando-se os seguintes tipos suportados pelas linguagens de programação orientadas por objectos: **herança**, **utilização** e **instanciação** [12]. De realçar que estes tipos de relações podem aparecer combinados.

Herança

A herança implementa quer a generalização quer a associação. É o mecanismo mais conhecido. É tão importante que irá merecer uma atenção mais detalhada num outro ponto (2.2.3.3), onde será apresentada numa

---

[16] Este é um tipo de relação nitidamente vertical - taxionómica.

[17] Relação nitidamente horizontal.

perspectiva de reutilização de software. Apesar de bastante poderoso não se julgue que é suficiente para explicitar todas as possíveis situações encontradas na modelação de problemas. O facto de não suportar a agregação torna-a um pouco limitada.

Para terminar a abordagem actual da herança podemos suspender o tema deixando no ar uma definição que pretende resumi-la. Uma classe (subclasse?) que estabeleça uma relação de herança com outra herdará dela a sua estrutura e comportamento.

Com esta definição torna-se mais claro o facto de se suportar a generalização. A partir do momento que um objecto é uma especialização de outro, terá efectivamente de herdar toda a sua estrutura e funcionamento. Em termos de conjuntos, a classe especializada abrange o conjunto de atributos e funcionalidade da classe generalista adicionada com os seus próprios atributos e funcionalidade.

As relações de associação assentam, fundamentalmente, no conceito de herança parcial. Enquanto que na generalização a herança dos atributos e funcionalidades é total (is-a), nas relações de associação, podem-se restringir os atributos e funcionalidades que são herdados.

<u>Utilização</u>

A herança poderá não ser suficiente para modelar todas as várias possibilidades de relacionamento de classes num dado sistema, como é o caso, por exemplo, da relação entre a classe robô, a classe garra e a classe *sensor_de_forças*, numa aplicação de montagem. As duas últimas classes não são, neste domínio de aplicação, uma especialização de robô, nem uma associação, apenas uma parte dele. Neste exemplo teríamos:

```
OBJECT robô
   ...
   posição: ARRAY[1..3] OF REAL
   teta1, teta2, teta3, teta4: REAL
   garra: CLASSE garra
   sens_forças: CLASSE sensor_de_forças
   ...
interface
   method get_posição()
   method insere_pino()
   method change_garra(CLASSE garra)
END robô
```

Neste caso, podem-se analisar os dois tipos de relação de utilização: (1) a interface de uma classe a utilizar uma outra classe - CLASSE garra e (2) a implementação a utilizar outra - CLASSE *sensor_de_forças*.

Na primeira situação a classe utilizada deve estar visível para qualquer cliente. Neste caso qualquer objecto supervisor, por exemplo, que procura utilizar o *method change_garra* tem de conhecer quer a classe robô (*method change_garra()*) quer a classe garra, para a poder utilizar como argumento.

Na segunda situação a CLASSE *sensor_de_forças* está escondida na implementação. O facto de não estar explícita na interface evita que seja dada a conhecer aos clientes da CLASSE robô.

Existe alguma semelhança entre esta relação e a sua congénere para os objectos apesar de estarem em contextos diferentes, neste contexto, a relação suporta a agregação entre classes. Quando perante uma agregação de classes, não se pretende que os atributos e funcionalidades sejam herdados para o objecto complexo. Na verdade, o objecto utilizado faz parte do complexo, mas o utilizador não tem acesso a essas características. Assim sendo, é conceptualmente mais correcto conservar as duas entidades separadas e estabelecer relações de utilização entre elas. O objecto robô, apesar de ser constituído por uma garra, não tem as mesmas propriedades que o objecto garra.

<u>Instanciação</u>

A instanciação, tal como a herança também suporta a generalização e a associação mas de uma forma

completamente distinta. Esta relação é implementada em CLU, ADA, EIFFEL, etc[18], sob um nome bastante conhecido: **classes genéricas** ou **parameterizadas**.

Uma classe genérica é como uma forma para gerar outras classes. A vantagem é que a forma pode ser parameterizada por outras classes, possibilitando assim a criação de novas classes, cujo formato foi o resultado de uma actuação indirecta de outras. A geração de uma nova classe obtém-se a partir da instanciação da classe genérica, passando como argumento a classe que irá servir de parâmetro na criação.

Seja novamente o exemplo da classe robô. Suponha-se agora, que esta classe era declarada como sendo genérica e que se pretendia obter uma nova classe *robô_scara*. Assim, haveria que instanciar esta nova classe sob a classe robô e passando como argumento a classe *scara*.

É, então, possível obter relações de generalização/especialização e de associação.

Existe alguma polémica entre os defensores das classes genéricas e os defensores da herança. Meyer [13], por exemplo, defende que a herança é um mecanismo mais poderoso que as classes genéricas e que os benefícios atingidos com as classes genéricas podem ser todos obtidos com a herança mas que o contrário não é verdadeiro. Rosen [11], por seu lado, defende que a herança só é um melhor mecanismo, quando o processo predominantemente utilizado na análise do problema foi baseado na classificação. Para ele, a utilização da composição na análise é uma forma melhor de analisar o problema e, neste caso, a utilização de pacotes genéricos[19] torna-se uma forma de relação bastante elegante e flexível.

Como não se chega a um acordo sobre qual a melhor forma de relação e, partindo da experiência prática que diz que em determinadas situações, uma forma é melhor que a outra e, vice-versa, para outras situações, o melhor é esperar que as linguagens suportem ambos os modelos.

2.2.3. Reutilização de Software

É antigo o sonho dos engenheiros de software de poderem utilizar ao máximo módulos de software existentes, no desenvolvimento de novas aplicações. O desenvolvimento massivo de componentes[20] de software representa a panaceia para muitos dos males detectados na produção de software.

A analogia com o hardware torna-se inevitável. Porque não utilizar componentes de software tal como o projectista de sistemas digitais utiliza circuitos existentes. Para este projectista, a não utilização deste tipo de componentes equivale ao abandono do projecto por, entre outras razões, não ser economicamente viável.

A razão do realce para este tema, dentro desta tese, relaciona-se com o facto de a área de produção de software para aplicações em automação industrial ser bastante complexo e de desenvolvimento lento. Assim, qualquer inovação no sentido de se aumentar a produção e facilidade de compreensão do problema, é obviamente bem vinda.

Os procedimentos/funções, macros e bibliotecas têm sido, desde há longo tempo, os mecanismos privilegiados de encapsulamento, permitindo uma melhoria ao nível da reutilização de componentes de software.

Os mecanismos básicos fornecidos pela POO, bastante importantes para uma melhoria do aspecto em análise são: **Instanciação**, **Classes**, **Herança**, **Polimorfismo**, **"Overloading "** e **Classes Genéricas**.

---

[18] Espera-se que o C++ suporte brevemente este conceito.

[19] Pacotes genéricos é o nome que o ADA dá às classes genéricas.

[20] Conforme defendeu D. McIlroy [14] em 1968 num “*workshop*” da NATO sobre "*software crisis*".

2.2.3.1. Tipos de Dados Abstractos - ADTs

O tipo de dados abstracto aparece neste ponto dada a sua importância no desenvolvimento da POO. A noção de classe está intimamente relacionada com este conceito. A classe pode ser vista como uma implementação de um ADT e a análise de um sistema como sendo a procura dos ADTs relevantes para a descrição do sistema.

A importância dos ADTs reside no facto de ser uma ferramenta matemática, desenvolvendo-se em torno deles uma determinada teoria que pode ser utilizada na análise de programas. Qualquer paradigma de programação deverá ter um suporte matemático que favoreça a aplicação de teorias.

O conceito de tipo de dados é fundamental na ciência de computadores, podendo afirmar-se que a história das linguagens é em grande parte a história dos tipos. A evolução das linguagens fez-se muito à custa de um cada vez maior suporte de tipos diferentes e da possibilidade do programador criar os seus próprios tipos.

Um tipo de dados é uma descrição abstracta de um grupo de entidades relacionadas. Com a abstracção consegue-se separar o que é importante do que é irrelevante para a compreensão de determinado fenómeno. Os tipos são o mecanismo fundamental para suportar a abstracção, uma vez que permitem que valores relacionados sejam agrupados de uma tal forma que as suas semelhanças sejam realçadas e as suas diferenças ignoradas, por não serem relevantes. É um mecanismo fundamental na ciência para se apreenderem os sistemas complexos. Sem eles seria muito difícil a construção de programas muito longos e complexos.

O tipo aparece também como um mecanismo de reutilização. Com efeito, sempre que se declara uma variável como pertencendo a um determinado tipo está-se a reutilizar um conjunto de conceitos que foram definidos na altura da declaração do tipo.

Seja, por exemplo, o tipo inteiro, que pode ser encontrado como tipo básico de quase todas as linguagens. Quando se declara uma variável como sendo deste tipo está-se simplesmente a reutilizar o conjunto de operações (funcionalidade) definidas sobre o mesmo. Como tal, as linguagens orientadas por objectos não trouxeram nada de novo, se se considerar apenas a questão dos tipos pré-definidos. Quando a discussão passa para o nível da criação de novos tipos (tipo definido no seu modo tradicional) pelo utilizador o caso muda de figura e estamos perante um conceito novo de reutilização.

Os tipos de dados podem desempenhar os seguintes papéis:

• Abstracção das propriedades intrínsecas dos objectos que estão a representar.

• Criação de abstracções de mais alto nível - composição de tipos.

• Protecção.

Numa definição informal de tipo considera-se que o tipo é constituído pela representação e pelas operações que podem ser realizadas sobre ele. Na noção de tipo suportada pelas linguagens tradicionais o conjunto de operações era pré-definido e eram constituídas pelas operações de acesso mais as operações de base. Por exemplo, quando se declara uma variável de tipo inteiro, neste tipo de linguagens, o programador utiliza sem quaisquer restrições as operações de acesso e aritméticas que estão previamente definidas.

Neste caso:

```
...
x, y: integer;
...
x := x + y;
...
```

Neste extracto de código aparecem as operações de afectação, de acesso e a operação aritmética soma, todas elas previamente definidas. Caso o programador pretendesse criar um novo tipo, com operações novas ficava imediatamente impedido, dada a noção de tipo suportada pela linguagem não permitir a definição de operações. Nestas linguagens, quando suportavam a criação de novos tipos, o programador ficava limitado às operações fornecidas por ela.

Por exemplo, em PASCAL, quando um programador cria um novo tipo pertencente a um conjunto ordenado, fica limitado às operações de *sucessor* e *predecessor* previamente definidas. O mesmo acontece com o construtor "*record"*, sobre o qual estão apenas definida operações de acesso.

Um conceito mais abrangente de tipo - ADTs, conduz-nos a que um tipo seja uma estrutura dividida em duas zonas: a **interface** e a **implementação**.

Um tipo de dados abstracto será então uma estrutura complexa em que as duas zonas estão perfeitamente demarcadas. A interface destina-se a afixar ao mundo exterior qual o comportamento do tipo, isto é, quais as operações que sobre ele podem ser realizadas. A implementação será constituída pela representação interna do tipo e pela implementação das operações.

Veja-se o seguinte caso:

```
...
ABSTRACT TYPE INTEGER
   afecta(v: BASIC_INTEGER)
   valor() -> BASIC_INTEGER
   mostra()
   mais(v: BASIC_INTEGER)
implementação
   val: BASIC_INTEGER;
   afecta(v: BASIC_INTEGER)
   begin ... end;
   valor() -> BASIC_INTEGER
   begin ... end;
   mostra()
   begin ... end;
   mais(v: BASIC_INTEGER)
   begin ... end;
END;
x, y: ABSTRACT TYPE INTEGER;
...
x.afecta(3);
y.afecta(5);
x.afecta(x.mais(y.valor));
x.mostra();
```

A utilização do tipo ABSTRACT TYPE INTEGER definido como um ADT, obriga a que sejam declaradas todas as operações que se pretendem realizar sobre as suas instâncias.

Esta noção estendida de tipo tem muito a ver com o conceito de classe já apresentado. Iremos ver no ponto seguinte onde é que a noção de classe se afasta da noção de tipo.

2.2.3.2. Tipos e Classes

Um tipo é a abstracção funcional de um dado conceito. O interessante numa abstracção é a sua funcionalidade, isto é, as operações que podem ser realizadas sobre a entidade em causa. Dois tipos com a mesma funcionalidade não são distintos por possuírem implementações diferentes, a distinção surge pela diversidade nas operações.

O tipo robô fica perfeitamente definido, a partir do momento em que são especificadas as operações que se podem ser realizadas sobre as suas instâncias. O facto de um dado modelo de robô implementar um movimento linear segundo um algoritmo diferente de outro, não significa que não sejam ambos robôs, do ponto de vista de abstracção.

Um exemplo talvez mais clarificador, aparece na definição do tipo pré-definido inteiro. Será justificável considerar um inteiro cuja implementação para a sua representação é a 16 bits, diferente de um outro cuja representação é a 32 bits? Parece que não, uma vez que as operações que se podem definir sobre ambos são precisamente as mesmas. O grau de compreensão para um e outro é precisamente o mesmo, isto é, do ponto de vista abstracto, os dois conceitos são iguais.

Em conclusão, uma classe representa a implementação de um tipo de dados abstracto, mas não se deve confundir uma classe com um tipo. Um tipo pode ser implementado por diversas classes, enquanto que a uma classe corresponde apenas um tipo de dados abstracto.

Subtipos e Subclasses

Este ponto está muito relacionado com a herança que será uma forma, como se irá mostrar, de implementar a noção de subtipo.

Um tipo T2 diz-se subtipo de T1 se cada instância de T2 for também uma instância de T1 [15]. Por exemplo, cada robô "*scara*" é também um robô tal como cada robô antropomórfico é também um robô. A noção de subtipo implica que qualquer instância de T2 pode ser utilizada onde se espera uma instância de T1, o que não é de estranhar já que, pela definição, T2 engloba todas as operações de T1. Este princípio denomina-se *princípio da substituição*.

Atenção que o inverso pode não ser verdadeiro, isto é, onde se espera uma instância de T2 pode não ser válido utilizar uma instância qualquer de T1.

A relação subtipo é *reflexiva*, *transitiva* e *antissimétrica*, impondo consequentemente uma relação de ordem.

Uma outro aspecto interessante a levar em conta na discussão tipo/subtipo é a noção de fecho que os conjuntos[21] devem ter. Seja, por exemplo, o conjunto dos inteiros e o seu subconjunto números primos. As operações que são aplicadas sobre os inteiros devem ter uma aplicação directa sobre os seus subconjuntos. Seja o caso da operação soma que será definida da seguinte forma:

INTEIRO
```
soma: inteiro + inteiro -> inteiro
```

Neste caso, a aplicação sobre os números primos deveria ser:

PRIMOS
```
soma: primo + primo -> primo
```

Ora este facto nunca acontece violando o fecho do conjunto. Este problema irá ser resolvido, segundo duas vertentes: (1) por definição de um conjunto de restrições que indicarão os valores para os quais a operação se comporta de acordo com a definição e (2) pela herança dos operadores do subtipo mas abandonando a restrição do fecho, isto é, permitindo que se tenha:

PRIMOS
```
soma: primo + primo -> inteiro
```

A segunda vertente parece ser mais razoável, mas tem o problema de os operadores do subtipo não funcionarem exactamente da mesma maneira que os do tipo.

Pela definição, e pelo que já foi apresentado sobre o mecanismo de herança, pode-se concluir que esta é uma forma de implementar a noção de subtipo e que as subclasses serão a forma computacional do subtipo. Pela mesma razão que não se confunde tipo com classe não se deve confundir subtipo com subclasse.

2.2.3.3. Herança

A herança é o mecanismo fundamental na implementação da reutilização. É bastante comum sobrevalorizar-se este aspecto da POO, dada a sua importância na reutilização e na implementação do conceito de subclasse/subtipo.

O facto de existir uma herança não quer dizer que se esteja a criar um subtipo. Existem, por sinal, alguns tipos de herança que facilitam a implementação da noção de subclasse, que pode ser relacionada com o conceito

[21] Não se deve estranhar a utilização do termo conjunto já que, por definição, diz-se que tipo é um conjunto de objectos.

de subtipo.

Quando se fala em herança está-se, em princípio, a falar da herança entre classes e não de herança entre as instâncias das classes - os objectos. A herança apresenta-se como sendo o mecanismo implementador da noção de subclasse, explorando a semelhança entre certas classes de objectos.

A herança pode ser simples ou múltipla. Na primeira situação, uma dada classe relaciona-se com apenas uma outra classe. Na segunda situação uma classe apresenta-se relacionada com várias classes, herdando as características provenientes das classes com quem está relacionada.

Um ponto interessante da discussão está no facto de as subclasses poderem ou não aceder aos atributos das suas classes. A possibilidade de acesso é uma violação do princípio do encapsulamento da informação. Recorde-se que, segundo este princípio, uma dada classe deve poder ser desenvolvida independentemente de uma outra qualquer classe. Uma subclasse ao poder aceder à estrutura interna da sua super classe limita grandemente este princípio uma vez que uma alteração na super classe repercute-se imediatamente na subclasse. Assim sendo, as subclasses deveriam apenas poder aceder à interface das suas super classes. Está-se a defender a teoria segundo a qual não deverá haver dois tipos de clientes diferentes de classes: objectos e subclasses.

Dentro deste assunto poderá levantar-se uma outra questão relacionada com a possibilidade das classes poderem sofrer uma evolução – "*schema evolution*" - ou seja uma alteração à sua estrutura. Nesta situação, o que se passará com as subclasses e o que acontecerá aos objectos que tenham sido previamente instanciados serão as interrogações fundamentais. À partida será interessante que o sistema garanta existência aos objectos previamente criados, podendo constatar-se que estes objectos representam versões de um dado objecto. A evolução da classe representaria assim uma forma de implementar versões, mecanismo muito importante nas aplicações complexas actuais - CAD-CAM (figura 2.4).

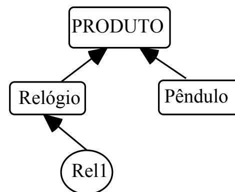


Figura 2.4 - "Schema-Evolution"

Inicialmente a classe relógio contém apenas descrições relevantes sobre o componente base do relógio.

```
classe relogio
    public
  method ref_ponto_central
  method ref_numeros
    private
  numbers: array[1..12] of referencial;
  central_point: referencial;
end classe;
```

À medida que fossem realizadas operações de montagem, a classe deveria reflectir a nova situação.

```
classe relogio
    public
  method ref_central_point
  method ref_numbers
  method ref_haste_central
  method pont_minutos
  method pont_horas
    private
  números: array[1..12] of referencial;
  ponto_central: referencial;
  haste: referencial;
  pont_horas: referencial;
  pont_minutos: referencial;
end classe;
```

Um ponto deixado em aberto seria o que aconteceria com o objecto que pertencia à classe anterior. Neste caso e, uma vez que se o objecto físico que se está a modelar sofreu uma transformação, deveria considerar-se o caso do objecto passar a reflectir o novo estado da sua classe.(Nos sistemas de "frames" é muito comum o "*schema evolution*", por adição de novas propriedades).

A herança também pode ser vista como uma especialização, nomeadamente na área da representação de conhecimento. Esta noção de especialização aparece associada às relações is-a, em que uma dada subclasse contém todas as propriedades da sua classe e possivelmente mais algumas.

Deve-se desfazer os equívocos de que os únicos mecanismos de herança são os de herdar todas as características da classe para a subclasse. Na realidade poderão haver vários tipos de herança que não só o mecanismo *is-a*. Convém desde já realçar o facto de que a maioria das linguagens de programação orientadas por objectos, de uso generalizado, apenas implementa o mecanismo de herança conducente à implementação de relações *is-a* (Ex: Cplus, EIFFEL, ..).

Esta discussão conduz forçosamente à questão da especialização versus agregação/composição. Aparentemente são dois conceitos muito semelhantes, uma vez que ambos estruturam a informação. Seja, por exemplo, o objecto robô que é constituído por uma estrutura mecânica e por um controlador. Poderia ser-se tentado a considerar a seguinte estrutura:

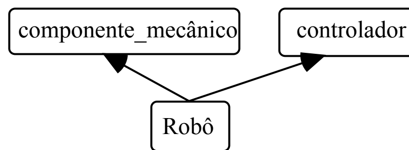


Figura 2.5 - Robô Definido por Especialização

Em que o objecto robô herdaria as características relevantes de *controlador* e do *componente_mecânico*. Mas, do ponto de vista conceptual, não parece ser muito correcto uma vez que um robô não é um controlador nem apenas um componente mecânico, daí que seja muito mais interessante considerar, para este caso, a agregação em que o robô é constituído pelos dois sistemas:

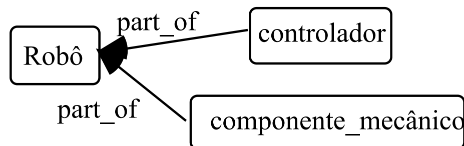


Figura 2.6 - Robô por Composição/Agregação

A relação existente entre as diferentes classes deixou de ser uma relação de classe<->subclasse, nitidamente vertical, para passar a ser uma relação nitidamente horizontal. O mecanismo de implementação destas relações deixou de ser a herança.

Até agora a discussão centrou-se essencialmente sobre a herança estática, ou seja uma herança que uma vez estabelecida é mantida para sempre. Mas poderá existir um outro tipo de herança suportando relações dinâmicas entre as classes. Quer isto dizer que os objectos têm a possibilidade de mudar a classe a que pertencem. Existe alguma tendência para se confundir este conceito com o de "*schema evolution*", mas a sua diferença é bastante notória no facto de na herança dinâmica a alteração de comportamento se dar ao nível dos objectos e não ao nível da classe.

Numa relação de herança entre classes não é obrigatório que todas elas possuam instâncias. Neste caso as classes chamam-se **classes abstractas**. Uma classe abstracta é criada na expectativa que as suas subclasses adicionem informação à sua estrutura e comportamento, geralmente completando a implementação dos seus métodos incompletos.

Numa hierarquia de classes relacionadas por relações de herança a classe mais geral chama-se **classe base**. O número de classes base[22] depende directamente do domínio do problema.

2.2.3.3. Polimorfismo

Os tipos das linguagens de programação tradicionais são essencialmente monomórficos no sentido de a um dado valor ser atribuído um único tipo. Quando se declaram variáveis, parâmetros de funções e procedimentos ou funções fica atribuído um único tipo a cada uma destas entidades. A verificação de tipos, em tempo de execução ou em tempo de compilação, zela para que a consistência seja mantida.

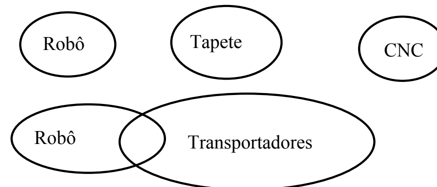


Figura 2.7 - Exemplos de Monomorfismo e Polimorfismo

Esta noção de aplicação de um único tipo pode ser bastante restritiva, aparecendo por isso a noção de polimorfismo. Polimorfismo é segundo [9] a capacidade que uma determinada entidade tem de pertencer a vários tipos.

A diferença entre o conceito de monomorfismo e polimorfismo pode ser realçada recorrendo aos diagramas de Venn da teoria de conjuntos (Figura 2.7).

Numa linguagem monomórfica determinada entidade só poderá ser do tipo robô ou-exclusivo, CNC ou-exclusivo transportador. Numa linguagem que suporte polimorfismo, a entidade poderá agora ser do tipo robô ou transportador. Uma determinada variável que represente um dado robô pode ser encarada como sendo do tipo robô ou então do tipo transportador, sem haver perda de consistência. É importante que haja regras para reger quais os tipos diferentes que uma dada entidade pode suportar. Por exemplo, no caso anterior deixaria de fazer sentido que a certa altura a variável que modela o robô passasse a suportar o conceito abstracto garfo.

O polimorfismo aparece como um conceito interessante do ponto de vista da Engenharia de Software desde que as diferentes formas que as entidades possam suportar estejam de algum modo relacionadas entre si. Vai-se ver que as linguagens implementam o polimorfismo de uma forma que garante este último facto.

O suporte ao polimorfismo não é apanágio apenas da POO, podendo-se encontrar formas de polimorfismo em linguagens mais tradicionais como o PASCAL e o C. A diferença está em que o polimorfismo poderá ser um conceito muito mais poderoso do que a versão restrita das linguagens mais tradicionais.

A **coerção/conversão** de tipos e o **"overloading"** são duas formas de polimorfismo bastante utilizadas nas linguagens tradicionais.

A **coerção** de tipos é uma forma limitada de polimorfismo e consiste em transformar os tipos existentes noutros mais convenientes. A coerção de tipos pode ser explícita ou implícita.

Um exemplo bastante sugestivo de coerção implícita acontece quando se utiliza a função:

```
soma: real x real -> real
```

com um parâmetro real e outro inteiro. Neste caso o parâmetro inteiro é implicitamente convertido num real. Seja o seguinte programa PASCAL:

```
...
```

[22] De notar que algumas linguagens de programação impõem elas próprias uma classe base, da qual estão dependentes todas as classes criadas pelo programador.

```
var arg1: integer;
    arg2: real;
...
arg2 := arg1 + arg2;
```

O código anterior é perfeitamente válido, não dando qualquer erro de compilação nem em tempo de execução, dado que foi feita uma coerção automática de tipos. A linguagem PASCAL é bastante restrita ao nível da possibilidade de coerção implícita não fazendo nenhuma operação de transformação que possa colocar em perigo a semântica do problema, o que já não acontece com o C, veja-se o seguinte exemplo:

```
char car;
int num;
float fnum;
...
car = ' ';
fnum = 1.5;
num =  fnum +  car;
printf("%d", num);
```

O programa não só é compilado com sucesso como corre e fornece um resultado 33. Mas qual o significado semântico de somar um *float* com um caracter ? Duma perspectiva de alto nível o significado é nulo, só fazendo sentido quando se pretende desvirtualizar as operações sobre os tipos. Reais e caracteres não são tipos comparáveis, representam entidades completamente diferentes, com operações distintas. Esta é uma operação típica de uma linguagem (em PASCAL não seria permitido) que permite fazer operações sem ter em conta o seu significado semântico. À partida, torna a linguagem muito flexível, mas transfere para o programador toda a responsabilidade dos seus actos e pode tornar o código ilegível. A razão do resultado 33 decorre da conversão do tipo char para inteiro (32 ASCII, de espaço) e da conversão do *float* 1.5 para inteiro (1); 32 + 1 = 33.

Um exemplo de <u>coerção/conversão explícita</u> acontece quando se é obrigado a escrever qual a transformação desejada. Do ponto de vista semântico pode ser mais interessante, uma vez que fica explícito no código quais as transformações que estão a ser realizadas, mas pode ter efeitos bastante perniciosos se for bastante aberta, isto é, se permitir que qualquer tipo seja convertido num outro qualquer. A linguagem PASCAL não permite este tipo de coerção que se pode encontrar na linguagem C. Seja o seguinte código C:

```
int imprime_tempo(float tempo)
{
   return printf("%d ", (int) tempo / 60);
}
```

A impressão vai ser feita como sendo um número real, mas o resultado que, em condições normais, seria do tipo real, vem do tipo inteiro obtendo-se a parte inteira do resultado.

O **"overloading"** consiste na capacidade de uma função ser usada com vários tipos de parâmetros. É o caso da função soma que pode ser sobrecarregada para operar com parâmetros inteiros e reais. O que na realidade acontece é que são implementadas várias funções, cada uma com parâmetros de tipos diferentes. A função a invocar é escolhida em função dos tipos associados aos parâmetros.

As linguagens mais tradicionais como o PASCAL e o C implementam overloading apenas ao nível das suas funções pré-definidas, como é o caso das operações e dos procedimentos <u>*read*</u> e <u>*writeln*</u>, no caso do PASCAL. A linguagem ADA fornece ao utilizador a possibilidade de criar "*overloadings*" [16,17].

O polimorfismo representado apenas pela coerção e "*overloading*" apresenta-se bastante restrito, podendo afirmar-se que a exploração de conceitos mais poderosos a si associados tornaram-se um vector importante a explorar na investigação de novas linguagens de programação. Este assunto foi intensamente investigado por [15] que propuseram uma taxonomia de técnicas associadas ao polimorfismo. Nesta taxonomia a primeira divisão ocorre entre polimorfismo <u>universal</u> e <u>ad-hoc (coerção e "*overloading*")</u>. A diferença entre ambos reside no facto do polimorfismo universal poder ser aplicado uniformemente sobre todos os tipos da linguagem. De realçar que o polimorfismo acontece para os tipos que de alguma forma estão relacionados entre si (hierarquia). Um caso interessante para ilustrar esta diferença acontece com as funções. Uma função polimórfica universal

vai executar o mesmo código, qualquer que seja o tipo, o que não acontece com as funções "*overloaded*" que executam código diferente para cada um dos tipos. Note-se neste caso a imensa flexibilidade do polimorfismo universal no caso de se pretender adicionar um novo tipo. A sua adição obriga apenas a que o novo tipo esteja de alguma forma relacionado com os tipos suportados pela função polimórfica genérica. No sistema com funções "*overloaded*" é-se obrigado a realizar uma codificação. O polimorfismo ad-hoc significa assim uma limitação na flexibilidade.

Segundo Cardelli e Wagner [15] o polimorfismo universal pode ser dividido em dois tipos: paramétrico e inclusão.

No polimorfismo paramétrico (pacotes genéricos do ADA) uma função é codificada uma única vez e suporta uniformemente uma série de tipos que apresentem alguma semelhança estrutural. No polimorfismo de inclusão (relacionado com o conceito de subtipo) a função suporta um conjunto de tipos relacionados entre si por relações de subtipos.

Existe polimorfismo de inclusão quando se define uma classe robô_scara que é um subtipo da classe robô que, por seu lado é um subtipo de outra classe executor. Uma função que aceite como argumento uma variável do tipo executor aceitará também variáveis do tipo robô e *robô_scara*.

As vantagens do polimorfismo sobre o monomorfismo são: flexibilidade, capacidade de abstracção, partilha de comportamento (mesmas operações suportadas para tipos diferentes) e partilha de código (código é feito apenas uma vez para um conjunto de tipos.

2.3. "*Frames*"

2.3.1. Introdução

Neste capítulo pretende-se apresentar, de uma forma breve, um conjunto de ideias e conceitos nascidos no âmbito da Inteligência Artificial e conhecidos sob o nome de "frames".

A palavra "***frame***" foi introduzida por Minsky em 1975 [18], para denominar um conjunto de ideias, ainda um pouco desligadas, relacionadas com a representação de conhecimento. Ainda hoje este conceito não tem um significado muito preciso, mantendo-se ainda como o unificador de ideias, com a agravante de se ter tornado ainda mais abrangente e com significados semânticos distintos para áreas de investigação distintas. Esta diversidade de interpretações obriga a que se clarifique cada uma das possibilidades.

Os "frames" podem ser interpretadas segundo três vertentes [19]:

- **linguagem formal para exprimir conhecimento**
- **aproximação metafísica**
- **aproximação heurística**

A primeira vertente surge desde o início e pretende ser uma alternativa às redes semânticas ou ao cálculo de predicados [20], de forma a tornar a representação de conhecimento (que pode ser descrita pelos dois processos referidos) mais facilmente tratável por computador.

A segunda vertente pode-se considerar como estando relacionada com a primeira mas com a grande diferença de que os "frames", neste caso, são utilizadas para descrever o conjunto de coisas que o programa necessita de saber em vez do modo como essas coisas são ou podem ser representadas (primeira vertente). Segundo este ponto de vista, a utilização de "frames" implica que se assuma a representação de um certo tipo de conhecimento, que é descrito através dessas mesmos "*frames*". A diferença para a primeira vertente, pode-se sumariar como sendo a diferença que existe entre uma linguagem e a forma como se implementam as descrições presentes na linguagem.

A terceira vertente é nitidamente uma aproximação implementacional, o que não acontecia com as duas primeiras, que se destinam a representar apenas conceitos. Aqui, os "*frames*" são vistos como sendo entidades computacionais destinadas a organizar a informação representada na memória dos computadores. Não existe mais a noção de frame como sendo uma linguagem de representação, podendo até dizer-se que os "*frames*" podem ser uma das muitas possibilidades para a implementação de uma dada linguagem.

Esta última vertente é a que tem prevalecido. Pode-se concluir que, apesar do paradigma de "*frames*" ter sido inicialmente proposto, por Minsky [18], como uma ferramenta para controlo de raciocínio automático, a sua utilização/divulgação generalizada tem sido ao nível da representação estrutural (modelação) [21]. Esta centralização na representação realçou as capacidades semânticas do paradigma para a organização e armazenamento de conhecimento, resultando daí que, muitas vezes, se trate a informação presente nos "frames" como sendo uma **base de conhecimento**.

Estas diferentes aproximações e, o facto da fronteira entre implementação e representação, não ser, muitas vezes, precisa, conduz a que se confundam os três conceitos. Os "*frames*" aparecem assim como uma representação e uma implementação simultaneamente, dependendo do contexto de utilização. A aproximação seguida, nesta tese, encara os "frames" como sendo uma linguagem de representação de conhecimento mas num sentido diferente da perspectiva seguida por alguns investigadores de IA, que parece ser um pouco restrita, nomeadamente ao nível da descrição comportamental.

O caso do cálculo de predicados, é um exemplo do modo como uma linguagem de grande capacidade expressiva e com uma semântica bem definida pode ser pouco útil para a modelação de sistemas de produção, já que os construtores da linguagem não permitem a definição de construtores complexos, tornando a modelação de entidades complexas difícil.

Numa perspectiva de utilização de ferramentas para sistemas de produção, não interessa que uma ferramenta destinada a modelar (representar conhecimento), apenas o consiga para os aspectos estáticos. É bastante importante que seja adequada para representar os aspectos comportamentais das entidades que pretende modelar. Seria importante que o paradigma suportasse a descrição, de uma forma declarativa, do modo como o conhecimento, presente nos "*frames*", poderia ser utilizado. Com esta possibilidade a descrição funcional ficaria mais completa e legível. A forma tradicional de resolver esta questão do comportamento é feita através da atribuição ao frame, de uma forma qualquer, de um conjunto de procedimentos escritos numa linguagem de programação tradicional.

Para resolver os problemas é muitas vezes necessário estender a linguagem com construtores por forma a suportar a descrição do comportamento de uma forma declarativa, por exemplo através da adição de um sistema de regras cujo raciocínio poderá ser do tipo *forward* ou *backward*. As regras podem ser encaradas como um subconjunto do cálculo de predicados e a sua relação com as "frames" parece ser óbvia. Não cabe no âmbito deste capítulo descrever este tipo de extensões, ficando a descrição dos aspectos comportamentais confinada à utilização de procedimentos associados ao frame (métodos e demónios).

O facto de já se ter debatido com algum pormenor, no ponto 2.1, o paradigma da programação orientada por objectos e dada a sua semelhança com os "*frames*" conduz a que a discussão deste capítulo se centre maioritariamente nas diferenças.

O paradigma de "*frames*" aparece mais associado ao desenvolvimento de protótipos, sendo por isso menos restritivo ao nível da tipificação, do dinamismo de relações e da própria estrutura da entidade que representa. O suporte para a definição de relações e alteração da estrutura da "*frame*", de uma forma dinâmica, é um dos pontos mais importantes na distinção dos dois paradigmas. As linguagens de "*frames*" fornecem um conjunto de construtores que podem ser utilizados de uma forma elegante para a criação dos tipos que descrevem o conhecimento de domínio do sistema que estão a representar. As descrições que representam um dado tipo de entidades podem ser inicializadas com a descrição de uma dada entidade desse tipo. Este protótipo será copiado para a estrutura criada, sempre que se adicionar uma nova entidade desse tipo. Esta noção poderá ser utilizada para a verificar se um dado frame pertence a um dado tipo.

2.3.2. Noções Básicas

Vão-se introduzir alguns dos conceitos chave associados aos "*frames*", realçando as semelhanças e as diferenças para a POO.

Chama-se a atenção para o facto dos exemplos que forem descritos não seguirem a sintaxe de nenhuma linguagem de "frames" em particular, uma vez que estas descrições pretendem apenas clarificar os pontos em análise e não apresentar uma determinada linguagem. Deste modo, seguiu-se uma sintaxe o mais clara e simples possível, mas dada a experiência do Grupo de Robótica em sistemas do tipo Knowledge Craft - KC™, com a sua linguagem de representação - CRL™ e Knowledge Engineering Environment - KEE™ [22-26] , é natural que as semelhanças se evidenciem.

2.3.2.1. Aspectos Estruturais

"*Frame*"

Um frame é encarado como sendo uma estrutura de dados, destinado a representar um estereótipo [19]. Esta definição, característica da IA, deve ser estendida no sentido de a encarar como representando um tipo de dados abstracto, conduzindo assim à noção de **classe**, definida anteriormente. Pode-se então dizer que um "*frame*" é uma representação estrutural de uma dada classe de objectos.

Um "frame" tanto pode representar uma classe de objectos como esse objecto - instância. As linguagens que suportam "frames" permitem que as classes possam ser organizadas em taxonomias (ver relações *is-a*). De notar que as vantagens da estruturação do conhecimento segundo esta forma já foram descritas no capítulo anterior.

Os "frames" podem conter conjuntos de descrições de atributos a que se chama *slots*.

Veja-se o seguinte exemplo de frame:

```
FRAME robô
   ...
   aceleração:
   velocidade:
   posição:
   teta1: teta2: teta3: teta4:
   garra:
   get_posição: method get_posição_fn()
   ...
```

No KEE™ [27] um frame é conhecido através do identificador **FRAME**, enquanto que no CRL™ [28] se denomina **SCHEMA**.

Instâncias de "*Frames*"

Uma instância representa uma entidade específica de um "*frame*". As instâncias estão para os "*frames*" como os objectos estão para as classes. Nem todos os *slots* de uma instância têm de ser necessariamente preenchidos, permitindo assim a modelação da dúvida, tal como não é necessário que uma instância de um dado "*frame*" mantenha a mesma estrutura deste. Como se referiu, a possibilidade de criação/remoção dinâmica de *slots* conduz a esta situação, o que nunca acontecia na POO.

Em termos implementacionais, os "*frames*" que representam classes e as suas instâncias (que também são "*frames*") são identificadas por um identificador único, tal como acontece, por exemplo, no SMALLTALK para as classes e objectos.

Uma instância do frame robô poderia ser:

```
FRAME robô_1
   instance-of: robô
   ...
   aceleração:
   velocidade:
   posição:
   teta1: teta2: teta3: teta4:
   garra: garra_1
   get_posição: method get_posição_fn()
   ...
```

em que *garra_1* representa o identificador de um frame garra.

Slots

Tal como nas classes, existem atributos que podem formar relações de agregação/composição, que se chamam ***slots***. Um "*frame*" é então constituída por *slots*, que podem ser afectados por valores que podem representar nomes, identificadores, valores numéricos ou ainda outros "*frames*". Quando um *slot* contém um identificador que representa uma "*frame*" está a formar uma relação de agregação/composição.

Os identificadores podem nomear métodos que são as entidades destinadas a modelar o comportamento dinâmico. Neste caso diz-se que o *slot* representa uma dada funcionalidade presente na entidade real modelada. Este conceito vai ser descrito num ponto separado.

Uma das características mais interessante deste paradigma é a possibilidade das classes conterem, através do frame que as representa, além das descrições, protótipos que serão utilizadas nas suas instâncias. Estas descrições são importantes para o conjunto de objectos representados na classe. Em termos práticos está-se a dizer que o paradigma suporta a noção de **meta conhecimento**, que ao ficar associado a cada frame que representa uma classe, corresponde à noção de meta conhecimento por cada classe.

Esta noção é bastante importante e permite a modelação de determinados conceitos de uma forma elegante. Veja-se o caso, por exemplo, de uma célula robotizada. Considerando que existem vários tipos de células (montagem, pintura, armazenamento, ...), cada uma delas é uma especialização do conceito mais geral célula (ver relações). O modelo de cada uma deve incorporar um dado tipo de conhecimento que é relevante para todos os membros da classe, considerados como um todo, mas que se apresenta irrelevante quando se consideram cada um dos membros da classe (instâncias) individualmente.

A informação necessária para configuração de cada instância de célula é um exemplo de meta conhecimento que se deve associar aos "frames" que representam a sua classe. O tipo de componentes válidos para realizar a entrada de uma célula de pintura, não são os mesmos para, por exemplo, um célula de montagem. Este conhecimento será representado também através de slots, mas com uma interpretação semântica distinta dos slots destinados a representar conhecimento relevante apenas para as instâncias.

Sejam as seguintes representações de célula de pintura e de montagem:

```
FRAME célula
nome:
coordenadas_base:
...
FRAME celula_montagem
is-a: célula
val-inp-ag: panela_vibra, buffer, alim_gravítico, mesa_index, agv,
tapete
val-out-ag: tapete, agv, buffer, mesa_index
val-proc-ag: robô
FRAME celula_pintura
is-a: célula
val-inp-ag: buffer, agv, tapete
val-out-ag: tapete, agv, buffer
val-proc-ag: robô
```

Os *slots val-inp-ag*, *val-out-ag* e *val-proc-ag* descrevem metaconhecimento. Destinam-se a auxiliar a criação de instâncias de "*frames*" do tipo referido, que, quando consideradas individualmente, não necessitam deste tipo de informação. O sistema de "*frames*" deve suportar uma semântica para a criação, remoção ou alteração deste tipo de *slots*.

Os slots *coordenada_base* e *nome* são relevantes (informação de protótipo) para as instâncias e, tal como os anteriores, o sistema deve apresentar uma semântica própria para a sua criação, remoção ou alteração.

O sistema de "*frames*" KEE™ resolve a diferença semântica entre os dois tipos de conhecimento através da existência de duas qualidades de *slots*: "*own slots*" e "*member slots*" [27]. Os primeiros descrevem o metaconhecimento enquanto que os segundos descrevem o geral.

O sistema KC™ resolve a questão do meta conhecimento pela possibilidade de associar um "*frame*" de meta conhecimento a qualquer "*frame*" existente no sistema [28]. Neste sistema *val-inp-ag*, *val-out-ag* e *val-proc-ag* seriam *slots* de um meta frame (*pintura_metaconhecimento* ou *montagem_metaconhecimento*) que ficaria associado ao schema *celula_pintura* ou *celula_montagem*, através, por exemplo, do comando *(attach-meta-schema 'celula_pintura 'pintura_metaconhecimento)*.

Ao nível da verificação de tipos, as linguagens de "*frames*" são, de um modo geral, pobres. Mas a ausência total de verificação de tipos conduziria a situações insustentáveis ao nível da manutenção da consistência, por exemplo, da base de conhecimento. Daí que algumas linguagens forneçam métodos para manter a integridade do sistema[23]. O KEE™ e o KC™, por exemplo, integram algumas facilidades que permitem restringir o número de valores que um dado *slot* pode conter[24], bem como as classes a que esses valores podem pertencer[25].

No caso do "*frame" robô* pode-se indicar que o slot garra pode conter 0 ou 1 valores, já que o robô pode não ter ferramenta corrente ou então possuir, no máximo, uma. Pode ainda definir-se que esse valor, apenas pode pertencer à classe garra.

As facilidades providenciadas pelo KEE™ para lidar com este problema são os construtores *CardinalityMin*, *CardinalityMax* e *ValueClass*., que se declaram no momento em que se define um *slot*. Os dois primeiros construtores destinam-se a restringir o número de valores que cada slot pode conter. *CardinalityMin* > 0 indica que o slot tem obrigatoriamente de conter pelo menos um valor, enquanto que *CardinalityMax* indica o número máximo de valores.

ValueClass, por seu lado, indica a classe a que podem pertencer os valores do *slot* (domínio + intervalo de valores), através da atribuição de uma expressão booleana. Desta forma é possível que os diferentes valores atribuídos pertençam a classes distintas. Com a expressão booleana pode-se construir o domínio juntamente com o intervalo, como, por exemplo, na seguinte expressão:

```
(INTERSECTION INTEGERS (INTERVAL 0 100) (NOT .ONE. OF 23 36))
```

em que se define que os valores possíveis para o slot são todos os inteiros no intervalo [0, 23[,]23,36[e]36, 100].

O CRL™ fornece também um conjunto de facilidades semelhantes que são declaradas num "schema" chamado *slot-control schema* e que tem o mesmo nome do *slot*. Os *slots* mais importantes declarados no *slot-control schema*, relacionados com o tema em análise, são: *Domain*, *Range* e *Cardinality*.

---

[23] Deve-se notar que o facto das linguagens providenciarem algumas facilidades não as torna tipificadas já que continuam a permitir que o utilizador não atribua classes (tipos) aos valores dos *slots*. Ao não obrigar o programador a definir tipos para os *slots* a linguagem permite brechas na sua integridade.

[24] De notar que nos sistemas de frames um *slot* pode conter mais do que um valor.

[25] Esta noção está directamente relacionada com a noção de tipo e sua verificação, já que ao restringir o universo de valores para o *slot*, está-se a criar um tipo e a possibilitar a sua verificação.

Domain destina-se a conter o domínio de valores que o slot pode incluir. *Range* contém o intervalo de valores, dentro do domínio, válidos para o *slot*. Finalmente, *Cardinality* restringe o número de valores que um *slot* pode conter.

Nas operações sobre os *slots* (afectação, remoção), é da responsabilidade da semântica da própria linguagem, verificar se o valor pertence ao domínio, está dentro do intervalo válido e ainda se a cardinalidade do slot não vai ser violada.

Relações

Neste paradigma existem vários tipos de relações que se podem estabelecer entre "frames", nomeadamente as definidas no capítulo anterior **generalização** (is-a), **agregação** (part-of) e **associação**.

Como já se referiu as relações do tipo **agregação/composição** existem a partir do momento que um atributo é afectado com o identificador de um outro frame, que é o que acontece quando o *slot garra* da instância *robô_1* é afectada com o identificador *garra_1*. O objecto é agora uma parte do objecto *robô_1*. De notar que um dado objecto, em tempos diferentes, pode ser parte de mais do que um objecto. Caso houvessem dois robôs a partilhar o mesmo conjunto de ferramentas, a *garra_1* poderia ser, exclusivamente, parte do *robô_1* ou parte do *robô_2*.

As relações de **generalização/especialização** destinam-se, tal como na POO, a relacionar classes, neste caso, relacionam "*frames*" que são protótipos de entidades específicas (outros "*frames*" também). Os construtores providenciados pelas linguagens de "frames" permitem a organização das "*frames*" de uma forma taxonómica em que cada classe (frame) pode ser vista como uma subclasse (especialização) ou super classe (generalização). O "*frame*" robô pode ser encarado como sendo uma especialização de uma classe mais geral *máquina_transformadora*, possuindo os atributos que o tornam distinto das outras máquinas de transformação. Esta relação é suportada à custa do conhecido mecanismo de herança, que permite que um protótipo subclasse seja constituído por protótipos definidos em classes mais gerais.

A herança implementada nestas linguagens segue a semântica tradicional ***is_a***, havendo alguma divergência quanto ao modo como deve ser interpretada.

O KEE™ e o CRL™ seguem a interpretação tradicional da POO para a relação *is-a*, isto é, seguem a noção de classe/subclasse ou tipo/subtipo. Assim todos os atributos definidos nas classes serão herdados pelas subclasses. Tal como na POO, a herança não está limitada a um nível. Em ambos os sistemas de "frames" existe a possibilidade de um frame ser uma especialização de mais do que uma classe - herança múltipla.

No KEE™, a implementação da relação de generalização/especialização é obtida através do construtor *Superclasses* enquanto que no CRL™ obtém-se através da utilização de uma relação definida no sistema, denominada *is-a*.

As relações de **associação** também se destinam a relacionar "frames", mas com uma interpretação semântica distinta da generalização/especialização, dado que não se destina a implementar a noção de subclasse/subtipo. Caiem dentro deste tipo todas as relações que o programador pretender definir entre diferentes classes. A noção de relação definida pelo programador apresenta-se bastante útil ao nível da compreensão do sistema. De uma forma sucinta, pode-se afirmar que a definição de relações, permite ao programador escolher a semântica do mecanismo de herança, associada a qualquer relação.

Por vezes torna-se complicado decidir entre a utilização de uma relação de agregação/composição e uma de associação. A diferença entre os dois tipos situa-se maioritariamente ao nível da interpretação semântica, devendo-se utilizar uma relação de agregação entre um objecto A para outro B, sempre que o objecto B for nitidamente uma parte do objecto A.

Nos outros tipos de relacionamento entre "frames" devem-se utilizar relações de associação, sendo a semântica da relação, obtida à custa da definição do mecanismo de herança que suporta a relação. Nalgumas linguagens de "*frames*", como por exemplo o CRL™, pode-se estabelecer, nesta definição, qual o domínio e

intervalo de valores estabelecidos para a herança, os slots que irão ser herdados (*inclusion*), os que poderão ser restringidos (*exclusion*) ou ainda a forma como os valores do slot irão ser alterados durante a herança (*map*).

Vai-se apresentar um exemplo, descrito em CRL™, em que se mostra o estabelecimento da relação *controlado_por*, definida pelo utilizador, entre a classe *robô* e a classe *controlador_de_robô* .

```
FRAME robô
   is-a: robô_component
   controlado_por: controlador_robô
   ...
FRAME controlador_robô
   is-a: controladores
   hardhome: method hardhome_fn()
   ...
FRAME controlado_por
   is-a: relation
   inclusion: controlador_robô_inc_spec
   ...
FRAME controlador_robô_inc_spec
   instance: inclusion-spec
   type: slot
   slot-restriction: (or hardhome ...)
```

O frame *robô* vai herdar os *slots* do frame *controlador_robô*, definidos na relação *controlado_por* através do slot *slot-restriction* do frame *inclusion-spec*.

2.3.2.2. Aspectos Comportamentais

Como se disse no início deste capítulo, as linguagens de "frames" não suportam descrição declarativa do modo como o conhecimento se há-de comportar, isto é, não é possível, de uma forma declarativa, modelar o comportamento dos "frames", vistos como um todo; mas será possível criar uma estrutura que represente uma parte do comportamento de cada frame.

Considerando que um dado frame representa uma classe e considerando a visão de tipo de dados abstracto, que considera que um tipo é caracterizado pela sua estrutura de dados (slots de valores) mais o conjunto de propriedades que governam o tipo (comportamento), conclui-se que um "*frame*" pode representar um tipo de dados abstracto desde que possibilite o encapsulamento de procedimentos que descrevem as propriedades referidas. Estes procedimentos não são mais do que subrotinas que são invocados sob o controlo do "*frame*". Está-se obviamente muito próximo da aproximação seguida pela POO.

O facto da aproximação tradicional, seguida nas linguagens de "*frames*", não considerar cada frame como um tipo de dados abstracto (ADT), mas sim uma estrutura que descreve determinado tipo de conhecimento, que pode ser adequadamente representado por rotinas, leva a que a sua semântica comportamental seja menos restritiva do que se seguisse a primeira aproximação. A aproximação ADT obriga a que o acesso aos atributos seja feita por métodos, o que não é norma nas linguagens de "*frames*".

Nos sistemas de "*frames*" existem duas formas de representar o comportamento: **métodos** e **demónios** [21]. Os métodos são rotinas que se atribuem ao frame enquanto que os demónios são rotinas que se associam a *slots*, cuja actuação depende do tipo de operação realizada sobre esses mesmos *slots*.

Métodos

Métodos são procedimentos encapsulados no frame em que são declarados, procurando assim representar funcionalidade presente na entidade que é modelada por esse mesmo frame. Numa perspectiva do paradigma da programação orientada por objectos, pode-se dizer que os métodos são os destinatários das mensagens que se enviam para os "*frames*". Apenas se podem enviar mensagens que tenham um correspondente método receptor.

Os métodos são activados através do envio de uma mensagem para o frame que o define. Será então necessário declarar o receptor (geralmente um slot) bem como a função que é chamada quando do envio da

mensagem (código do método). Conclui-se então que os métodos são constituídos por duas entidades fundamentais: receptor e a respectiva implementação.

As acções descritas nos métodos podem também ser encaradas como responsáveis pela alteração do frame, considerando-se estado como sendo o conjunto de valores presentes nos *slots* do "*frame*", num dado momento. Esta visão encara a funcionalidade, com o seu conjunto de métodos, como primitivas disponíveis para alterar o estado do objecto, devendo, por isso, a implementação provocar a alteração de valores presentes nos *slots* do "*frame*". Considerar que o acesso aos *slots* se faça através de métodos (como na POO) é uma tentativa de implementar "*information hiding*" que não é suportada pelas linguagens de "*frames*", uma vez que o utilizador pode sempre aceder ou alterar os slots através das primitivas de manipulação.

Mas os métodos não devem apenas ser vistos como destinados a alterar o estado do frame de uma forma directa. A utilização de métodos continua a fazer sentido mesmo que esses métodos não alterem nenhum *slot*. Pode-se encarar um "*frame*" como tendo apenas métodos, sem nenhum atributo. Uma "*frame*" deste tipo modela uma entidade cujas acções físicas são actuadas através de métodos, não sendo o seu estado importante.

Um bom exemplo da utilização de métodos para descrever a funcionalidade de entidades pode ser encontrado no caso da modelação de um controlador de robô. Este controlador pode ser visto como um conjunto de primitivas que provocam um determinado comportamento no robô físico. Assim o seu modelo pode ser descrito através de um conjunto de procedimentos que virtualizam esse comportamento.

Os métodos apresentam-se como um mecanismo bastante adequado para modelar componentes controlados por sistemas automáticos (controladores), já que se torna possível virtualizar nos métodos, a funcionalidade do controlador local. A virtualização é conseguida através da ligação entre o modelo e o controlador real (ver capítulo sobre modelação).

No KEE™, os métodos são procedimentos LISP que respondem a mensagens enviadas para o frame. Os receptores são *slots* do tipo *method*. Os valores atribuídos a estes *slots* identificam o nome do procedimento LISP. A mensagem que é enviada para o frame inclui os respectivos argumentos da implementação do método.

Seja o seguinte modelo de um controlador de robô em KEE™:

```
Unit: controlador_ROBÔ
Superclasses: controladores

MemberSlot:HARDHOME
   ValueClass: METHODS
   Cardinality.Min: 1
   Cardinality.Max: 1
   Values: HARDHOME_FN
```

A activação do método que irá realizar o hardhome do robô é feito através do envio da mensagem para o *slot HARDHOME*, provocando a activação do procedimento *HARDHOME_FN*.

Em CRL™ os métodos são também procedimentos LISP que respondem a mensagens enviadas para o "*frame*". Tal como no KEE™, os *slots* de um "*frame*" podem conter valores ou nomear o nome de uma implementação de um método. A implementação do frame *controlador_robô* seria descrita em CRL™ da seguinte forma:

```
 (defschema controlador_robô
   (is-a controladores)
   (hardhome hardhome_fn) )

(defschema hardhome
   (is-a method) )

(defun hardhome_fn () ... )
```

Defchema é uma macro fornecida pela linguagem para criar um frame. *Controlador_robô* foi criado com um método *hardhome*. A implementação está definida em *hardhome_fn* enquanto que o receptor da mensagem será o *slot hardhome*.

A activação do método será feita pelo comando:

```
(call-method 'controlador_robô 'hardhome)
```

Demónios

Tal como foi dito, o demónio é uma outra forma de representar comportamento mas com uma semântica bastante afastada da POO.

O demónio é um conceito que aparece como uma particularidade de um outro conceito mais vasto: **programação reactiva**. Não se deve confundir a programação reactiva com o paradigma da programação por "frames", já que são conceitos diferentes, não sendo propriedade exclusiva dos "*frames*", apesar de se ter desenvolvido no seu contexto. A programação reactiva pode ser adaptada à programação procedimental (Pascal, C) ou à programação declarativa (Prolog, Lisp).

Antes de começar por definir explicitamente o conceito de demónio vai-se começar por apresentar os conceitos gerais que lhe estão subjacentes.

Este conceito fundamenta-se na lei básica da física: o princípio da acção-reacção, que diz que qualquer objecto que sofra uma acção reage através de uma reacção. Isto é, quando se sujeita um objecto a uma qualquer acção existe um efeito lateral que é governado pela reacção do objecto à acção a que esteve sujeito. Transferindo a noção para o mundo dos "*frames*", há que definir o seguinte:

- quais os objectos que podem sofrer acções
- quais as acções
- quando se dá a reacção
- como se dá a reacção

Os objectos que podem sofrer acções são os *slots*; as acções que podem ser desencadeadas são as de manipulação (leitura/escrita/execução); o modo como o *slot* reage ao ser manipulado será implementado pelo demónio que fica atribuído ao *slot*, ficando em aberto se a reacção se dá antes ou depois da operação que lhe deu origem.

O demónio surge então como a reacção à manipulação do *slot*, como se fosse uma acção demoníaca, daí derivando o seu nome.

A programação reactiva aparece então como um conceito de programação em que o programa evolui por desencadeamentos de acções. Cada acção provoca uma reacção que pode desencadear nova(s) acção(ões). O programa vai assim evoluindo até que não haja nenhuma reacção. Do ponto de vista da estruturação é um paradigma interessante, mas pode ser bastante difícil de ler e interpretar. Pode também não ser muito fácil realizar operações de "*debugging*".

Uma aplicação muito interessante para este paradigma consiste em utilizar os demónios para garantir a consistência dos valores nos *slots*, funcionando, neste caso, como guardas.

É necessário clarificar as operações sobre *slots* a que os demónios estão associados. À partida podem definir-se duas operações sobre os *slots*: leitura e escrita conduzindo ao aparecimento dos seguintes tipos de demónios:

- ***if_read*** - desencadeia-se sempre que se realizar uma operação de leitura sobre o slot
- ***if_write*** - desencadeia-se sempre que se realizar uma operação de escrita sobre o slot
- ***if_needed*** - É um caso particular do *if_read* e é desencadeado apenas quando o slot que está a ser lido se encontra vazio, isto é, sem nenhum valor.

A utilização de demónios nos "*frames*" que modelam componentes de produção mostra-se bastante interessante para descrever o comportamento desses mesmos componentes. As variáveis de controlo, como por exemplo no caso de um robô, posição, velocidade, aceleração, etc, podem ser facilmente representadas por slots, ficando a ligação entre os slots (variáveis de controlo no ambiente do modelo) e as variáveis de controlo do componente físico escondida em cada um dos demónios. Desta forma o modelo de um controlador, por exemplo, é representado por um "*frame*" em que se definiram *slots*, com demónios associados, que representam as variáveis de controlo.

Em CRL™ o frame *controlador_robô* poderia ser definido da seguinte forma:

```
(defschema controlador_robô
  (is-a controladores)
  (posição '(0 0 270) )
  (velocidade 20)
  (aceleração 1) )

(defschema posição
  (is-a slot)
  (demon posição-dem) )

(defschema posição-dem
  (instance demon)
  (access put-value)
  (effect alter-value)
  (when before)
  (action posição_fn) )

(defschema velocidade
  (is-a slot)
  (demon velocidade-dem) )

(defschema velocidade-dem
  (instance demon)
  (access put-value)
  (effect alter-value)
  (when after)
  (action vel_fn) )

(defschema aceleração
  (is-a slot)
  (demon aceleração-dem) )

(defun posição_fn () ... )
```

Este código pretende apenas realçar algumas das características do CRL™ no que diz respeito à programação reactiva. Os *slots* com demónios devem ser declarados, como acontece nos "*frames*" *posição*, *velocidade* e *aceleração*. Cada um destes "frames" declara o nome da frame que caracteriza o demónio, neste caso *posição-dem*, *velocidade-dem* e *aceleração-dem*. Para que o demónio fique caracterizado é preciso: definir a operação que o desencadeia (access), sendo os valores possíveis *put-value*, *get-value* ou *delete-value*.; a altura em que é desencadeado define-se atribuindo ao *slot when* o valor *after* ou *before*, consoante se deseja que seja actuado depois ou antes, respectivamente, do slot ser actualizado; e especificar as consequências/resultado do demónio através do *slot effect*.

O CRL™ não fica limitado aos três tipos de demónios (if_read, if_write e if_needed) apresentados atrás.

2.3.3. Conclusão

De uma forma breve podem-se sumariar as diferenças entre o sistema de "frames" e a POO nos seguintes pontos:

- O paradigma de "frames" possui vários tipos de mecanismos de herança.

- Os slots podem conter múltiplos valores no paradigma de "*frames*", enquanto que na POO, normalmente, possuem apenas um.
- É normal encontrar no paradigma de "*frames*" demónios associados aos slots (programação reactiva), que normalmente não aparece associado às linguagens tradicionais de objectos.
- A definição de relações pelo utilizador é normalmente suportada pelos sistemas de "frames".
- Normalmente os sistemas de "frames" possibilitam de alteração dinâmica dos "frames".

2.4. Redes de Petri

2.4.1. Introdução

A Rede de Petri (RdP) é uma ferramenta de modelação desenvolvida pelo matemático alemão Carl Adam Petri [29], criada originalmente para o estudo de sistemas concorrentes. Dada a sua adequabilidade para a modelação de sistemas, tem merecido, ao longo do tempo, a atenção de diversas equipas de investigação [30]. Alguns projectos têm contribuído para o seu desenvolvimento e normalização [31]. O seu uso tem sido mais generalizado na Europa que nos Estados Unidos, apesar de um dos trabalhos mais notáveis sobre RdP ter sido realizado no MIT, que desenvolveu um projecto sobre o uso de RdP na análise dos aspectos de controlo de computadores [32-34].

O facto de permitir a representação de conceitos com um nível de abstracção elevado torna-a numa ferramenta indicada para modelar sistemas dinâmicos complexos. Nesta forma de representação (modelação) importa realçar 2 aspectos que demonstram a importância desta ferramenta: (1) a representação visual do sistema e (2) o fundamento teórico em que assenta o seu desenvolvimento.

Quando se descreve um sistema, utilizando uma RdP, a visualização do comportamento do sistema surge naturalmente e, um qualquer leitor que conheça as regras do formalismo, pode apreender as noções comportamentais do sistema a descrever. Um outro pormenor, associado à forma de representação do sistema, é a possibilidade de extrair informação acerca dos aspectos estruturais do mesmo. Quer então dizer que o formalismo RdP não só modela os aspectos comportamentais (parte mais conhecida) como também os aspectos estruturais.

O facto da RdP ter por base um fundamento teórico, permite que o sistema a ser modelado, seja validado matematicamente. Este facto é extremamente importante e é uma das razões do êxito da ferramenta porque, a possibilidade de realizar uma análise qualitativa sobre um sistema complexo, que de outra forma seria intratável, representa um grande avanço para o estudo da previsão do comportamento dos sistemas.

A RdP, pelas características referidas, é indicada para a representação de sistemas dinâmicos e, como tal, tem sido utilizada na modelação dos mais variados tipos de sistemas: software, hardware, sistemas químicos, sistemas legais, sistemas de comunicação, cálculo preposicional, robótica, planos de montagem, etc [30, 35-37].

A importância do formalismo RdP, no contexto desta tese, deriva da sua adequabilidade para a representação de sistemas dinâmicos, no qual se insere, sem qualquer dúvida, a representação de uma célula robótica. Este tipo de ambiente, nitidamente concorrente e complexo, actuado por eventos externos e, muitas vezes, assincronamente, parece ser ideal para a utilização de uma ferramenta com as características da RdP.

2.4.2. Noções Básicas

Uma RdP é constituída por **lugares**, **transições** e **arcos** [38]. Os lugares significam estados do sistema e são representados por círculos. As transições significam condições (eventos) que ocorrem e são representadas por barras. Os arcos ligam as transições aos lugares e vice-versa. Cada arco tem de ter, em cada extremidade, um lugar e uma transição.

Formalmente uma RdP pode ser vista como um tuplo **Rp = (P, T, I, O)**, em que P representa o conjunto finito de lugares, T o conjunto finito de transições, I , por cada transição, o conjunto de lugares que são entrada dessa transição e O, também por cada transição, o conjunto de lugares que são saída dessa transição.

Na figura 2.9 mostra-se um exemplo de uma RdP que modela o sistema de controlo de enchimento de um tanque. Este sistema é caracterizado por dois lugares que representam os dois estados possíveis do sistema: motor ligado e motor desligado. A transição do estado motor ligado para o estado motor desligado dá-se quando o tanque estiver cheio e a passagem de desligado para ligado dá-se quando o tanque estiver vazio. Este exemplo pretende apenas apresentar os primeiros conceitos associados às RdP, realçando a noção de estado para os lugares e a noção de transição como sendo a condição de mudança de estado. Pela análise da figura constata-se imediatamente quais os possíveis estados do sistema e o modo como evoluem.

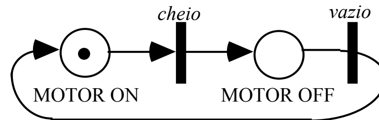


Figura 2.9 - Exemplo de Rede de Petri

A descrição básica da RdP não fica completa enquanto não for introduzido o conceito de marca. A marca indica qual ou quais os estados que estão activos. As marcas indicam qual o estado corrente do sistema. O sistema representado na figura 2.9, é constituído por 2 estados possíveis, mas o estado actual do sistema, é MOTOR ON. Num sistema complexo, vários estados podem estar activos. Neste caso, o estado actual do sistema é representado pelo conjunto de estados com marcas - marcação da rede.

A introdução do conceito de marca obriga à redefinição do tuplo para se introduzir a marcação. Uma rede de Petri marcada será então definida por: **Rpm = (P, T, I, O, μ).**

A RdP da figura 2.9 pode ser caracterizada da seguinte forma:

P = { MOTOR_ON, MOTOR_OFF }

T = { cheio, vazio }

I(cheio) = { MOTOR_ON } **I(vazio) = { MOTOR_OFF }**

O(cheio) = { MOTOR_OFF } O(vazio) = { MOTOR_ON }

$$\mu = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Cada local pode conter um número de marcas maior ou igual a zero, sendo este número portador de uma determinada semântica de estado. O número de marcas de um lugar i é dado por $\mu(P_i)$ ou simplesmente m_i. No exemplo da figura 2.4.1 tem-se m1=1 e m2=0. A rede de marcas designa-se por μ e pode ser representada por (m_1, m_2, m_3, ..., m_n), sendo μ = (1, 0) para o exemplo da figura 2.9.

A marcação da rede caracteriza o estado do sistema. O conjunto de marcações representa os estados possíveis desse sistema. No caso da figura 2.9 μ pode tomar os valores μ1 = (1, 0) e μ2 = (0, 1). A passagem de um estado para outro é condicionada pelo disparo das transições.

Uma transição dispara[26] quando estão criadas determinadas condições. Uma transição apenas pode disparar se estiver habilitada, o que acontece se todos os lugares ligados à transição tiverem pelo menos uma marca. Apesar de poderem haver várias transições habilitadas simultaneamente, de cada vez, apenas pode disparar uma,

[26] Diz-se que uma transição dispara quando ocorre.

isto é, não há simultaneidade no disparo. Associado a este conceito aparece também a noção de indivisibilidade da transição, que garante que, a partir do momento que se inicie, apenas pode terminar quando todas as marcas tiverem sido actualizadas. Quando se dá o disparo, todos os lugares aos quais está ligada a transição, recebem mais uma marca, enquanto que os lugares que estavam ligados à transição perdem, cada um deles, uma marca.

Na figura 2.10 mostram-se alguns exemplos de disparos, com diferentes situações de marcas. Pode-se verificar a situação em que apesar de as 2 transições estarem habilitadas (t1 e t2), apenas se deu o disparo de t1.

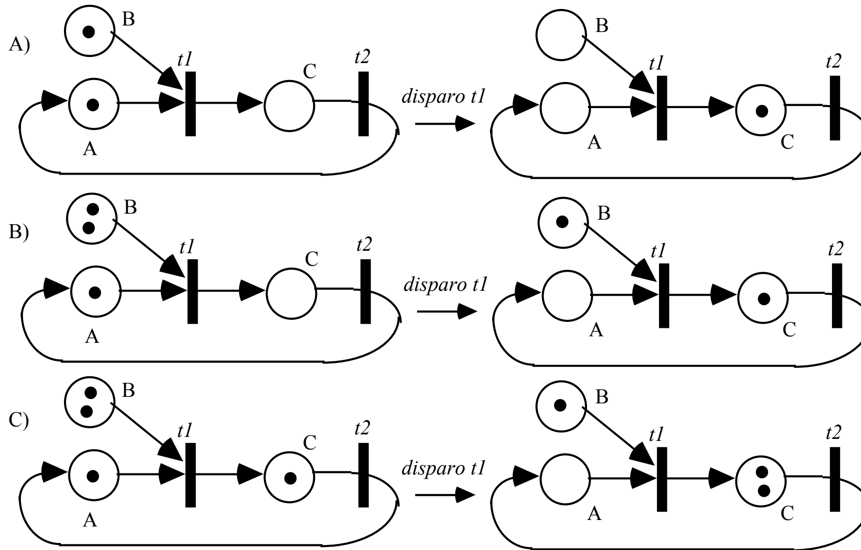


Figura 2.10 - Exemplos de Disparo de RdPs

As marcas de uma RdP representam-se através de um **vector coluna**. A rede de Petri da figura 2.4.4 representa-se da seguinte forma: $\mu_0 = [0, 1, 3, 0, 0]^T$. A **matriz de incidência** (W) representa a estrutura da RdP, representando as colunas as transições enquanto que as linhas representam os locais. No caso novamente da figura 2.12 tem-se a seguinte matriz de incidência:

$$W = \begin{bmatrix} \mathbf{t1} & \mathbf{t2} & \mathbf{t3} & \mathbf{t4} \\ 0 & 0 & -1 & +1 \\ -1 & +1 & +1 & -1 \\ -1 & 0 & +1 & 0 \\ +1 & -1 & 0 & 0 \\ 0 & +1 & 0 & -1 \end{bmatrix} \begin{matrix} \mathbf{P1} \\ \mathbf{P2} \\ \mathbf{P3} \\ \mathbf{P4} \\ \mathbf{P5} \end{matrix}$$

A matriz W descreve o comportamento das marcas, para cada lugar, quando se dão as transições. Por exemplo, quando se dá a transição t3, é retirada uma marca ao local P1 (RECOLHA) e adiciona-se uma marca aos locais P2 (ROBOT_LIVRE) e P3 (PAL_ARMAZEM). Os outros locais não sofrem qualquer alteração.

A partir de um dado estado de marcas pode obter-se uma dada sequência de disparos, correspondendo a cada sequência um vector característica $\underline{S}$. A partir da figura 2.12 pode-se obter as seguintes sequências S_1 = t1t2t1t2, S_2 = t4t3t1, S_3 = t4t3t1t2t1 e ainda muitas outras. Para cada uma destas sequências têm-se os seguintes vectores características: $\underline{S_1} = (2, 2, 0, 0)$, $\underline{S_2} = (1, 0, 1, 1)$ e $\underline{S_3} = (2, 1, 1, 1)$. Os números indicam o número de transições que ocorreram, para cada sequência. Por exemplo, $\underline{S_1}$ indica que ocorreram 2 transições t1, 2 t2 e nenhuma t3 e t4.

Um dado estado de marca μ_k é obtido através da seguinte equação fundamental:

$$M_k = M_i + W.\underline{S}$$

Supondo a sequência S = t1t2t1t2t1t2, obtém-se $\underline{S} = (3, 3, 0, 0)$. Sendo $\mu i = [0, 1, 3, 0, 1]$ e aplicando a fórmula anterior obtem-se $\mu k = [0, 1, 0, 0, 4]$, que está de acordo com o que seria expectável.

Diz-se que um vector X é P-invariante se $X^T.W = 0$, deduzindo-se $X^T.\mu_k = X^T.\mu_0$. O vector X representa os lugares que se estão a testar como invariantes. A expressão $X^T.\mu_k = X^T.\mu_0$ significa que, no caso do vector X ser P-invariante, o número de marcas se mantém constante para qualquer estado. Seja, por exemplo a figura 2.12, em que se vai verificar se existe um invariante associado a m1, m3, m4 e m5. O vector X será então [1, 0, 1, 1, 1].

$$[1\ \ 0\ \ 1\ \ 1\ \ 1] \bullet \begin{bmatrix} 0 & 0 & -1 & +1 \\ -1 & +1 & +1 & -1 \\ -1 & 0 & +1 & 0 \\ +1 & -1 & 0 & 0 \\ 0 & +1 & 0 & -1 \end{bmatrix} = [0\ \ 0\ \ 0\ \ 0]$$

Verifica-se que o vector X é invariante. Para se saber o valor do invariante basta realizar a operação $X^T.\mu_0$ que dá 4, sendo que m1+m3+m4+m5 = 4. Esta é uma forma expedita verificar se uma rede é conservativa.

Um vector Y diz-se T-invariante se W.Y = 0. O vector Y representa o número de disparos que cada transição realizou. Quando existir um S que seja igual a Y, então a rede será repetitiva. Recordando novamente o exemplo da figura 2.12, verifica-se que a rede é repetitiva, já que para um vector Y = [1, 1, 1, 1] se obtem W.Y = 0 e existe um S = (1, 1, 1, 1) que corresponde à sequência de transição t1t2t4t3.

2.4.2.1. Exemplo 1

Pretende-se modelar um sistema de controlo de um tapete integrado numa sequência, isto é, o tapete está inserido entre dois outros tapetes, destinados a transportar paletes. O tapete pode estar num dos seguintes estados possíveis: livre, com palete em entrada, com palete parada e com palete em saída. O tapete dá entrada a uma palete desde que esteja livre e o tapete anterior esteja em saída, enquanto que dá saída desde que o tapete seguinte esteja livre.

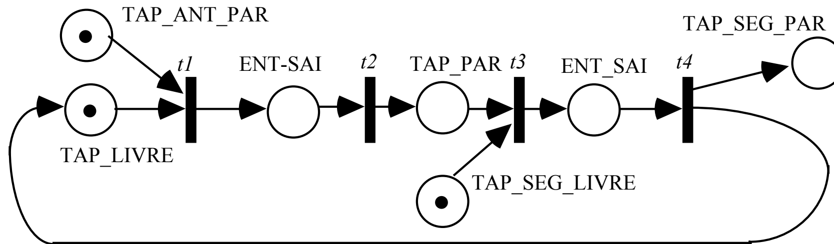


Figura 2.11 - Controlo de Tapete

Inicialmente o tapete está livre (1 marca no lugar TAP_LIVRE), o tapete anterior tem palete parada sobre ele (1 marca no lugar TAP_ANT_PAR) e o tapete seguinte está também livre, sendo μ = (1, 1, 0, 0, 1, 0, 0). Por transição de t1 obtêm-se μ = (0, 0, 1, 0, 1, 0, 0) que indica a situação do tapete seguinte se manter livre, o corrente a dar entrada à palete e o anterior em saída (uma marca em ENT-SAI indica que o tapete anterior está em saída enquanto que o corrente está em entrada). Quando a palete está totalmente posicionada no tapete, ocorre t2 e o tapete fica parado com palete, neste caso, μ = (0, 0, 0, 1, 1, 0, 0). Dado que o tapete seguinte está livre, ocorre t3 e o tapete passa a estar em saída, enquanto que o seguinte está em entrada (1 marca em ENT-SAI), sendo μ = (0, 0, 0, 0, 0, 1 , 0). Quando a palete está totalmente posicionada no tapete seguinte ocorre t4, ficando o tapete livre e o seguinte parado, em que μ = (0, 1, 0, 0, 0, 0, 1).

Se por alguma razão o tapete seguinte não ficar livre e o anterior tiver fornecido uma palete, o sistema fica no estado μ = (x, 0, 0, 1, 0, 0, 1), aí permanecendo até que o tapete seguinte esteja livre. O x em μ indica que a marca para a posição pode ser 1 ou 0, porque é irrelevante se o tapete anterior está com palete parada ou não.

Esta RdP é **não autónoma** já que existe a necessidade de sincronizacão externa (lugares TAP_SEG_LIVRE e TAP_ANT_PAR).

O conjunto de **marcações atingíveis**, a partir de μ0 é { μ1, μ2, μ3 }, em que μ0 = (1, 1, 0, 0, y, 0, z), μ1 = (0, 0, 1, 0, y, 0, z), μ2 = (x, 0, 0, 1, 1, 0, 0), μ3 = (x, 0, 0, 0, 0, 1, 1), em que x, y e z representam valores possíveis de 0 ou 1.

Desde que os lugares externos (TAP_SEG_LIVRE E TAP_ANT_PAR) sejam alimentados com marcas, a rede é **permanente**, uma vez que nenhuma transição fica sem disparar.

Existe pelo menos uma **marca invariante**: $\mu_i(P_2) + \mu_i(P_3) + \mu_i(P_4) + \mu_i(P_6) = 1$. A rede é **não conservativa** já que não existe um invariante que englobe todos os lugares.

Dado que as marcas de cada lugar apenas têm o valor 0 ou 1 - marcação booleana, diz-se que a rede é **segura**. Por esta razão é automaticamente **limitada**.

2.4.2.2. Exemplo 2

Pretende-se modelar um armazém automático servido por um robot. O armazém deve poder recolher e fornecer paletes. A recolha e entrega não podem ser simultâneas, uma vez que existe apenas um robot. No modelo deve aparecer o mundo exterior ao armazém, que poderá significar um sistema de transporte. Os estados possíveis deste sistema são: robot livre, em recolha, em entrega , palete armazenada e palete no exterior.

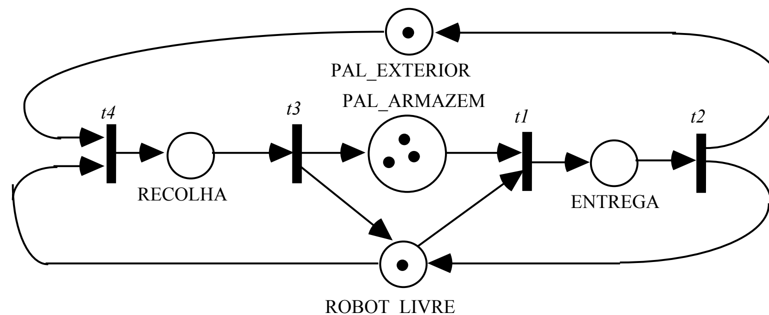


Figura 2.12 - Controlo de Armazém Automático

Este exemplo tem uma característica importante que é o facto de se poder saber, em qualquer altura, o número de paletes presente no armazém. Efectivamente, o número de marcas presente no lugar palete armazenada corresponde ao número de vezes que o estado ocorreu, sendo por isso igual ao número de paletes presente em armazém. Por outro lado, para haver entrega terá de haver, pelo menos, uma palete, e, sempre que ocorrer a transição para o estado entrega, o número de marcas em palete armazenada é diminuído de 1, obtendo-se que o número de marcas presente em palete armazenada corresponde ao número de paletes em armazém.

Inicialmente o robot está livre e existem 3 paletes em armazém e uma no sistema exterior, sendo μ = (0, 1, 3, 0, 1). Caso haja um evento externo correspondente a um pedido de recolha, a transição t4, que está habilitada, dispara e passa-se ao estado em que o robot está a fazer a recolha, com μ = (1, 0, 3, 0, 0). Nesta situação, mesmo que haja um pedido de entrega, a transição t1 não está habilitada e como tal nunca irá disparar, o que está de acordo com o que foi definido anteriormente sobre a não simultaneidade da recolha e entrega. Após a recolha, acontece a transição t3, ficando o robot novamente livre e mais uma palete em armazém. Neste ponto tem-se μ = (0, 1, 4, 0, 0). Caso ocorra agora, um pedido de entrega, o estado será μ = (0, 0, 3, 1, 0). Quando acontecer a entrega, dá-se a transição t2 e o sistema volta novamente a μ = (0, 1, 3, 0, 1).

Esta RdP é **autónoma** já que não existe a necessidade de sincronização externa.

O conjunto de **marcas atingíveis**, a partir de μ_0 é um número razoável já que o facto de existirem 3 marcas no lugar PAL_ARMAZEM condiciona o número de marcas possíveis. Quanto mais elevado for este número maior será a cardinalidade do conjunto marcas atingíveis.

A rede é **permanente**, uma vez que nenhuma transição fica sem disparar.

Existe pelo menos uma **marca invariante**: $\mu_i(P_1) + \mu_i(P_3) + \mu_i(P_4) + \mu_i(P_5) = 1$. A rede é n**ão conservativa** já que não existe um invariante que englobe todos os lugares. Mas é quase conservativa a menos do lugar ROBOT_LIVRE, o que é expectável já que o número de paletes presentes no sistema global tem de ser constante.

A rede é não **segura**, mas é **limitada**, já que o número de marcas é limitado para cada lugar.

2.4.2.3. Exemplo 3

Pretende-se modelar um sistema constituído por 2 AGVs, que se deslocam sobre carris e o objectivo final do sistema é garantir o transporte de um dado material, desde um ponto A até um ponto B. Cada AGV apenas percorrre metade do percurso, sendo por isso necessário fazer uma transferência de material de um AGV para o outro. Os estados que se podem encontrar neste sistema são: AGV1 a mover-se para a esquerda(AGVAESQ), AGV2 a mover-se para a esquerda (AGVBESQ), AGV1 a mover-se para a direita (AGVADTA), AGV2 a mover-se para a direita (AGVBDTA), AGV1 a carregar (AGVACAR), AGV2 a carregar (AGVBCAR), AGV1 a descarregar (AGVDESC) e AGV2 a descarregar (AGVBDESC).

A passagem do material, a meio do percurso, de um AGV para o outro faz-se através de uma operação de descarga do AGV1 e de uma operação de carga do AGV2. Quando o AGV2 chega ao destino realiza uma operação de descarga e volta ao ponto intermédio. O AGV1, por seu lado, após a passagem do material para o AGV2 volta à origem para realizar novo carregamento.

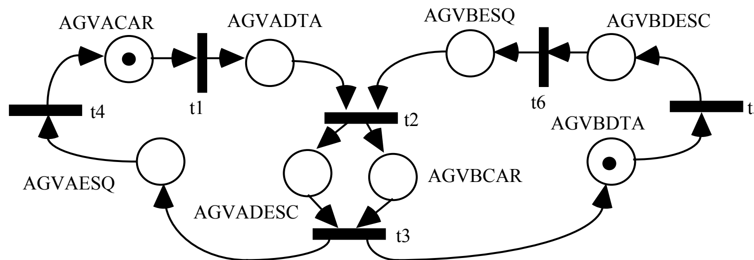


Figura 2.13 - Sistema com 2 AGVs

Inicialmente o AGVA está a carregar na origem, enquanto que o AGVB se está a deslocar para a direita para proceder ao descarregamento no destino, $\mu0 = (1, 0, 0, 0, 0, 1, 0, 0)$. Deste estado o sistema pode evoluir para 2 possíveis estados, dependendo da transição que tiver sido disparada. Se t1, indicando a situacão de fim de carga, o AGVA passa a deslocar-se para a direita, $\mu1 = (0, 1, 0, 0, 0, 1, 0, 0)$. Se t5, indicando que o AGVB chegou ao destino, passa-se à descarga do AGVB, $\mu2 = (1, 0, 0, 0, 0, 0, 1, 0)$. Caso o AGVA se mantenha em carregamento e o AGVB faça a sua descarga rapidamente, este pode passar para o estado de deslocamento para a esquerda, graças ao disparo de t6, $\mu3 = (1, 0, 0, 0, 0, 0, 0, 1)$. A evolução do sistema continua até que ambos os AGVs estejam a meio caminho, $\mu4 = (0, 1, 0, 0, 0, 0, 0, 1)$. Nesta altura faz-se a passagem do material, sendo $\mu5 = (0, 0, 1, 0, 1, 0, 0, 0)$. Existem 10 possíveis estados. A ocorrência de transições simultâneas não altera em nada a análise.

Esta RdP é **autónoma** já que não existe a necessidade de sincronizacão externa.

O conjunto de **marcas atingíveis**, a partir de μ_0 é $\{ \mu0, \mu1, \mu2, \mu3, ..., \mu9 \}$

A rede é **permanente**, uma vez que nenhuma transição fica sem disparar.

Existe pelo menos uma **marca invariante**: $\mu_i(P_1) + \mu_i(P_2) + \mu_i(P_3) + \mu_i(P_4) + \mu_i(P_5) + \mu_i(P_6) + \mu_i(P_7) + \mu_i(P_8) = 1$. A rede é **conservativa** já que existe um invariante que engloba todos os lugares.

A rede é **segura** e como tal é **limitada**.

O fenómeno da **concorrência** pode ser observado nesta RdP, por exemplo, quando um AGV está a deslocar-se para um lado e o outro para o outro. Também se pode observar a sincronização, através da necessidade de os 2 AGVs se deslocarem, no sentido de se encontrarem, para proceder à transferência do material.

2.4.3. Modelação Usando RdP

As RdP, podem ser classificadas quanto à sua autonomia, que define a independência da RdP face a eventos externos, classificando-se em autónomas e não autónomas [38].

2.4.3.1. Autónomas

Uma RdP autónoma é uma rede não condicionada por nenhum evento externo. Existem diversos tipos de redes autónomas cujo comportamento difere ligeiramente do referido anteriormente. A diferença corresponde, nalguns casos, à introdução de primitivas que tornam a rede mais legível (abreviação) e, noutros, à introdução de novas funcionalidades que enriquecem o modelo, tornando-o adequado para um número maior de casos (extensões).

Em relação às abreviações garante-se que as propriedades gerais se mantêm, não se podendo dizer o mesmo para as extensões, apesar de se manterem os conceitos básicos.

Nas redes autónomas descreve-se o fenómeno mas não quando acontece, dada a impossibilidade de condicionar as transições através de eventos externos.

Abreviações

Rede de Petri Generalizada

Uma rede de Petri Generalizada é uma RdP em que os arcos são caracterizados por possuírem pesos, que determinam a quantidade de marcas envolvidas no disparo [38]. Suponha-se, por exemplo, que o arco que liga um lugar a uma transição tem peso 3 e que o arco que liga essa transição a outro lugar tem peso 2. Para que a transição fique habilitada, é necessário que o lugar que lhe está ligado tenha 3 marcas. Quando se dá a transição, ao lugar destino adicionam-se 2 marcas enquanto que ao lugar origem se subtraem 3 marcas.

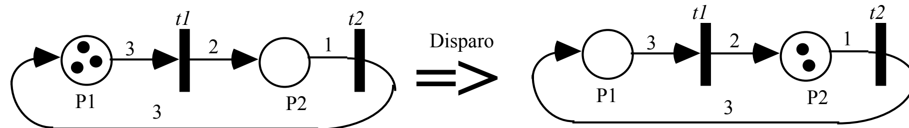


Figura 2.14 - Exemplo de Disparo para RdP Generalizada

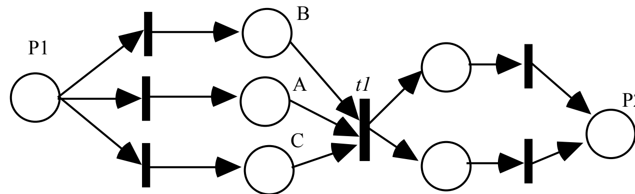


Figura 2.15- Transformação de RdP Generalizada da Figura 2.14 em Normal

Não é complicado fazer a transformação de uma RdP generalizada para uma RdP normal. Ver figura 2.15, onde se mostra como se transformou a RdP generalizada da figura 2.14 numa RdP normal. Mas esta transformação tem um interesse relativo já que as propriedades se mantêm, nomeadamente a equação fundamental, havendo apenas que redefinir a matriz W, por forma a que englobe o peso dos arcos. A matriz W que representa a RdP da figura 2.14 seria:

$$W = \begin{bmatrix} -3 & +3 \\ +2 & -1 \end{bmatrix}$$

Exemplo:

Suponha-se que se pretende modelar, através de uma RdP, o processo de montagem de uma peça, formada por 2 componentes diferentes: A e B. Cada peça necessita de 2 componentes A e 3 B. A modelação desta tarefa é mais simples utilizando uma RdP generalizada, uma vez que o facto da montagem só poder ocorrer quando estiverem satisfeitas as condições do número de componentes, é facilmente captada pela atribuição de pesos aos arcos. Na figura 2.16 mostra-se parte da rede.

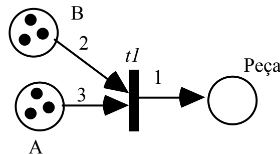


Figura 2.16 - Exemplo Parcial de Montagem de uma Peça

Rede de Petri Colorida - RdPc

Quando existe uma grande complexidade associada ao sistema que se pretende modelar ou várias subestruturas repetitivas, a sua modelação utilizando as RdP descritas, pode ser bastante fastidiosa e o resultado final bastante difícil de interpretar. A necessidade de uma RdP com um nível semântico superior surge naturalmente como uma forma de ultrapassar o problema que é, reforce-se a ideia, não uma impossibilidade de modelação por parte dos modelos anteriores, mas sim uma maior dificuldade de exprimir a complexidade intrínsica dos sistemas.

Numa tentativa de resolver este assunto desenvolveram-se RdP de alto nível, que procuram associar a cada lugar e às transições uma maior nível semântico, tornando a modelação de sistemas complexos mais acessível. Como exemplos deste tipo de redes têm-se as **Redes com Predicados de Transição** (Predicate Transition Nets) e as **Redes de Petri Coloridas -** RdPc.

Numa RdPc, as marcas que aparecem em cada lugar não são todas iguais, isto é, enquanto que nas redes anteriores o importante era o número de marcas, na RdPc o importante é, não só o número, mas também o tipo de marcas que existe. Cada lugar terá vários tipos de marcas, sendo atribuído a cada tipo uma cor, derivando assim o nome de RdP coloridas. Uma das características mais marcantes é a existência de funções associadas aos arcos, que se destinam a transformar as cores dos lugares em cores de transições e vice-versa.

Deve-se realçar que a leitura intuítiva de uma RdPc é mais complexa, mas este facto não invalida as vantagens do ponto de vista de modelação.

Segundo Jensen [39, 40], uma RdPc define-se da seguinte forma:

definição 1

Uma RdPc define-se por um tuplo de ordem 6:

RdPc = (P, T, C, I, O, μ_0)

em que:

- **P** representa o conjunto finito de lugares. P = { p_1, p_2, p_3, ..., p_m }
- **T** representa o conjunto finito de transições. T = { t_1, t_2, ..., t_n }
- **C** representa o conjunto de cores associadas aos lugares e transições.
- **I** representa a função de entrada definida em PxT. I(p, t): C(p) x C(t) -> N
- **O** representa a função de saída definida em PxT. O(p, t): C(p) x C(t) -> N
- μ_0 representa a marcação inicial da rede e define-se sobre P.

Os elementos de C(p) e C(t) chamam-se cores, enquanto que $\mu(p)$ representa o número de marcas de cada cor que existe num lugar p.

<u>definição 2</u>

O disparo de uma RdPc define-se da seguinte forma:

- Uma transição t fica habilitada para uma dada cor ci e C(t) sse ∀ p □ t, $\mu(p) >= I(p, t, ci)$
- O disparo de uma transição t para uma dada marcação μ e cor ci □ C(t), provoca uma nova marca μ' definida por $\mu'(p) = \mu(p) - I(p, t, ci) + O(p, t, ci)$

As RdPc têm sido utilizadas para modelar e validar sistemas de produção complexos [41, 42], uma vez que se adaptam muito facilmente a problemas que envolvam a noção de fila e de atribuição de recursos.

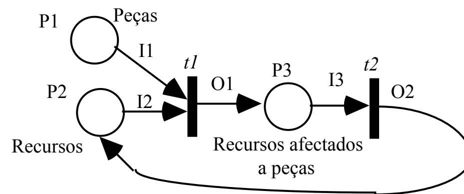


Figura 2.17 - Exemplo de RdPc Utilizada como Atribuição de Recursos

A figura 2.17 representa a atribuição de garras (recursos) às peças a montar, numa célula de montagem do benchmark de Cranfield[27]. O lugar P1 representa as peças, P2 os recursos e P3 a afectação do recurso à peça. As cores atribuídas aos lugares estarão de acordo com esta reprsentação, como se pode ver de seguida:

- C(P1) = { <base 1>, <base 2>, <pendulo>, <pino 1>, <pino 2>, <centro> }
- C(P2) = { <garra 1>, <garra 2>, <garra 3> }
- C(P3) = { <base 1, garra 1>, <base 2, garra 1>, <pendulo, garra 2>, <pino 1, garra 3>, <pino 2, garra 3>, <centro, garra 2> }
- C(t1) = C(P3)
- C(t2) = C(P3)

As funções associadas a cada arco para transformação das cores serão definidas da seguinte forma:

- I1: { < X, Y > } -> { <X> } Ex: I1(<base 1, garra 1>) = <base 1>
- I2: { < X, Y > } -> { <Y> } Ex: I2(<base 1, garra 1>) = <garra 1>
- O1: { < X, Y > } -> { < X, Y > } Ex: O1(<base 1, garra 1>) = <base 1, garra 1>
- I3: { < X, Y > } -> { < X, Y > } Ex: I3(<base 1, garra 1>) = <base 1, garra 1>
- O2: { < X, Y > } -> { < Y > } Ex: O1(<base 1, garra 1>) = <garra 1>

Supondo uma marcação inicial

- μ_0 = [{ <base 1>, <base 2>, <pino 1> }, { <garra 1>, <garra 2> }, {}]

obtem-se as seguintes transições e marcações:

- μ_1 = [{ <base 2>, <pino 1> }, { <garra 2> }, { <base 1, garra 1>}], por habilitação de t1

---

[27] Benchmark de Cranfield é um produto desenvolvido pela Universidade de Cranfield para servir de teste a operações de montagem com robôs. O produto é um pêndulo formado por duas bases, uma haste, um pino central, uma travessa, 4 pinos verticais e 8 pinos laterais.

- μ_2 = [{ <base 2>, <pino 1> }, { <garra 1>, <garra 2> }, {}], por habilitação de t2
- μ_3 = [{ <base 2> }, { <garra 1> }, { <pino 1, garra 2> }], por habilitação de t1
- μ_4 = [{ <base 2> }, { <garra 1>, <garra 2> }, {}], por habilitação de t2
- μ_5 = [{}, { <garra 2> }, { <base 2, garra 1> }], por habilitação de t1
- μ_6 = [{}, { <garra 1>, <garra 2> }, {}], por habilitação de t2

Rede de Petri com Predicados de Transição

A característica mais marcante de uma rede de Petri com predicados de transição é a possibilidade de se associarem predicados às transições. Estes predicados vão condicionar o disparo da transição a que estão atribuídos. As condições de disparo deste tipo de redes, face às normais, alteram-se no facto de, para além da satisfação de existência de marcas nos nós de entrada, é necessário que as condições expressas nos predicados sejam satisfeitas.

Exemplo:

Suponha-se que se pretende modelar o mecanismo de controlo de um alimentador de peças com um mecanismo de fornecimento constituído por um cilindro pneumático. O cilindro é actuado para colocar a peça em posição, após o que deve ser recolhido. Existe um sensor binário que indica que a peça está posicionada no local de fornecimento. O fornecimento de uma peça dá-se quando existe um pedido de peça.

Na figura 2.18 mostra-se uma RdP com predicados de transição que modela este sistema. Existem dois estados (nós): fornece e recolhe. O primeiro indica a actuação do cilindro, enquanto que o segundo indica a recolha. Normalmente o cilindro está recolhido (1 marca neste lugar). A transição só ocorre quando o predicado existe pedido é verdadeiro, passando-se para o estado de fornece. A rede mantém este estado até que o predicado sensor actuado seja verdadeiro. Nesta situação ocorre a transição e o cilindro recolhe novamente.

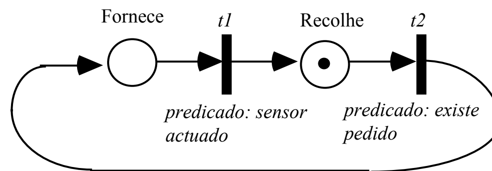


Figura 2.18 - Exemplo de RdP com Predicados na Transição - PTN

Rede de Petri de Capacidade Finita

Neste tipo de rede atribui-se a cada lugar uma capacidade máxima de marcas, isto é, limita-se o número máximo de marcas que podem existir num lugar. Para que uma dada transição dispare, não basta que a transição esteja habilitada, é necessário que o resultado da transição não provoque um número de marcas superior ao máximo estabelecido.

Como exemplo de utilização refere-se o caso de um sistema de armazenamento com uma filosofia FIFO, como se pode ver na figura 2.19.

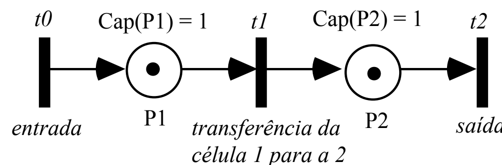


Figura 2.19- Exemplo de RdP com Capacidade Finita

A transferência da célula 1 para a 2 ocorre apenas quando não houver uma marca em P2, passando-se o mesmo com P1. Só se dá entrada quando não existir marca em P1.

Extensões

Rede de Petri com Arco Inibidor

Neste tipo de redes, existe um arco que liga um lugar a uma transição e que pode inibir essa transição, desde que haja uma marca nesse lugar. O arco inibidor distingue-se dos outros pela bola que aparece desenhada do lado da transição (figura 2.20).

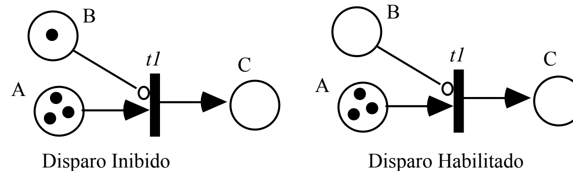


Figura 2.20 - Exemplo de RdP com Arco Inibidor

Este tipo de rede é bastante indicado para modelar sistemas em que existe a necessidade de armazenar previamente uma quantidade não determinada de objectos. Por exemplo se se pretender modelar o comportamento de um sistema que processe peças, com a restrição do processamento apenas se poder iniciar quando houver um determinado número, não fixo de peças, será preferível a utilização de uma rede com arco inibidor. Na figura 2.21 mostra-se um exemplo de uma rede que modela um sistema de maquinação de peças. Neste sistema enquanto não chegar toda a matéria-prima necessária para produzir o lote a máquina não começa a trabalhar, mantendo-se a entrada disponível para a sua chegada. Assim que chega a quantidade de matéria prima suficiente, a entrada fica indisponível e inicia-se a maquinação da matéria prima. Só quando a máquina terminar o processamento de toda a matéria-prima, é que volta a colocar a entrada novamente disponível.

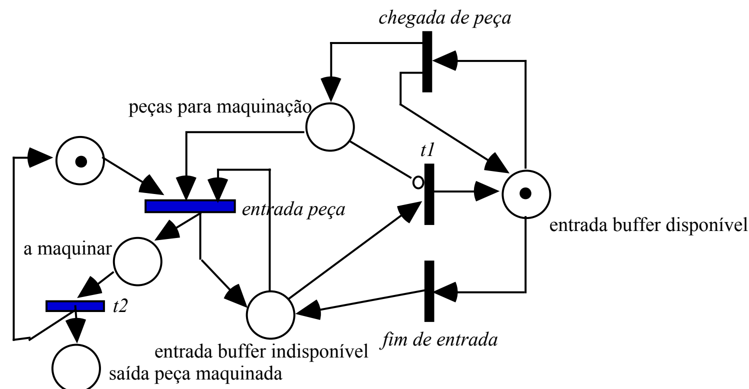


Figura 2.21 - Exemplo de RdP com Arco Inibidor

Enquanto não ocorrer o evento que determina o fim de entrada vão entrando peças para serem processadas, quando a entrada ficar indisponível, deixam de entrar peças o que só acontecerá novamente quando todas tiverem sido processadas.

Quando os lugares, ligados a arcos inibidores, não são limitados, como acontece na figura 2.21, não é possível transformar este tipo de rede em redes normais, já que não é possível explicitar qual o limite para o lugar "peças para processamento".

Rede de Petri Prioritárias

Neste tipo de rede as transições assumem uma dada prioridade que será relevante para a escolha da transição que irá disparar, quando existem várias, simultâneamente habilitadas. Não podem ser transformadas em redes normais.

Como é sabido, nas redes normais, quando existem várias transições habilitadas, a escolha é deixada ao critério do simulador/executor.

2.4.3.2. Não Autónomas

RdP não autónomas apresentam face às autónomas a vantagem de se poder definir quando a transição é realizada. Enquanto que nas autónomas as transições acontecem assim que a rede está habilitada , nas autónomas para além da condição de habilitação normal é necessário que haja um evento externo ou que haja uma dependência temporal. Este último aspecto pode estar relacionado com os lugares ou com as transições. Dentro das redes não autónomas, dependentes de eventos externos destaca-se a Rede de Petri Sincronizada - RdPS, e nas dependentes do tempo destacam-se as Redes de Petri Temporizadas, que se podem ainda subdividir em Redes de Petri Temporais por Lugar ou por Transição, consoante o tempo se relaciona com lugares ou transições, respectivamente.

As redes não autónomas são bastante importantes para a modelação de sistemas controladores. Na especificação deste tipo de sistemas as necessidades de modelação relacionadas com temporizações ou eventos são tão evidentes que limitam a necessidade de uma explicação mais detalhada. Interessa apenas realçar a sua importância na área de controlo. Como exemplo deste facto refere-se o Grafcet que pode ser considerado como uma rede de Petri não autónoma, usando um formalismo diferente.

Redes de Petri Sincronizadas - RdPS

A característica mais marcante deste tipo de rede, face às normais, reside na dependência que as transições têm da ocorrência de um evento externo. Para que ocorra uma transição são necessárias as seguintes situações:

- **a transição esteja habilitada**
- **ocorra o evento associado à transição**

A possibilidade de associar à transição um evento aumenta a capacidade expressiva da rede, nomeadamente pelo facto de existir uma quantidade razoável de sistemas cuja evolução é realizada à custa de eventos. Os eventos podem ser de dois tipos: gerados externamente ou eventos que ocorrem sempre. Uma transição dependente deste último tipo de eventos ocorre imediatamente após ter ficado habilitada. Assume-se que 2 eventos externos nunca ocorrem simultaneamente.

Para uma maior compreensão atente-se no seguinte problema:

Pretende-se modelar um sistema constituído por um robot e 3 ferramentas que o robot pode utilizar, de cada vez. O robot pode estar em espera, a agarrar uma das 3 ferramentas, em trabalho, a guardar uma das 3 ferramentas ou ainda ficar numa situação de erro. Supõe-se a existência de um controlador de alto nível que seja capaz de realizar os eventos associados aos estados referidos.

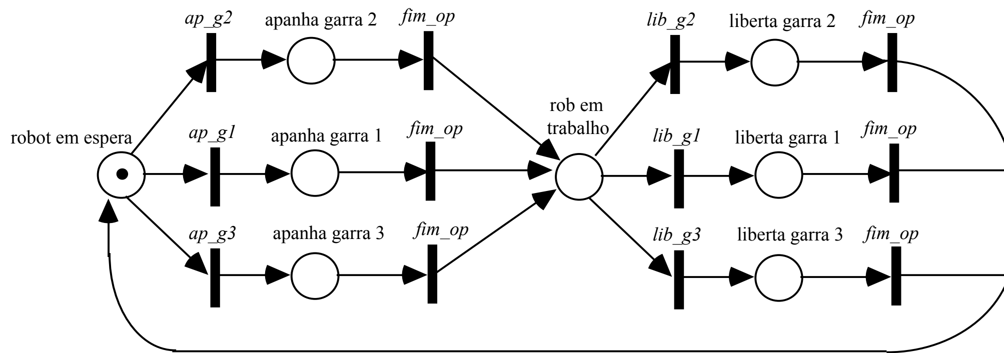


Figura 2.22 - Exemplo de RdP Sincronizada

Se a rede da figura 2.22 fosse uma rede normal, a evolução poderia dar-se para qualquer dos estados "apanha garra 1", "apanha garra 2" ou "apanha garra 3", dependendo da implementação. Só que esta situação não corresponde exactamente àquela que se pretende modelar, em que a evolução de estado é condicionada pelo evento, gerado através do controlador de alto nível. Utilizando uma rede sincronizada a situação fica perfeitamente descrita, porque ao se associarem eventos às transições, a ordem de disparo fica claramente definida.

Este formalismo pode também ser utilizado para descrever máquinas de estado assíncronas, isto é, máquinas cuja evolução não é temporal, mas dependente apenas do acontecimento de eventos.

Redes de Petri Temporizadas - RdPT

Neste tipo de redes, a sua evolução faz-se depender do tempo. Um determinado estado (operação) pode ter necessidade de demorar um certo tempo, obrigando a que o lugar que o representa seja condicionado a demorar pelo menos esse tempo. A temporização pode estar associada ao lugar ou à transição, dependendo a escolha, de um ou outro método, das características específicas do sistema a modelar. É sempre possível passar de uma rede temporizada por transição para uma temporizada por lugar e vice-versa.

Redes de Petri Temporizadas por Lugar

Numa rede normal, assim que uma marca é adicionada a um lugar, pode participar imediatamente nas condições de habilitação da(s) transição(ões) a que o lugar estiver associado. Pode-se dizer que uma dada marca pode transitar imediatamente[28] para o lugar seguinte. Neste tipo de redes, a existência de uma marca num lugar não a torna imediatamente elegível para participar nas condições de habilitação da transição.

Quando uma marca é adicionada a um lugar, ficará **indisponível** até que decorra um certo intervalo de tempo, altura em que se tornará **disponível**, podendo participar nas condições de habilitação. O intervalo de tempo durante o qual a marca está indisponível é variável e pode ser diferente para cada um dos lugares. Note-se ainda que não se está a pensar em temporizações associadas a marcas mas sim a lugares.

A adição de semântica temporal aos lugares faz com que a modelação dos processos discretos de fabricação seja feita de uma forma mais coerente e realista. Na figura 2.23 mostra-se uma rede deste tipo que modela a entrada de paletes para 2 células contíguas e o tempo mínimo que cada delas aí vai estar. A figura mostra os diferentes passos do que acontece quando se dá a transição *t1*.

---

[28] Quando se diz imediatamente refere-se apenas à noção de habilitação e não à própria evolução, ou seja, quer dizer-se que a transição, associada ao lugar, ficou imediatamente habilitada no que diz respeito a esta condição.

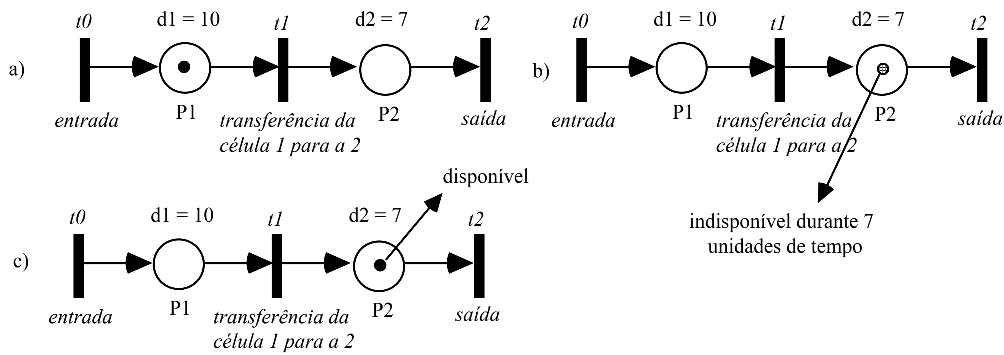


Figura 2.23 - Exemplo de RdP Temporizada por Lugar

Redes de Petri Temporizadas por Transição

Neste tipo de redes as transições, após estarem habilitadas, sofrem um atraso até que ocorra o disparo. A cada transição é atribuído um determinado intervalo de tempo. As marcas que participam na habilitação de uma dada transição são colocadas imediatamente em estado **reservadas**, assim permanecendo até que decorra um determinado intervalo de tempo, após o qual se dá a transição. As marcas que não habilitam transições estão em estado **não reservadas**.

Uma marca só passa ao estado reservada se existirem condições de habilitação para a transição associada ao lugar onde se encontra a marca, isto é, as marcas dos lugares de entrada de uma transição apenas ficam reservadas quando existirem marcas em todos os lugares de entrada.

A diferença entre a temporização por lugar e por transição ocorre precisamente ao nível do momento em que as marcas são afectadas às transições. Na temporização por lugar, a transição só fica habilitada ao fim de um dado intervalo de tempo (marca indisponível). Só ao fim deste tempo é que a marca pode habilitar a transição (marca disponível). Na temporização por transição, a marca pode habilitar imediatamente a transição, mas só fica reservada se houver condição de habilitação.

Uma outra diferença pode ser sentida ao nível do momento do disparo. Na temporização por lugar a transição pode ou não disparar ao fim do intervalo de tempo, enquanto que na por transição, a transição tem de forçosamente disparar ao fim do intervalo de tempo.

Redes de Petri Interpretada - RdPI

Este tipo de rede engloba as características mais relevantes das redes sincronizadas e temporizadas. Apesar disto, não se pode dizer que uma RdPI é apenas o conjunto de características dos dois tipos de rede referidos, uma vez que paralelamente às características herdadas dos 2 tipos referidos, adicionaram-se novas funcionalidades, que a tornam mais indicada para a modelação de controladores lógicos e sistemas em tempo real.

Uma RdPI apresenta então as seguintes caracteristicas:

- É sincronizada
- É temporizada por lugar
- Existe a noção de variável, existindo um conjunto de variáveis V = { V1, V2, ... }, cujo estado é alterado através do conjunto de operações O = { O1, O2, ... }. As operações aparecem associadas aos lugares, isto é, quando uma marca chega a um determinado local, desencadeiam-se as operações atribuídas a esse mesmo local.
- Existe a noção de condição C = { C1, C2, ... }, que aparece associada às transições.

A adição das novas funcionalidades obriga a que o disparo de uma transição Tj ocorra quando:

- A transição Tj está habilitada
- A condição Cj é verdadeira
- Ocorre o evento Ej

De notar que a diferença entre evento e condição pode não ser imediata à primeira vista, havendo até alguns casos em que os dois conceitos se podem confundir. A diferença reside essencialmente no facto da condição ser um predicado associado à rede, ou seja, dentro do total controlo da rede, enquanto que um evento é algo gerado externamente.

Na figura 2.24 apresenta-se o modelo do sistema de controlo de um tapete transportador de paletes, constituído por um motor e um sensor de palete posicionada. O tapete encontra-se inserido numa cadeia de tapetes e a transferência da palete do tapete anterior para o seguinte faz-se em 2 fases: na primeira dá-se entrada da palete, mantendo-se o motor ligado até que a palete fique totalmente posicionada no tapete, desligando-se então o motor do tapete; a segunda fase é a saída da palete, ligando-se o motor, quando o tapete seguinte está disponível, que se mantém ligado até que a palete seja totalmente expelida do tapete.

P1 representa a situação do tapete sem palete e com o motor parado, P2 a entrada de palete, P3 tapete com palete em espera que o seguinte esteja disponível e finalmente P4 representa a saída da palete para o próximo tapete.

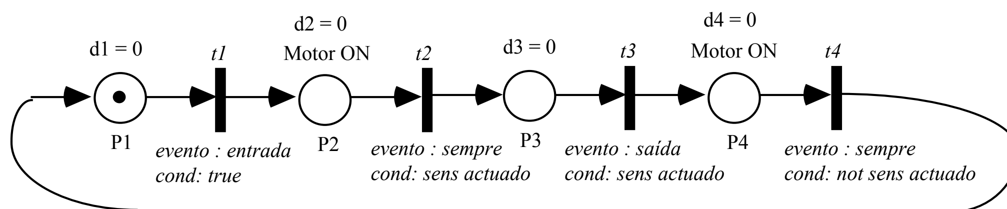


Figura 2.24 - Exemplo de RdP interpretada

Para haver entrada de palete é necessário que o tapete esteja desligado, sem palete, correspondendo esta situação à existência de uma marca em P1, provocando a habilitação de t1. Este estado mantém-se até que seja dada "ordem" para que o tapete dê entrada a uma nova palete. Esta "ordem" representa um evento e é gerado por um sistema externo ao controlador que a rede está a modelar. A passagem do estado "tapete em entrada" para o estado "tapete com palete", ocorre agora, não por ocorrência de um evento mas sim por verificação de uma condição que está associada ao controlador modelado pela rede. Mas a passagem deste último estado para o estado "tapete em saída" já é condicionado pela ocorrência de um evento. O controlador não pode, por si só, determinar que vai proceder à entrega da palete, apenas pode determinar que está em condições para o fazer (existência de marca em P3). A passagem efectiva ao estado de saída ocorre quando for gerado o evento *saída*. Como medida de poupança de esforço do motor exige-se que a condição de sensor actuado (existência de palete) seja cumprida.

MOTOR ON é uma variável que é actuada los locais P2 e P4.

A utilização deste tipo de modelos para modelar sistemas controladores é muito interessante e tem sido objecto de estudo por parte dos investigadores da área de controladores lógicos. Neste sentido, desenvolveu-se um formalismo que é bastante semelhante, na sua funcionalidade geral, à Rede de Petri Interpretada - **Grafcet**.

O Grafcet resultou do trabalho levado a cabo em França, por um grupo de trabalho da AFCET (Association Française pour la Cybernétique Economique et Technique), para normalizar as representações das

especificações de controladores lógicos programáveis. O documento final foi assinado em 1977 e o resultado foi o Grafcet, que começou por ser uma norma francesa para, em 1987, se tornar uma norma internacional[29].

Apesar do reconhecimento internacional e da sua efectiva adequabilidade para a modelação de controladores, especialmente quando o sistema a controlar envolve a noção de estado, não se pode dizer que o Grafcet tenha uma grande divulgação industrial. Este facto acontece por vários factores, a que não será estranho a falta de empenhamento de alguns fabricantes em fornecer os seus sistemas com ferramentas adequadas ao suporte do Grafcet. Por outro lado, o nível de formação existente nos ambientes fabris não permite a sua divulgação. Quer se queira quer não, este tipo de formalismo ainda não está no domínio do técnico intermédio, tornando a sua divulgação/utilização mais complicada.

2.4.4. Análise das RdP

A importância das redes de Petri reside no facto de ser um formalismo sobre o qual se podem fazer análises matemáticas que ajudam a perceber e detectar a existência de potenciais problemas nos sistemas que pretendem modelar. Esta capacidade de análise, que torna possível, por exemplo, detectar se determinados estados ocorrem ou se o sistema entra em situações de bloqueio, é bastante importante pois permite, de uma forma racional e matemática, descrever o comportamento do sistema. Por mais complexo que seja o problema, a ambição do programador de controladores é ter a certeza de que conhece completamente o comportamento do sistema (propriedades), o que nem sempre é possível, mesmo com RdP. De qualquer maneira são um contributo importante nesse sentido.

A análise de uma rede de Petri destina-se a verificar uma dado conjunto de propriedades. Dada a sua importância, vão-se descrever, de uma forma resumida, alguns dos aspectos mais importantes da análise qualitativa e quantitativa que é possível realizar sobre as RdPs.

2.4.4.1. Análise Qualitativa

A análise qualitativa destina-se a verificar, entre outras propriedades, se uma rede é segura, se é limitada, se é conservativa, se um dado estado cobre outro ou ainda se um dado estado é acessível. Nenhuma destas propriedades está relacionada com tempo ou sincronização. Na verdade, a análise qualitativa não se aplica a nenhuma propriedade relacionada com estes dois conceitos. Conclui-se também que esta análise se destina essencialmente a ser aplicada sobre redes autónomas, que são, como se sabe, redes completamente independentes de condições ou eventos externos.

Na análise qualitativa utilizam-se essencialmente métodos baseados em:

- **árvores de acessibilidade**
- **álgebra linear**
- **reduções**

Árvores de acessibilidade são árvores construídas à custa de um algoritmo que permite a construção de um grafo que mostra a evolução possível de marcas na rede. Os nós da árvore representam um determinado estado de marcação, enquanto que os arcos representam as transições que podem ocorrer a partir de cada nó. É uma ideia mais ou menos intuitiva já que, quando analisamos informalmente uma rede, construímos mentalmente um grafo, tentando para cada marcação, verificar quais as transições que podem ocorrer e gerar novo estado de marcação, continuando-se, recursivamente e para cada lugar, este processo.

No processo de análise, primeiro, constrói-se a árvore, fazendo-se posteriormente sobre ele, pesquisas para verificação das propriedades.

---

[29] Publicação 848 da Comissão Internacional de Electrotecnia, denominada "Etablissement des diagrammes fonctionnels pour systèmes de commande"

Uma rede em que não haja um número limitado de marcas por lugar, é representada por uma árvore de tamanho infinito, tornando impraticável a pesquisa. Esta árvore pode ser transformada numa de tamanho finito se se encontrarem ramos cíclicos, isto é, com arcos iguais e nós que diferem apenas no número de marcas. Os ramos cíclicos são transformados num único nó, sendo o número de marcas do lugar que varia identificado pela letra w.

Na figura 2.25 mostra-se um exemplo de uma RdP simples com a sua respectiva árvore de acessibilidade. Pode-se verificar que a rede é **limitada** (não aparece w em nenhum dos nós), **não segura** (existem nós que podem ter um número de marcas superior a 1) e **não conservativa** (perde 1 marca).

Os métodos relacionados com **álgebra linear** são particularmente indicados para a obtenção dos invariantes e estão baseados na equação fundamental $\mu_k = \mu_i + W.\underline{S}$. A grande vantagem destes métodos é a sua possível automatização.

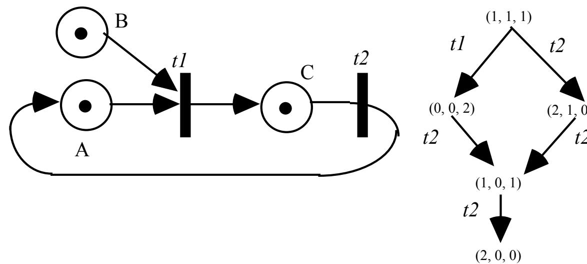


Figura 2.25 - Exemplo de RdP e Árvore de Acessibilidade

A redução é um método que consiste na tentativa de transformar uma RdP noutra mais simples sem perda de conteúdo semântico. Nem sempre é possível reduzir uma RdP a outra que seja equivalente. O facto de se obter uma rede mais simples, que mantém as propriedades essenciais do sistema que descreve, permite a aplicação dos métodos referidos anteriormente de uma maneira mais simples, tornando a sua utilização óbvia.

2.4.4.2. Análise Quantitativa

Aparece directamente relacionada com os aspectos de tempo e sincronização [43-45]. Permite a análise de desempenho baseando-se, naturalmente no conceito tempo.

Existem 2 métodos para a realizar: (1) por simulação e (2) por utilização de métodos analíticos.

3. Modelação Conceptual de Células

3.1. Introdução

Neste capítulo fazer-se-á uma análise aos conceitos envolvidos na modelação conceptual de células. Será feita uma abordagem no sentido de se levantarem as questões importantes e procurar-se-á também apresentar um conjunto de soluções que contribuam para a criação de controladores flexíveis. A abordagem ao problema da modelação será feita apresentando primeiramente os conceitos relacionados com a **modelação estática**, onde se equacionam os aspectos taxonómicos dos componentes participantes numa célula, as suas relações e a sua descrição estrutural individual. Os conceitos relacionados com a **modelação dinâmica** serão apresentados posteriormente. Equacionar-se-ão os aspectos relacionados com o comportamento individual dos componentes, o modo como cada componente é decomposto na sua faceta de objecto interno e externo, a forma de ligação entre os dois modelos de execução (execução interna versus execução no controlador externo) e todo o comportamento relacionado com a execução dos objectos, analisados sob vários quadrantes possíveis: desde a visão de sistema até à visão componente, passando pela visão célula.

Relacionado com a modelação dinâmica será dado realce aos aspectos de **persistência dinâmica**. De uma forma simples, a persistência dinâmica consiste em fazer persistir uma parte da estrutura dos objectos, no controlador externo da entidade física que os referidos objectos estão a modelar.

Será realizado um esforço no sentido de se criarem células genéricas que possam ser facilmente reutilizadas e configuradas. Esta abordagem será feita na medida do possível independentemente da ferramenta de modelação a usar: frames ou programação orientada por objectos.

Serão também utilizados conceitos de vistas e papéis no intuito de tornar as estruturas mais flexíveis.

Um ponto importante será também a descrição de problemas relacionados com a ligação lógica que existirá entre o modelo conceptual e o controlador físico, questão esta que está intimamente relacionada com os aspectos de modelação dinâmica.

Este capítulo será apresentado recorrendo a um grande número de exemplos. A razão desta opção prende-se com a facilidade de percepção que é inerente à exemplificação. A apresentação apenas de conceitos pode tornar-se extremamente fastidiosa e, muitas vezes, bastante complexa, já que a transmissão de alguns conceitos é quase impossível sem o recurso a exemplos. Assim, ao longo do capítulo, o recurso aos exemplos será uma constante.

Este facto obrigará à escolha de um modelo de programação para se apresentarem os exemplos, que deverá sair dos apresentados no capítulo anterior: "*Frames*" e Programação Orientada por Objectos.

Dada a experiência anterior do grupo na programação por Frames [22-25] é natural que a escolha recaia sobre este modelo. Mas, é sempre altura de introduzir algo de novo. Considerando que, até agora, todas as abordagens ao tema têm sido feitas numa perspectiva baseada em frames, e conhecendo-se a importância que a

programação orientada por objectos tem para o mundo da automação, optou-se também por incluir exemplos em POO.

Escolhidos os modelos há que escolher as implementações. Os candidatos naturais da POO são as linguagens C^{++} e EIFFEL. A escolha mais natural seria talvez o C^{++} dada a sua grande divulgação, ferramentas de suporte, etc. Mas o facto de não suportar todos os mecanismos da POO levou a que não fosse considerada, tendo-se antes optado pelo EIFFEL®. Para as frames optou-se pela utilização do Golog, que é um sistema de frames, desenvolvido no seio do Grupo de Robótica e CIM, que corre sobre Prolog [46].

No ponto 3.2 será apresentada a formulação do problema. Serão introduzidos os conceitos básicos associados à modelação de células, com especial destaque para a formalização do conceito geral de célula. Será feita uma primeira abordagem à modelação dos componentes existentes numa fábrica e a sua relação com as unidades célula. No ponto 3.3, será feita uma descrição das classes de informação presentes. Descrever-se-ão os elementos preponderantes do sistema: taxonomia de papéis, de componentes, etc. No ponto 3.4 serão analisados os conceitos relacionados com o comportamento estático, realçando-se os aspectos estruturais de componentes e elementos complexos como células e sistemas. No ponto 3.5 serão agora analisados os aspectos relacionados com o comportamento dinâmico.

3.2. Formulação do Problema

Os paradigmas da programação orientada por objectos e frames são uma ferramenta poderosa para modelar a complexidade inerente aos sistemas de produção. Complexidade essa que nasce da quantidade de relações entre os componentes do sistema e a grande diversidade dos mesmos [4].

A modelação tem sido um campo bastante importante no CIM, pela necessidade que se faz sentir de um modelo que suporte as diversas actividades existentes numa fábrica. Pretende-se, neste capítulo, desenvolver os conceitos necessários para modelar um sistema de manufactura constituído por várias células distintas. Será necessário fornecer definições de alguns conceitos a utilizar.

Componentes são entidades básicas (elementos primitivos) que participam no processo produtivo, controladas ou não por um sistema computacional, mas com uma função específica no sistema em que se inserem.

Componente é toda a entidade física, que participa numa célula. Como seria de esperar, existem diversos tipos de componentes, com funcionalidades e características diferentes. Os componentes podem, por isso, ser agrupados numa taxonomia de componentes. Esta taxonomia será organizada de acordo com o tipo de componente, isto é, a taxonomia é organizada pela semelhança entre os componentes.

Uma **célula** é uma estrutura complexa composta por componentes e interligados de determinada forma. Uma célula será, de um ponto de vista abstracto, uma entidade em que são realizadas transformações sobre objectos[30] que foram colocados à entrada. O resultado dessa transformação é colocado na saída.

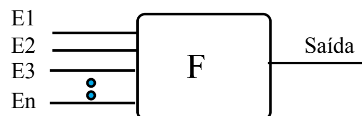


Figura 3.1 - Representação Abstracta de Célula

Pode-se dizer que:

```
Saída = F(E1, E2, E3, .., En)
```

[30] Objecto no sentido de entidade

em que F representa uma dada transformação sobre as entradas. Esta visão de célula como uma representação matemática de função implica que se clarifique o que são as entradas e o que é a saída. Tendo em vista que as unidades fabris se destinam a produzir produtos, é natural que as células sejam encaradas como parte deste objectivo final. Assim, qualquer célula terá sempre uma transformação sobre um conjunto de produtos, que poderão ser matérias-primas ou subprodutos, e terá como resultado um subproduto ou um produto final. A função será então encarada como a realização de uma qualquer operação sobre subprodutos ou matéria-prima. Não se deve pensar que esta noção de célula se restringe a actividades de montagem, na verdade adequa-se a todas as situações, veja-se o caso, por exemplo, da pintura e da soldadura.

No exemplo concreto de uma célula de pintura, a entrada pode ser uma porta de um automóvel, a saída a porta pintada e a função a acção de pintar a porta. No caso da soldadura, as entradas podem ser duas peças, a saída as duas peças soldadas e a função a acção de soldar.

Uma discussão que se pode levantar neste ponto é a seguinte: será que numa célula são sempre realizadas operações sobre as entradas o que conduz, inevitavelmente, a que o produto à saída seja sempre diferente do que era à entrada ou acontece que, em certos casos, onde apesar de se aplicar a noção de operação sobre o objecto, ela não se centra no sentido de provocar uma transformação da entrada para a saída. Analise-se, por exemplo, o caso do armazenamento de um dado produto (matéria prima ou produto processado) que pode ser visto como uma função com entrada e saída. Senão repare-se, a entrada é constituída pelo produto a armazenar, a função corresponde ao acto de armazenar o produto e a saída é novamente o produto que foi armazenado.

Segundo este ponto de vista pode-se considerar uma outra situação. Suponha-se que se pretende transportar um produto de um local para o outro, por exemplo, de uma célula de montagem para outra. A acção que irá ser desencadeada sobre o produto resultante, na saída da 1ª célula, corresponde a leva-lo de uma posição para outra, sendo agora presente à entrada da 2ª célula. Esta célula, cuja acção corresponde a movimentar um produto de um referencial para outro, chama-se célula de transporte.

A abstracção utilizada permite que as células sejam encaradas como entidades genéricas, onde são realizadas operações sobre as entradas, realçando-se o facto dessas operações não serem, forçosamente, no sentido de adicionar algo aos produtos na entrada.

Uma **célula** será uma entidade capaz de desencadear operações de índole geral (transformação, armazenamento, transporte, ...) sobre uma ou mais entradas, produzindo uma ou mais saídas, em resultado dessas operações.

A vantagem deste tipo de abstracção reside na facilidade com que se podem construir sistemas, a partir de unidades básicas como as que foram definidas.

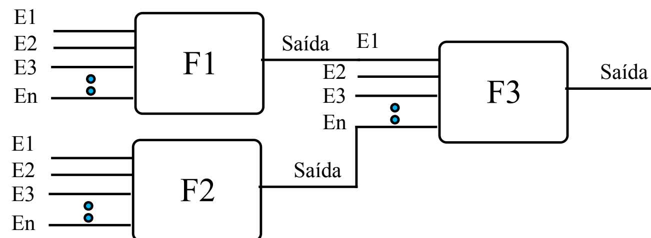


Figura 3.2 - Concatenação possível de Células

Um sistema constituído por duas unidades de montagem, ligadas em linha por um transportador pode então ser modelado como se mostra na Figura 3.3.

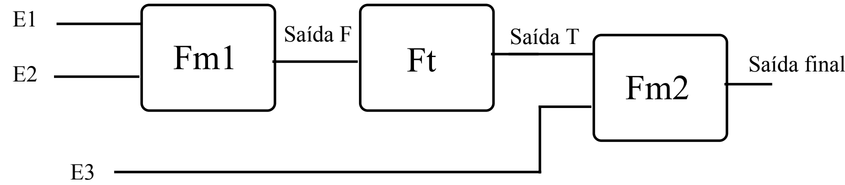


Figura 3.3 - Sistema com 2 Células de Montagem Ligadas por uma Célula de Transporte

```
Saída final = Fm2(Saída T, E3)
Saída T = Ft(Saída F)
Saída F = Fm1(E1, E2)
Saída final = Fm2(Ft(Fm1(E1,E2)), E3)
```

O sistema foi representado de uma forma clara, sem recurso a casos particulares. O facto de se considerar o transporte como sendo uma operação sobre o objecto simplifica grandemente a construção de sistemas, sendo utilizado sempre que seja necessário interligar duas células fisicamente separadas.

Um **sistema** é um agrupamento de células, em que as saídas de uma podem aparecer ligadas às entradas de outras.

Introduzida que está a noção abstracta de célula é importante consolidar a discussão, através de um olhar para o interior da "caixa" que representa a célula fazendo a ligação entre o conceito abstracto e os componentes existentes na realidade.

As operações sobre as entradas são desempenhadas por um agente que deverá ser capaz de as desencadear. Por outro lado, para que o agente consiga desencadear as operações é necessário que os produtos, sobre quem vão ser desencadeadas as operações, estejam disponíveis na entrada. Deverão, assim, haver entidades que garantem o fornecimento de produtos à célula. Finalmente, o resultado das operações deve ser colocado disponível na saída da célula sendo, por isso, necessários componentes para este efeito.

Figura 3.4 - Estrutura de Célula Abstracta

A representação abstracta de célula consistirá numa Entrada, numa Saída e num Agente, conforme se pode verificar na Figura 3.4.

Cada célula irá ser constituída por componentes que suportam as funções de entrada, um componente agente que realizará as operações e componentes que suportam as funções de saída.

Componentes	Entrada	Agente	Saída
Panelas Vibratórias	•		
Buffers	•		•
Mesa Indexação	•		•
Alimentador Gravítico	•		
Tapete Transportador	•	•	•
Robô		•	
Máquina CNC		•	
AGV		•	
Fixador	•		•
Doca de AGV	•		•

Tabela 3.1 - Exemplos de Componentes e Indicação das Funções que Podem Desempenhar

A tabela 3.1 representa um conjunto de componentes e a indicação das funções que podem suportar, notando-se que existem componentes que podem desempenhar mais do que uma função. O caso extremo é dado pelo tapete transportador que pode ser um componente de entrada, de saída e um agente, que é o que acontece numa célula transportadora, quando o agente é um tapete.

Apesar da possibilidade que alguns componentes têm de poderem desempenhar, por exemplo, funções de entrada e saída, o seu desempenho pode não ser igual quando integrados numa ou noutra situação. Como exemplo de um caso destes, veja-se a mesa indexadora que, quando colocada como componente de entrada, o referencial para ir buscar os elementos pode ser diferente do referencial para colocar os elementos, quando em saída.

Um componente pode ser partilhado por duas funções distintas de uma célula, isto é, um componente de saída de uma dada célula pode ser o componente de entrada da célula seguinte. Como exemplo de uma situação deste tipo basta pensar, por exemplo, em duas células de montagem interligadas através de um buffer. Os produtos que vão sendo montados na célula 1 são colocados no buffer - saída, que representa também uma entrada da célula seguinte. Apesar desta partilha, o papel que o buffer representa é distinto num e noutro caso.

Esta questão dos papéis que os componentes representam é uma ideia interessante pois permite que a partilha se faça de uma forma elegante. As características que são relevantes para a entrada aparecem associadas à entrada, enquanto que as que são relevantes para a saída aparecem associadas à saída. O que efectivamente se vai criar são duas vistas de um mesmo componente, uma em que está a desempenhar o papel de entrada, outra em que aparece a desempenhar o papel de componente de saída.

A possibilidade do componente poder ser descrito de duas formas diferentes "separa as águas" do ponto de vista da programação, tornando-a mais elegante e expressiva. Caso não fosse possível a criação destas duas vistas, a questão que poderia imediatamente levantar-se era como se havia de considerar o objecto, se de saída da célula 1 ou se de entrada da célula 2. Outra questão ainda:: quem é que controlava o objecto ? a célula 1 ou a 2 ? Era com toda a certeza uma solução não flexível, cheia de "ses". Com as vistas, o problema resolve-se, uma vez que o componente é agora considerado como se fossem dois, uma parte é saída e outra parte é entrada. Garante-se, ao mesmo tempo, que as operações realizadas pelos dois lados correspondem a operações realizadas, efectivamente, sobre o mesmo componente.

Os seres humanos, quando são utilizados para realizar determinadas tarefas, necessitam de ser "adaptados" às suas novas funções. Muitas vezes é mesmo necessário desenvolver algumas características por forma a que a tarefa tenha um desempenho superior. De uma forma análoga, não se irão aplicar directamente à célula os componentes tal e qual estão descritos na taxonomia de componentes.

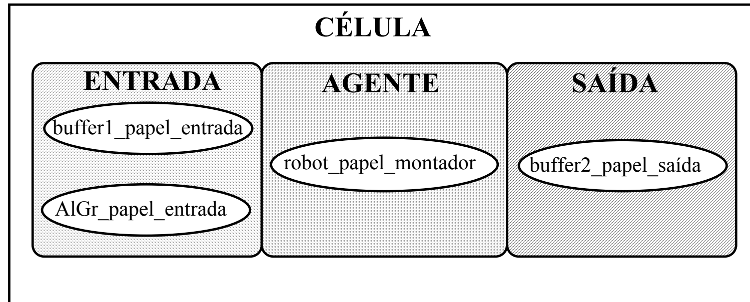


Figura 3.5 - Exemplo de Célula Abstracta com Componentes Representando Diversos Papéis

Na verdade não serão os componentes que participarão directamente, mas sim uma nova estrutura que resulta de características herdadas do componente indicado para a tarefa, adaptado com características próprias da operação a realizar. Esta estrutura, criada a partir do componente, representa o componente a desempenhar um determinado papel.

Assim tem-se, por exemplo, uma célula em que se modelam a entrada, através de um buffer a desempenhar o papel de entrada, o agente, através de um robô a desempenhar o papel de montador e a saída, utilizando novamente um buffer mas agora a representar o papel de saída.

Na Figura 3.5 mostra-se uma hipotética célula de montagem com duas entradas e uma saída, indicando-se nas formas ovais as entidades que representam os componentes a desempenhar os diferentes papéis.

Se a célula anterior fosse ligada a uma segunda célula do mesmo tipo, e a ligação entre a 1ª e a 2ª célula se fizesse através do *buffer2*, este apareceria a desempenhar o papel de saída tal como na Figura e apareceria também a desempenhar o papel de entrada na segunda célula.

De forma a realçar a importância do desempenho de papéis pelos componentes, veja-se o problema representado na Figura 3.6.

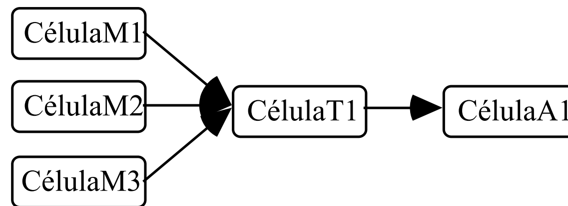


Figura 3.6 - Sistema Constituído por 5 Células

O sistema representado na figura 3.6 é constituído por 3 células de montagem cuja saída é feita para um tapete que transportará os produtos até à entrada da célula de armazenagem. A célula de transporte será constituída por um tapete, que participará na actividade de agente, de entrada e de saída. Cada célula de montagem terá um ponto de saída para o tapete, que será idêntico ao ponto de entrada, visto do lado da célula de transporte. As entradas e saídas, não contando com as entradas das células de montagem e a saída da célula de armazenamento, envolvidas no sistema serão: *saídaM1*, *saídaM2, saídaM3, entradaT1M1, entradaT1M2, entradaT1M3, saídaT1, entradaA1*.

Cada componente, pertencente à célula, pode participar em vários papéis, conforme se pode verificar na Figura 3.7.

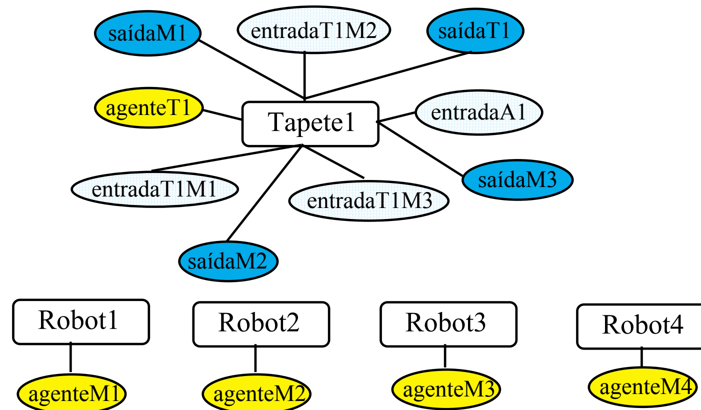


Figura 3.7 - Papéis Desempenhados pelos Componentes

Um mesmo componente, neste caso o *tapete1*, participa em vários papéis, fazendo com que cada célula seja considerada como uma entidade única, encapsulada em torno do conceito célula, com entrada, agente e saída, mas representando ao mesmo tempo uma partilha do recurso. O facto do tapete ser o mesmo, em todos os papéis, implica a partilha do componente físico.

É necessário, para uma compreensão mais efectiva do conceito de célula, que todo o conjunto de componentes participantes numa célula, tais como, sensores de força, ferramentas, fixadores, mesas de montagem, etc, sejam enquadrados na célula. Estes componentes não podem propriamente realizar tarefas de entrada, saída ou de agente, devendo antes ser considerados como auxiliares do agente.

Assim, é natural que apareçam associados à estrutura complexa que representa os agentes num qualquer papel. Seja, por exemplo, o componente robô a desempenhar o papel de montador, necessitando para essa tarefa de uma ferramenta de montagem, um sensor de forças, um fixador para realizar a montagem, etc. Esta estrutura é de certeza diferente da que representa um robô a desempenhar o papel de soldador, necessitando agora apenas de uma ferramenta de soldadura e um fixador diferente. Pode-se dizer, em jeito de conclusão, que os componentes que não são desenhados para serem, nem entradas, nem saídas, nem agentes, vão aparecer associados a este último, sendo a sua relação com o componente principal descrita através dos diferentes papeis que este último pode representar.

Até agora realçou-se sempre o facto de as células não serem todas do mesmo tipo. Mas será interessante clarificar o tipo de células considerado durante esta análise: células de **montagem**, **pintura**, **transporte**, **armazenamento**, **soldadura** e **maquinação**.

A diferença entre elas vai ser marcada, essencialmente, pelo domínio das entidades que podem ser atribuídas às entradas, saídas e agentes. Existem algumas restrições, para cada tipo de célula, que merecem ser notadas. Elas encontram-se a 2 níveis: (1) ao nível do componente e (2) ao nível do papel que o componente desempenha.

Como exemplo do 1º caso pode citar-se a diferença que existe, ao nível dos componentes que podem participar na entrada, entre uma célula de pintura e outra de montagem. Uma panela vibratória é um componente válido para participar como entrada de uma célula de montagem mas não é para a entrada de uma célula de pintura. Um outro exemplo que se pode dar situa-se ao nível do agente. Um robô do tipo SCARA não é um componente válido para participar como agente de uma célula de pintura enquanto que é perfeitamente válido para participar numa célula de montagem.

Como exemplo do 2º caso cita-se a diferença que existe entre uma célula de transporte e uma célula de montagem que possuam, em ambos os casos, agentes desempenhados pelo componente robô. O papel que um robô de montagem desempenha, em que existem mais relações com componentes auxiliares (fixador, sensor de forças, ferramenta, ...), é com certeza distinto do papel desempenhado por um robô destinado apenas a movimentar materiais. Neste último caso a relação do robô com componentes auxiliares pode resumir-se quase só à ferramenta.

O facto de os componentes desempenharem papéis implicará o desenvolvimento de uma taxonomia de papéis, que será apresentada no ponto 3.3.

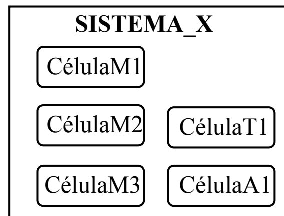


Figura 3.8 - Sistema como uma Composição de Células

Para terminar a análise das células falta referir o modo como será modelado um sistema. Um sistema, como vimos, é um conjunto de células interligadas, sendo por isso natural que a sua representação se faça através de uma relação *parte_de*[31], veja-se a Figura 3.8. De um ponto de vista abstracto a classe sistema será constituída por uma estrutura que suporte vários elementos (*array, bag, lista*, ...) em que cada elemento do conjunto seja uma célula.

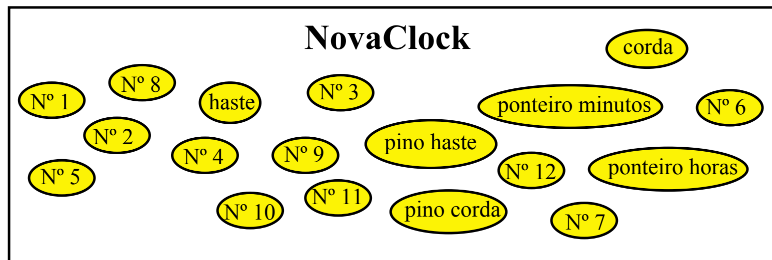


Figura 3.9 - Um Produto Complexo: NovaClock, Constituído por Produtos Simples

Mas num sistema também participam produtos sendo, por isso, necessário desenvolver conceitos que os descrevam. Considera-se como produto cada uma das entidades que são movimentadas dentro do sistema. Os produtos podem ser, como é óbvio, bastante diversificados e com vários níveis de complexidade, e como tal a taxonomia de produtos irá ser desenvolvida tendo estes aspectos em atenção. Terá de haver uma forma de relacionar um produto complexo com um produto simples, ou dito de uma outra forma, terá de se conseguir expressar o facto de um produto complexo ser constituído por vários produtos simples. A ideia que surge imediatamente é considerar cada produto complexo como uma agregação de produtos simples.

Cada produto simples, por exemplo o *Nº12* pode ser visto como um produto individualizado, pertencendo à extensão[32] da classe *números* ou pode ser visto como um dos elementos da agregação de um produto complexo, neste caso o NovaClock®[33].

[31] Conforme se viu no capítulo 2, esta relação denomina-se composição o que está de acordo com a definição de sistema que se pode considerar como sendo um composto de células.

[32] Chama-se extensão ao conjunto de objectos pertencentes a uma dada classe.

No modelo de um produto simples devem ficar expressas as suas propriedades geométricas, o sítio onde se encontra armazenado (os produtos simples estão armazenados em fixadores (*holders*) que são, geralmente, transportados por paletes).

No modelo de um produto complexo deve ficar expresso o modo como o objecto é formado. A descrição do objecto, em termos dos seus constituintes básicos pode ser realizada de diversas formas: (1) através de uma lista ordenada de componentes simples (BOM: *Bill of Materials*) ou (2) através de um grafo ou estrutura complexa como redes de Petri [35].

A primeira situação aFigura-se bastante mais simples e, neste caso, parte-se do princípio que uma outra entidade gerou essa lista. O modo como a lista está ordenada corresponde a uma possível maneira de montar o objecto. No caso do NovaClock®, a lista poderia ser:

(haste, ponteiro minutos, ponteiro horas, pino haste, corda, pino corda, Nº1, Nº2, Nº3, Nº4, Nº5, Nº6, Nº7, Nº8, Nº9, Nº10, Nº11, Nº12)

No objecto ficavam descritos dois conceitos, utilizando uma única estrutura: (1) precedência das operações de montagem e (2) os componentes simples que formam o objecto.

A segunda forma de representação poderá ser bastante mais complexa. No caso da estrutura indicar o grafo com as operações possíveis de montagem, está-se a sugerir que o cálculo da ordem de precedência das operações seja realizado quando do início de montagem do produto. Desta forma, permite-se que o produto possa ser montado em qualquer sistema, uma vez que a geração das operações pode ser feita antes de começar a operação de montagem. Isto era manifestamente impossível no caso da lista ordenada, em que o produto fica imediatamente comprometido com o sistema onde terá de ser montado. É importante realçar a existência de dois tipos de ordenação: uma que depende as características intrínsecas do produto, outra que depende dos agentes existentes no sistema de manufactura.

De notar que o grafo deve ser representado considerando cada nó como pertencendo a uma classe e que se podem descrever nos atributos da classe quais as operações que se fazem para que, por exemplo, se monte N12. Neste caso, o atributo *tipo_operação* deveria conter *inserção*. Uma outra forma seria considerar que este atributo estava declarado no objecto Nº12.

3.3. Classes de Informação Presentes

Neste ponto vai-se agrupar alguma da informação existente num sistema de automação, segundo uma certa classificação. As classes serão nomeadas, mas a sua descrição será ainda feita de uma forma bastante generalista. A descrição dos elementos preponderantes das classes será feita nos pontos seguintes.

A informação irá ser agrupada nos seguintes grupos: **Taxonomia de Componentes**, **Taxonomia de Papéis**, **Taxonomia de Células**, **Taxonomia de Sistemas**, **Taxonomia de Produtos**, **Taxonomia de Suportes** e **Taxonomia de Controladores**.

Uma nota em relação à classificação anterior. Evidentemente que os grupos referidos não são as únicas classes possíveis, num sistema de automação. Elas representam um subconjunto significativo dessas classes e são aquelas que esta tese pretende analisar. Seria impensável tentar resolver aqui todo um problema com a envergadura que este tem.

---

[33] O NovaClock foi o produto desenhado pelo Grupo de Robótica e CIM para servir de demonstração na célula piloto, instalada no Centro de Robótica Inteligente do UNINOVA, conhecida pelo nome NovaFlex. O produto consiste num relógio, sem mecanismo, apenas com números, ponteiros e botão de corda.

3.3.1. Taxonomia de Componentes

Os componentes vão ser agrupados na taxonomia de acordo com o seu tipo. A ideia base para os classificar será o seu tipo de funcionalidade. Será então natural que componentes como os tapetes, AGVs, ..., sejam agrupados numa classe geral denominada transportadores.

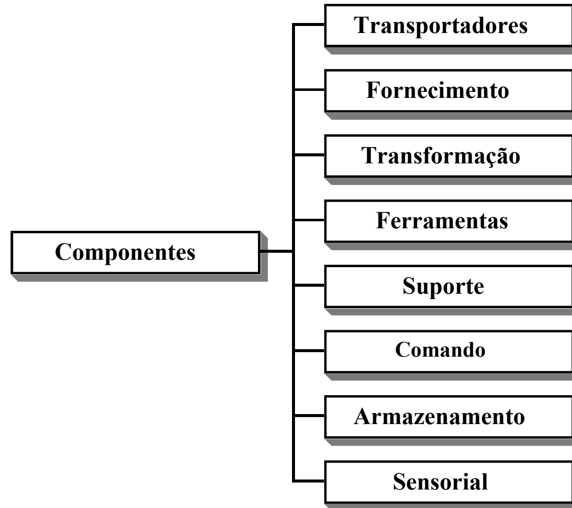


Figura 3.10 - Taxonomia de Componentes - Nível Componentes

Vai-se apresentar uma taxonomia que pretende realçar os aspectos de classificação mais importantes, mostrando alguns dos componentes mais significativos (Figura 3.10).

Os componentes transportadores serão ainda divididos em moveis e fixos, com os AGVs e os tapetes a pertencerem ao primeiro e segundo grupos, respectivamente. Os componentes de fornecimento serão ainda classificáveis em não controláveis e controláveis, com as panelas vibratórias e os alimentadores gravíticos a pertencerem ao primeiro e segundo grupo, respectivamente. Os componentes de transformação serão todos aqueles envolvidos na transformação de produtos (robôs, fresas, ...), vide Figuras 3.11, 3.12 e 3.13 para uma explosão deste grupo.

Figura 3.11 - Taxonomia de Componentes - Nível Transportadores

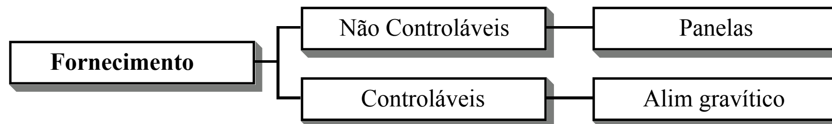


Figura 3.12 - Taxonomia de Componentes - Nível Fornecimento

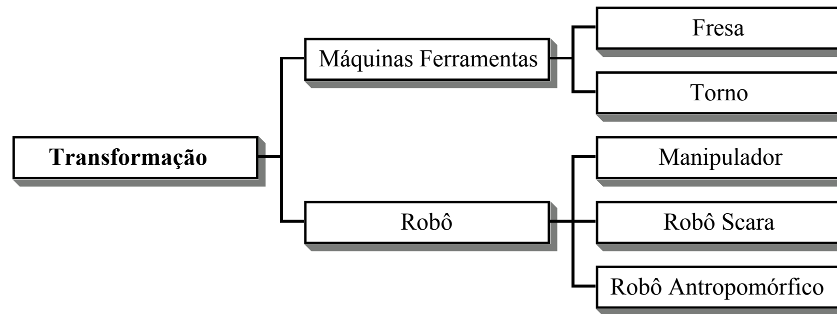


Figura 3.13 - Taxonomia de Componentes - Nível Transformação

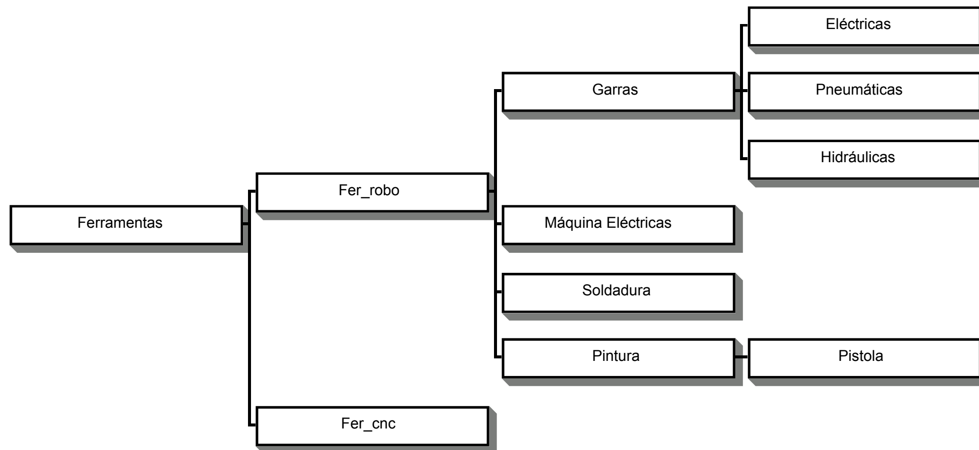


Figura 3.14 - Taxonomia de Componentes - Nível Ferramentas

As ferramentas estão representadas na Figura 3.14. A ideia base consistiu na divisão por ferramentas de robô e de CNC, sendo criadas novas classificações para as do robô.

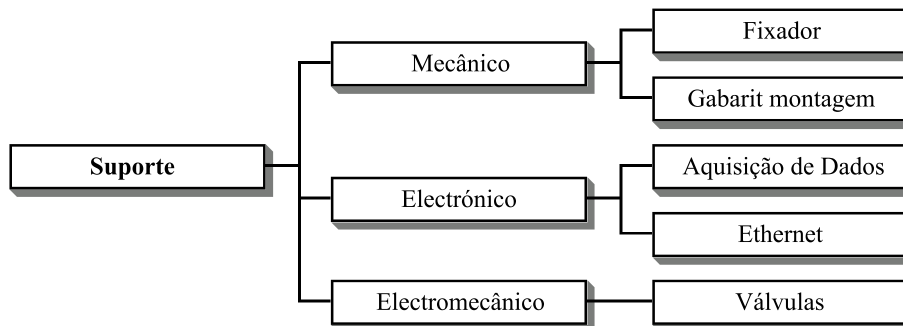


Figura 3.15 - Taxonomia de Componentes - Nível Suporte

NA Figura 3.15 mostra-se a classificação dos componentes de suporte. Os componentes a colocar nesta taxonomia são todos aqueles que participam de alguma forma, no desempenho dos outros. É o caso, por exemplo, do fixador que interactua com o robô a desempenhar a função de montador.

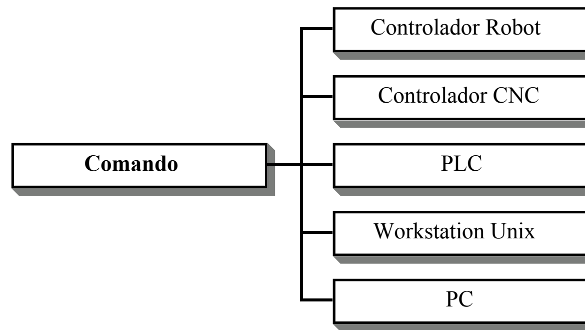


Figura 3.16 - Taxonomia de Componentes - Nível Comando

No comando enquadram-se todos os componentes que são uma infra estrutura computacional. Têm uma linguagem de comando própria e possuem, além disso, um ambiente de execução que será externo ao ambiente onde correrá a aplicação que utilizará o modelo que as classes representam (Figura 3.16).

Na Figura 3.17 apresenta-se uma possível organização de sensores, inspirada na proposta de Gonzalez [47]. A primeira grande divisão ocorre entre sensores que se destinam a fazer medições de variáveis internas ou externas, vistas do lado do componente em que o sensor é aplicado. Um sensor para medir o ângulo entre dois eixos de um robô mede, do seu ponto de vista, uma variável interna. Mas se for aplicado ao robô um sensor de forças, a medida efectuada será, do ponto de vista do robô, externa, uma vez que corresponde a medições de acções que foram realizadas sobre ele, embora isto seja mais resultado de uma tradição tecnológica do que qualquer outra razão.

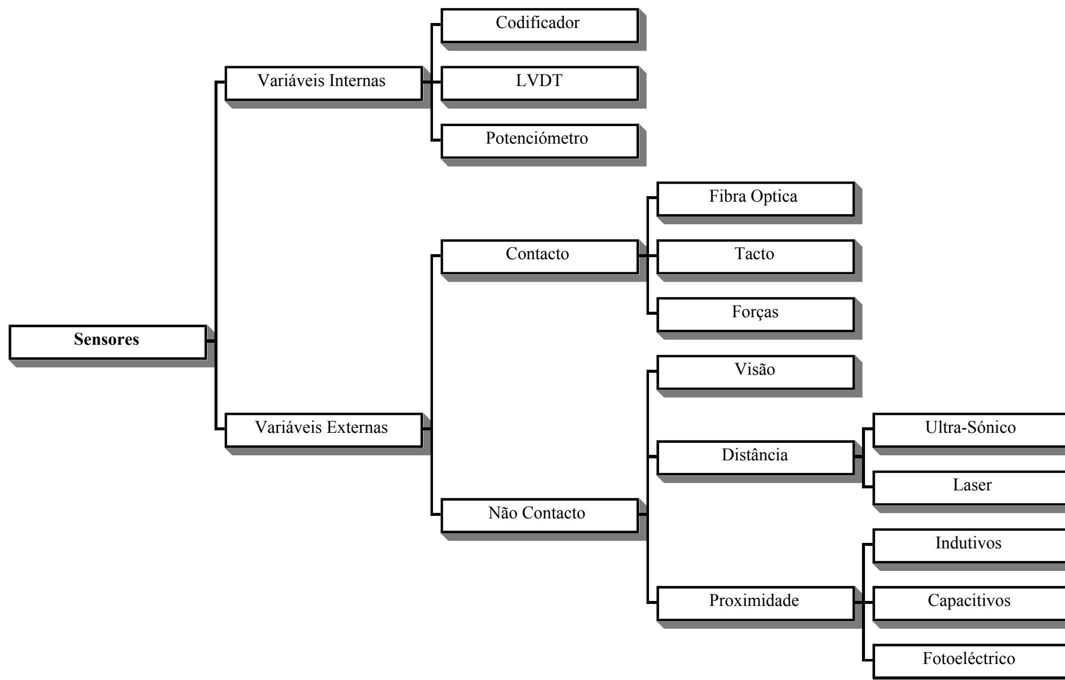


Figura 3.17 - Taxonomia de Componentes - Nível Sensor

Consideram-se componentes de armazenamento todos aqueles que, de alguma forma, armazenam (guardam) outros componentes (produtos, ferramentas, ...) existentes no sistema de manufactura (Figura 3.18).

Termina-se a apresentação realçando-se o facto de ter sido proposta uma forma de organização para os componentes de um sistema de automação. A taxonomia servirá unicamente para organizar o catálogo de onde

serão extraídos os componentes para participar nas células e sistemas. A estrutura interna dos nós desta taxonomia será apresentada a seguir.

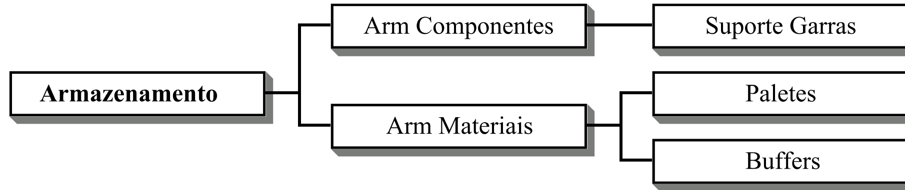


Figura 3.18 - Taxonomia de Componentes - Nível Armazenamento

3.3.2. Taxonomia de Papéis

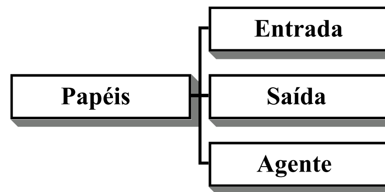


Figura 3.19 - Taxonomia de Papéis - Nível Papéis

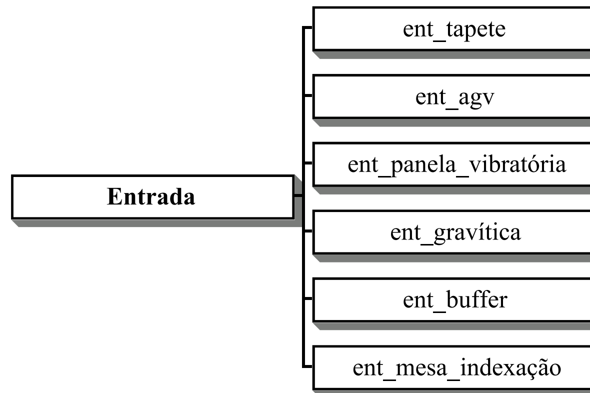


Figura 3.20 - Taxonomia de Papéis - Nível Entrada

Esta taxonomia pretende ilustrar uma classificação dos diferentes papéis existentes num sistema de automação. Os papéis serão utilizados da forma descrita em 3.2. Na Figura 3.19 mostra-se o nível mais geral, enquanto que na Figura 3.20 se mostra o nível entrada.

Uma chamada de atenção para a classificação feita para a entrada: neste caso verifica-se que haverá um papel para cada tipo de dispositivo de entrada. Aparentemente, poderia haver apenas um papel para entrada. A diferença entre as entidades que modelam o componente a desempenhar o papel de entrada, será feita pelo componente, isto é, diferentes componentes a desempenhar papéis iguais podem desempenha-los de forma distinta.

Mas existe um problema associado ao domínio do componente. Um cão, por exemplo, não desempenha as funções de guarda da mesma forma que um homem. Existem algumas semelhanças, mas também se podem apontar diferenças. A descrição dos papéis contempla de alguma forma o domínio do sujeito.

Passando para o mundo da automação está-se a sugerir que o papel de entrada é, de alguma forma, "temperado" pelo componente que vai desempenhar a tarefa. Assim, quer para a entrada, quer para a saída, quer ainda para o agente, o domínio do componente vai aparecer associado aos papéis.

No ponto 3.4 serão descritos, utilizando EIFFEL® e "frames", algumas das classes representadas na taxonomia.

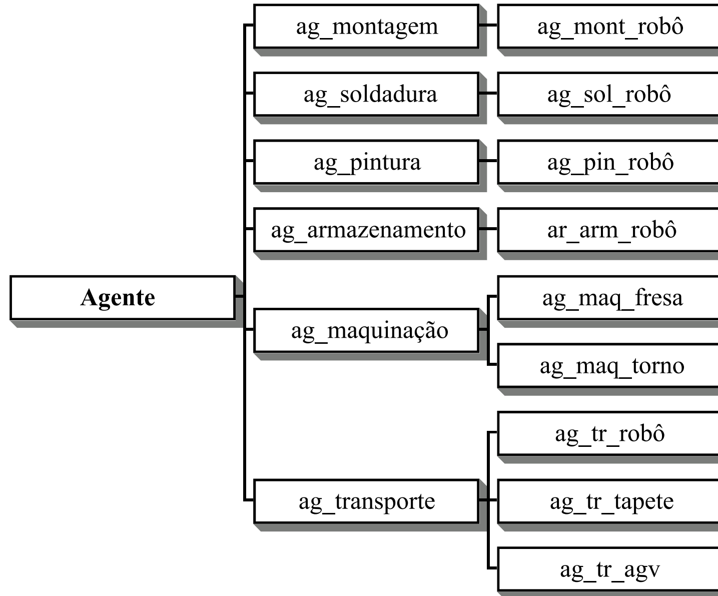


Figura 3.21 - Taxonomia de Papéis - Nível Agente

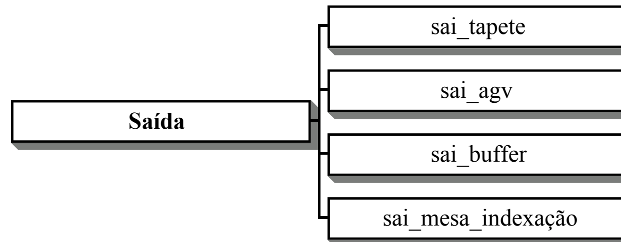


Figura 3.22 - Taxonomia de Papéis - Nível Saída

3.3.3. Células

Em 3.2 realçou-se a existência de diversos tipos de células. Uma taxonomia pode ser vista na Figura 3.23. No ponto 3.4 serão descritos, utilizando EIFFEL® e "frames", algumas das classes representadas na taxonomia.

3.3.4. Materiais

Os produtos serão classificados de acordo com a taxonomia representada na Figura 3.24. Neste ponto não existe qualquer preocupação de definir a estrutura interna das classes. Na Figura apenas estão representadas as relações ***is-a***, implementando a especialização das classes. Mas existirão outros tipos de relações para permitirem, por exemplo, relacionar um produto simples com um produto composto.

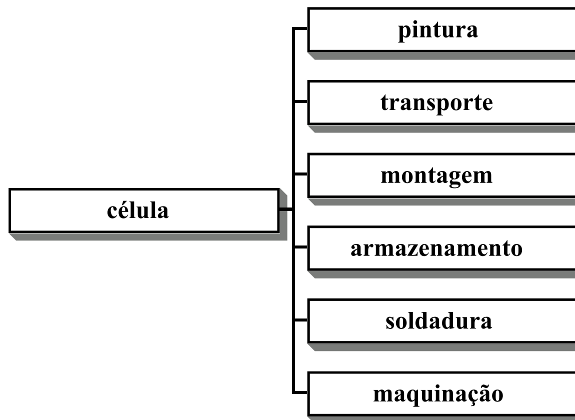


Figura 3.23 - Taxonomia de Células

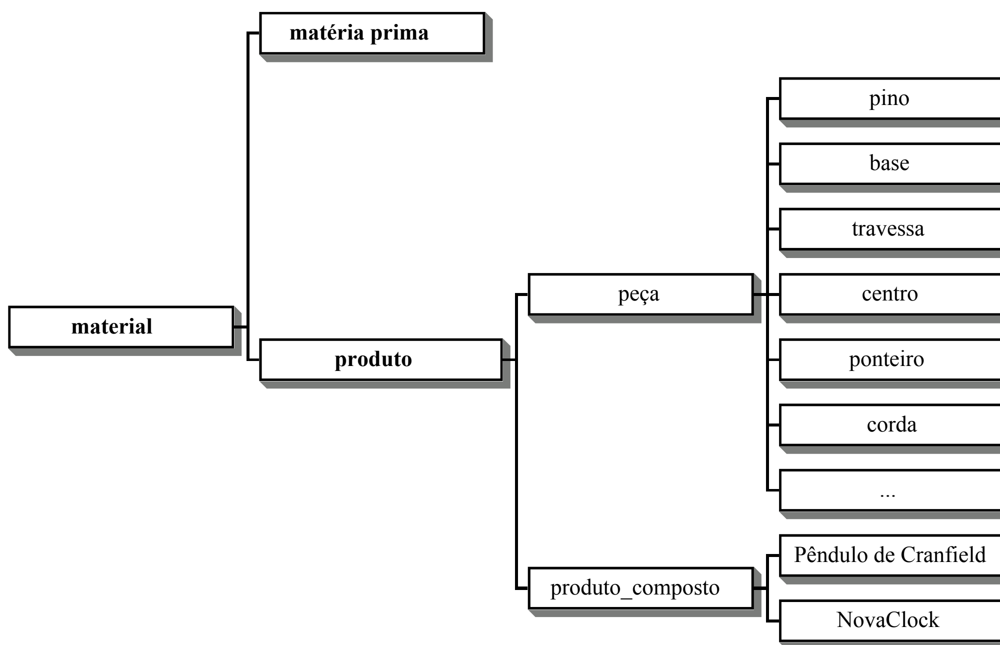


Figura 3.24 - Taxonomia de Materiais

Na figura 3.24 aparecem, a título de exemplo, alguns produtos simples que fazem parte dos produtos[34] NovaClock e Benchmark de Cranfield, montados na NovaFlex e na célula flexível construída em torno de um robô SONY, respectivamente.

3.3.5. Suportes

Suportes serão todas as entidades, destinadas a fixar materiais. Na célula do CRI, por exemplo, os ponteiros, a corda e os números são transportados em estruturas que estão colocadas sobre as paletes. Os suportes podem ser de diversos formatos, podendo transportar uma ou mais unidades de materiais. É necessário descrever as propriedades geométricas e mecânicas do *holder*, para permitir, por exemplo, a inserção e extracção dos materiais. Na figura 3.25 mostra-se a taxonomia de fixadores.

---

[34] Nos apêndices mostram-se figuras dos produtos pêndulo de Cranfield e NovaClock.

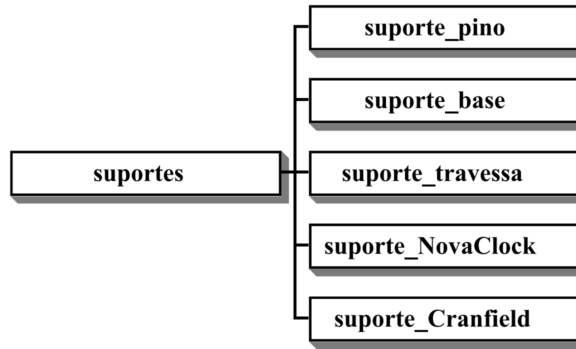


Figura 3.25 - Taxonomia de Suportes

É importante realçar o facto do Grupo de Sistemas Robóticos e CIM, ter já desenvolvido trabalho ao nível da organização das actividades CIM [48].

3.4. Comportamento Estático

Neste ponto vão-se descrever as classes de informação apresentadas em 3.3, utilizando a linguagem EIFFEL® e frames. Não se espere contudo, uma descrição detalhada e completa de todas as classes, o que seria demasiado extenso. Pretende-se apenas dar alguns exemplos significativos. Será pois natural que na descrição de uma dada classe, não se mostrem todos os seus atributos e funcionalidades.

3.4.1. Componentes

Conforme se mostrou em 3.3.1 a taxonomia é formada por relações *is-a*. Uma vez que não se pretendem modelar todas as classes vão-se modelar apenas as que correspondem a componentes que participam na célula NovaFlex/SONY, bem como os seus antecessores.

MODELO EIFFEL®

```
class COMPONENTE feature
   ...
   ref_base: REFERENCIAL;
   nome: STRING;
   fabricante: STRING;
   ...
   invariant
            nome.length /= 0;
end -- class COMPONENTE
```

MODELO FRAMES

```
FRAME COMPONENTE
      ...
      ref_base:
      nome: demon if_write verifica_tamanho
      fabricante:
      ...
end -- COMPONENTE
```

A classe componente, bem como as seguintes até indicação em contrário, são classes sem quaisquer instâncias, sendo a sua existência relacionada com aspectos meramente taxonómicos.

A classe *transportador* herda os atributos de componente e adiciona-lhe alguns, dos quais se salienta o atributo *controlado_por*. Este atributo aponta para um objecto que virtualiza a funcionalidade do transportador. Este aspecto, mais relacionado com a dinâmica do objecto será descrito em 3.5. Note-se a diferença de semântica do atributo *controlado_por*, no caso do EIFFEL e no caso dos Frames. Em EIFFEL participa numa relação de agregação enquanto que nos "*frames*" é o nome de uma relação.

Para indicar o que vai ser herdado na relação *controlado_por* (FRAMES), teve que se introduzir o *slot* *herança_controlado_por*. O seu conteúdo indica ao mecanismo que implementa a herança quais os *slots* que irão ser herdados.

```
MODELO EIFFEL®
class TRANSPORTADOR inherit
            COMPONENTE
 feature
   ...
   mat_a_transportar: MATERIAIS;
   transporta: STRING;
   velocidade: INTEGER;
   controlado_por: COMANDO;
   ...
   invariant
            not controlado_por.Void;
end -- class TRANSPORTADOR
```

```
MODELO FRAMES
FRAME TRANSPORTADOR
      is-a: COMPONENTE
      ...
      mat_a_transportar:
      transporta:
      velocidade:
      relation controlado_por: comando
      heranca_controlado_por
      ...
end -- TRANSPORTADOR
```

Em EIFFEL, a obrigatoriedade de ser controlado representa-se pelo invariante da classe. Neste caso, obriga à existência de um objecto COMANDO. O facto da relação estar declarada no frame obriga automáticamenteà sua existência.

Na classe MOVEL estão definidos os locais de acostagem válidos. O destino para onde o objecto móvel se há-de deslocar terá sempre de pertencer a um dos locais válidos. O método *dst* garante esta situação através da pré-condição (EIFFEL). O demónio *if_write*, presente no *slot destino*, assegura também a validade dos destinos (FRAMES).

```
MODELO EIFFEL®
class MOVEL inherit
         TRANSPORTADOR
 feature
     ...
     locais_acostagem_validos:
                  LINKED_LIST[REFERENCIAL];
     destino: REFERENCIAL;
     ...
     dst (ref: REFERENCIAL) is
     require
         locais_acostagem_validos.present(ref)
     do
         destino := ref
     ensure
         destino /= old destino
     end; -- dst

end -- class MOVEL
```

```
MODELO FRAMES

FRAME MOVEL
      is-a: TRANSPORTADOR    ...
      destino: demon if_write ver_val_dst()
      locais_acostagem_validos:
      ...
end -- MOVEL
```

```
MODELO EIFFEL®
class FIXO inherit
              TRANSPORTADOR
 feature
         ...
         funcionamento: BOOLEAN;
         entrada, saída: REFERENCIAL;
         ...
end -- class FIXO
```

```
MODELO FRAMES
FRAME FIXO
      is-a: TRANSPORTADOR
      funcionamento:
      entrada:
      saída:
      ...
end -- FIXO
```

O demónio atribuído ao *slot* destino do "frame" MOVEL serve para validar o destino, isto é, verifica se o valor colocado no slot é válido. Para isso recorre ao slot *locais_acostagem_válidos*, definido no mesmo frame.

O atributo *funcionamento* da classe *fixo* indica se o transportador está ou não em funcionamento.

MODELO EIFFEL®

```
class TAPETE export
        arranca, para
inherit
                FIXO
 feature
        ...
        s_digital_arranca: INTEGER;
        n_objectos: INTEGER;
        arranca is
        require
                not funcionamento
        do
          controlado_por.saída_digital(s_digital_arranca,
                                        TRUE);
          funcionamento := TRUE
        end -- arranca
        ...
end -- class TAPETE
```

MODELO FRAMES

```
FRAME TAPETE
        is-a: FIXO
        s_digital_arranca:
        n_objectos:
        method arranca: arranca_fn
end -- TAPETE

arranca_fn(TAPETE):-
  get_value(TAPETE, funcionamento, FALSE),
  get_value(TAPETE, s_digital_arranca, V),
  call_method(TAPETE, saída_digital, V)
  new_value(TAPETE, funcionamento, TRUE).
```

Poderá ainda definir-se uma nova classe como os tapetes transportadores de paletes que, neste caso, apenas transportam paletes e, como tal, é necessário redefinir *transporta* como sendo uma lista de paletes.

MODELO EIFFEL®

```
class FORNECIMENTO inherit
                COMPONENTE
 feature
        ...
        ref_entrega: REFERENCIAL;
        accionamento: STRING;
        lista_materiais: LINKED_LIST[MATERIAIS];
        n_materiais: INTEGER is
        require
                not lista_materiais.Void
        do
                Result := lista_materiais.nb_elements
        end; -- n_materiais
        ...
        invariant
                not ref_entrega.Void;
end -- class FORNECIMENTO
```

MODELO FRAMES

```
FRAME FORNECIMENTO
        ...
        is-a: COMPONENTE
        ref_entrega:
        accionamento:
        lista_materiais:
        n_materiais: demon if_read before dem_fn
end -- FORNECIMENTO

dem_fn(FORNECIMENTO, n_materiais, X):-
        get_values(FORNECIMENTO,
                      lista_materiais, Lista),
        length(Lista, Tam),
        new_value(FORNECIMENTO,
                      n_materiais, Tam).
```

MODELO EIFFEL®

```
class CONTROLAVEL inherit
                FORNECIMENTO
 feature
        ...
        controlado_por:  COMANDO;
        ...
        invariant
                not controlado_por.Void;
end -- class CONTROLAVEL
```

MODELO FRAMES

```
FRAME CONTROLAVEL
        is-a: FORNECIMENTO
        relation controlado_por:  COMANDO;
        ...
end -- class CONTROLAVEL
```

De seguida vão-se definir as características básicas dos componentes fornecedores de materiais. A classe CONTROLAVEL relaciona o controlador com o componente. Cada FORNECEDOR deve ter uma lista de objectos (atributo *lista_materiais*) pertencentes à extensão da classe materiais. De cada vez que é retirado um material deve ser retirado da lista o respectivo objecto. Repare-se que na implementação por frames definiu-se um demónio para o atributo *n_materiais*, que calcula o número de elementos no atributo *lista_materiais*.

Esta classe representa um tipo de alimentador gravítico accionado por um cilindro pneumático. O atributo *accionamento* foi redefinido no sentido de ser uma constante (EIFFEL). Os clientes do alimentador apenas têm acesso às operações *fornece*, *n_materiais* e *carregamento* (EIFFEL).

```
MODELO EIFFEL®
class ALIM_GRAV_SONY  export
                  fornece, n_materiais, carregamento
inherit CONTROLAVEL redefine accionamento
 feature
    accionamento: STRING is "Pneumatico";
    sensor_pecas: S_PROXIMIDADE;
    sensor_ext, sensor_rec: S_PROXIMIDADE;
    s_digital_extende, s_digital_recolhe: INTEGER;
    fornece: MATERIAIS is
    require
          sensor_rec.estado
    do
         controlado_por..saída_digital(s_digital_extende);
         from  -- não inicializado until sensor_ext.estado
         loop  -- nada end;
         controlado_por.saída_digital(s_digital_recolhe);
         Result := lista_materiais.first;
         lista_materiais.delete;
     ensure
         lista_materiais.nb_elements  = old
                         lista_materiais.nb_elements  -  1;
         rescue
            controlado_por..saída_digital(s_digital_recolhe);
     end; -- fornece
     carregamento( L: LINKED_LIST[MATERIAIS]) is
     require
            sensor_rec.estado
    do - Coloca L em lista_materiais end; -- carregamento
            invariant
            not lista_materiais.Void; sensor_pecas.estado
            lista_materiais.nb_elements /= 0;
end -- class ALIM_GRAV_SONY
```

```
MODELO FRAMES

FRAME AL_GRAV inherit
      is-a: CONTROLAVEL
      ...
      accionamento: Pneumatico
      sensor_pecas
      sensor_ext, sensor_rec:
      s_digital_extende, s_digital_recolhe:
      fornece: demon if_read before forn_dem
      method carregamento: carg_fn
end -- AL_GRAV

forn_dem(AL_GRAV , fornece, X) :-
     get_value(AL_GRAV , sensor_rec, Id),
     get_value(Id, estado, TRUE),
     get_value(AL_GRAV,s_digital_extende,V),
     call_method(AL_GRAV , saída_digital, V),
     espera_sensor_extendido,
     get_value(AL_GRAV,s_digital_recolhe,K),
     call_method(AL_GRAV , saída_digital, K),
     get_value(AL_GRAV, lista_materiais, Mat),
     new_value(AL_GRAV, fornece, Mat),
     delete_value(AL_GRAV, lista_materiais, Mat).

carga_fn(AL_GRAV, LMat) :-
     get_value(AL_GRAV , sensor_rec, Id),
     get_value(Id, estado, TRUE),
     new_values(AL_GRAV, lista_materiais,
LMat).
```

A primeira é uma operação complexa que desencadeia uma acção sobre o controlador e garante ao cliente a entrega do objecto que representa o material que foi fornecido. Houve a preocupação de garantir sempre que o alimentador esteja recolhido antes de se iniciar a operação. Se qualquer invariante, pré ou pós condição, falhar coloca-se o alimentador recolhido, utilizando o construtor **rescue** (EIFFEL).

No modelo de Frames o fornecimento é realizado quando se lê o atributo *fornece*, que devolve um identificador do material que acabou de ser fornecido. As acções de controlo sobre o alimentador são desencadeadas pelo demónio que está associado ao atributo.

A operação *carregamento* deve ser utilizada por um cliente sempre que os materiais acabarem, obrigando-se a fornecer uma lista contendo a descrição dos materiais que foram introduzidos.

```
MODELO EIFFEL®
class TRANSFORMAÇÃO  inherit
                  COMPONENTE
 feature
          accionamento: STRING;
          controlado_por: COMANDO;
          ...
          invariant
                  not  controlado_por.Void
end -- class TRANSFORMAÇÃO
```

```
MODELO FRAMES
FRAME TRANSFORMAÇÃO
      is-a: COMPONENTE
      accionamento:
      relation controlado_por: COMANDO;
end -- TRANSFORMAÇÃO
```

Na classe robô não está descrita nenhuma funcionalidade, apenas se descrevem atributos que representam o estado. Os clientes acederão à funcionalidade através do atributo *controlado_por*. Realce-se que esta questão da funcionalidade é um dos aspectos mais importantes da modelação de componentes de manufactura, daí que esteja previsto um ponto só para discussão destes aspectos.

```
MODELO EIFFEL®
class ROBÔ  inherit
            TRANSFORMAÇÃO
 feature
    ...
    aplicações: LIST[STRING];
    dof: INTEGER;
    àrea_trabalho: INTEGER;
    carga: INTEGER;
    repetabilidade: REAL;
    posição_corrente: REFERENCIAL;
    erro: ROBOT_ERROR;
    custo: INTEGER;
    tempo_ciclo: REAL;
    próxima_manutenção: DATE;
    horas_trabalho: INTEGER;
    peso: INTEGER;
    resolução: INTEGER;
    vel_max_eixo: INTEGER;
    ...
    invariant
        data_superior(next_maintenace, current_date);
        not controlado_por.Void;
end -- class ROBÔ
```

```
MODELO FRAMES
FRAME ROBÔ
    is-a: TRANSFORMAÇÃO
    aplicações:
    dof:
    àrea_trabalho:
    carga:
    repetabilidade:
    posição_corrente:
    erro:
    custo:
    tempo_ciclo:
    próxima_manutenção:
    horas_trabalho:
    peso:
    resolução:
    vel_max_eixo:
    ...
end -- ROBÔ
```

Deve-se encarar o robô, enquanto descrito na taxonomia de componentes, como uma entidade estática, onde não aparecem entidades que lhe são intrinsecamente externas, como a ferramenta, por exemplo. O robô será uma estrutura completa quando estiver a desempenhar um determinado papel. Essa estrutura será definida (1) através da herança das características do papel, onde poderá estar definida a ferramenta e (2) por utilização do robô, acedendo às suas características relevantes.

Desde que a data de manutenção seja superior à data actual, não podem ser realizadas operações sobre o robô. Esta restrição é simples de implementar em EIFFEL, através da declaração da condição no invariante da classe. Mas em Frames já não é tão simples, porque não existe um mecanismo que permita associar a qualquer acesso (leitura/escrita) um método/demónio que verifique a condição.

```
MODELO EIFFEL®
class SCARA  export
    controlado_por { AG_MNT_ROBOT,
                     AG_ARM_ROBOT,
                     AG_TR_ROBOT },
    próxima_manutenção { MANUTENÇÃO },
    repetabilidade { CONFIGURAÇÃO },
    aplicações { CONFIGURAÇÃO },
    àrea_trabalho { CONFIGURAÇÃO },
    carga { CONFIGURAÇÃO },
    custo { CONFIGURAÇÃO },
    tempo_ciclo { CONFIGURAÇÃO },
    horas_trabalho { MANUTENÇÃO },
    peso { CONFIGURAÇÃO }, ...
inherit
            ROBÔ redefine  dof,  applications
 feature
        ...
        dof: INTEGER is 4;
        desl_eixo_z: INTEGER;
        ...
end -- class SCARA
```

```
MODELO FRAMES
FRAME SCARA
    is-a: ROBÔ
    heranca_controlado_por: input, output,
                            move_lin, move_circ,
                            move_eixo,
                            velocidade,
                            aceleração
    dof: 4
    desl_eixo_z:
    ...
end -- SCARA
```

O mais importante da definição da classe SCARA é a declaração do export. O EIFFEL® permite que uma classe defina os clientes que podem aceder aos atributos declarados como públicos. Por exemplo, o atributo *controlado_por*, poderá apenas ser utilizado pelas classes AG_MNT_ROBOT, AG_ARM_ROBOT e AG_TR_ROBOT.

Como já foi referido, o robô, neste caso SCARA, irá ser utilizado pelas entidades complexas (AG_MNT_ROBOT, AG_ARM_ROBOT, AG_TR_ROBOT, ...) que participam na célula. A esta entidade apenas interessam as características relevantes do componente indicado para desempenhar determinada tarefa, daí que os atributos relacionados com a conFiguração não sejam exportados para a entidade que representa o robô a desempenhar o papel de montador.

Os atributos relacionados com aspectos de manutenção - *próxima_manutenção* e *horas_trabalho*, apenas poderão ser utilizados por classes relacionadas com a manutenção. Os relacionados com a conFiguração apenas podem ser relacionados com essa classe.

Com o sistema de frames não é possível utilizar o mesmo mecanismo de restrição de clientes. Qualquer slot/método que pertença ao frame, pode ser acedido por qualquer cliente.

MODELO EIFFEL®

```
class FERRAMENTA  inherit
            COMPONENTE
 feature
        accionamento: STRING;
        controlado_por: COMANDO;
        peso: INTEGER;
        trabalha_em: TRANSFORMAÇÃO;
        repousa_em: SUPORTE_GARRAS;
        tcp: REFERENCIAL;
        ...
        invariant
        not  controlado_por.Void;
        (not trabalha_em.Void and repousa_em.Void) or
        (trabalha_em.Void and not repousa_em.Void)
end -- class FERRAMENTA
```

MODELO FRAMES

```
FRAME FERRAMENTA
    is-a: COMPONENTE
    accionamento:
    relation controlado_por: COMANDO;
    heranca_controlado_por:  saida_digital
    trabalha_em: demon if_write after ver_cons
    repousa_em: demon if_write after ver_cons
    tcp:
    ...
end -- FERRAMENTA
```

MODELO EIFFEL®

```
class GARRAS  export
            abre, fecha
inherit
            FER_ROBÔ
 feature
        modo_preensão: STRING;
        com_peça: BOOLEAN;
        repetibilidade, stroke, força_preensão,
         tempo_abertura: REAL;
        sensor_aberta,
         sensor_fechada: S_PROXIMIDADE;
        s_digital_aberta, s_digital_fechada: INTEGER;
        abre is
        require
          not trabalha_em.Void; not sensor_aberta.estado
        do
          controlado_por.saída_digital(s_digital_aberta);
          com_peca := FALSE;
        assure
            sensor_aberta.estado; not com_peça;
        end; -- ABRE
        fecha is
        require
          not trabalha_em.Void;
          not sensor_fechado.estado
          do
          controlado_por.saída_digital(s_digital_fechada);
          if not sensor_fechado. estado then
            com_peca := TRUE;
          end;
        assure
          sensor_fechado.estado
        end; -- FECHA
        invariant
            not ( aberta and fechada )
end -- class GARRAS
```

MODELO FRAMES

```
FRAME GARRAS
    is-a: FER_ROBÔ
    modo_preensão:
    com_peça:
    repetibilidade: abertura:
    força_preensão: tempo_abertura:
    sensor_aberta:
     sensor_fechada:
    s_digital_aberta: s_digital_fechada:
    method abre: abre_fn
    method fecha: fecha_fn

abre_fn(GARRAS) :-
   get_value(GARRAS , sensor_aberta, Id),
   get_value(Id, estado, FALSE),
   get_value(GARRAS,s_digital_aberta,V),
   call_method(GARRAS , saída_digital, V),
   new_value(GARRAS, com_peça, FALSE),
   get_value(Id, estado, TRUE).

fecha_fn(GARRAS) :-
   get_value(GARRAS , sensor_fechado, Id),
   get_value(Id, estado, FALSE),
   get_value(GARRAS,s_digital_fechada,V),
   call_method(GARRAS , saída_digital, V),
     get_value(GARRAS , sensor_fechado, Id),
   fechou(Id).

fechou(ID) :-
   get_value(Id, estado, TRUE).
fechou(ID) :-
   get_value(Id, estado, FALSE),
   new_value(GARRAS, com_peça, TRUE).
```

```
                    MODELO EIFFEL®
class PNEUMÁTICAS  inherit
                    GARRAS  redefine accionamento
 feature
          accionamento: STRING is "PNEUMÁTICO";
          max_pressão: INTEGER;
          ...
          invariant
                    existe_ar
end -- class PNEUMÁTICAS
```

```
               MODELO FRAMES
FRAME PNEUMÁTICAS
      is-a: GARRAS
      accionamento:  pneumático
      max_pressão:
end -- PNEUMÁTICAS
```

Na classe ferramenta os atributos mais relevantes são *trabalha_em* e *repousa_em* que terão de ser mutuamente exclusivos, isto é, uma ferramenta ou está associada a um componente de TRANSFORMAÇÃO ou então está colocada num SUPORTE DE FERRAMENTAS. O invariante garante este facto, em EIFFEL. A manutenção da consistência a este nível, usando Frames, é feita recorrendo aos demónios *if_write* que se associaram aos atributos *trabalha_em* e *repousa_em*. Uma qualquer alteração provoca imediatamente uma verificação de consistência.

As operações sobre os objectos do tipo GARRA são: abre e fecha. Os restantes atributos destinam-se a suportar essa funcionalidade. O objecto garra memoriza a preensão de uma entidade sempre que for dado sinal para a garra se fechar e o sensor de garra fechada não tiver sido actuado. na implementação de frames mostra-se um trecho de código que permite a abertura ou o fecho da garra. Repare-se no modo como o componente é actuado fisicamente: através da chamada ao método *saída_digital* que foi herdado do controlador através da relação *controlado_por*. Note-se ainda o modo diferente como as duas implementações (Frames e EIFFEL) tratam a sua chamada; nos frames o método está em GARRAS (não se indicou no frame, por ser herdado do controlador), através da relação *controlado_por*; no EIFFEL, o método está no controlador, tendo-se obtido o seu ID através do atributo *controlado_por*.

A classe FIXADOR é uma subclasse da classe MECÂNICO, que por seu lado é uma subclasse de SUPORTE.

A funcionalidade da classe FIXADOR é dada por *abre* e *fecha(holder_arg: PALETES)*. Estas duas rotinas são muito semelhantes às da classe GARRA, encontrando-se a maior diferença na *fecha* que recebe como parâmetro um objecto da classe PALETES. O objecto FIXADOR, enquanto não fôr aberto, memoriza a palete que está a fixar.

```
                    MODELO EIFFEL®
class SUPORTE  inherit
                    COMPONENTE
 feature
          accionamento: STRING is "PNEUMÁTICO";
          controlado_por: COMANDO;
          ...
          invariant
                    not controlado_po.Void
end -- class SUPORTE
```

```
               MODELO FRAMES
FRAME SUPORTE
      is-a: COMPONENTE
      accionamento:
      relation controlado_por:
      heranca_controlado_por:  saida_digital
end -- SUPORTE
```

```
                    MODELO EIFFEL®
class MECÂNICO  inherit
                    SUPORTE
 feature
          ...
end -- class MECÂNICO
```

```
               MODELO FRAMES
FRAME MECÂNICO
      ...
end -- MECÂNICO
```

```
                    MODELO EIFFEL®
class FIXADOR  export
                    abre, fecha
inherit
                    MECÂNICO
 feature
          agarra: PALETES;
          sensor_aberta,
           sensor_fechada: S_PROXIMIDADE;
          s_digital_aberta, s_digital_fechada: INTEGER;
          abre is
          require
            not sensor_aberta.estado
          do
            controlado_por.saída_digital(s_digital_aberta);
            agarra := agarra.Forget;
          assure
                    sensor_aberta.estado;  agarra.Void;
          end; -- ABRE
          fecha(pal:PALETES) is
          require
            not sensor_fechado.estado
            do
            controlado_por.saída_digital(s_digital_fechada);
            agarra := pal;
          end; -- FECHA
          invariant
                    not ( aberta and fechada )
end -- class FIXADOR
```

```
                 MODELO FRAMES
FRAME FIXADOR
        is-a: MECÂNICO
        agarra:
        sensor_aberta:
         sensor_fechada:
        s_digital_aberta: s_digital_fechada:
        method abre: abre_fn
        method fecha: fecha_fn

abre_fn(FIXADOR  ) :-
     get_value(FIXADOR  , sensor_aberta, Id),
     get_value(Id, estado, FALSE),
     get_value(FIXADOR  ,s_digital_aberta,V),
     call_method(FIXADOR , saída_digital, V),
     new_value(FIXADOR  , agarra, nil),
     get_value(Id, estado, TRUE).

fecha_fn(FIXADOR  , Pal) :-
     get_value(FIXADOR  , sensor_fechado, Id),
     get_value(Id, estado, FALSE),
     get_value(FIXADOR  ,s_digital_fechada,V),
     call_method(FIXADOR  , saída_digital, V),
     new_value(FIXADOR  , agarra, Pal).
```

```
                    MODELO EIFFEL®
class ARMAZENAMENTO  inherit
                    COMPONENTE
 feature
          ...
          lista_materiais: LINKED_LIST[T];
end -- class ARMAZENAMENTO
```

```
                 MODELO FRAMES
FRAME ARMAZENAMENTO
        is-a: COMPONENTE
        lista_materiais:
end -- ARMAZENAMENTO
```

```
                    MODELO EIFFEL®
class SUPORTE_GARRAS  export
          armazena_garra, retira_garra, garra_presente
inherit
          ARM_COMPONENTES
rename  lista_materiais as garra redefine lista_materiais
 feature
          garra: GARRAS;
          sensor_garra_p: S_PROXIMIDADE;
          armazena_garra( g: GARRAS ) is
          require not sensor_garra.estado
          do
                    garra := g;
          assure not garra.Void;
          end; -- ARMAZENA_GARRA
          retira_garra: PALETES is
          require sensor_garra.estado
          do
                    Result := garra;
                    garra := garra.Forget;
          assure not garra.Void;
          end; -- RETIRA_GARRA
          garra_presente: BOOLEAN is
          require
                    not garra.Void;
          do
                    Result := sensor_garra.estado;
          end; -- GARRA_PRESENTE
end -- class SUPORTE_GARRAS
```

```
                 MODELO FRAMES
FRAME SUPORTE_GARRAS
        is-a: ARM_COMPONENTES
        garra:
        sensor_garra_p
        garra_presente: demon  before if_read g_pres
        retira_garra: demon  before if_read ret_g_fn
        method armazena_garra: arm_g_fn

arm_g_fn(SUPORTE_GARRAS, G) :-
   get_value(SUPORTE_GARRAS, sensor_garra_p, Id),
   get_value(Id, estado, FALSE),
       new_value(SUPORTE_GARRAS, garra, G).

ret_g_fn(SUPORTE_GARRAS, retira_garra, Info) :-
       get_value(SUPORTE_GARRAS, garra, Id),
       new_value(SUPORTE_GARRAS, retira_garra, Id),
       new_value(SUPORTE_GARRAS, garra, nil).

g_pres_fn(SUPORTE_GARRAS, garra_presente, Info) :-
   get_value(SUPORTE_GARRAS, sensor_garra_p, Id),
   get_value(Id, estado, Est),
   new_value(SUPORTE_GARRAS, garra_presente, Est).
```

As funcionalidades principais *retira_garra* e *armazena_garra* vão ser utilizadas através da funcionalidade *trocar_ferramenta* existente em AG_MNT_ROBOT que, como é sabido, representa o robô a desempenhar o papel de montador. O acto físico de substituir uma garra será desencadeado por uma rotina existente na estrutura referida que utilizará funcionalidades da garra e do controlador.

MODELO EIFFEL®

```
class SENSOR inherit
        COMPONENTE
 feature
        controlado_por: COMANDO;
        usado_em: COMPONENTE;
        estado: REAL is
        deferred
        end; -- ESTADO
        ...
        invariant
        not controlado_por.Void; not usado_em.Void;
end -- class SENSOR
```

MODELO FRAMES

```
FRAME SENSOR
        is-a: COMPONENTE
        estado:
        usado_em:
        relation controlado_por:
        heranca_controlado_por:  entrada_digital
end -- SENSOR
```

MODELO EIFFEL®

```
class INDUTIVO export estado
 inherit
        PROXIMIDADE redefine estado
 feature
        ent_digital: INTEGER;
        distância_minima_detecção: REAL;
        estado: BOOLEAN is
        do
     Result := controlado_por.entrada_digital(ent_digital);
        end; -- ESTADO
        ...
end -- class INDUTIVO
```

MODELO FRAMES

```
FRAME INDUTIVO
        is-a: PROXIMIDADE
        ent_digital
        distância_minima_detecção:
        estado: demon  before if_read estado_fn
estado_fn(INDUTIVO, estado, Info) :-
     get_value(INDUTIVO, ent_digital, Id),
     concatena(entrada_digital, Id, Ent),
     get_value(INDUTIVO, Ent, Id).
```

Das classes relacionadas com sensores optou-se por indicar apenas um exemplo para um sensor indutivo. Na definição da classe SENSOR e na INDUTIVO mostram-se as características mais relevantes. Um sensor indutivo é um sensor binário cujo valor é obtido através da leitura da porta digital de entrada de um controlador.

Com esta definição terminou-se a ilustração da descrição de classes de componentes. Não se descreveram as classes relacionadas com COMANDO, pela simples razão de se ter optado pela sua definição no ponto 3.5 - Comportamento Dinâmico.

3.4.2. Papéis

No modelo da célula, a entrada, saída e o agente são caracterizados por entidades que representam um dado componente a desempenhar um dado papel. Na Figura 3.26 mostra-se a relação que existe entre uma destas entidades e o papel e o componente.

O objecto *ag_mont_robô* será o agente efectivo da célula, enquanto que o componente *robô scara* e o papel *ag_montagem* serão intervenientes indirectos da célula. As características do robô e do papel serão reflectidas no agente efectivo.

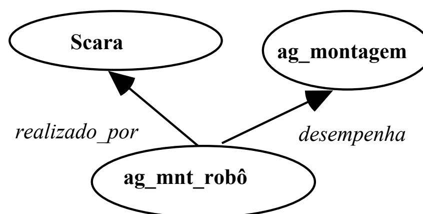


Figura 3.26 - Modelo de um Agente Robô a Desempenhar o Papel de Montador

A implementação das entidades que representam um componente a desempenhar um dado papel poderia ser realizada de diversas formas. A forma mais natural talvez seja considerar *realizado_por* e *desempenha* como sendo relações entre as classes. Como é sabido, a relação mais comum entre classes na POO, é a relação *is-a*. Na semântica que caracteriza esta relação, todos os atributos e métodos são herdados. De um ponto de vista conceptual, parece ser incorrecto afirmar que *ag_mont_robô* é um *ag_montagem* e que é também um *robô*. uma vez que ele é apenas uma entidade que desempenha um dado papel ao mesmo tempo que é realizado pelo robô.

Sabendo que o modelo de programação do EIFFEL® não suporta a definição de relações de herança, definidas pelo utilizador, houve que adaptar o paradigma no sentido de implementar o conceito de entidade a representar um dado papel.

A primeira alteração significativa é considerar, por exemplo, *ag_mnt_robô* como sendo um *ag_montagem*, sendo por isso estabelecida uma relação *is-a* entre estas 2 classes. Cada entidade que representa um dado papel estará sempre relacionada com um papel por uma relação *is-a* - vide Figura 3.21.

Usando o modelo de modelação por frames, torna-se mais simples implementar os agentes a desempenharam qualquer papel. Neste caso ag_mnt_robô será constituído pelas 2 relações:

```
FRAME AG-MNT_ROBÔ
is-a: agente
relation desempenha:AG-MONTAGEM
relation realizado-por:SCARA
```

Em EIFFEL, a relação entre AG_MNT_ROBÔ e SCARA será estabelecida à custa da agregação/composição, ou seja, dentro da classe será definido um atributo que representa o componente que participa no papel - Figura 3.27.

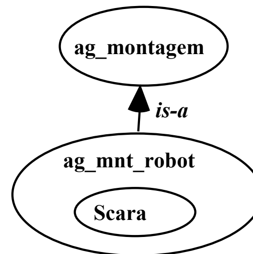


Figura 3.27 - Implementação em POO Tradicional

As características do componente robô estão encapsuladas no objecto *ag_mnt_robô*.

Consoante o papel que o componente está a desempenhar assim irão variar os atributos e funcionalidades que a entidade papel-objecto necessita conhecer. A garantia de que a entidade apenas acede aos atributos e funcionalidades relevantes é garantida, no caso da modelação em EIFFEL®, pelo mecanismo de restrição de clientes. Para recordar, veja-se a modelação da classe SCARA.

A vantagem de se criarem entidades papel-objecto reside na possibilidade de se criarem novas funcionalidades na ENTRADA, independentemente do componente. Evidentemente que é uma noção relativa de independência já que um determinado comportamento, condicionado por um dado papel, nunca pode ser completamente independente do sujeito que o executa (componente).

Na definição das classes anteriores os aspectos mais relevantes situam-se ao nível da função *Create* e *obtem_parte*, definidas na classe ENT_GRAVITIC. *Create* foi redefinida de forma a permitir a reinicialização do objecto e a transferência do referencial, existente no componente, para a entidade papel/objecto. *Obtem_parte*, neste caso, utiliza directamente a funcionalidade do componente (alimentador gravítico), mas poderá ser enriquecida com outras acções, garantindo-se a possibilidade de expansão do software sem grandes comprometimentos com os clientes da classe.

Uma questão que se pode levantar na criação das entidades papel/objecto é a altura em que é criado o objecto que representa o componente. Na descrição da classe ENT_GRAVITIC o componente aparece como sendo criado na altura da criação de um objecto. Esta abordagem foi utilizada apenas para demonstrar que o objecto alimentador gravítico tem de ser criado.

Uma outra forma será considerar que o objecto componente já foi criado e que, na altura da criação da classe ENT_GRAVITIC, apenas se passa a sua referência.

Em ambas as aproximações pode supor-se a existência de uma fase inicial de configuração que consiste em escolher os objectos que irão participar no sistema [49-51] e fazer a sua validação formal. Será durante a configuração que se criarão as entidades papel/objecto, as células, componentes, etc.

MODELO EIFFEL®

```
class PAPÉIS
 feature
        ...
        nome_papel: STRING;
        desempenha: COMPONENTE;
        pertence: CÉLULA;
        invariant
                not desempenha.Void
end -- class PAPÉIS
```

MODELO FRAMES

```
FRAME PAPÉIS
    is-a: concept
    nome_papel:
    pertence:
```

MODELO EIFFEL®

```
class ENTRADA  inherit
                PAPÉIS redefine nome_papel
 feature
        ...
        nome_papel: STRING is "ENTRADA"
        ref_entrada: REFERENCIAL;
        obtem_parte: MATERIAIS is
        defered
        end; -- obtem_parte
        obtem_referencial: REFERENCIAL is
        require
                not ref_entrada.Void
        do
                Result := ref_entrada;
        end; -- obtem_referencial
end -- class ENTRADA
```

MODELO FRAMES

```
FRAME ENTRADA
    is-a: PAPÉIS
    nome_papel: ENTRADA
    ref_entrada:
    parte_id:
```

Nos sistemas de Frames não é necessário definir métodos para aceder aos atributos. Assim não se declarou, em ENTRADA, o método *obtém_referencial*, pois facilmente se acede ao *slot ref_entrada*, usando a primitiva *get_value(ENTRADA, ref_entrada, Ref)*.

MODELO EIFFEL®

```
class ENT_GRAVITIC  export
                obtem_referencial, obtem_parte
inherit
                ENTRADA redefine desempenha
 feature
        desempenha: ALIM_GRAV_SONY;
        obtem_parte: MATERIAIS is
        do
                Result := desempenha.fornece;
        end; -- obtem_parte
        Create is
        do
                ...
                desempenha.Create;

                ref_entrada := desempenha.ref_entrega
                ...
        end; -- Create
end -- class ENT_GRAVITIC
```

MODELO FRAMES

```
FRAME ENT_GRAVITIC
    is-a: ENTRADA
    nome_papel: ENT_GRAVITIC
    ref_entrada:
    atributos-principais: ref-entrada,
                          obtém_parte
    atributos-componentes: fornece
    method obtém_parte: obt_parte

obt_parte(ENT_GRAVITIC) :-
   get_value(ENT_GRAVITIC, fornece, X),
   new_value(ENT_GRAVITIC, parte_id, X).
```

Parte_id é um atributo que foi criado no frame ENTRADA e destina-se a armazenar o identificador da parte que foi fornecida. Note-se que em EIFFEL não é necessário fazer o mesmo, já que *obtém_parte* é uma função que, para além de provocar o fornecimento, devolve o identificador da peça fornecida.

Começam-se agora a notar as diferenças de implementação, provocadas pelo facto do EIFFEL não suportar relações definidas pelo utilizador. No frame ENT_GRAVITIC, os atributos *atributos-principais* e *atributos-componentes* destinam-se a parameterizar a relação *realiza* e *desempenhado_por*, respectivamente. *Realiza* é a relação que se estabelece entre o agente e o papel, enquanto que *desempenhado_por* é a relação que se estabelece entre o agente e o componente. Note-se que *atributos-principais* indica os slots importantes do frame ENT_GRAVITIC que serão utilizados pelo agente (*ref-entrada* e *obtém_parte*) e *atributos-componentes* os relevantes do componente (fornece).

O fornecimento do material vai ocorrer pela acção de get_value(ENT_GRAVITIC, fornece, X). Devido à relação *desempenhado_por*, que se estabelece entre o agente e a classe alimentador gravítico, o atributo *fornece* desta última classe é herdada pelo agente. A acção de leitura provocada por *get_value* faz o disparo do demónio que, entre outras coisas, provoca a actuação da saída digital que controla o cilindro pneumático que faz o fornecimento.

MODELO EIFFEL®

```
class SAÍDA  inherit
                PAPÉIS redefine nome_papel
 feature
        ...
        nome_papel: STRING is "SAÍDA"
        ref_saída: REFERENCIAL;
        põe_parte(mat: MATERIAIS) is
        defered
        end; -- põe_parte

        obtem_referencial: REFERENCIAL is
        require
                not ref_saída.Void
        do
                Result := ref_saída;
        end; -- obtem_referencial
end -- class SAÍDA
```

MODELO FRAMES

```
FRAME SAÍDA
    is-a: PAPÉIS
    nome_papel: SAÍDA
    ref_saída
```

Os atributos relevantes da classe SAÍDA são *ref_saída* e *põe_parte*.

MODELO EIFFEL®

```
class SAI_BUFFER  export
                obtem_referencial, pôe_parte
inherit
                SAÍDA redefine desempenha
 feature
        ...
        desempenha: BUFFER;

        pôe_parte ( mat: MATERIAIS ) is
        require
                not desempenha.completo
        do
                ref_saída := desempenha.insere(mat);
        end; -- pôe_parte

end -- class SAI_BUFFER
```

MODELO FRAMES

```
FRAME SAI_BUFFER
    is-a: SAÍDA
    nome_papel: SAI_BUFFER
    atributos-principais: ref_saída, põe_parte
    atributos-componentes: insere, completo,
            prox_pos
    method põe_parte: põe_parte

põe_parte(SAI_BUFFER, Mat) :-
  get_value(SAI_BUFFER, completo, false),
  call-method(SAI_BUFFER, insere, Mat),
  get_value(SAI_BUFFER, prox_pos, Ref),
  new_value(SAI_BUFFER, ref_saída, Ref).
```

Na classe SAI_BUFFER definiu-se *põe_parte*, já que é dependente do componente. Sempre que se chama este método, o atributo *ref_saída* é actualizado com o valor da próxima posição de inserção no buffer. A implementação em Frames obriga à existência de um atributo *prox_pos*, no frame BUFFER, e contém a próxima posição de inserção. Este atributo é alterado pela acção do método *insere*.

Para um melhor esclarecimento do modo como as entidades papel-objecto podem ser utilizadas pelos clientes, mostra-se um trecho de código que represента um agente de uma célula de montagem a transferir um dado material para a saída.

MODELO EIFFEL®

```
...
apanha_peça;
ref := saída.obtem_referencial;
movimenta_robô(ref);
pôe_peça(material);
...
```

MODELO FRAMES

```
transfere(Self, Mat):-
        ....
        apanha_peça,
        get_value(Self, ref_saída, Val),
        movimenta_robô(Val),
        call-method(Self, põe_parte, Mat).
```

CLASSE AG_MONTAGEM

```
class AG_MONTAGEM inherit
            AGENTE redefine  nome_papel
 feature
   nome_papel: STRING is "AG_MONTAGEM"
   armazem_ferramentas: ARRAY[SUPORTE_GARRAS];
   fix: FIXADOR;
   ferramentas_disponíveis: ARRAY[GARRAS];
   ferramenta_corrente: GARRAS;
   s_forças: SENSOR_FORÇAS;
   troca_ferramenta: SISTEMA_TROCA;
   agarrar (ref: REFERENCIAL) is
   require
            desempenha.error = 0;
            not ferramenta_corrente.Void
   do
            desempenha.controlado_por.move(ref);
            ferramenta_corrente.fecha;
   end; -- agarrar
   largar( ref:REFERENCIAL) is
   end; -- largar
   trocar_ferramenta( index_armazem: INTEGER): BOOLEAN is
   require
            index_armazem <= armazem_ferramentas.size;
            procura_vago >= 0;
            armazem_ferramentas.entry(index_armazem).ocupado
   local
            ref: REFERENCIAL;
            val: SUPORTE_GARRAS;
   do
            val := armazem_ferramentas.entry(procura_vago(armazem_ferramentas));
            ref := val.ref_ap;
            desempenhado_por.controlado_por.move(ref);
            ref := val.ref_larga;
            desempenhado_por.controlado_por.move(ref);
            troca_ferramenta.larga;
            val.holds := ferramenta_corrente;
            ferramenta_corrente.Forget;
            ref := val.ref_ap;
            desempenhado_por.controlado_por.move(ref);
            val := armazem_ferramentas.entry(index_armazem);
            -- movimentos normais
            ferramenta_corrente := val.holds;
            val.holds.Forget;                              -- O suporte fica sem ferramenta
            ferramenta_corrente.altera_trabalha_em(Current);
   assure
            not ferramenta_corrente.Void;
   end; -- trocar_ferramenta
movimento_guardado( ref: REFERENCIAL) is   end; -- movimento_guardado
invariant
   ferramenta_disponivel.size <= armazem_ferramentas.size;
end -- class AG_MONTAGEM
```

A classe AG_MONTAGEM, definida em baixo, apresenta-se como sendo a que melhor representa o conceito de papel. Os atributos descritos representam aquilo que verdadeiramente caracteriza o papel de montagem. Com efeito, para que este papel possa ser representado, é necessário que exista um local que indique qual a ferramenta corrente do agente: *ferramenta_corrente*, ao mesmo tempo que é necessário saber qual o número de ferramentas que existem para desempenhar o papel: *ferramentas_disponíveis* e, como seria de esperar, o local onde essas ferramentas vão estar armazenadas: *armazem_ferramentas*. Também é necessário saber qual o fixador que irá ser utilizado: *fix*.

As funcionalidades principais terão de acompanhar o comportamento que se espera para este papel. Podem-se destacar os seguintes comportamentos: *agarrar* ou *largar* num referencial, um dado material; *trocar_ferramenta*; *fixar* ou *libertar*. Estes comportamentos destacam-se apenas por serem os mais comuns e simples.

Outros comportamentos podem ser definidos para este papel, que obrigam a características mais ricas, quer do componente, quer do papel. É o caso, por exemplo, dos movimentos guardados, que implicam a existência, quando não implementados no comando do componente, de um sensor que aparecerá descrito no papel. Movimentos em que a velocidade ou aceleração são parâmetros, podem também ser descritos.

Não se apresentou uma descrição em frame por uma questão de optimização de espaço, além de que não iria introduzir nada de novo. De qualquer forma, o leitor pode consultar o capítulo seguinte onde aparece um exemplo de modelação em frames de um agente de montagem.

Em relação a AG_MONTAGEM_ROBÔ existem algumas diferenças entre a implementação EIFFEL e a de Frames. Esta classe faz apenas sentido no ambiente EIFFEL, pelo facto de não se poderem definir relações próprias. Em ambiente de Frames necessita-se de criar uma entidade que representa o agente a desempenhar um dado papel: *papel_agente*.

MODELO EIFFEL®

```
class AG_MONTAGEM_ROBOT inherit
        AG_MONTAGEM  redefine desempenha
feature
        ...
        desempenha: ROBOT;
        ...
end -- class AG_MONTAGEM_ROBOT
```

MODELO FRAMES

```
FRAME PAPEL_AGENTE
is-a: concept
relation desempenha:PAPÉIS
relation realizado-por:COMPONENTES
```

3.4.3. Células

Uma célula é uma entidade que é capaz de executar um dado plano. No caso de uma célula de montagem, por exemplo, o plano é constituído pela sequência de operações que conduzirão à montagem de um dado produto. As operações são funcionalidades existentes nas entidades papel-objecto que participam na célula e que, como é sabido, virtualizam os componentes físicos que participam na célula. Como exemplo de operações válidas tem-se: *move_entrada*, *obtem*, *agarra*, *move_locmont*, *monta*, etc, que correspondem a funcionalidades do robô a desempenhar o papel de objecto.

Os planos possíveis para uma dada célula serão gerados por planeadores de uma forma que pode ser independente da sua execução. Uma célula executará o plano que lhe tenha sido atribuído. Assim sendo, deverão ser criadas funcionalidades que permitam a atribuição de um dado plano a uma célula.

Tendo-se definido célula como sendo uma entidade capaz de executar planos, é natural que a funcionalidade mais importante seja a execução de um dado plano. No sentido de uma melhor precisão, a definição de célula altera-se, ligeiramente, para ser uma entidade que executa com sucesso o plano que lhe foi atribuído. Esta noção de execução com sucesso é bastante importante e é um bom pretexto para introduzir duas questões. (1) O que acontece quando o plano não é executado correctamente ? e (2) como se sabe que um plano não está a ser executado correctamente ?

Para uma melhor compreensão ainda, é necessário interrogar (3) o modo como o plano será executado.

Existem várias formas de representar um plano: grafos, RdP, .. [52]. De todas estas formas, RdP parece ser a mais interessante, dada a facilidade com que modelam eventos concorrentes e assíncronos, que estão maioritariamente associados aos processos produtivos [36, 53].

Como qualquer entidade abstracta, a RdP é passível de representação computacional. Existem, como foi referido, diversas formas de a representar. A escolha da representação mais adequada depende fortemente do contexto de utilização. Neste contexto, não é necessário recorrer a formas elaboradas para representar uma RdP, bastando uma simples lista.

Seja a Rede de Petri descrita na Figura 3.28 e que representa parte de um plano de montagem a ser utilizado numa célula de montagem. Esta rede poderá ser definida através da seguinte lista:

```
( ( (Robot_pronto obtem) t1 (move_entrada) )  ( (move_entrada) t2 (agarra) )  ( (agarra) t3 (move_fix) )
  ( (move_fix) t4 (monta) )  ( (monta) t5 (Robot_pronto) )  )
```

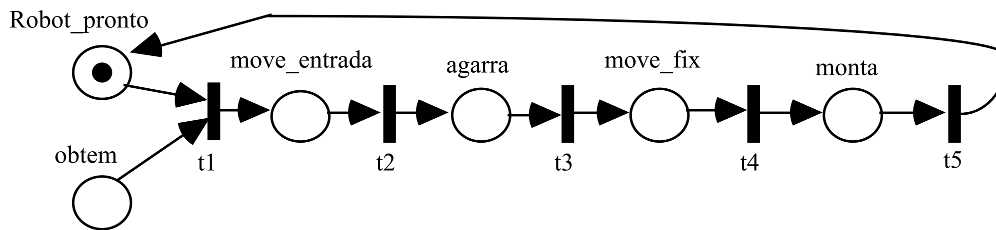


Figura 3.28 - Exemplo de RdP

Respondendo à questão do modo como o plano será executado, assume-se a existência de um EXECUTOR, que não será mais do que um executor de RdP com determinadas propriedades, nomeadamente a possibilidade de executar acções no objecto CÉLULA que é seu cliente.

Cada célula estará relacionada com um objecto EXECUTOR através de uma relação de agregação/composição. Para já, o importante é realçar a existência deste EXECUTOR agregado à célula que modelará os seus aspectos dinâmicos.

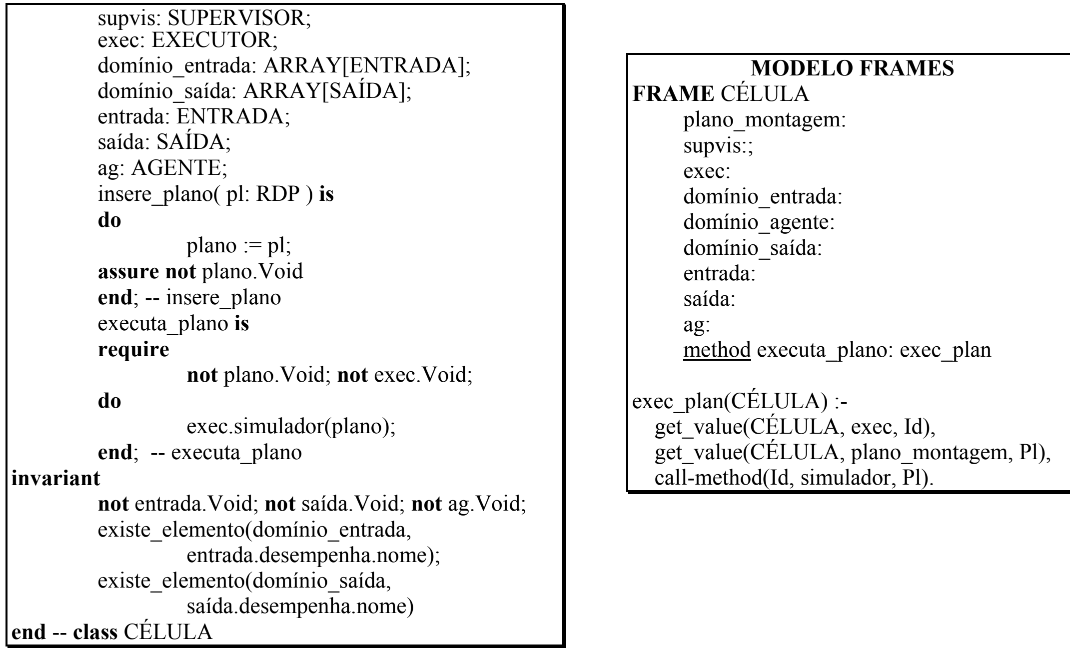
Executa_plano será a funcionalidade principal da célula. A sua implementação consiste em lançar o simulador de RdP (EXECUTOR).

Para responder à questão sobre o que acontece quando o plano não é correctamente executado, a célula deve "poder" informar uma entidade externa (monitorização de execução), sempre que se verifique uma condição anómala. Perante uma destas condições a entidade externa deverá aceder a funcionalidades especiais da célula que lhe permitam recuperar para um estado normal de funcionamento. No ponto 3.5 serão referidos outros aspectos relacionados com esta questão, ao mesmo tempo que se procurará também responder à segunda questão, relacionada com a forma como é detectada uma condição anómala.

Na modelação do conceito genérico de célula apresentaram-se 2 aspectos relevantes: (1) a existência de um controlador agregado e (2) a possibilidade de ser realizada supervisão.

MODELO EIFFEL®

```
class CÉLULA  feature
          plano: RDP;
```



```
        supvis: SUPERVISOR;
        exec: EXECUTOR;
        domínio_entrada: ARRAY[ENTRADA];
        domínio_saída: ARRAY[SAÍDA];
        entrada: ENTRADA;
        saída: SAÍDA;
        ag: AGENTE;
        insere_plano( pl: RDP ) is
        do
                plano := pl;
        assure not plano.Void
        end; -- insere_plano
        executa_plano is
        require
                not plano.Void; not exec.Void;
        do
                exec.simulador(plano);
        end;  -- executa_plano
invariant
        not entrada.Void; not saída.Void; not ag.Void;
        existe_elemento(domínio_entrada,
                entrada.desempenha.nome);
        existe_elemento(domínio_saída,
                saída.desempenha.nome)
end -- class CÉLULA
```

```
MODELO FRAMES
FRAME CÉLULA
        plano_montagem:
        supvis:;
        exec:
        domínio_entrada:
        domínio_agente:
        domínio_saída:
        entrada:
        saída:
        ag:
        method executa_plano: exec_plan

exec_plan(CÉLULA) :-
   get_value(CÉLULA, exec, Id),
   get_value(CÉLULA, plano_montagem, Pl),
   call-method(Id, simulador, Pl).
```

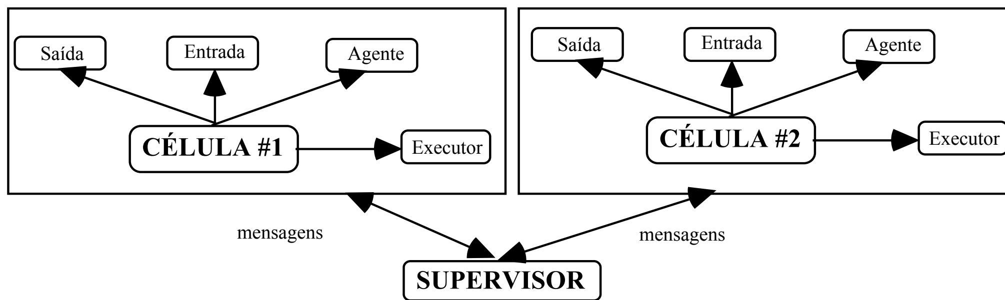


Figura 3.29 - Relações Existentes numa Célula

Pretende-se que a célula possa ser encarada sob diversos aspectos: execução, manutenção, monitoração, etc. Desta forma, as funcionalidades da célula serão restringidas a clientes pré-definidos[35]. *Executa_plano*, por exemplo, só poderá ser utilizada por um cliente que pertença a uma classe EXECUÇÃO, enquanto que operadores de mais baixo nível do tipo mover o robô, accionar o alimentador, etc, apenas poderão ser utilizados por clientes da classe SUPERVISÃO.

Cada tipo de célula apresenta características próprias, relacionadas essencialmente com o tipo de entidades papel/objecto que podem participar como entrada, agente ou saída.

O facto de se estar a utilizar a linguagem EIFFEL® para a modelação, implica algumas particularidades no modo como se definem as entidades válidas para cada tipo de célula. No caso do agente, a restrição será feita em tempo de compilação, uma vez que existe na taxonomia de papéis, uma classificação por tipo de acção: montagem, armazenamento, etc.

.

[35] Recorde-se que esta é a forma que se tem estado a utilizar para a implementação do conceito de visões no ambiente EIFFEL®. No ambiente de Frames, as visões implementam-se pela utilização das relações definidas pelo utilizador: *realizado_por* e *desempenha*.

```
                    MODELO EIFFEL®
class CÉLULA_MONTAGEM export
        executa_plano {SUPERVISOR_EXECUÇÃO },
        insere_plano  {SUPERVISOR_EXECUÇÃO }, .
inherit
        CÉLULA redefine ag, domínio_entrada,
                        domínio_saída
feature
        ag: AG_MONTAGEM;
        domínio_entrada: ARRAY[STRING] is
        once
                Result.Create(6);
                Result.enter("TAPETE");
                Result.enter("AGV");
                Result.enter("AL_VIB");
                Result.enter("GRAVITICO");
                Result.enter("BUFFER");
                Result.enter("MESA_INDEX");
        end -- domínio_entrada
        domínio_saída: ARRAY[STRING] is
        once
                Result.Create(4);
                Result.enter("TAPETE");
                Result.enter("AGV");
                Result.enter("BUFFER");
                Result.enter("MESA_INDEX");
        end -- domínio_entrada
end -- class CÉLULA_MONTAGEM
```

```
                    MODELO FRAMES
FRAME CÉLULA_MONTAGEM
    plano_montagem:
    supvis:;
    exec:
    domínio-entrada: TAPETE, AGV,
            AL-VIB, GRAVITICO,
            BUFFER, MESA_IND,
            STEAM
    domínio-saída: TAPETE, AGV, BUFFER,
            MESA_IND, STEAM
    domínio-agente: robô-scara,
            robô-antropomórfico,
            manipulador
    entrada:
    saída:
    ag:
```

```
                    MODELO EIFFEL®
class CÉLULA_PINTURA export
        executa_plano {SUPERVISOR_EXECUÇÃO },
        insere_plano  {SUPERVISOR_EXECUÇÃO }, .
inherit
        CÉLULA redefine ag, domínio_entrada,
                        domínio_saída
feature
        ag: AG_PINTURA;
        domínio_entrada: ARRAY[STRING] is
        once
                Result.Create(4);
                Result.enter("TAPETE");
                Result.enter("AGV");
                Result.enter("BUFFER");
                Result.enter("MESA_INDEX");
        end -- domínio_entrada
        domínio_saída: ARRAY[STRING] is
        once
                Result.Create(4);
                Result.enter("TAPETE");
                Result.enter("AGV");
                Result.enter("BUFFER");
                Result.enter("MESA_INDEX");
        end -- domínio_entrada
end -- class CÉLULA_PINTURA
```

```
                    MODELO FRAMES
FRAME CÉLULA_PINTURA
    plano_montagem:
    supvis:;
    exec:
    domínio-entrada: TAPETE, AGV,
            BUFFER, MESA_IND,
            STEAM
    domínio-saída: TAPETE, AGV, BUFFER,
            MESA_IND, STEAM
    domínio-agente: robô-antropomórfico
    entrada:
    saída:
    ag:
```

Para a entrada e saída já não acontece o mesmo. É então necessário adicionar, ao objecto célula, informação sobre o domínio da entrada e da saída, o que é conseguido através da atribuição ao *domínio_entrada* e *domínio_saída* de uma lista de nomes de componentes válidos. O invariante da classe garante, através da função booleana existe_elemento, se o componente atribuído à entrada ou saída pertence ao domínio.

A utilização de Frames, não permite a mesma verificação de tipos, utilizada no EIFFEL, para validar os agentes válidos para cada tipo de célula. A única forma consiste em associar demónios *if_write* aos atributos *entrada*, *saída* e *ag*. Durante a configuração, quando se atribuírem os agentes àqueles atributos, vão-se verificar se são válidos, através da comparação com os valores presentes em *domínio_entrada*, *domínio_saída* e

domínio_agente. A forma de validar dinamicamente os agentes será através da definição de demónios if_write, associados aos atributos *ag*, *entrada* e *saída*.

Repare-se que em EIFFEL não era necessário o atributo domínio_agente.

As diferenças mais significativas de uma célula de pintura para uma célula de montagem ocorrem ao nível das entidades que participarão como entrada, agente e saída.

3.4.4. Materiais

A modelação de materiais (componentes, produtos, matéria prima, ...) é um assunto bastante vasto e tem sido analisado ao longo do tempo por diversos investigadores [23]. A proposta a apresentar deve ser encarada numa perspectiva de enquadramento dos materiais nos sistemas de produção e não como uma análise exaustiva da modelação de materiais. Assim, não serão apresentados quaisquer atributos que tenham a ver com a descrição das propriedades mecânicas ou tecnológicas dos materiais. Por este facto também não se apresentarão as definições em Frames.

Apesar de não se pretender fazer uma descrição pormenorizada, é bom referir que a ISO[36], através do seu comité técnico nº 184/SC4, tem procurado normalizar a modelação de produtos, através do desenvolvimento de uma norma, denominada STEP [54, 55]. O objectivo final desta normalização é permitir a troca de informação entre entidades (empresas, computadores, ...), sobre produtos, sem qualquer ambiguidade.

Apesar de serem normas distintas são, por vezes, confundidas. O STEP considera a manufactura numa perspectiva orientada para o produto. O STEP é a norma que rege o modo como se cria um modelo de produto por forma a que possa ser representado informaticamente.

Os pontos importantes são a divisão dos materiais em produtos simples e compostos. Um produto simples poderá num momento qualquer pertencer ou não a um produto composto. Por seu lado, os produtos compostos são constituídos por produtos simples. Ambos os tipos de produtos estão armazenados em componentes de armazenamento.

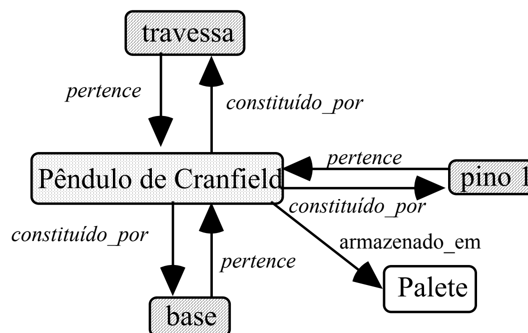


Figura 3.30 - Relações Existentes entre Produtos Simples e Compostos

Na figura 3.30 mostra-se a relação que existe entre um produto composto (Pêndulo de Cranfield) e alguns dos produtos simples que o formam (base, travessa, pino, ...). Um produto simples, quando pertence a um produto composto, deixa, individualmente, de estar armazenado num qualquer componente de armazenamento, uma vez que passou a pertencer a uma estrutura composta, sendo esta estrutura que está armazenada.

Sobre os materiais realizam-se, tal como no caso dos componentes, operações. Sempre que um robô realiza uma operação de montagem, um produto simples passa a composto. Este efeito lateral, provocado pela acção de montagem, deve ter correspondência no modelo do produto. A funcionalidade do produto, para reflectir esta situação, tem de ser enriquecida com uma operação que altere o estado do produto (*monta*).

---

[36] International Standards Organization

CLASSE PRODUTOS

```
class PRODUTOS  feature
   ...
   armazenado_em: ARMAZENAMENTO;
   ...
end -- class PRODUTOS
```

CLASSE PRODUTO_SIMPLES

```
class PRODUTO_SIMPLES  export
            pertence, ref_montagem, ref_agarrar, monta{PRODUTO_COMPOSTO}, altera_armazenamento
inherit
            PRODUTO
feature
   pertence: PRODUTO_COMPOSTO;
   ref_montagem: REFERENCIAL;                    -- Local onde vai ser montado
   ref_agarrar: REFERENCIAL;
   monta( pc: PRODUTO_COMPOSTO ) is
   require
            pertence.Void;  not armazenado_em.Void;
   do
            armazenado_em.retira_produto(Current);          -- O produto deixa de estar armazenado
            armazenado_em.Forget;                           -- NIL
            pertence := pc;
   assure
            armazenado_em.Void; not pertence.Void;
   end; -- monta

   altera_armazenamento( arm: ARMAZENAMENTO ) is
   require
            pertence.Void
   do
            arm.insere_produto(Current);
            if not armazenado_em.Void then  armazenado_em.retira_produto(Current);
            end;
            armazenado_em := arm;
   assure
            armazenado_em.Void; not pertence.Void;
   end; -- monta
invariant
   (not armazenado_em.Void and pertence.Void ) or ( armazenado_em.Void and not pertence.Void )
end -- class PRODUTO_SIMPLES
```

CLASSE PRODUTO_COMPOSTO

```
class PRODUTO_COMPOSTO  export
            ref_agarrar, monta{CÉLULA_MONTAGEM}, ...
inherit
            PRODUTO
feature
   constituido_por: LINKED_LIST[PRODUTO_SIMPLES];
   lista_produtos: LINKED_LIST[PRODUTO_SIMPLES];   -- Quais os Produtos  que pertencem
   ref_agarrar: REFERENCIAL;
   monta(ps: PRODUTO_SIMPLES) is
   require
            lista_produtos.present(ps); -- Produto a montar tem de pertencer ao composto
            not constituido_por.present(ps); -- Não pode ainda ter sido montado
   do
            ...
            ps.monta(Current);              -- Actualiza o produto simples
            constituido_por.insert(ps);     -- Actualizado
            ...
   assure
            constituido_por.present(ps);
   end; -- monta
invariant
   not armazenado_em.Void
end -- class PRODUTO_COMPOSTO
```

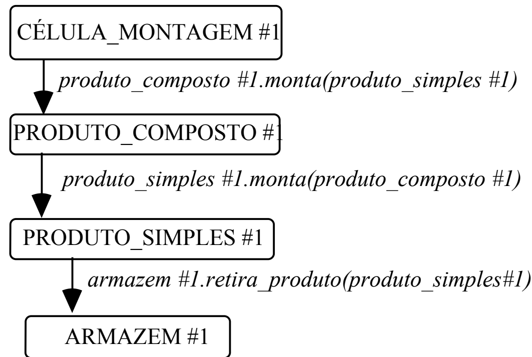


Figura 3.31 - Mensagens entre Objectos quando da Operação "monta"

A classe PRODUTO_SIMPLES relaciona-se com a classe ARMAZENAMENTO e vice-versa. Sempre que um produto muda de local de armazenamento ou passa a estar inserido numa estrutura complexa é necessário "desfazer" a ligação que existe entre o armazém e o produto, já que no armazém deixou de existir o referido produto. Esta situação mostra-se nos métodos *altera_armazenamento* e *monta*, que utilizam a operação *insere_produto* e *retira_produto* , exportada pela classe ARMAZENAMENTO.

Na classe PRODUTO_COMPOSTO, os atributos mais relevantes são: *lista_produtos* e *constituído_por*. O primeiro representa os produtos simples que vão participar na montagem e, por isso, sempre que for realizada uma operação *monta* é necessário verificar se o PRODUTO_SIMPLES que vai ser montado faz parte do produto. O segundo representa uma lista com os produtos que já estão efectivamente montados. Sempre que há uma operação de montagem é necessário adicionar a esta lista o novo produto e actualizar a nova situação no PRODUTO_SIMPLES.

Note-se que a operação monta, definida em PRODUTO_COMPOSTO, é exportada selectivamente, apenas para a classe CÉLULA_MONTAGEM, que é a classe que pode realizar operações de montagem. A relação dos objectos que são alterados quando se realiza uma operação de montagem a partir duma célula pode ser vista na Figura 3.31.

3.5. Comportamento Dinâmico

O comportamento dinâmico relaciona-se directamente com a actividade que implica a alteração do estado dos objectos[37]. A dinâmica de um dado objecto pode ser verificada pela análise da evolução do seu estado[38]. Esta visão generalista de comportamento dinâmico implica que se considerem, no seu estudo, aspectos tão elementares como o modo como se altera o estado de um objecto, como é o caso, por exemplo, da definição, do(s) método(s) que permite(m) que um dado atributo seja modificado.

O objectivo deste ponto não é analisar todos os aspectos relacionados com a dinâmica dos objectos. Com efeito, não se pretende analisar a forma como é realizada a mudança de estado, nem tão pouco descrever as possíveis evoluções de estado dos objectos envolvidos, pretendendo-se antes privilegiar os conceitos relacionados com o comportamento da entidade física modelada.

A maneira do modelo reflectir as alterações dos componentes físicos e o modo deste reflectir as alterações do modelo são os aspectos fundamentais a analisar neste ponto sobre comportamento dinâmico. Parece ser bastante tentador considerar que a dinâmica, no contexto desta tese, aparece muito próxima do controlo, uma

[37] Objecto deve ser encarado, neste contexto, como uma entidade pertencendo à extensão de uma classe que representa um dado componente físico.

[38] Considerando-se estado como uma imagem dos atributos do objecto.

vez que é através de acções de controlo que serão realizadas acções sobre os componentes físicos, ou seja, está-se a afirmar que o controlo representa o modo como a dinâmica é expressa.

O estudo do comportamento dinâmico dos componentes, numa perspectiva de controlo, pode ser encarado segundo duas vertentes: 1) considerando-os isoladamente ou 2) considerando-os integrados numa estrutura complexa que pode ser, por exemplo, uma célula. No primeiro aspecto privilegia-se o modo como o componente é actuado e a forma como o modelo há-de conhecer o seu estado físico, mas sem quaisquer preocupações de integração. No segundo aspecto privilegia-se uma forma de actuar sincronizada, ou seja o modo como os componentes cooperam na procura do objectivo final que pode ser, por exemplo, a montagem de um dado produto numa célula.

O segundo aspecto do comportamento dinâmico será objecto de análise no capítulo 4 desta tese. Pretende-se apresentar, nesse capítulo, uma proposta de ligação coerente entre uma Rede de Petri que representa o controlo de alto nível a realizar e os componentes físicos que participam no processo.

A partir de agora considera-se apenas em discussão o estudo do comportamento dinâmico dos componentes, considerados individualmente. A acção de controlo sobre um componente físico é sempre realizada por um controlador, com capacidade de cálculo. O controlador de um alimentador gravítico, por exemplo, necessita apenas de suportar primitivas de INPUT e OUTPUT, enquanto que o controlador de um robô é bastante mais complexo, suportando primitivas de movimento, aceleração, velocidade, etc. O ponto importante a reter nesta altura da discussão é a existência de uma unidade computacional externa ao ambiente em que a aplicação de controlo reside.

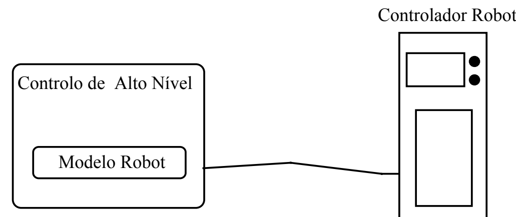


Figura 3.32 - Dois Mundos Distintos: Controlador do Robô e Controlador de Alto Nível

A maior dificuldade de modelação da dinâmica dos componentes ocorre precisamente pela existência de dois mundos computacionais distintos: o ambiente computacional onde está descrito o modelo dos componentes e o ambiente computacional associado ao controlador. Uma vez que estes ambientes podem ser completamente distintos será necessário criar uma camada de software que faça a ligação uniforme entre os dois sistemas.

Por forma a ligar o modelo do componente ao seu controlador físico real deve-se criar um modelo por cada tipo de controlador existente. Este modelo deve implementar, ao nível da sua funcionalidade, todas as primitivas que o controlador real possui. Se se estiver perante um controlador, por exemplo, do tipo SONY SCR-4H, o modelo deve implementar a imagem das funcionalidades que correspondem às do controlador real.

A classe CONTROLADOR_ROBÔ deve encapsular a ligação que deverá ser estabelecida entre ela e o controlador do robô. Considerando que o ambiente onde vai ser executada a aplicação que contém o modelo dos componentes é um ambiente multi-tarefa, por exemplo UNIX®, deverá ser criada, para cada controlador físico com quem se pretenda estabelecer comunicação, uma camada de software que funcionará como um servidor. Os serviços fornecidos pelo servidor têm correspondência directa com as potencialidades do controlador físico. Por cada tipo de controlador terá de haver um tipo de servidor. A razão deste facto prende-se com as diferenças que existem entre os controladores físicos, ao nível do modo como podem ser controlados externamente. No caso do controlador SONY, por exemplo, foi necessário desenvolver um protocolo especial, que permite que a sua funcionalidade seja actuada a partir da porta série.

Este trabalho foi desenvolvido no âmbito de um projecto de fim de curso, de que o autor da tese foi co-orientador e consistiu no desenvolvimento de uma camada de software, interna ao controlador, por forma a que fosse possível controlar o robô, utilizando a porta série.

O controlador do robô ABB IRB 2000 permite o seu controlo, através da porta série, utilizando um protocolo privado da ABB.

Por seu lado, o controlador de robô BOSCH SR80, obrigou também ao desenvolvimento de um protocolo especial por forma a poder ser controlado a partir da porta série. Este trabalho foi também realizado no âmbito de um projecto de fim de curso, co-orientado pelo autor.

MODELO EIFFEL®

```
class CONTROLADOR_ROBÔ  export
        velocidade, aceleração, move_lin, move_circ,
        move_eixo,  input, output, liga_controlador,
                desliga_controlador
inherit
                CONTROLADOR
feature
        ...
        liga_controlador is
        do  -- código  end; -- liga_controlador
        desliga_controlador is
        do  -- código  end; -- desliga_controlador
        velocidade(vel: INTEGER) is
        do  -- código  end; -- velocidade

        aceleração(ac: INTEGER) is
        do  -- código  end; -- velocidade
        move_lin(ref: REFERENCIAL) is
        do  -- código  end; -- velocidade
        move_circ(ref: REFERENCIAL) is
        do  -- código  end; -- move_circ
        move_eixo(ref: REFERENCIAL) is
        do  -- código  end; -- velocidade

        input(ent: INTEGER): BOOLEANis
        do --código  end;  --input
        output(sai: INTEGER; V:BOOLEAN) is
        do --código  end;  --output
end -- class CONTROLADOR_ROBÔ
```

Uma vez que o acesso aos controladores externos se fará sempre através de um servidor, é natural que haja um método na definição da classe CONTROLADOR que permita ligar - *liga_rpc*, desligar- *desliga_rpc* e enviar mensagens - *enviar_msg_to_rpc*, para o RPC[39], que será identificado através de um número existente na definição da classe - *rpc_id*. Os três métodos: *liga_rpc*, *desliga_rpc* e *enviar_msg_to_rpc*, são definidos ao nível da classe CONTROLADOR, já que todas as suas subclasses os irão utilizar. A distinção entre as subclasses faz-se ao nível do identificador do RPC - *rpc_id*. O método enviar_msg_to_rpc, recebe como argumento um dado código que identificará um serviço do RPC que, tal como já foi referido, tem uma correspondência directa com a funcionalidade do controlador.

As variáveis de controlo de componentes estão definidas no controlador físico. Por exemplo, o estado das variáveis de controlo de um robô: velocidade, aceleração, posição, etc, pode ser conhecido através do acesso ao referido controlador. Desta forma, será natural que o modelo do controlador não tenha necessidade de criar atributos para guardar o estado dessas variáveis, já que, em caso de necessidade, basta realizar uma consulta sobre o controlador, partindo do princípio que se definiram primitivas para estas operações ao nível do modelo.

[39] Remote Procedure Call

Do ponto de vista da aplicação que contém os modelos, estas variáveis são **persistentes**, já que mantêm a sua existência para além da execução da aplicação.

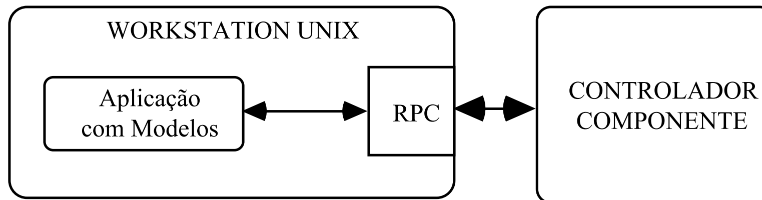


Figura 3.33 - Ligação dos modelos ao controlador local, através de RPC

No caso das frames, terão que se declarar *slots* que virtualizam as variáveis do controlador. Para que em cada acesso, aos slots, o cliente receba informação actualizada do controlador físico, é necessário associar demónios if_read que, ao serem actuados, fazem a leitura dos valores presentes no controlador. Ver código referente aos Frames para a classe CONTROLADOR_ROBÔ.

Em ambos os sistemas têm-se variáveis persistentes. Esta persistência das variáveis não significa que tenham um comportamento estático, já que a ligação, estabelecida entre a entidade que as controla (aplicação utilizando o modelo) e o local onde estão armazenadas (controlador físico do componente), é realizada de uma forma dinâmica, isto é, qualquer alteração ao nível da aplicação implica uma alteração do estado do componente e, por outro lado, qualquer alteração do estado do controlador é conhecida imediatamente pelo modelo. Estamos por isso perante um conceito de **persistência dinâmica** dos atributos que identificam o controlo do componente.

A recuperação do estado em que o componente estava, no momento de uma eventual falha, originada no ambiente da aplicação, torna-se bastante mais simples. Evidentemente que se está a supor que o ambiente do controlador não sofreu qualquer falha.

Para uma possível recuperação de erro, vão-se duplicar as variáveis de estado no modelo que representa o controlador. Sempre que um determinado cliente necessitar de saber qual o estado de uma dada variável, que resultou , por efeito lateral, de uma determinada acção sobre o controlador, a consulta dessa variável ao controlador torna-se morosa. A utilização de um atributo no modelo que reflicta o estado da variável em questão é uma forma mais optimizada, já que é realizado apenas um acesso à variável. É necessário distinguir, neste caso, dois tipos de variáveis: 1) as cujo estado resultou de uma acção empreendida a partir da aplicação (velocidade, aceleração, posição, ...) e 2) as que resultam de acções realizadas externamente sobre o controlador (entradas digitais, ...).

Para o primeiro tipo de variável, a alteração de estado no modelo, acontece após a realização da acção que conduz à sua modificação. Por uma questão de segurança deve ser pedido ao controlador que informe o estado em que a variável ficou. Veja-se o exemplo da variável *velocidade* , no método *altera_velocidade*, da definição da classe CONTROLADOR_ROBÔ.

No segundo tipo de variável, sempre que um cliente necessite saber qual o seu estado tem de se "perguntar" ao controlador, qual o seu estado, porque desde a última consulta poderá ter havido alteração. Este problema pode ser mais facilmente resolvido, quando em presença de um ambiente de programação concorrente por objectos.

De notar que, na definição da classe CONTROLADOR_ROBÔ, *envia_saida*, *envia_velocidade* e *envia_saida* são funções definidas internamente à classe e fazem apelo ao procedimento de ligação ao servidor.

MODELO EIFFEL®

```
class CONTROLADOR_ROBÔ  export
                velocidade, aceleração, ..
inherit
                CONTROLADOR redefine rpc_id
feature
        ...
        SEND_VELOC:  INTEGER is 1;
        CONF_VELOC: INTEGER is 2;
        SEND_SD:  INTEGER is 3;

        MAX_SD: INTEGER is 10;
        MAX_ENT: INTEGER is 10;
        ...
        rpc_id: INTEGER is 2001;
        velocidade: INTEGER;
        aceleração: INTEGER;
        posição: REFERENCIAL;
        saída_dgital : ARRAY[BOOLEAN];

        liga_controlador is
        require
                not ligado
        do
                erro := liga_rpc(rpc_id);
                ligado := TRUE;
        assure
                ligado; erro = 0;
        end; -- liga_controlador
        ...
        velocidade(vel: INTEGER) is
        require
                ligado; erro = 0; vel < 10 and vel > 0;
        do
                erro := envia_velocidade(vel,);
                velocidade := pergunta_velocidade
        assure
                velocidade = vel; erro = 0;
        end; -- velocidade

        entrada_digital(ent: INTEGER): BOOLEAN is
        require
                ligado; erro = 0;  ent <= MAX_ENT
        do
                Result := pergunta_entrada(ent);
        end;  --input

        saída_digital(sai: INTEGER, val: BOOLEAN) is
        require
                ligado; erro = 0; sai <= MAX_SD
        do
                erro := envia_saida(sai, val);
                saída_digital.Enter(sai, val);
        end;  --output

end -- class CONTROLADOR_ROBÔ
```

MODELO FRAMES

```
FRAME CONTROLADOR_ROBÔ
    is-a: CONTROLADOR
    envia_veloc: 1
    conf_veloc: 2
    envia_sd: 3
    max_sd: 10
    max_ent: 10
    rpc_id: 2001
    ligado:  FALSE
    velocidade: demon if_write after vel_fn
    aceleração: demon if_write after acel_fn
posição: demon if_write  aftermovimentar
tipo_movimento: eixo
saída_digital_1:false demon if_write alterar_saída1
...
saída_digital_n:false demon if_write alterar_saída1
entrada_digital_1: demon if_read before
consultar_entrada1
...
entrada_digital_n: demon if_read before
consultar_entradan
method ligar_controlador: ligar_fn()
method desligar_controlador: desligar_fn()
method saída_digital: output_fn(saída)

ligar_fn(CONTROLADOR_ROBÔ) :-
    get_value(CONTROLADOR_ROBÔ, ligado, FALSE),
    get_value(CONTROLADOR_ROBÔ, rpc_id, ID),
    liga_rpc(ID),
    new_value(CONTROLADOR_ROBÔ,ligado, TRUE).
    ...
vel_fn(CONTROLADOR_ROBÔ, velocidade, Info) :-
    get_value(CONTROLADOR_ROBÔ, ligado, TRUE),
    envia_velocidade(Info),
    pergunta_velocidade(Info).

consultar_entrada1(CONTROLADOR_ROBÔ,
                              entrada_digital_1, Info) :-
    get_value(CONTROLADOR_ROBÔ, ligado, TRUE),
    pergunta_entrada(1, Res),
    new_value(CONTROLADOR_ROBÔ,
                              entrada_digital_1, Res).

output_fn(CONTROLADOR_ROBÔ, N, V) :-
    get_value(CONTROLADOR_ROBÔ, ligado, TRUE),
    concatena(saída_digital, N, Res),
    new_value(CONTROLADOR_ROBÔ, Res, V).

alterar_saída1(CONTROLADOR_ROBÔ, saída_digital1,
                                        Info) :-
    envia_saída(1, info).
```

Um demónio *if_read*, deverá estar associado às variáveis de controlo que são actuadas externamente, como é o caso das entradas digitais. Sempre que um cliente do controlador necessitar de usar (ler) o valor do atributo entrada_digital, provoca o disparo implícito do demónio que desencadeará as acções que conduzirão à leitura do valor no controlador externo e a sua colocação no atributo do modelo.

Por seu lado, um demónio *if_write* , deverá estar associado a uma variável de controlo actuada internamente, como é o caso, por exemplo, da variável posição. Quando um cliente pretende alterar a posição do robô, basta-

lhe alterar o valor do atributo posição. Nesta altura desencadeia-se o demónio que fará a transmissão, para o controlador, do comando necessário para que o robô se mude para a referida posição.

Em termos do objectivo final, a utilização da programação reactiva (Frames) ou a POO (EIFFEL) não tem grandes diferenças. Elas ocorrem essencialmente no modo como se tratam as variáveis dos controladores físicos (posição, velocidade, ...). Enquanto que na programação reactiva, os clientes acedem às variáveis pelos slots, na POO acedem pelos métodos.

Do exposto parece ficar-se com a ideia que os métodos apenas podem ser definidos para a POO, o que não corresponde à verdade. Acontece que, ao optar-se por ter informação redundante no mundo do controlador de alto nível, das variáveis existentes no controlador físico (posição, velocidade, ...), houve que declarar slots/atributos para esta função. Para se aceder a estes atributos/*slots* existem agora duas formas, consoante se está em POO ou em Frames; no primeiro caso as operações de escrita/leitura sómente se podem realizar à custa de métodos; no segundo caso, estas operações podem ser feitas, ou por métodos ou então acedendo directamente. Neste caso optou-se pelo acesso directo o que conduziu à não utilização de métodos.

A funcionalidade dos controladores, não directamente relacionada com variáveis é implementada com métodos, num e noutro modelo. Veja-se o caso dos métodos ligar_controlador e desligar_controlador.

Conclui-se o capítulo com uma comparação final entre os dois paradigmas. A POO parece ser mais indicada para o desenvolvimento de software em grandes quantidades, uma vez que é um paradigma pensado para Engenharia de Software. Por este facto, apresentam algumas limitações, como é o caso da impossibilidade de alteração dinâmica da estrutura (classe/tipo) de um objecto.

Os Frames, por seu lado, parecem ser mais indicados para criação de protótipos, sendo por isso mais flexíveis; é possivel alterar dinamicamente a sua estrutura e o mecanismo de relações, definidas pelo utilizador, é bastante importante para se modelarem entidades que se relacionam entre si.

4. Aplicação a um Sistema Flexível de Manufactura

4.1. Introdução

Uma Unidade de Produção pode envolver no seu funcionamento um número elevado de controladores. Estes apresentam-se sobre diversas formas: PLCs, PCs, controladores de Robôs, controladores de CNCs, sistemas integrados de controlo, etc.

As acções de comando desencadeadas por estes controladores, podem apresentar níveis de abstracção muito distintos. Enquanto que, por exemplo, determinado PLC controla apenas um determinado processo isolado (controlo da plataforma que movimenta o robô entre dois pontos distintos de montagem), pode existir outro ou outros cuja função seja a de coordenar os diferentes controladores envolvidos no processo. Este último controlador exerce nitidamente, funções de nível hierárquico superior às do primeiro.

Apesar destas possíveis diferenças ao nível hierárquico, a necessidade de uma ferramenta que modele correctamente (de uma forma fácil e elegante), e sem qualquer comprometimento com o tipo de controlador envolvido, é uma necessidade básica.

A grande complexidade na modelação de sistemas reais existe devido a vários factores: (1) diferentes tipos de controladores envolvidos, (2) dificuldades de exprimir o processo em si, (3) necessidade de explicitar a sincronização entre os diferentes processos envolvidos e (4) a necessidade de validar os resultados obtidos.

4.1.1. Controladores Diferentes

Os diferentes tipos de controladores envolvidos conduzem à utilização de diversos formas de programação (modelação do processo) e, consequentemente, à utilização de uma ferramenta de modelação que é adaptada à máquina em causa e, muitas vezes, de conteúdo semântico duvidoso, veja-se o caso das linguagens providenciadas por alguns controladores de Robôs como é o caso do IRB 2000 da ABB e, em menor grau, o SR800 da BOSCH.

A utilização destas diferentes ferramentas obriga a um grande esforço de formação que pode ser inglório, já que a alteração de um tipo de controlador para outro, num mesmo processo, pode obrigar à utilização de uma nova ferramenta, tornando-se a flexibilidade um conceito morto.

Como conclusão tira-se que a existência de controladores distintos conduz-nos à questão da necessidade de existir uma ferramenta de modelação que seja independente do tipo de controlador existente.

Paralelamente a este problema situa-se a necessidade de integrar os diferentes tipos de controladores, sendo necessário, na maior parte dos casos, desenvolver um grande esforço que pode não ser muito recompensador já que, do ponto de vista conceptual, não é muito interessante. A razão para este facto prende-se maioritariamente com o facto dos robôs presentemente apresentarem, quase todos, arquitecturas muito fechadas; se se aliar a isto, o facto dos fabricantes serem muito ciosos da sua informação, estamos perante um cenário em que cerca de 75%

do esforço de integração é colocado a "partir" protocolos próprios de fabricante. Mas, apesar de tudo isto, resta ainda uma quantidade de trabalho razoável conceptualmente bastante interessante porque, para além de "quebrar" os protocolos é necessário realizar um esforço de integração que torne os controladores existentes unidades cooperativas, fazendo parte de uma estrutura de controlo manipulada e supervisionada de uma forma harmoniosa e coerente.

É de todo interessante realçar que o trabalho desenvolvido no sentido da integração de plataformas distintas se apresenta muito importante, sendo a experiência adquirida nesse sentido, de grande utilidade, já que a grande maioria das soluções credíveis no âmbito de projectos de automação passa pela integração, não só de novas plataformas, mas também de plataformas já existentes ("legacy systems").

Nos diferentes trabalhos em que o autor desta tese esteve envolvido, no grupo de Robótica e CIM, procurou-se sempre seguir uma filosofia em que a inserção de controladores distintos fosse feita de uma forma tão elegante quanto possível, isto é, criou-se uma filosofia baseada no paradigma cliente-servidor, em que os distintos controladores têm uma representação (imagem?) num ambiente que permite uma fácil integração/comunicação, como é o caso do sistema operativo UNIX®. A ligação entre o cliente e a imagem é feita sempre da mesma forma, a menos das particularidades dos comandos, variando apenas a ligação entre a imagem e o controlador real. Desta forma, garante-se a possibilidade de ligação de qualquer controlador existente ou novo e em qualquer altura. A solução não fica comprometida com um número fixo de controladores já que é sempre possível juntar, em qualquer altura um novo servidor.

4.1.2. Dificuldades de Exprimir o Processo

A dificuldade de exprimir o processo surge naturalmente como outro dos factores que tornam a modelação complexa. A questão não se põe ao nível de ser ou não ser capaz de solucionar o problema, mas sim da forma mais elegante de o solucionar. A modelação de um problema (solução) deve, em primeiro lugar, ter uma fácil representação, evitando assim o desenvolvimento de uma solução bastante onerosa do ponto de vista temporal. A ideia é tornar o processo o mais rápido possível.

Por outro lado, a solução deve ser bastante flexível, permitindo que seja facilmente compreendida por qualquer pessoa. Este ponto é extremamente importante na criação de sistemas flexíveis, ou com grande mutação tecnológica. O facto de um sistema ter sido modelado com uma ferramenta de grande conteúdo semântico conduz a uma solução de maior clareza, tornando a compreensão do sistema mais fácil e, consequentemente, a uma também maior facilidade na alteração do processo.

Conclui-se assim da necessidade de existir uma ferramenta de modelação de grande conteúdo semântico, com uma grande riqueza ao nível dos operadores utilizados.

4.1.3. Necessidades de Sincronização

A necessidade de explicitar a sincronização entre os diferentes processos envolvidos surge como outro dos factores conducentes à complexidade de modelação, não só pela sua complexidade inerente mas também pelo facto de ser uma área onde dificilmente se encontra um controlador com uma ferramenta de modelação desenvolvida especificamente para a solução deste problema.

O desenvolvimento de soluções para este tipo de problemas faz-se à custa de mecanismos de modelação (fornecidos pelas linguagens/sistemas operativos tradicionais) que são completamente inadequados à situação. Consequentemente, as soluções, são, quase sempre, soluções pouco elegantes, pouco flexíveis e com grande dificuldade de compreensão.

Mas o facto dos fabricantes de controladores providenciarem ferramentas inadequadas ao problema não significa, de modo algum, que este seja um problema menor. Pode afirmar-se, com toda a certeza, que é um dos problemas mais importantes na procura de uma solução global. Só com uma boa ferramenta que permita a explicitação formal da cooperação entre os diferentes processos envolvidos, se assegura a possibilidade de

divisão do problema global em várias tarefas que cooperarão para o objectivo comum. Levanta-se assim a questão do processamento distribuído e a necessidade de agentes autónomos distribuídos que cooperam entre si.

A utilização de técnicas de agentes distribuídos sem a utilização de uma ferramenta que suporte convenientemente a modelação da sincronização, torna-se quase impossível, ou pelo menos, bastante complicada. O comportamento global de um sistema constituído por diversos controladores, ligados de forma distribuída, requer uma modelação coerente. Este comportamento distingue-se do comportamento individual apresentado por cada objecto e, como tal, tem requisitos de modelação diferentes.

Os aspectos do comportamento individual dos componentes estão directamente relacionados com o modelo, destacando-se os métodos e demónios como as entidades candidatas à sua modelação. Ainda dentro da problemática do comportamento individual dos componentes situa-se a forma como o modelo é ligado ao componente real.

Os aspectos de comportamento da entidade complexa, célula, por exemplo, requerem uma modelação, que para além de descrever a forma como os diferentes componentes se relacionam, terá também de explicitar a forma como as diferentes acções dos diferentes componentes se sincronizam.

Assim, na modelação deste tipo de comportamento, irá recorrer-se ao paradigma da programação orientada por objectos/frames, que definirá as relações entre os componentes enquanto que as redes de PETRI suportarão as necessidades de modelação ao nível da sincronização.

4.1.4. Validação dos Resultados

Em alguns processos pode ser extremamente importante que uma determinada solução possa ser validada antes de ser utilizada. Pode ser bastante difícil uma situação de teste inicial , por exemplo, no caso de um processo que envolva matéria prima muito onerosa. A solução neste caso passa quase sempre pela simulação, a qual resolve muitos dos casos, mas não garante a certeza de uma solução provada matematicamente. Ao ser correctamente provada a validade formal de determinada solução tem-se a certeza do comportamento do sistema. A dificuldade está, na maior parte dos casos, em descobrir a prova formal.

Como conclusão, tira-se que a validação de resultados conduz à necessidade de utilização de uma ferramenta de modelação sobre a qual seja possível aplicar prova matemática.

4.1.5. Conclusão

No capítulo anterior foi apresentado um conjunto de propostas relacionadas com a modelação de componentes presentes nos sistemas de produção. Algumas das ideias foram aplicadas em projectos em que o autor esteve envolvido, nomeadamente nos projectos relacionados com a célula NovaFlex[40].

Para testar as ideias apresentadas utilizou-se um sistema de frames - GOLOG, desenvolvido no seio do Grupo de Sistemas Robóticos e CIM [46].

Como exemplos mais relevantes do trabalho desenvolvido refira-se a participação no desenvolvimento e implementação de duas células flexíveis (célula Sony e NovaFlex), as quais foram utilizadas como suporte a quatro trabalhos de fim de curso [56-59], na área da modelação e integração, que decorreram sob a co-orientação do autor. Detalhes do projecto e implementação destas células serão descritos no ponto 4.2.

No ponto 4.3 serão descritos aspectos concretos de trabalho realizado sobre modelação e integração, utilizando como suporte os sistemas apresentados no ponto 4.2. Neste ponto procurar-se-á também enfatizar a ligação dos modelos ao controlador real.

---

[40] NovaFlex é o nome dado a um sistema flexível de manufactura e montagem constituído por 2 células de montagem, uma de maquinação, um sistema de transporte e um armazém automático, instalados no Centro de Robótica Inteligente do UNINOVA.

No ponto 4.4 apresenta-se finalmente a ligação entre um controlador descrito em RdP com a infraestrutura entretanto apresentada. Neste sentido, dar-se-á especial atenção aos aspectos que conduzem à síntese do controlador de alto nível a partir da RdP.

Finalmente no ponto 4.5 fará-se-á uma breve discussão da experiência obtida na modelação e integração de sistemas flexíveis de produção.

4.2. Sistemas em Estudo

De seguida apresentam-se dois sistemas flexíveis de produção pertencentes ao grupo de Robótica e CIM: célula de montagem e sistema flexível de produção - NovaFlex.

4.2.1. Célula de Montagem

A motivação que presidiu à implementação desta célula foi o suporte aos projectos de investigação Esprit B-LEARN e JNICT SARPIC e CIM-CASE em que o grupo está envolvido. Estes projectos pretendem estudar a aplicação de aprendizagem automática na recuperação e monitorização de operações de montagem robotizadas.

Consequentemente, a arquitectura da célula reflecte um pouco estes requisitos tendo-se procurado criar uma infra-estrutura que fornecesse alguma flexibilidade, apresentasse um conjunto sensorial mais ou menos rico e, acima de tudo, fornecesse uma arquitectura de controlo flexível.

Os componentes mecânicos básicos da célula são:

- robô SONY do tipo SCARA, equipado com um mecanismo para a troca automática de ferramentas e um sensor de forças/momentos.
- 1 fixador com movimento provocado por um cilindro pneumático controlado por electro-válvula.
- 2 alimentadores com mecanismo fornecedor suportado por cilindro pneumático e controlado também por electro-válvula.
- sistema de comutação de ferramentas constituído por 3 suportes para ferramentas
- 3 ferramentas pneumáticas (garras) com diferentes cursos e dedos

O sistema sensorial é maioritariamente constituído por sensores binários, destinados a indicar a presença de materiais, a posição dos cilindros, a existência de ferramenta, etc.; o sensor de forças/momentos que está instalado no quarto eixo do robô, permite quantificar as forças e momentos, para cada um dos eixos, que estejam a ser actuados sobre a ferramenta.

O facto da célula estar um pouco comprometida com um produto - Benchmark de Cranfield (em termos de alimentadores), não limita, do nosso ponto de vista, a flexibilidade no suporte à investigação já que a infra-estrutura suporta um conjunto diverso de operações, que se podem encontrar em células robotizadas. Dentro destas operações salientam-se os movimentos normais de montagem (inserção, agarrar, largar, ...), a troca de ferramenta e o fornecimento de materiais.

Como se pretendia uma arquitectura de controlo flexível, um requisito básico daí derivado, seria a possibilidade de controlar o robô directamente a partir de um "*host*". A arquitectura do controlador SONY não estava preparada para tal, pois unicamente possibilitava a programação através de uma linguagem própria - "LUNA language", indicada para o desenvolvimento de aplicações completas. A exploração da funcionalidade do robô apenas poderia ser realizada através de um programa, estando o controlador apenas preparado para fazer o "*download*" e "*upload*" de programas e dados, através da porta de comunicação série. Esta é, aliás, a situação mais frequente nos robôs industriais.

A forma de tornear a dificuldade surgiu através do envio de comandos para a sua porta série, sendo para isso implementado um programa, desenvolvido em LUNA, que funciona como interpretador de comandos, aproveitando as primitivas do LUNA para ler e escrever da porta série.

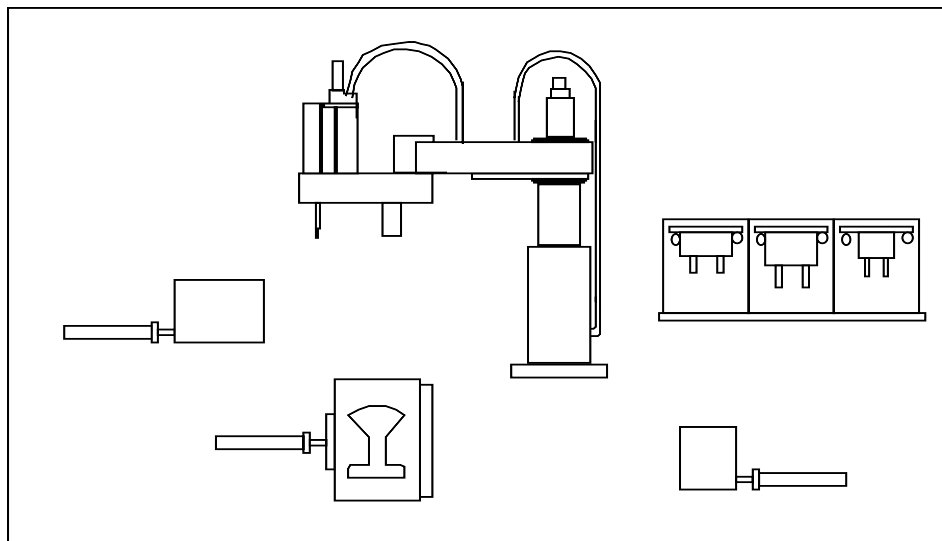


Figura 4.1 - Arquitectura Hardware da Célula do SONY

A implementação da filosofia não foi tão simples quanto parece já que o protocolo de comunicação não seguia os parâmetros habituais e, dada a inexistência de documentação (apesar dos esforços feitos junto do fabricante), houve que despender um grande esforço no sentido de perceber a forma como a comunicação se realizava, o que foi conseguido com recurso a um analisador de protocolos.

Dominado o problema da comunicação ficou por resolver um outro que, à partida, se apresentava ainda mais complexo. Com efeito, a utilização do interpretador do lado do controlador do SONY não resolvia o problema quando em situações de erro. Com as operações a funcionarem normalmente tudo corria dentro dos trâmites, enviando o interpretador um sinal para o "*host*" no final da realização de cada comando. Mas o que acontecia numa situação de erro, como por exemplo, no caso de ser premido o botão de emergência ou um determinado ponto não poder ser atingível ? Nestas situações, o "*host*" ficava numa situação de completo desconhecimento sobre o tipo de erro, impossibilitando a sua recuperação. Por outro lado, determinados tipos de erro implicam a realização de uma operação de "*hardhome*" que era impossível de ser actuada por software, a única hipótese é através do "*teach pendant*".

Sabendo que, após cada erro, o controlador o afixava no "*teach pendant*", surgiu a possibilidade de tornear o problema, através da emulação do "*teach pendant*" por um "*host*", com a "pequena" dificuldade de ter de se perceber qual o seu modo de funcionamento. Mais uma vez, a documentação era nula, obrigando a que um esforço considerável fosse colocado na sua percepção, recorrendo-se, outra vez, ao analisador de protocolos. Isto também era necessário para a implementação de comandos guardados, fundamentais num sistema de supervisão.

Estes exemplos ilustram as dificuldades que se encontram quando se pretende evoluir dos sistemas legados do passado ("*legacy systems*") para uma nova geração de sistemas integrados e flexíveis.

A infra-estrutura de controlo da célula[41] pode ser vista na figura 4.2, destacando-se o servidor operacional que está ligado ao controlador físico do robô, que lhe envia as acções de controlo; o emulador do "*teach*

[41] Este trabalho foi iniciado num projecto de fim de curso (aluno: João Carlos Silva) e co-orientado pelo autor que também participou na implementação, e continuado nos projectos B-Learn, Sarpic e CIM-CASE.

pendant" é constituído por um servidor Linux/Unix e está ligado ao servidor operacional, que o utiliza para determinar as situações de erro e realizar operações de recuperação.

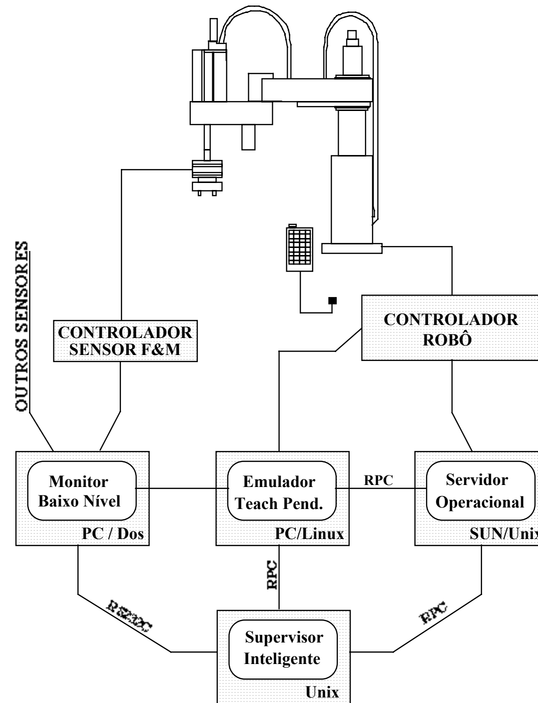


Figura 4.2 - Infraestrutura de Controlo da Célula do SONY

O servidor operacional virtualiza o conjunto de comandos existentes ao nível do controlador físico dos quais se destacam [60]:

- movimento ponto a ponto
- movimento linear
- movimento circular
- definição de um ponto
- atribui factor de velocidade
- atribui factor de aceleração
- *delay*
- pára

4.2.2. Sistema NovaFlex

O sistema piloto desenvolvido no âmbito do projecto PEDIP - NovaFlex[42], instalado no Centro de Robótica Inteligente (CRI) do UNINOVA foi concebido como uma unidade para a demonstração de um conjunto típico de actividades relacionadas com um sistema de Produção Integrada por Computador (CIM).

[42] Em cuja concepção e desenvolvimento o autor desta tese participou como co-autor [1].

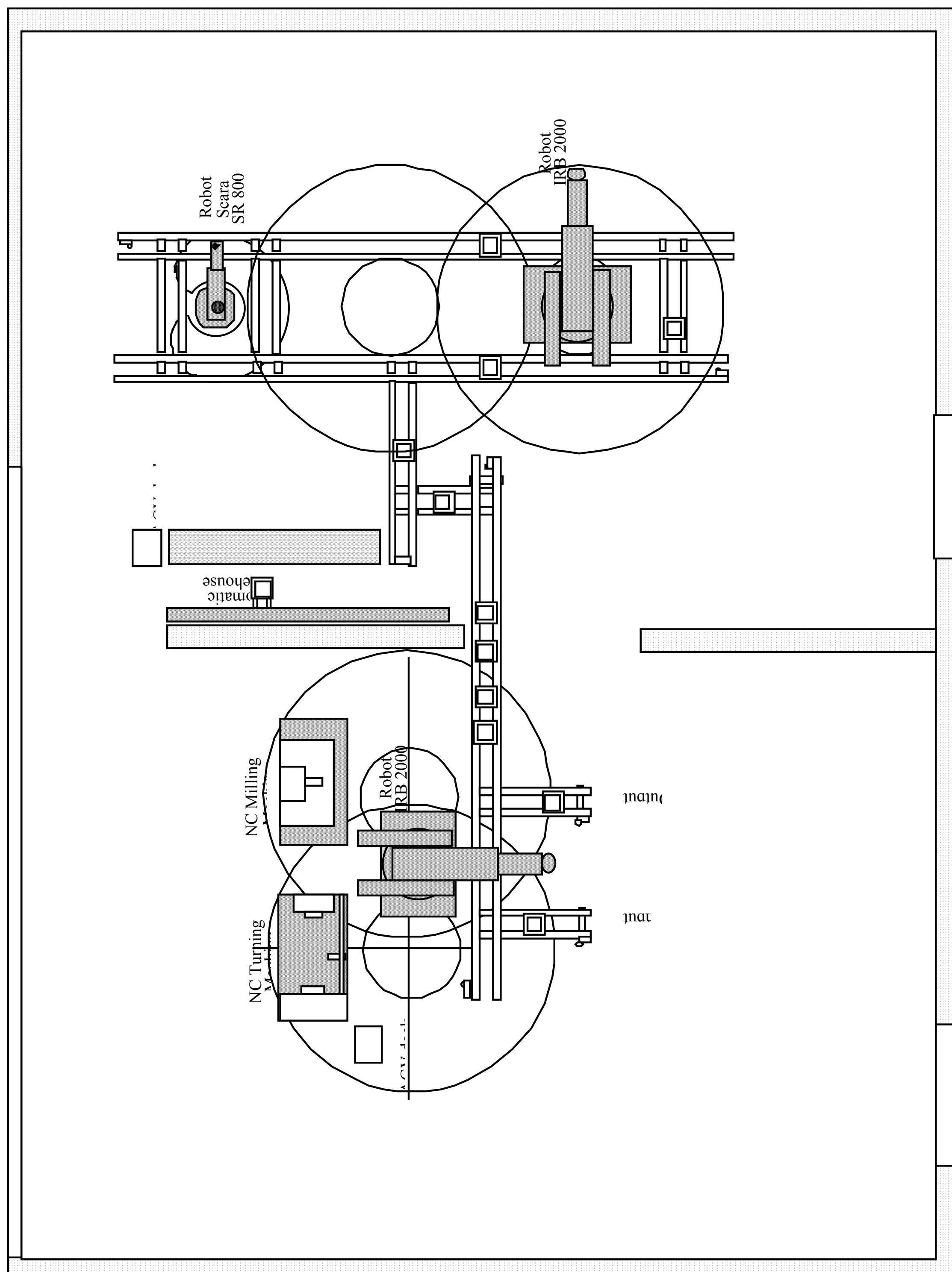


Figura 4.3 - Arquitectura Hardware da Célula NovaFlex

A unidade NovaFlex é formada por 5 subsistemas:

(1) Subsistema FMS,

(2) Subsistema FAS Multi-Robô,

(3) Subsistema Armazém Automático,

(4) Subsistema de Transporte,

(5) Subsistema sensorial.

A orientação mestra que norteou o projecto da unidade foi a de não ficar comprometida com nenhum produto em particular. O objectivo foi a construção de uma infra-estrutura relativamente genérica, que pudesse ser adaptável a uma gama variada de produtos sem grandes alterações iniciais. O CRI necessitava de uma infra-estrutura flexível, com vários conjuntos representativos de recursos de produção e não um sistema vocacionado para determinada aplicação.

Esta Unidade Piloto suporta 3 tipos de actividades:

i. formação,

ii. demonstração,

iii. investigação - integração de sistemas, controlo de células e escalonamento, planeamento, monitorização, diagnóstico e recuperação de erros, percepção sensorial, etc.

Os requisitos ou as restrições impostas por estas actividades nem sempre são convergentes impondo, por este facto, dificuldades acrescidas na tomada de decisões na fase de projecto. Preferiu-se privilegiar a flexibilidade em vez da eficiência.

A necessidade de integrar algum equipamento já existente, nomeadamente as máquinas de CNC, foi outra não menos importante restrição. Esta necessidade de integrar equipamento já existente, foi aceite como um importante desafio que pode ser encontrado em sistemas de produção reais, cuja evolução tem, quase sempre, de entrar em linha de conta com os sistemas já existentes. Os vários aspectos de integração e interoperabilidade entre os vários componentes tiveram assim de ser levados em consideração.

Uma outra restrição à arquitectura foi a necessidade do sistema suportar o funcionamento simultâneo de diferentes grupos de trabalho (experiências simultâneas). A unidade pode assim ser operada como um sistema FMS/FAS integrado ou então como um conjunto isolado de subsistemas.

A necessidade de suportar diferentes áreas de investigação, tipos diversos de demonstração e formação implicou o desenho de uma arquitectura flexível, desde o ponto de vista da topologia ao controlo. Para além das restrições impostas pelos requisitos das diferentes actividades houve também duas outras restrições que acompanham quase sempre os projectos: restrições financeiras e de espaço, que influenciaram também, de forma apreciável, a topologia e a arquitectura de controlo.

4.2.2.1. Subsistema FMS

O sistema FMS inclui uma fresa (DENFORD StarMill) e um torno (DENFORD StarTurn) de controlo numérico, que são alimentados através de um robô com 6 graus de liberdade (IRB 2000 da ABB). Este equipamento de controlo numérico (fresa e torno) já existia tendo sido integrado com o robô para formar a célula. O robô está instalado sobre um eixo controlável que permite o seu posicionamento em 2 posições distintas para servir as máquinas (Figura 4.4).

Os materiais a serem maquinados são transportados para a área de trabalho do robô em paletes próprias, pelo sistema transportador, enquanto que os materiais maquinados são expedidos da célula também recorrendo ao mesmo subsistema de transporte. O tapete deste subsistema, que passa em frente à célula FMS, possui dois pontos de paragem, colocados dentro da área de trabalho do robô, específicos para as paletes com matéria-prima e com produtos maquinados. Sempre que a célula estiver em funcionamento, no tapete, existem pelo menos duas paletes paradas, uma em cada um destes pontos. Estes pontos podem ser vistos como entrada e saída da célula, sendo o robô o responsável pela ligação entre a célula e a entrada/saída.

A matéria-prima para a entrada pode provir do (1) armazém automático ou da (2) entrada de matérias primas. Por seu lado, os materiais maquinados da saída, são enviados para um dos seguintes pontos: (1) expedição, (2) armazém automático ou (3) subsistema de montagem.

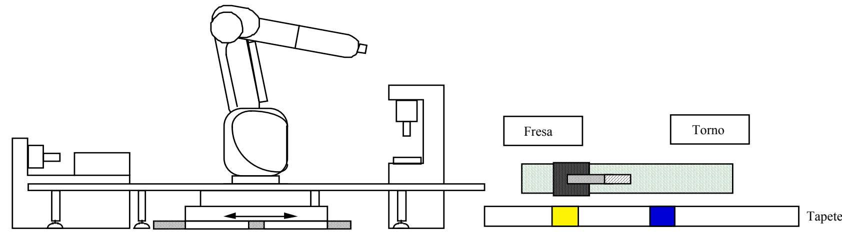


Figura 4.4 - Célula FMS

4.2.2.2. Subsistema FAS Multi-Robô

Este subsistema é composto por duas células robotizadas que podem funcionar de uma forma integrada ou isolada. As principais características operacionais do subsistema são:

- Operações de montagem realizadas de uma forma autónoma, isto é, cada um dos robôs pode funcionar independentemente do outro.
- Operações de montagem em linha envolvendo os 2 robôs
- Operações que envolvam a cooperação paralela dos 2 robôs
- Operações de montagem realizadas em movimento. Um dos robôs encontra-se montado sobre uma plataforma móvel que pode ser sincronizada com o tapete que transporta a palete onde se está a realizar a montagem

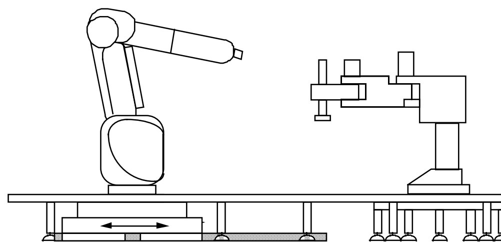


Figura 4.5 - Célula FAS Multi-robô

Os robôs existentes são: Robô **BOSCH TURBO SCARA 840** e Robô **ABB IRB 2000.**

4.2.2.2.1. Célula de Montagem 1

Esta célula foi criada em torno de um robô do tipo SCARA (BOSCH SR840), com capacidade para 10 Kg e que inclui um sistema automático de troca de ferramentas.

Para além do sistema que permite a troca de ferramentas, montado no braço do robô (BOSCH Exchange System GWS 20), estão instalados 6 "magazines" para armazenamento de outras tantas ferramentas que não estejam a ser correntemente utilizadas. Cada "magazine" possui um sensor binário que permite testar a presença de ferramenta. Cada ferramenta apresenta um tipo de dedos diferente.

O robô pode ser programado através de uma linguagem de programação do tipo PASCAL - BAPS [61]. Esta linguagem suporta, por exemplo, movimentos guardados que é um tipo de comando bastante importante para a realização de operações de montagem complexas.

Também foi montado no braço do robô um sensor de forças/momentos (SCHUNK FTS 30) .

As operações de montagem são realizadas sobre "*fixtures*" instalados em paletes que são fornecidas à célula por um tapete do subsistema transportador. Durante a realização da montagem, a palete está fixada por um posicionador mecânico, instalado no referido tapete, com 0.1 mm de precisão. Por seu lado, o fornecimento de materiais é assegurado também por paletes, transportadas pelo subsistema de transporte e posicionadas noutros tapetes que envolvem a célula.

Os produtos acabados ou os subprodutos, aqui processados, podem ser enviados para o armazém automático, para a outra célula ou então para a expedição.

4.2.2.2.2. Célula de Montagem 2

Esta célula foi criada em torno de um robô do tipo antropomórfico (ABB IRB 2000), com capacidade para 10 Kg e que inclui um sistema automático de troca de ferramentas.

Para além do sistema que permite a troca de ferramentas, montado no braço do robô (SCHUNK Pneumatic Exchange System GWS), estão instalados 4 "magazines" para armazenamento de outras tantas ferramentas que não estejam a ser correntemente utilizadas. Cada "magazine" possui um sensor binário que permite testar a presença de ferramenta. Cada ferramenta apresenta um tipo de dedos diferente.

O robô é controlado através de "*softkeys*"[43], "*joystick*" e pode ser programado através de uma linguagem denominada ARLA.

O robô está instalado sobre um "*charriot*"[44] que se move de um modo incremental ao longo de um eixo, sendo controlado por um PLC. Num dos extremos, o robô encontra-se a trabalhar na área de trabalho normal da célula 2; quando no outro extremo, o robô "insere-se" na área de trabalho da célula 1, permitindo assim a realização de trabalhos de cooperação entre robôs. Apesar do movimento do "*charriot*" poder ser sincronizado com o tapete que passa em frente à célula, permitindo operações de montagem em movimento, existem algumas limitações na realização destas operações. A limitação mais importante é o baixo nível de precisão deste tipo de operações derivado (1) da incerteza do posicionamento da palete, que não está rigidamente fixa ao tapete e (2) de diferenças de velocidade entre o tapete e o "*charriot*".

Tal como na célula 1, as operações de montagem são realizadas em paletes com gabarits, fixadas por um posicionador mecânico de 0.1 mm de precisão.

4.2.2.3. Subsistema de Transporte

É talvez a parte mais "visível" da unidade, sendo constituído por uma rede de tapetes que ligam os vários subsistemas, transportando paletes suportando um volume máximo de 200x200x200 mm^2 de 10 Kg de peso. Devido ao fluxo de materiais, produtos e subprodutos ser assegurado integralmente por este subsistema, foi necessária uma atenção especial no desenho da sua topologia. Assim, e tendo em atenção as restrições já referidas anteriormente (espaço, financeiras, ...) não foi possível evitar o recurso a secções críticas (zonas de transição exclusiva de paletes em ambos os sentidos) ou mesmo secções que, em determinadas condições, podem ser zonas de entupimento.

Uma vez que se pretendia um sistema dinâmico e autónomo em que a palete pudesse ser reencaminhada dinamicamente durante o seu percurso, optou-se pela utilização de um sistema de identificação que permitisse

---

[43] O nome dado pela ABB para indicar as teclas do "*teach-pendant*" que permitem movimentar o robô.

[44] Plataforma móvel, sobre a qual se montou o robô, permitindo assim a sua movimentação.

operações de leitura e escrita dinâmicas, de maneira a que a palete pudesse transportar, de alguma forma, identificadores que seriam utilizados pelo sistema de transporte para a encaminhar.

Desta forma optou-se pela instalação de um sistema desenvolvido pela BOSCH - ID80/E que é baseado fundamentalmente em 2 componentes [62]:

- **MDT** - Mobil Data Tag - A unidade que vai ser incorporada em cada palete, daí o termo "mobil". Permite que sejam realizadas, sobre ela, operações de escrita e leitura, podendo ser encarada como uma memória móvel, onde se pode afixar informação que pode ser lida posteriormente.
- **SLS** - São as estações que realizam a transferência da informação de e para o MDT - operações de leitura e escrita. O número de estações depende da dinamicidade pretendida ao nível da utilização da informação presente nos MDTs. De qualquer forma, existe uma relação de 1 para 1 entre o número de SLS e o número de cruzamentos, porque em cada cruzamento é necessário tomar decisões acerca do caminho a seguir.

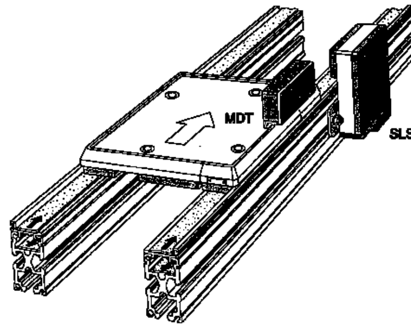


Figura 4.6 - Palete com MDT e SLS

O SLS pode ser utilizado como uma unidade autónoma de controlo (microPLC) em que o programa é carregado através duma porta série. O controlo é realizado através de entradas e saídas existentes no próprio SLS. A outra forma de utilizar o SLS é considerá-lo como uma unidade escrava de um "*host*" que o controla a partir da porta série através de comandos pré definidos [62]. Estes comandos permitem a escrita de informação no MDT que estiver a passar em frente ao SLS ou então a sua leitura. A forma de leitura e escrita pode ser estruturada.

A flexibilidade deste sistema permite que seja utilizado não só para o controlo de encaminhamento de paletes, podendo também ser, por exemplo, o suporte para um sistema de controlo distribuído em que cada palete transporta a informação de controlo necessária para a célula onde vai ser utilizada. Suponha-se, por exemplo, a célula 2, que pode realizar um diversificado conjunto de operações de montagem. A palete, antes de entrar, pode ser lida e o seu conteúdo determinar o conjunto de operações a realizar. O MDT, neste caso, é encarado como o transportador da lista de atributos que determinam a operação a realizar.

A memória do MDT pode então ser utilizada com vária finalidades: identificadores, dados de montagem, dados de maquinação e informação sobre o estado de produtos.

4.2.2.4. Subsistema Armazém Automático

É constituído por 50 alvéolos, cada um deles com capacidade para uma palete e é servido por um manipulador de 3 eixos. A ligação entre o armazém e o subsistema transportador é assegurada por um tapete reversível, onde foi montado um "*stopper*", que fixa as paletes para a operação de carregamento. Na operação de descarga a palete é simplesmente depositada no tapete.

Foi montado um SLS no braço do manipulador que, além de verificar a existência de palete no alvéolo, permite ainda confirmar qual o tipo. Esta verificação/confirmação do tipo de palete é bastante importante, nomeadamente durante a inicialização e para garantir que o armazém não descarrega uma palete errada.

4.2.2.5. Subsistema Sensorial

A topologia da NovaFlex não restringe qualquer possibilidade de evolução a nível sensorial, podendo afirmar-se que a unidade pode ser, em qualquer altura, enriquecida a este nível. A descrição seguinte representa por isso o estado na fase inicial da sua implementação.

A arquitectura sensorial da unidade será obviamente a arquitectura sensorial de cada um dos subsistemas referidos anteriormente: FMS, FAS, transporte e armazém automático.

Cada uma das máquinas (fresa e torno) engloba um conjunto de sensores, desenvolvidos e aplicados no contexto de um trabalho de doutoramento [63], utilizados na monitorização e diagnóstico de operações de maquinação. Os restantes sensores são maioritariamente sensores binários para determinar a existência de palete, de produto ou ainda o posicionamento do "*charriot*".

O sensor nobre da célula 1 do subsistema FAS é o sensor de força/momento instalado no braço do robô. Os restantes sensores são do tipo binário para o controlo de posição, existência de palete, existência de ferramenta no fixador, etc.

A célula 2 possui somente sensores binários utilizados com o mesmo fim dos da célula 1.

No subsistema de transporte está incluído o ID80E - Sistema de Identificação de Paletes, e o conjunto de sensores binários necessários para o controlo dos diferentes actores (tapetes, elevadores, estações elevatórias, "*stoppers*", ...) envolvidos no processo de encaminhamento de paletes.

O armazém automático não é muito rico do ponto de vista sensorial. Para além do SLS, montado no braço do manipulador e dos sensores inerentes ao manipulador, existe apenas um sensor de presença de palete para entrar no armazém.

4.2.2.6. Arquitectura de Controlo e Supervisão

A característica fundamental na definição da arquitectura de controlo e supervisão a utilizar para esta unidade reside na necessidade de incorporar diversos controladores locais e heterogéneos. Cada um destes controladores apresenta uma filosofia distinta, com arquitecturas computacionais diversas, o que implica um grande esforço de integração. A integração dá-se, não só ao nível global, mas também ao nível local. Na integração ao nível global consideram-se os aspectos relacionados com a integração da funcionalidade de cada um dos subsistemas particulares na arquitectura global de controlo do sistema (coexistência de subsistema), enquanto que na integração ao nível local consideram-se os aspectos relacionados com a integração dos controladores locais (coexistência de controladores locais) em cada um dos subsistemas.

Os controladores locais mais relevantes que existem na unidade são: controladores de robôs da ABB, controlador de robô da BOSCH, PLC C300 que controla o subsistema de transporte, controlador do armazém automático, PLCs de controlo dos "*charriots*" e controladores das máquinas ferramentas.

A base da arquitectura geral concebida do sistema assenta na existência de um computador UNIX/LINUX, por subsistema, que suporta o controlador de alto nível desse subsistema. Todos os computadores estarão ligados em rede, sendo o controlo de alto nível do sistema, baseado nesta infra-estrutura distribuída, que deixará as tarefas directamente relacionadas com o subsistema a cargo deste último (controlo local). O controlo de sistema preocupar-se-á apenas com as tarefas de gestão de alto nível (controlo global).

Dada a complexidade e grandeza do trabalho envolvido na concepção e implementação da arquitectura de controlo e supervisão da unidade, resolveu-se realizar este trabalho por fases, baseadas na natureza sistémica da unidade. Fazia sentido que fossem realizados primeiramente os trabalhos de integração local, após o que se

poderia passar a uma definição de arquitectura geral de subsistema. Após todos os sistemas terem a sua arquitectura geral definida, poderia passar-se à definição da arquitectura global de controlo da unidade, tendo agora em atenção os aspectos de integração global.

O esforço de trabalho realizado sobre a NovaFlex, para esta tese, assentou essencialmente sobre os primeiros aspectos de integração (integração local) em vários subsistemas: transporte, célula 1 do FAS e armazém automático, isto no que diz respeito à componente de integração de sistemas. A parte referente à modelação considerou também este primeiro nível da definição da arquitectura

Dentro desta filosofia integram-se os trabalhos de alunos co-orientados pelo autor da tese: **Steam** - Sistema de Transporte e Encaminhamento Automático de Materiais, **BOSCHSYS** - Sistema de Integração de um Controlador BOSCH num ambiente UNIX e **GAA** - Gestão e Monitorização do Armazém Automático.

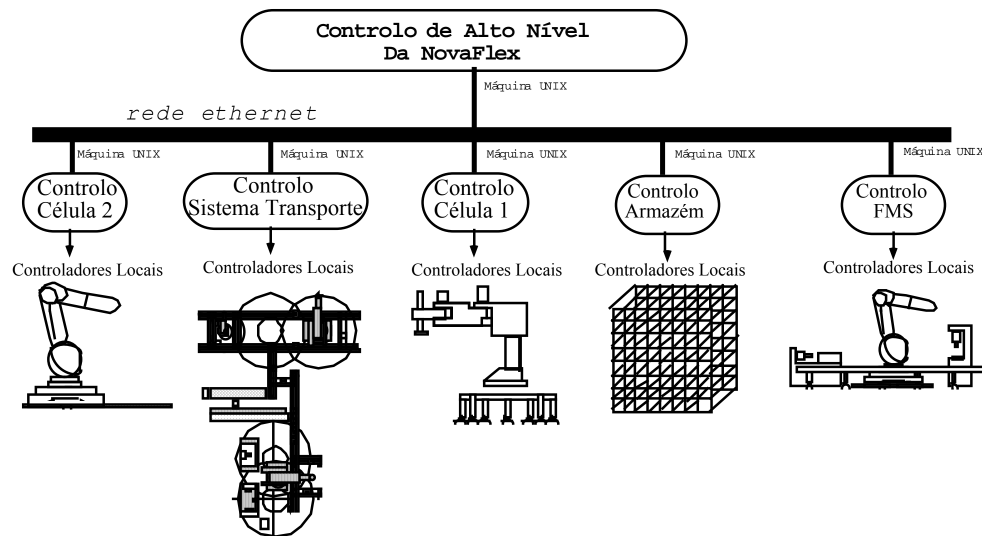


Figura 4.7 - Arquitectura de Controlo da NovaFlex

O Steam consistiu no desenvolvimento de uma infraestrutura de controlo, baseada no paradigma cliente-servidor, para a gestão do sistema de transporte e encaminhamento da NovaFlex. O software desenvolvido permite a movimentação de paletes através do sistema de transporte da forma mais flexível possível, ao mesmo tempo que não compromete a fiabilidade do sistema. As primitivas principais que podem ser utilizadas pelos clientes estão directamente relacionadas com a movimentação de paletes e o tratamento de emergências.

O BOSCHSYS consistiu na integração do robô SR800 e respectivo controlador RS/82 através do paradigma servidor-cliente. Assim foi desenvolvida uma infra-estrutura que torna transparente o robô e controladores físicos aos clientes existentes na máquina onde foi criada esta infra-estrutura. Para além da implementação das primitivas relacionadas com a movimentação do robô (aceleração, move, velocidade, ...), virtualizaram-se também as suportadas pelo sistema operativo próprio do controlador RS82 (gestão de ficheiros, "*debug*", compilação, ...).

O GAA consistiu no desenvolvimento de um sistema de gestão e monitorização do armazém automático, através da criação de um servidor capaz de providenciar três tipos de serviços: (1) "*queries*" destinadas a indagar sobre o estado de ocupação do armazém, (2) acções sobre o armazém (movimentar, descarregar e carregar paletes) e (3) especiais, destinados a clientes específicos como é o caso da interface gráfica.

4.3. Modelação e Integração

4.3.1. Integração

Todo o trabalho de integração dos controladores locais assentou no paradigma servidor-cliente. Para cada controlador criou-se uma "imagem" das suas funcionalidades, que é acedida através de um servidor. A ligação do lado dos clientes segue a filosofia normal de "Remote Procedure Calls" - RPC. A ligação ao controlador, é dependente da arquitectura física do controlador, utilizando-se, na maioria dos casos, uma ligação baseada no protocolo RS232C ou "*loop*" de corrente.

Como já se teve oportunidade de escrever, aquando da discussão sobre os trabalhos de integração na célula SONY, a ligação do servidor ao controlador não passa apenas pelo desenvolvimento de software do lado do servidor. É necessário também um esforço do lado do controlador físico no sentido de o "adaptar" aos novos requisitos. Enquanto que do lado do servidor, qualquer que seja o tipo de controlador, se segue sempre a mesma metodologia: desenvolvimento de um servidor "pendurado" num RPC, do lado físico isso já não é possível dada a sua grande diversidade.

Para um melhor esclarecimento deste tema optou-se pela apresentação breve das arquitecturas utilizadas na realização dos trabalhos co-orientados pelo autor: Steam, BOSCHSYS e GAA.

4.3.1.1. Steam[45]

Pretendia-se a criação de um servidor que virtualizasse as funcionalidades do sistema de transporte, nomeadamente o encaminhamento de paletes para as diferentes secções. Foram definidas primitivas para a alteração/consulta de rotas e para o encaminhamento de paletes, que devem ser definidas com diferentes níveis de abstracção. Não se defendia a existência de primitivas de muito alto nível, por se poderem tornar facilmente inflexíveis e pretendia-se, acima de tudo, um sistema flexível.

Os sistemas de transporte, vulgo linhas, instalados nas empresas actuais, são, na sua grande maioria, pouco flexíveis ao nível do controlo, vejam-se os casos das linhas instaladas na Blaupunkt, ARP, Grundig, Delco-Remi, Philips[46]. Cada uma destas linhas possui um controlador - PLC, programado única e exclusivamente para um determinado conjunto de rotas fixas, por palete. Qualquer alteração de rota em tempo real é impossível, já que não existe nenhuma possibilidade de interferir com o PLC de controlo. Implementar uma arquitectura que permita uma utilização flexível, do ponto de vista de controlo, da "linha" representa uma vantagem, com bastante utilidade para a instalação de novas linhas. A experiência do autor em discussões com industriais sobre projectos de subsistemas de transporte mostrou-lhe que muita da complexidade que se encontra em algumas topologias, poderia ser substancialmente reduzida recorrendo a uma arquitectura de controlo flexível, com todas as vantagens de redução de custos iniciais e de manutenção.

A construção de uma arquitectura de controlo flexível baseada nos sensores SLS/MDT apresenta-se, à partida, mais simplificada, desde que sejam colocados SLS em número suficiente e nos pontos importantes[47]. O tipo de encaminhamento a realizar seria conhecido em qualquer dos cruzamentos. Por outro lado, alterações dinâmicas da rota de uma dada palete, também poderiam ser realizadas, por alteração de informação no MDT.

A construção de uma arquitectura de controlo não baseada em SLS/MDT apresenta-se muito mais complicada mas tem a vantagem de oferecer soluções mais baratas. Os custos com os SLS/MDT são ainda elevados. Evidentemente que uma solução deste tipo será menos fiável já que as posições das paletes são baseadas em suposições e não em certezas como aconteceria com a arquitectura anterior. Na primeira fase do

---

[45] Os alunos envolvidos neste projecto foram: Sandra Pinto Gadanho e Nuno Chalmique Chagas.

[46] Esta impressão resultou de visitas realizadas pelo autor às referidas empresas. O caso que o autor melhor conhece é o da Grundig/ARP/Blaupunkt, a cujas instalações (Braga), já por diversas vezes se deslocou. Este grupo de empresas é também um bom exemplo pelo número de linhas de montagem que têm instaladas.

[47] Antes de cada cruzamento deveria estar colocado um SLS.

trabalho pretendeu-se experimentar uma solução baseada nesta arquitectura, isto é, não considerando os SLS/MDT.

Existem no subsistema diversos tipos de tapetes com vários cruzamentos e pontos de paragem, permitindo a definição de várias rotas. O subsistema foi dividido em secções que correspondem a pontos onde existem "*stoppers*" ou então alteração de encaminhamento. Na figura 4.8 mostram-se essas secções. Uma rota é uma sequência de opções de paragem e encaminhamento, em que as opções são representadas por bits. (0 ou 1).

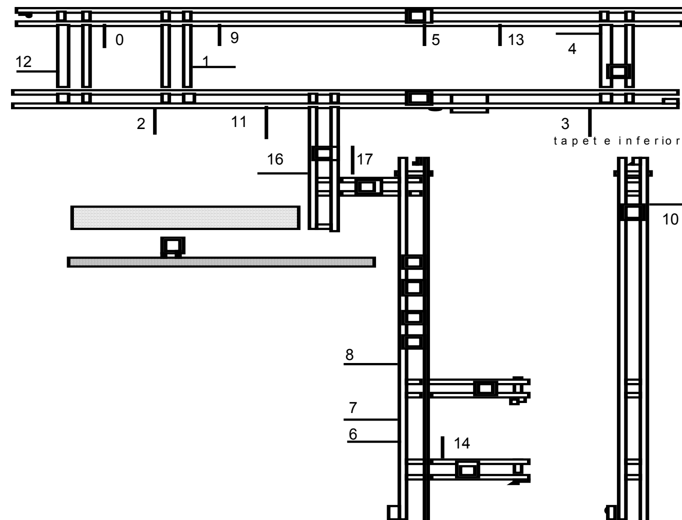


Figura 4.8 - Esquema das Secções do Subsistema de Transporte

Os serviços mais importantes fornecidos aos clientes são os seguintes [59]:

- **open_req** - abertura lógica do sistema
- **close_req** - fecho lógico do sistema
- **monitor_req** - envio para o cliente de informação de monitorização
- **emergency_req** - activa a entrada de emergência do controlador
- **agente_req** - sinalização para os subsistemas armazém, FAS e FMS
- **palllet_in_req** - introdução de paletes no sistema na entrada de materiais ou do armazém. Em certas condições podem ser introduzidas manualmente no sistema, em qualquer secção que não a zona crítica.
- **pallet_release_req** - liberta uma palete que esteja parada por imposição da sua rota.
- **setroute_req** - atribui uma nova rota para uma dada palete. Não entra em consideração com a posição corrente.
- **changeroute_req** - permite alterar a rota a partir da posição corrente.
- **getroute_req** - retorna informação de rota e posição da a palete.

O problema na construção do servidor residia essencialmente no facto do controlador do sistema de transporte - PLC CL300, ser pouco flexível ao nível da comunicação com o exterior. Não existia qualquer primitiva de programação para a comunicação com a porta série, que serve apenas para fazer a "carga"/"descarga" de programas e o envio de informação de monitorização. A "adaptação" do PLC ao servidor parecia um pouco difícil, estando a utilização de um interpretador sobre o PLC , à partida, completamente posta de lado.

O problema foi torneado recorrendo ao sistema operativo do CL300. Existe um modo de funcionamento do PLC que o torna controlável a partir de um *host* externo através de uma aplicação proprietária da BOSCH: "BOSCH - PROFI - Professionl Integrator". Esta aplicação permite ler de portos de entrada ou posições de memória do PLC e actuar portos de saída ou posições de memória. Tendo acesso ao protocolo, o PLC poderia passar a ser um escravo do "*host*", que enviaria os comandos quando e como fosse necessário. Note-se que, com esta solução, o PLC passa a ser um completo escravo do *host*, limitando-se o seu programa a estar condicionado por variáveis actuadas pelo *host*.

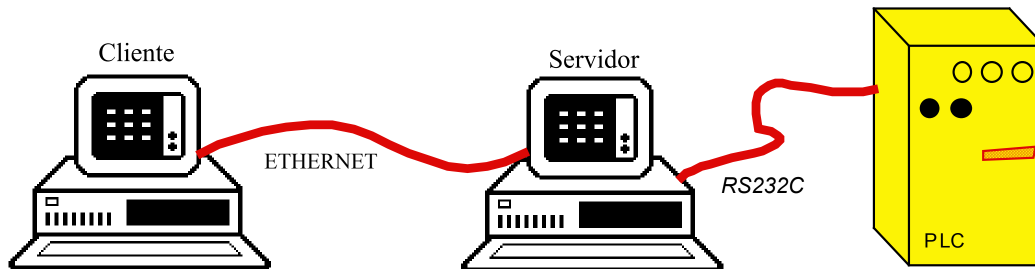


Figura 4.9 - Arquitectura Hardware do Subsistema de Transporte

Como não existia qualquer informação acerca do protocolo foi necessário mais uma vez recorrer ao analisador de protocolos tendo a equipa que participou no projecto (Sandra Gadanho + Nuno Chagas) desenvolvido um esforço notável que conduziu à sua descoberta. A partir deste momento estavam criadas as condições para o desenvolvimento da arquitectura de software.

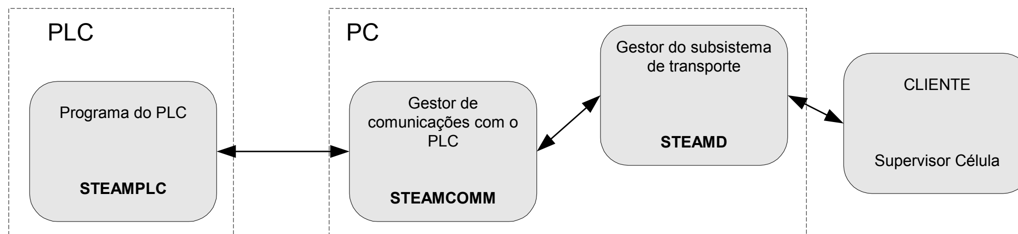


Figura 4.10 - Arquitectura de Software do Subsistema de Transporte

O gestor do subsistema de transporte recebe os pedidos dos clientes; possui uma base de dados com informação de posição e trajectória de todas as paletes da unidade. O facto do subsistema ir abaixo não implica a perca do estado das paletes, desde que não se mudem manualmente.

O gestor de comunicações com o PLC assegura o envio de comandos para o PLC (actuação de saídas, memórias, etc) e também a recepção de informação acerca do estado de entradas, saídas e memórias do PLC. O STEAMCOMM é simultaneamente cliente e servidor do STEAMD. Como servidor presta serviços através do envio de comandos provenientes do STEAMD para o PLC (actuação de portos ou memórias). Como cliente envia o estado das variáveis que estão a ser monitoradas no PLC.

O programa do PLC tem como já se referiu pouca autonomia, recebendo quase só, do gestor de comunicações, apenas informação de opções de encaminhamento ou paragens relativas, nos cruzamentos ou stoppers, respectivamente. Mas apesar desta simplicidade aparente, o programa do PLC tem ainda alguns eventos a seu cargo: tratamento de emergências, controlo dos tapetes reversíveis e controlo dos stoppers ou elevadores.

4.3.1.2. BOSCHSYS[48]

Neste subsistema pretendia-se a criação de um servidor que virtualizasse as funcionalidades do controlador RS82 do robô SR800 da BOSCH. O conjunto de primitivas exportáveis pelo servidor deve ser o mais próximo possível do conjunto de primitivas oferecidas, quando se manipula directamente o controlador. A motivação da criação deste servidor é integrar um dos componentes básicos da célula que será posteriormente utilizado pelo cliente controlador da célula FAS. Note-se que este projecto não está a criar o controlador da célula integral, está apenas a integrar o seu componente fundamental.

O Controlador Local

O controlador RS82 tem dois modos fundamentais de funcionamento: automático e manual. Quando se liga o controlador pela primeira vez é necessário referenciar o robô, o que só é conseguido em modo de funcionamento manual e por activação do "teach-pendant".

No modo de funcionamento manual o utilizador tem acesso, através da porta série, ao sistema operativo do controlador que permite entre outras coisas o seguinte:

- mostrar os ficheiros existentes na memória de programas do controlador
- editar ficheiros de programas e pontos
- compilar ficheiros de programas e pontos
- verificar o estado do controlador ao nível das suas variáveis internas e entradas e saídas digitais
- executar um programa

A única forma de movimentar o robô, quando em modo manual, é através do "teach-pendant", ou então por execução de um programa, não existindo, infelizmente, comandos ao nível do sistema operativo para o movimentar. Também não é possível colocar um programa a "correr", directamente do sistema operativo; para além do comando ao nível do sistema operativo é necessário actuar um botão no painel de controlo. Enquanto dura a execução do programa a porta série fica sob possível controlo do programa em execução.

Em modo automático, a execução de programas é feita a partir do painel de controlo, por escolha do número do programa que se pretende correr.

A paragem de um programa é feita, em automático ou em manual, por activação do painel de controlo.

Uma limitação apontada ao controlador, especialmente por utilizadores de fábricas com poucos conhecimentos de programação, é a impossibilidade de fazer programas que sejam simples colecções de pontos, como acontece por exemplo com o controlador do IRB 2000 da ABB (modo "playback"). A programação deste robô implica conhecimentos de linguagens de programação do tipo PASCAL, já que a sua linguagem - BAPS, tem uma estrutura semelhante, com o mesmo tipo de estruturas de controlo, atribuição de tipos a variáveis, procedimentos, etc. A edição, compilação e execução do programa é suportada pelo sistema operativo proprietário.

Todos os botões do painel de controlo estão ligados à carta de entradas digitais do PLC.

A integração

Tendo em atenção a arquitectura do controlador físico e considerando que a implementação de uma sua "imagem" fiel, do lado do servidor implica uma total acção de controlo por parte deste último, verifica-se que houve necessariamente que colocar um esforço no sentido de o adaptar aos requisitos.

---

[48] A aluna envolvida neste projecto foi: Florbela Tique Aires.

O facto do robô não poder ser controlado directamente a partir da porta série conduziu à necessidade de o controlar através de um programa interpretador que é colocado na memória do controlador físico (ver descrição sobre integração do robô SONY). Está desenvolvido na linguagem do robô (BAPS) e os comandos que o interpretador aceita são enviados pela porta série.

Os problemas mais importantes relacionados com a "adaptação" do robô ao servidor foram então a necessidade de implementar um interpretador e o facto das acções para correr programas, parar programas, etc, terem de ser actuadas através de ligações directas ao PLC do controlador. Houve por isso a necessidade de "cablar" essa entradas e liga-las à porta paralela do PC. Aparentemente não há nada de especial nesta ligação e, do ponto de vista hardware nada há, mas do ponto de vista de software já o mesmo não se passa. Os "device-drivers" fornecidos pelo UNIX não abrangiam o tipo de controlo que se pretendia para a porta paralela, isto é, não havia nenhum meio de accionar os bits pretendidos da porta paralela, tendo por isso sido desenvolvido um "device-driver" para a porta paralela.

A ligação física entre o servidor e o controlador ficou então assente em duas ligações externas:

- uma ligação série por onde são transmitidos comandos para o interpretador residente no controlador e recebidos resultados decorrentes da acção desses comandos
- uma ligação paralela que se destina a controlar remotamente acções como executar e parar um programa

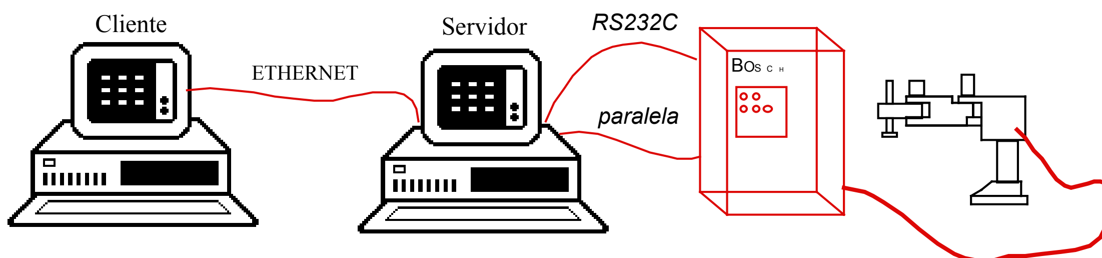


Figura 4.11 - Arquitectura de Hardware da Integração do Controlador BOSCH RS82

Pelo facto do controlador possuir um sistema operativo ao qual se acede via porta série, procurou-se que o servidor descrevesse também esta possibilidade, garantindo assim uma "imagem" mais fiel. Assim os serviços prestados pelo servidor aos clientes podem ser divididos em:

- **sistema de ficheiros**
- **sistema de monitorização**
- **sistema de programação**

Os serviços mais importantes fornecidos aos clientes, organizados por tipo, são os seguintes [58]:

- sistema de ficheiros
  - **ListAll** - lista todos os ficheiros existentes na memória do controlador
  - **ListUser** - lista todos os ficheiros de um utilizador
  - **GetFile** - obtém um ficheiro
  - **SendFile** - envia um ficheiro
  - **DeleteFile** - apaga um ficheiro
  - **CopyFile** - copia um ficheiro
  - **CompileFile** - compila um ficheiro

- **ExistsFile** - verifica se um ficheiro existe

- sistema de monitorização
    - **GetBasic** - recebe o estado de um conjunto de variáveis do controlador, como por exemplo, saídas, entradas, programa corrente, erros, tempo, data, aceleração, velocidade, ...
    - **GetOutputs** - recebe o estado das saídas
    - **GetInputs** - recebe o estado das entradas
    - **GetAxisPos** - recebe a posição corrente do robô
    - **SetFactors** - ajusta os factores de aceleração e velocidade do robô
- sistema de programação
    - **ProgramStart** - activa um programa executável que esteja no controlador
    - **ProgramEnd** - pára o programa em execução no controlador
    - **StartInterp** - activa o interpretador de comandos para controlo *on-line*
    - **StopInterp** - pára o interpretador
    - **MoveInst** - movimenta o robô
    - **WriteVar** - afecta as variáveis de aceleração e velocidade
    - **GetInput** - recebe o valor de uma entrada
    - **SetOutput** - actua uma saída
    - **....**

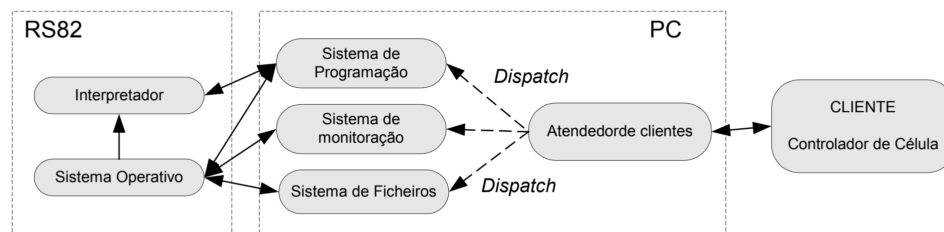


Figura 4.12 - Arquitectura de Software do BOSCHSYS

Os três componentes do servidor (sistema de programação, sistema de monitorização e sistema de ficheiros) são actuados de acordo com os pedidos que chegam. O programa com que cada um deles dialoga, no RS82, depende do tipo de pedido. Qualquer pedido do tipo sistema de ficheiros e sistema de monitorização implica sempre um diálogo com o sistema operativo do RS82. Por outro lado, o sistema de programação já implica um diálogo a dois níveis diferentes: (1) antes de correr o interpretador e (2) depois de correr o interpretador. Enquanto o interpretador não é activado, existe diálogo com o sistema operativo. Depois, passa a haver com o interpretador.

4.3.1.3. GAA[49]

Neste subsistema pretendia-se a criação de um servidor que virtualizasse as funcionalidades do subsistema de armazenagem. A "imagem" deste servidor deveria conter não só as primitivas de movimento, típicas de um

---

[49] Os alunos envolvidos neste projecto foram: Eduardo Braz e Luís Fernandes.

manipulador, mas também outras primitivas de mais alto nível, tipicamente relacionadas com a gestão de um armazém.

O armazém faz a recepção e entrega de paletes de e para o subsistema de transporte, respectivamente. Um ponto interessante nesta discussão é a presença de um SLS no braço do manipulador que permite a leitura e escrita de informação nos MDTs das paletes. Foi já referido que este sensor é utilizado para verificar a existência de palete ou ainda o seu tipo. A gestão do armazém é realizada com base na informação presente nos MDTs, pelo que se teve de organizar o formato de memória do MDT (figura 4.13).

Durante a fase de desenho e montagem deste subsistema, o autor participou em diversas discussões com os alunos responsáveis pela sua implementação, acerca do tipo de arquitectura de controlo que o sistema deveria ter. Foi sempre defendido que o controlador, baseado num PC, deveria conter um sistema operativo tipo UNIX que permitisse a realização de multi tarefas. A arquitectura de controlo de software consistiria apenas no desenvolvimento de um "device-driver" que permitiria o comando de muito baixo nível do manipulador. A integração do armazém consistiria agora no desenvolvimento de um servidor que atenderia os pedidos dos clientes através da manipulação directa do "device-driver".

1DEFH
System Data
Dados Opcionais
Lista de Peças
Tamanho do Cabeçalho
Tipo de Palete
Identificação da Palete
0000H

Figura 4.13 - Formato de Memória do MDT

A falta de conhecimentos informáticos dos implementadores, o tempo disponível para esta implementação e o tipo de placas utilizadas para controlo dos motores levaram a que a sugestão desta arquitectura não fosse feita, tendo sido apresentada uma solução tipicamente DOS, isto é, foi fornecido um conjunto de primitivas para o controlo do manipulador, mas em mono-programação.

Desta forma a arquitectura de hardware teve de comportar a introdução de um segundo computador, onde foi instalado o servidor. Este PC terá um sistema operativo do tipo UNIX, fazendo a ligação com o PC controlador através da porta série. Terá uma ligação à rede "*ethernet*", permitindo a sua integração num sistema global, de forma a que os clientes possam aceder remotamente.

O tipo de arquitectura hardware condicionou a arquitectura de software à existência de 2 módulos principais: (1) **controlador** e (2) **servidor**.

A existência de um PC como controlador permitiu a criação de uma arquitectura de software mais flexível para o controlador. Pretendeu-se que o utilizador também pudesse intervir de alguma forma no controlador, de maneira a que fosse possível controlar o armazém com comandos de baixo nível. As acções fundamentais do controlador são [57]:

- controlo de motores

- comunicação com o PLC do subsistema de transporte para sincronismo na entrega e recepção de paletes
- gerir a informação relativa ao armazém, guardando-se por isso informação acerca das paletes, nomeadamente o conteúdo dos MDTs. A cada alvéolo associa-se a seguinte informação: estado presente de ocupação, estado futuro de ocupação, coordenada em x, coordenada em y, tipo de palete, identificador da palete, tamanho de dados do MDT, identificador do pedido e dados do MDT.
- receber os pedidos enviados do servidor, pela porta série

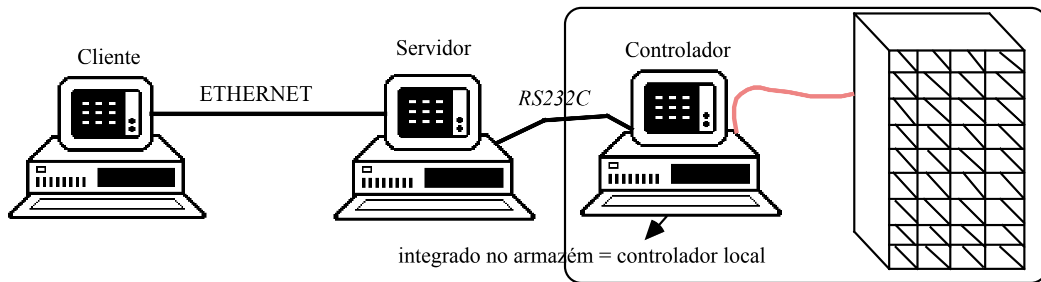


Figura 4.14 - Arquitectura Hardware do Armazém Automático

A possibilidade dos pedidos enviados pelo servidor estarem sujeito a demoras levou a que se implementasse uma fila de pedidos. Para uma maior flexibilidade de controlo implementou-se a possibilidade do utilizador, através da linha de comando, realizar pedidos semelhantes aos enviados do servidor pela porta série.

Os pedidos possíveis ao controlador estão divididos em duas categorias: (1) os que esperam pela sua vez, decorrente da espera do manipulador e (2) os atendidos imediatamente. Os com espera são:

- **Insere_palete** - destina-se a inserir uma palete manualmente no alvéolo
- **Mov_palete** - destina-se a movimentar uma palete da posição x1,y1 para a posição x2,y2
- **Descarrega_paletes** - destina-se a descarregar n paletes para a linha
- **Carrega_paletes** - destina-se a carregar n paletes da linha
- **Le_MDT** - destina-se a ler o valor do MDT numa determinada posição x,y
- **Escreve_MDT** - destina-se a escrever um valor no MDT numa determinada posição x,y

Os logo atendidos são:

- **Pausa** - suspende a execução do controlador
- **Start** - inicia a execução
- **Remove_palete** - remove uma palete da estrutura de dados que contém a imagem do armazém
- **Imagem** - devolve o estado de ocupação do armazém.
- **Output** - Activa saídas
- **Input** - Lê entradas

Do ponto de vista da gestão do armazém importa ainda referir algumas decisões tomadas:

- Sempre que é feito um pedido para descarga de n paletes e não exista esse número, o pedido é recusado

- Sempre que é feito um pedido para carga de n paletes e não exista esse número de alvéolos livres, o pedido é recusado
- Assim que ocorre um pedido de descarga de n paletes e existam essas paletes, são imediatamente reservadas para evitar que um próximo pedido seja atendido com sucesso. Suponha-se a existência de 50 alvéolos ocupados com paletes e chega um pedido de descarga para 50 paletes. O sistema começa a descarregar e chega um novo pedido de 10. Caso não se tivessem reservado as paletes, quando do pedido anterior, o sistema nada diria e o cliente poderia ficar eternamente à espera. A reserva de uma palete é feita através da activação do campo "estado futuro" que está associado a cada alvéolo.
- Foram definidas algumas estratégias de carregamento: linha a linha, coluna a coluna e célula mais próxima.

O servidor destina-se a satisfazer o pedido de clientes acerca da gestão do armazém, existindo três tipos de serviços: (1) sem espera, ou seja, assim que é enviado um pedido é registado, sendo o cliente informado se o pedido foi ou não aceite, pelo servidor; (2) espera confirmação, ou seja, o cliente espera pela confirmação que o servidor tenta obter do controlador; (3) espera pela execução, ou seja, o cliente fica à espera que o comando seja completamente executado recebendo essa confirmação do controlador via servidor.

A arquitectura de software do servidor baseia-se na existência de três processos (figura 4.15): autenticador, gestor e o despachante.

O despachante é o processo directamente relacionado com a comunicação com o controlador. A existência de um processo a correr separadamente, para o tratamento de comunicações permite uma maior flexibilidade.

O autenticador destina-se a garantir a autenticidade do cliente para realizar os serviços que está a pedir. É uma forma de restringir o acesso a clientes indesejáveis. Foi utilizado o algoritmo de Kerberos [64] na implementação do protocolo de autenticação.

O gestor é o ponto de entrada, recebe os pedidos dos clientes, tendo sido implementado com RPCs da SUN™, tendo-se alterado os *stubs*, para permitir o desempenho de outras funções.

O controlador já foi suficientemente descrito anteriormente, podendo, como conclusão, dizer-se que é o processo base na gestão do armazém.

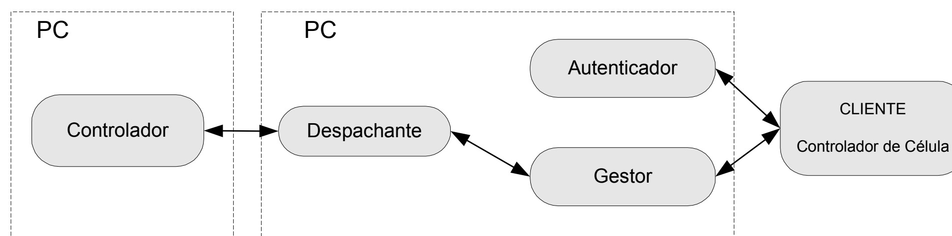


Figura 4.15 - Arquitectura de Software do Armazém Automático

Os serviços disponibilizados pelo gestor, dividem-se em 3 tipos:

- ***queries*** sobre o estado do armazém, podendo obter-se a seguinte informação:
  - estado de ocupação de um alvéolo
  - estado de ocupação do armazém: presente e futuro
  - conteúdo de cada palete, por leitura do MDT
  - tipo de paletes existente em armazém,

    - existência de espaço para n paletes
    - quantas paletes existem de um determinado tipo
    - posição de uma dada palete
    - estado dos sensores
- **acções** sobre o armazém
    - descarga de uma palete especificando o tipo
    - descarga de uma palete especificando o id
    - descarga de n paletes de um dado tipo segundo a estratégia escolhida pelo utilizador
    - carga de n paletes segundo a estratégia escolhida pelo utilizador
    - transitar do modo de carregamento por pedido para o modo de carregamento sempre que exista palete na entrada do armazém
    - mover paletes no armazém
- **serviços especiais** relacionados com a interface gráfica.

4.3.1.4. Outros Subsistemas

A infra-estrutura de controlo básica da NovaFlex não fica completa sem o desenvolvimento de outros subsistemas: célula de montagem com robô ABB, sistema de plataformas e máquinas ferramentas. Este trabalho irá continuar a ser feito, devendo ser considerado fora do contexto desta tese, dada a dimensão do trabalho.

De qualquer forma, pensa-se que os casos tratados são suficientes para a ilustração da metodologia.

4.3.2. Modelos

O exemplo de modelação que se vai apresentar será baseado na célula 1 do subsistema FAS da NovaFlex. Será ainda apresentada uma proposta de modelo para o subsistema de transport, que será encarado como sendo um sistema composto complexo constituído por vários tapetes; apresentar cada um tapete separadamente é um erro já que o sistema, fisicamente, tem um controlo global. Como formalismo de modelação vai-se utilizar o sistema de Frames.

A célula BOSCH será definida da seguinte forma

```
FRAME CELULA_1
instance-of: célula-montagem
nome: BOSCHSYS
coordenadas_base:
plano_montagem: get_ferramenta1, get_part, ... demon if_needed
                start_expansão()
produtos_processáveis: montagens-por-empilhamento
domínio-entrada: TAPETE, AGV, AL-VIB, GRAVITICO, BUFFER, MESA_IND,
                 STEAM
domínio-saída: TAPETE, AGV, BUFFER, MESA_IND, STEAM
domínio-agente: robô-scara, robô-antropomórfico, manipulador
entrada:tapete_esq, tapete_dir
saída:tapete_saída
agente:robô-BOSCH
method iniciar-montagem: iniciar_montagen_fn()
```

Domínio-entrada, domínio-agente e *domínio-saída* são slots de metaconhecimento, importantes para a fase de configuração. Estão aqui representados a título meramente indicativo, já que estão definidos no conceito de célula e são importantes apenas durante a configuração.

O método *iniciar-montagem* dá início ao plano de montagem, guardado em *plano_montagem*.. Este plano está definido por operadores de alto nível sendo necessário fazer a expansão para os operadores de baixo nível , definidos na estrutura célula_1.

Recordando os aspectos de modelação que foram apresentados no capítulo 3, que consideravam a entrada, a saída e o agente da célula como sendo modelada, não através de instâncias de componentes, mas sim por instâncias de entidades que descrevem o componente a representar um dado papel, tem-se que os valores colocados nos slots *entrada, saída e agente* terão de ser instâncias destas últimas entidades.

A entrada de materiais será realizada através de 2 tapetes colocados, um de cada lado do robô (*tapete_esq* e *tapete_dir*). Ambos os tapetes pertencem ao sistema de transporte. Desta forma considera-se que tapete_esq e *tapete_dir* são instâncias de um conceito que representa agentes de entrada. Os agentes são aparentemente iguais, o mesmo papel (*entrada-por-palete*) para o mesmo componente (subsistema de transporte). Mas existe uma pequena diferença que se baseia no facto do subsistema de transporte ser um componente complexo, com utilizações diversas; cada um dos agentes de entrada utiliza uma secção diferente do componente subsistema de transporte, daí que se tenham criado, para o papel ENTRADA-POR-PALETE, o *slot secção* que indica qual a secção que se vai utilizar no subsistema de transporte.

O papel de entrada-por-palete é descrito da seguinte forma:

```
FRAME ENTRADA-POR-PALETE
is-a: ENTRADA
ref-entrada:
paletes: if_needed entra_palete
secção:
atributos-principais: ref-entrada, paletes, secção, entra-palete
atributos-componentes: insere-palete
method entra_palete: entra_palete_fn
method obtém_parte: obtem_parte_fn
```

Paletes indica os identificadores de palete que estão na entrada. Quando se pretende dar entrada a uma dada palete, deve-se adicionar a este slot o seu identificador. O demónio associado à operação if_needed (não existe palete) evoca o método *entra_palete* que modela a funcionalidade de entrada de palete no tapete, isto é, modela a chegada da palete à secção correspondente do sistema de transporte. O código deste método evocará um outro método que todos os componentes que participam em agentes de entrada por palete têm de exportar: insere_palete(secção). O valor do argumento será construído a partir do valor presente no atributo secção.

Secção indica sobre que secção do subsistema de transporte é que se realiza a entrada. Secções são pontos especiais do subsistema de transporte, onde as paletes podem ser paradas, por controlo de um "stopper".

Os slots *atributos-principais e atributos-componentes* indicam quais os slots herdados no mecanismo de herança que governa a relação *desempenha* e *realizado_por*, respectivamente, que se encontra definida para os agentes (ver capítulo 3).

A funcionalidade principal está descrita pelo método *obtém_parte* que devolve o referencial da próxima peça a ir buscar. Este referencial é calculado a partir do ref-entrada (referencial base da palete) e do referencial da peça que é conhecido no frame que descreve palete. Quando se chama este método e não existe nenhuma palete actua-se o demónio do slot palete, fazendo com que o subsistema de transporte lhe entregue uma palete. Se, por outro lado, não existir referencial de peça, porque se acabaram os materiais, remove-se a palete do slot *paletes* e activa-se o método *liberta_palete* do STEAM que conduzirá a palete até ao ponto de saída de materiais.

Como já foi dito, cada uma das entradas será assegurada por um agente que desempenha um determinado papel e é realizado por um determinado componente. Tapetes_Entrada que se mostra a seguir será um frame apenas para englobar os agentes que participarão na entrada por palete.

```
FRAME TAPETES_ENTRADA
is-a: papel_agente
relation desempenha:ENTRADA-POR-PALETE
```

```
relation realizado-por:STEAM
```

As relações *desempenha* e *realizado-por* são definidas da seguinte forma:

```
RELATION DESEMPENHA
type: intransitive
inherit_slot: atributos-principais
inverse_relation: desempenhado-por

RELATION REALIZADO-POR
is-a: relation
type: intransitive
inherit_slot: atributos-componentes
inverse_relation: realiza
```

Tapete_esq e tapete_dir serão instâncias de TAPETES_ENTRADA, já que as suas diferenças ocorrem apenas ao nível do valor dos atributos.

```
FRAME TAPETE_ESQ
instance-of: TAPETES_ENTRADA
ref-entrada:
paletes: demon if_write dem_inspal_fn
secção:2
method obtém_parte: obtem_parte_fn
method entra_palete: entra_palete_fn
method insere_palete: insere_palete_fn(secção)

FRAME TAPETE_DIR
instance-of: TAPETES_ENTRADA
ref-entrada:
paletes: demon if_write dem_inspal_fn
secção:0
method obtém_parte: obtem_parte_fn
method entra_palete: entra_palete_fn
method insere_palete: insere_palete_fn(secção)
```

Será importante agora definir o componente *STEAM*, que será uma instância de um componente complexo *sistema_transporte*.

```
FRAME STEAM
instance-of: sistema_transporte
relation controlado_por: controlador_steam
heranca_controlado_por: liberta_palete
secções: 1, 2, 3, ..., 14
rota_para_secção_1:0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1
rota_para_secção_0:0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1
rota_para_secção_2:0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1
rota_liberta_secção_1:0, 0, 1, 0, 1
rota_liberta_secção_2:0, 0, 1, 0, 1
rota_liberta_secção_0:0, 0, 0, 0, 1, 0, 1
method insere_palete: insere_palete_fn(secção)
method liberta_palete: liberta_palete_fn(secção)
method existe_palete: existe_palete_fn(secção)
method fim_montagem_fas1 fim_montagem_fas1_fn()
```

Uma das características mais importantes na modelação de um dado componente é a definição do seu comportamento, isto é, como é que se vai descrever o comportamento que o componente tem fisicamente, usando uma linguagem computacional. Como se viu, os métodos e demónios são bastante adequados para este problema desde que se garanta uma forma de ligar o seu código com o componente propriamente dito. Todos os controladores físicos têm uma "imagem" que modela as suas funcionalidades. O componente STEAM não foge à regra, estando a funcionalidade do sistema físico descrita num frame *controlador_steam* com quem se relaciona através da relação *controlado_por*. Esta relação permite a herança de todos os métodos definidos no controlador, permitindo assim a sua utilização.

```
RELATION CONTROLADO_POR
is-a: relation
type: intransitive
```

```
inherits: inclusion heranca_controlado_por
inverse_relation: controla
```

Os atributos do frame steam *rota_para_secção_n*, contêm a rota necessária para que uma palete, que esteja parada na entrada de materiais siga até uma das referidas secções. Assume-se, nesta descrição, que as paletes apenas podem provir deste ponto. As rotas são descritas de acordo com os possíveis pontos de paragem ou de encaminhamento. Não cabe, nesta altura da apresentação do trabalho, fazer uma descrição exaustiva do modo como se obtêm as rotas, uma vez que é um ponto directamente relacionado com a implementação do sistema de gestão do subsistema de transporte da NovaFlex.

A relação *controlado_por* é extremamente importante para a criação de modelos de componentes que sejam controláveis. Dado que o mecanismo de herança não vai ser igual em todos eles, inclui-se um slot de metaconhecimento *heranca_controlado_por* que contém os slots a herdar.

A saída de produtos da célula é feita através do agente de saída que desempenha o papel de saída-por-palete e é realizado pelo STEAM, através da sua secção 1.

```
FRAME SAÍDA-POR-PALETE
is-a: SAÍDA
ref-saída:
secção:
paletes:
atributos-principais: ref-saída, paletes, secção, poe_parte
atributos-componentes: liberta-palete
method poe_parte: poe_parte_fn
```

Os atributos são semelhantes aos da classe ENTRADA-POR-PALETE, pelo que não se fazem mais comentários.

```
FRAME TAPETE_SAÍDA
is-a: papel_agente
relation desempenha:SAÍDA-POR-PALETE
relation realizado-por:STEAM

FRAME SAÍDA_TAPETE
instance-of: TAPETES_SAÍDA
ref-saída:
secção:1
paletes: demon if_needed dem_inspal_fn
method poe_parte: poe_parte_fn
method liberta_palete: liberta_palete_fn(secção)
```

No caso particular desta célula em que a entrada e a saída da palete, onde é realizada a montagem, se faz pelo mesmo ponto, o atributo *ref_saída* não é necessário; uma vez que podem existir situações onde os dois pontos (entrada e saída) sejam distintos, o atributo torna-se indispensável para permitir ao agente executor saber qual o ponto onde terá de colocar o produto acabado de montar. Bastava, por exemplo, que após a montagem o produto fosse colocado noutra palete.

O método *poe_parte*, herdado da classe papéis de saída, destina-se a colocar o produto na saída. Neste caso particular, corresponde a actuar o STEAM, através do método liberta_palete por forma a que o subsistema conduza a palete até ao ponto de saída de materiais.

O agente executor robô-montagem vai ser realizado por um robô SCARA a desempenhar o papel de agente de montagem.

```
FRAME AG-MONTAGEM
is-a: agente
tools_domain: grippers, screwdriver
aux_res_domain: buffers
armazém_ferramentas: arm_fer_1, arm_fer_2, arm_fer3, arm_fer4
ref_base:
local_montagem:  posicionador_1
ferramentas-disponíveis: gr1, gr2, gr3, gr4
ferramenta_corrente: gr1
troca_ferramenta: stroca1
```

```
atributos-principais: ref_base, armazém_ferramentas, fix,
                       ferramentas-disponíveis,
                       ferramenta_corrente, agarrar, largar,
                       trocar_ferramenta
atributos-componentes: controlado_por, working_area, load, dof,
                        Current_position

method agarrar: agarra_fn(point)
method largar: largar_fn(point)
method trocar_ferramenta: troca_fn(nova_ferramenta)
method obter_gabarit: obter_gabarit_fn()
```

As características relevantes da montagem estão definidas no frame AG-MONTAGEM.

Local_de_montagem indica uma instância de um objecto do tipo fixador. Sempre que não existir palete com gabarit de montagem no posicionador definido neste atributo, o que é conseguido por leitura do atributo *holds* do frame posicionador_1, desencadeia-se um método que "pede" ao STEAM para lhe enviar uma palete, tal como foi descrito para a entrada de materiais.

Os métodos *agarrar* e *largar* descrevem a funcionalidade principal de uma célula de montagem, agarrar um material numa dada posição e largar esse mesmo material também numa dada posição. Estes métodos estarão relacionados com as operações fornecidas pelo componente que realiza a operação de montagem.

Armazém_ferramentas contém uma lista de instâncias de objectos do tipo suporte_garras, definida da seguinte forma:

```
FRAME ARM_FER_1
instance-of: SUPORTE-GARRAS
relation controlado_por: controlador_robô
herança_controlado_por: input
garra: gr1
sensor_presença: s_binário_armfer1
existe_garra: demon if_read testa_garra_fn
```

Sensor_presença na frame ARM_FER_1 contém um instância de um sensor. Os sensores são modelados por forma a indicarem qual o controlador de que dependem e o número da entrada digital a que estão ligados. Desta forma, garante-se que cada sensor encapsula a informação referente à porta de entrada a que estão ligados. De notar que um controlador, por exemplo, de robô pode conter um elevado número de portas e que seria menos estruturado armazenar toda a informação referente ao uso das portas no modelo do controlador do robô, sendo preferível que sejam os utilizadores das portas lógicas a reconhecerem o seu uso.

Ferramenta_corrente contém o identificador de uma instância ferramenta, que é definida da seguinte forma:

```
FRAME GR1
instance-of: GARRAS
relation controlado_por: controlador_robô
heranca_controlado_por: input, output
pêso: 400
accionamento: pneumático
sensor_aberta: s_binário_gr1_1
sensor_fechada:s_binário_gr1_2
saída_abrir: 2
saída_fechar:3
method abrir: largar_fn()
method fechar: largar_fn()
```

Saída_abrir e *saída_fechar* identificam o número do porto de saída, neste caso no controlador do robô que deve ser accionado para abrir e fechar a garra, respectivamente. São utilizadas pelo código dos métodos abrir e fechar.

```
FRAME ROBÔ-MONTAGEM
is-a: papel_agente
relation desempenha:AG-MONTAGEM
relation realizado-por:ROBOT-BOSCH-SCARA
```

```
FRAME ROBOT-BOSCH-SCARA
instance-of: SCARA
dof: 4
desl_eixo_z: 400
base_coordinate_system:
applications: assembly, gluing, ..
working_area:800
load: 10
repeatability:0.025
...
current_position:
relation controlado_por: controlador_robô
heranca_controlado_por: input, output, move_lin, move_circ,
                        move_eixo, velocidade, aceleração
```

O modelo do controlador físico do robô terá de virtualizar as funcionalidades disponibilizadas pela "imagem" computacional do controlador físico, acedida através de um servidor. O frame *controlador_robô* não pode descrever directamente o componente físico, porque não tem forma de o aceder directamente. Uma vez que o acesso está estabelecido via servidor (ver discussão anterior), o frame tem de descrever apenas a funcionalidade implementada na imagem. Com uma "imagem" completa garante-se uma descrição completa do componente físico.

```
FRAME CONTROLADOR-RS82
instance-of: CONTROLADOR-ROBÔ
rpc_id: 2001
velocidade: demon if_write alterar_velocidade(value)
aceleração: demon if_write alterar_aceleração(value)
posição: demon if_write movimentar(value)
tipo_movimento: eixo
saída_digital_1:false demon if_write alterar_saída(value)
...
saída_digital_n:false demon if_write alterar_saída(value)
entrada_digital_1: demon if_read consultar_entrada(1)
...
entrada_digital_n: demon if_read consultar_entrada(n)
method ligar_controlador: ligar_fn()
method desligar_controlador: desligar_fn()
method output: output_fn(saída)
```

Rpc_id indica o número de RPC onde está ligado o servidor modelado pelo frame. Os demónios associados às variáveis velocidade, aceleração, posição, entrada e saída digitais fazem a ligação com o servidor. Existem 2 formas de movimentar o robô: (1) por alteração do *slot posição*, sendo o tipo de movimento aquele que estiver em *tipo_movimento* ou (2) por activação de um dos métodos *move_eixo*, *move_linear* ou *move_circular*.

4.4. Rede de Petri na Modelação Dinâmica e Síntese do Controlo de Alto Nível

A descrição deste ponto começa pela apresentação da arquitectura geral do sistema necessário para suportar a modelação dinâmica e a síntese de controlo de alto nível. Pretende-se que o leitor fique com uma perspectiva global do sistema que foi criado. A arquitectura criada nos pontos anteriores deve ser encarada como uma plataforma base para a construção do controlo global/sistema de supervisão - que dependerá das aplicações.

Depois da definição desta infraestrutura de execução vai-se agora ilustrar, como se podem sintetizar programas de controlo sobre essa plataforma, a partir do modelo de RdP.

4.4.1. Descrição global

Na figura 4.16 mostra-se a arquitectura geral do sistema. Tudo começa com a descrição do comportamento de um dado sistema (célula) através de uma RdP. Esta RdP é construída de acordo com determinadas regras que

possibilitarão a geração automática de um programa que virtualiza o comportamento do sistema, expresso na rede.

A infraestrutura desenvolvida pode ser utilizada por diversos tipos de clientes. A forma como esses clientes irão ser descritos terá de ser através de uma linguagem de programação, usando um dos dois paradigmas: POO ou Frames. Neste caso, e dado que se usaram Frames (Golog), usa-se Prolog.

Para que a ligação entre os dois mundos (RdP versus ambiente computacional) se estabeleça decidiu-se transformar a semântica presente na RdP numa descrição Prolog. O programa deve captar completamente a semântica da rede (mesmo tipo de comportamento), isto é, as acções a desencadear devem ser as mesmas que as descritas na rede e além disso devem ocorrer segundo a mesma ordem, independentemente de serem sequenciais ou paralelas.

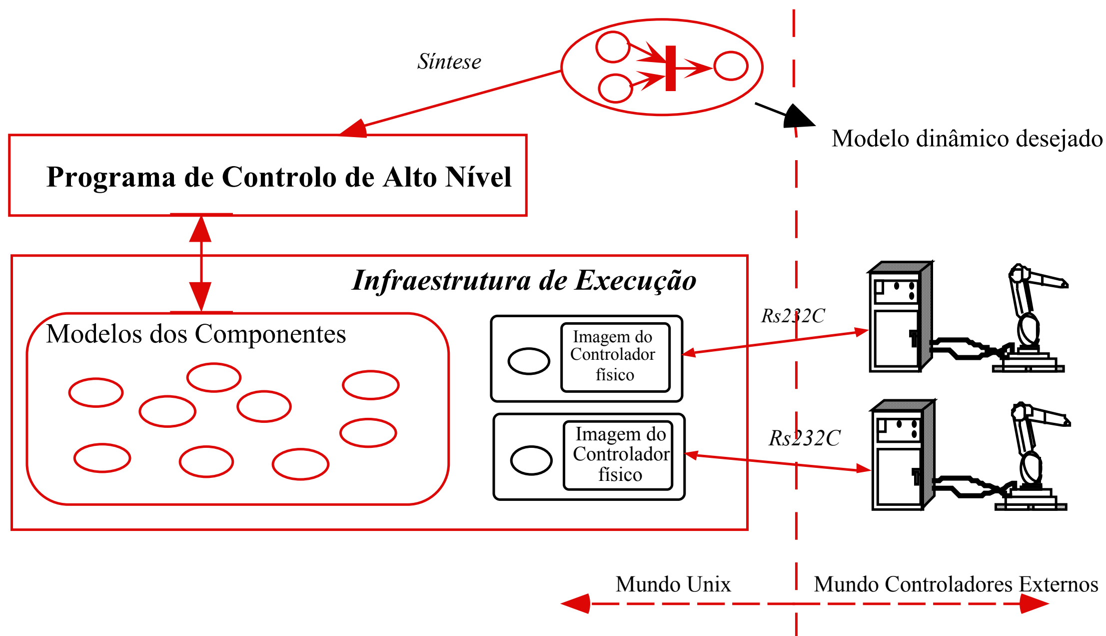


Figura 4.16 - Arquitectura Geral Proposta

Para que a ligação se faça há, então, que desenvolver um programa que, a partir de uma estrutura onde se define a RdP, gere um programa que tenha o mesmo comportamento da referida rede. Este programa não tem obrigatoriamente de ser desenvolvido na mesma linguagem do programa gerado.

O programa gerado utilizará representações de todos os componentes envolvidos na rede, que serão extraídas da biblioteca de modelos de componentes. A cada rede ficará associado um determinado conjunto de modelos que representam o conjunto de componentes reais modelados. A existência de várias redes provoca também a existência de várias instâncias de modelos.

4.4.2. Síntese da RdP

Para que a execução da RdP provoque a reacção dos componentes físicos cujo comportamento está a modelar é necessário que seja construída de acordo com determinadas regras [65].

Como se sabe, numa RdP, os lugares representam acções, que poderão ser ou não implementáveis por um dado controlador físico. O aparecimento de uma marca num lugar, durante a execução da rede, a que corresponde uma determinada acção implementada por um controlador físico, provocará a correspondente acção deste último.

Os lugares poderão modelar acções externas ou internas. Os lugares com acções externas representam estados, cujas acções são executáveis num ambiente computacional distinto daquele em que ocorrem (nos componentes físicos), enquanto que os lugares de acções internas representam estados não dependentes de componentes físicos.

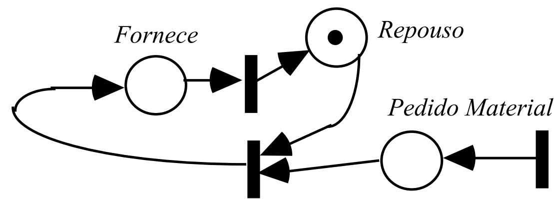


Figura 4.17 - RdP para Descrever Funcionamento do Alimentador Gravítico na Célula SONY

Com a RdP da figura 4.17 descreve-se o comportamento de um alimentador gravítico. O fornecimento de material ocorre quando a marca está no lugar *Fornece*, que é nitidamente um lugar de acção externa, enquanto que *Repouso* e *Pedido Material* são lugares de acção interna. Para que ocorra fornecimento é necessário que o controlador que controla a electro-válvula que empurra o material seja actuado. Deste facto deriva que a acção externa associada ao lugar *Fornece* será realizada pelo controlador da electro-válvula, devendo, por esta razão, ser actuado no momento em que o lugar receber uma marca.

A funcionalidade dos componentes está descrita, na sua representação em memória (modelo em frame/objecto), através de métodos. No caso do alimentador gravítico, o método *fornece* faz parte da definição do frame *alimentador_gravítico*, para descrever a acção de fornecer um material. Parece então natural associar o aparecimento de uma marca no lugar *Fornece*, à activação do método *fornece* da referida frame. Desta forma, sempre que uma marca atinge um lugar de acção externa activa-se o método que modela essa mesma acção.

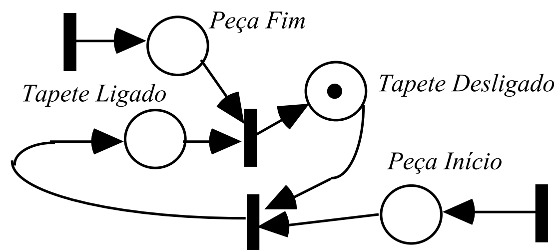


Figura 4.18 - RdP para Descrever Funcionamento de um Tapete

Será então necessário que os lugares de acções externas indiquem, de alguma forma, o nome do modelo do componente que realiza a acção e o nome dessa acção que é implementada por um método.

As acções ficam perfeitamente definidas com o disparo dos métodos, mas levanta-se a dúvida quanto às condições que são um elemento importante na modelação de acções de controlo. Uma forma de resolver o problema das condições seria através da criação de uma semântica especial para os lugares que estivessem associados a condições.

Suponha-se, por exemplo, que se pretende modelar o controlo de movimento de um tapete com 2 sensores de presença: um no início, e o outro no final. Quando é colocado um objecto no início do tapete actua-se o sensor e o tapete é posto em movimento; quando a peça atinge o final do tapete, o sensor é actuado e o tapete deve ser desligado. Na figura 4.18 mostra-se uma RdP para modelar o comportamento do tapete.

O problema com os lugares *Peça Início* e *Peça Fim* é que não representam acções mas antes condições. Qualquer um dos lugares são estados mas estes dois têm uma semântica diferente já que são estados condicionados por condições externas e, como tal, o aparecimento de marca é dependente da condição ser

verdadeira. Neste caso, o aparecimento de marca em *Peça Início* ou *Peça Fim* será dependente do facto do sensor de início ou de fim, respectivamente, estarem actuados. Com esta interpretação obriga-se a que o executor de RdP considere a existência de 3 tipos de lugares: (1) com acções externas, (2) internos e (3) dependentes de condições externas. Durante a execução, os lugares do tipo 3 teriam de, em cada iteração, ser analisados para verificar se geraram marca.

Uma outra interpretação poderia ser feita, através de uma valorização semântica da transição, atribuindo a cada transição um conjunto de predicados que deveriam ser satisfeitos para que a transição ocorresse. Neste caso está-se perante uma rede interpretada. O modelo seria então descrito da forma representada com a figura 4.19.

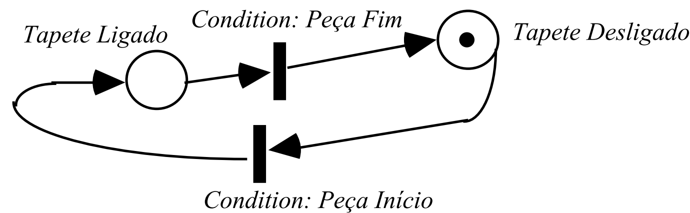


Figura 4.19 - RdP para Descrever Funcionamento de um Tapete - PTN

Não é grande a diferença de implementação entre uma PTN e uma rede normal que considere a existência de lugares dependentes de condições externas. Em ambos os casos, após se obter a habilitação de transição, há que verificar se a condição é verdadeira.

É importante recordar que o objectivo do trabalho não é a criação de um gerador automático de RdP a partir dos planos de produção. Neste trabalho, considera-se que a RdP é construída à mão. Pretende-se que a edição automática de RdP venha a ocorrer numa fase posterior de trabalho. Desta forma, assume-se que o executor da RdP recebe um conjunto de parâmetros em que se descrevem as características da rede. Nesta fase, essas características encontram-se num ficheiro que será descrito posteriormente.

Assim sendo e partindo do princípio que o componente alimentador gravítico está modelado da seguinte forma:

```
Frame ALGRAV1
instance-of: alimentador_gravitico
nome_produto: peca_grande
relation controlado_por: controlador_sony
numero_produtos: 2
sensor_extendido: s_binário_algrav1
saída_fornecer: 2
fornece: method fornece_fn
```

A RdP da figura 4.17 deveria ser alterada por forma a que o nome do lugar com acção externa mudasse para ***algrav1_fornece*** em vez de *Fornece*. Quer então dizer que o nome dos lugares com acções externas devem conter o nome do modelo mais o nome do método, separados por "underscore". A sintaxe de definição do nome do lugar será então:

<nome do frame>'_'<nome do método>

No caso das redes interpretadas segue-se a mesma sintaxe para a definição dos nomes das condições.

Suponha-se agora que se pretende modelar o ciclo de chegada de paletes com "*gabarits*" à célula 1 do FAS, na NovaFlex.

Na RdP da figura 4.20, irá ser estabelecida uma ligação com 2 entidades que modelam conceitos importantes: subsistema de transporte - STEAM, já previamente modelado e a célula 1 do FAS - célula_1,

também já modelada. Existem apenas lugares de acção que irão provocar a activação dos métodos presentes: *inserepalete*, *montar* e *libertapalete*. Duas das transições são condicionadas pelo resultado devolvido pelos métodos.

Implementou-se um programa em Prolog, estendido com um sistema de frames - Golog, que recebe a descrição da rede num ficheiro e produz código, cujo comportamento segue o descrito na RdP.

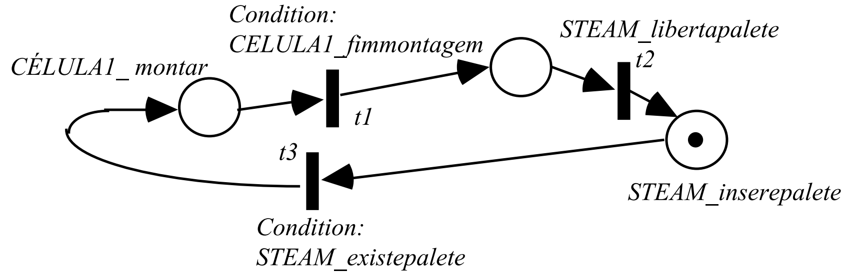


Figura 4.20 - RdP para Descrever Célula 1 do FAS

O ficheiro utilizado para a rede da figura 4.20 contém a seguinte estrutura:

< descrição topológica da rede>.

<lista dos nomes de lugares>.

<atribuição de condições às transições>.

O ficheiro utilizado para gerar o código de controlo descrito na figura 4.20 é:

```
/* Descrição da topologia */
[ [ [CÉLULA1_montar], t1, [STEAM_libertapalete] ],
  [ [STEAM_libertapalete], t2, [STEAM_inserepalete] ],
  [ [STEAM_inserepalete],  t3, [CELULA1_montar] ] ].

/* Nomes dos lugares */
[CÉLULA1_montar, STEAM_libertapalete, STEAM_inserepalete].

/* Atribuição de  condições às transições */
[ [ t1, [ CÉLULA1_fimmontagem ] ]
  [ t2, [ nil ] ]
  [ t3, [ STEAM_existepalete ] ]  ].
```

De seguida vai-se apresentar o programa gerado pelo gerador, subdividindo-o por secções. A primeira secção define os lugares; cada lugar é definido por uma frame, cujo atributo principal, slot marca, contém a marca do lugar. A consulta à base de conhecimento que contém os modelos dos componentes envolvidos é também feita durante esta fase.

```
:- consult('modelos.pl').

/* DEFINICAO FRAMES DOS LOCAIS */
:-
    new_frame(locais), new_slot(locais, marca).
:-
    new_frame(CÉLULA1_montar),  new_slot(CÉLULA1_montar, isa, locais),
    new_slot(CÉLULA1_montar, marca, 0).
:-
    new_frame(STEAM_libertapalete), new_slot(STEAM_libertapalete, isa, locais),
    new_slot(STEAM_libertapalete, marca, 0).
:-
    new_frame(STEAM_inserepalete),  new_slot(STEAM_inserepalete, isa, locais),
    new_slot(STEAM_inserepalete, marca, 1).
```

Depois são definidas as transições. Para que uma transição tenha sucesso é necessário verificar as suas condições de habilitação. Quando isto acontece, os lugares de entrada são actualizados e a transição ocorre, actualizando-se a saída e a correspondente activação do método.

```
t1 :-
/* GERA INPUT */
    get_value(CÉLULA1_montar, marca, X0),
    X0 > 0,
    call_method(CÉLULA1, fimmontagem, [true]),
/* ACTUALIZA TRANSICOES */
    get_value(CÉLULA1_montar, marca, X0),
    NVal0 is X0 - 1,
    new_value(CÉLULA1_montar, marca, NVal0),
/* GERA OUT */
    call_method(STEAM, libertapalete , [true]),
    get_value(STEAM_libertapalete , marca, VOal0), NVOal0 is VOal0 + 1,
    new_value(STEAM_libertapalete , marca, NVOal0),
    write('TRANSICAO '), write(t1), write(' ACTUADA '), nl, nl.
```

No caso, por exemplo da transição 1, verifica-se se existe marca no lugar CÉLULA1_montar e se a condição de transição é verdadeira (*call-method(CÉLULA1, fimmontagem, [true])*). O disparo da transição provoca a actualização das marcas dos lugares a jusante e activa o disparo do método: *call-method(STEAM, libertapalete, [true])*. O código deste método, encapsulado no frame STEAM, enviará uma mensagem ao servidor do subsistema de transporte, que reagirá desencadeando as acções necessárias para que a palete saia da secção.

O código respeitante às outras transições é semelhante não sendo por isso apresentado. O programa principal consiste num ciclo infinito que continuamente escolhe, de uma forma aleatória, o nome da transição que vai testar. Deve-se notar que não houve uma grande preocupação no sentido de garantir que o nome de cada transição tenha a mesma possibilidade de ser escolhido, porque se pretende que este seja um trabalho exploratório. De qualquer forma, por cada ciclo, todas as transições são escolhidas. A ordem como o são é que é feita aleatoriamente.

```
rep_run([]).
rep_run(Lista) :- length(Lista, Tam), Pos is ip(rand(Tam)),
    position(Pos, Lista, Tr), remove(Tr, Lista, RLista),
    call(Tr, Sucesso), !,
    fail==Sucesso,
    rep_run(RLista).

run :- repeat, rep_run([t1,t2,t3]).
```

4.5. Sumário/Experiência

Neste trabalho apresentou-se uma arquitectura que pretende fazer a ligação entre dois mundos de representação distintos: RdP e representação computacional. Começou-se por apresentar um conjunto de infra-estruturas "hardware", nas quais o autor contribuiu, durante as fases de desenho e implementação.

Para suportar a integração dos diferentes controladores envolvidos apresentou-se uma filosofia de integração baseada no paradigma servidor-cliente, tendo sido descritos alguns trabalhos que seguiram essa filosofia.

Para completar a infra-estrutura de execução mostraram-se alguns exemplos de modelação de componentes reais da NovaFlex. Estes modelos serão a forma como as aplicações de alto nível (supervisores, controladores de célula, ...) manipularão os conceitos físicos. Desenvolveram-se ainda alguns conceitos de programação que possibilitam que as linguagens de programação do paradigma objectos/frames sejam enriquecidas com um conjunto de funcionalidades presentes nos controladores reais, nomeadamente através dos aspectos de modelação dinâmica descritos no final do capítulo anterior.

Para completar o quadro mostrou-se um pequeno exemplo de utilização da infra-estrutura de execução criada, que realça a possibilidade de integrar uma representação não habitualmente presente nos controladores: Redes de Petri, com esta estrutura, ficando de alguma forma demonstrada a possibilidade de geração automática de controladores a partir de RdP.

A experiência adquirida ao longo deste trabalho mostrou-se muito interessante, não apenas a um nível pessoal, mas também para o grupo a que pertenço. A discussão deste tipo de ideias, nomeadamente os aspectos de integração de sistemas existentes, faz muito sentido para os requisitos da indústria nacional. É importante a consciencialização, por parte das universidades, de que a integração não é um trabalho menor mas requer também um determinado esforço conceptual. Serviu este trabalho também para alertar para a grande quantidade de trabalho que aparece, muitas vezes escondida, por detrás das arquitecturas proprietárias que se têm de integrar.

O estudo desenvolvido nos aspectos de modelação foi também importante já que, o mundo industrial da automação está tradicionalmente habituado a um conjunto restrito de tipos para modelar as suas necessidades. À medida que a complexidade dos sistemas vai aumentando exigem-se programas de controlo mais complexos e, como tal, surge a necessidade de estratégias que possibilitem o domínio do problema de uma forma mais simples e natural.

5. Conclusões

O trabalho consistiu na apresentação de uma metodologia para a criação de uma plataforma de suporte à integração de supervisores inteligentes. Para isso dividiu-se o trabalho em duas partes. Na primeira parte foi feito um esforço no sentido de realçar os aspectos importantes da modelação de sistemas flexíveis, enquanto que na segunda parte se apresentaram os aspectos directamente relacionados com a concepção e implementação da plataforma de suporte.

Nos aspectos directamente relacionados com a modelação de células apresentou-se uma proposta para a criação de células genéricas, baseada no conceito de que cada célula tem uma entrada, uma saída e um agente executor. Cada um destes conceitos é implementado por um agente, realçando-se o facto destes agentes serem componentes a desempenharem papéis.

Dada a importância que a organização de informação tem para qualquer sistema informático, apresentou-se uma taxonomia de componentes, organizada de acordo com a função que os componentes desempenham e uma taxonomia de papéis, organizada de acordo com o local onde o papel se vai realizar (entrada, saída e agente).

Os componentes mais relevantes de um sistema de manufactura, nós da taxonomia de componentes, foram descritos utilizando dois modelos adequados para a modelar este tipo de sistemas: objectos e "frames", sendo realçados, numa primeira abordagem, os seus aspectos estruturais. Os aspectos de modelação do comportamento mereceram uma especial atenção, tendo-se apresentado alguns modelos de controladores e a forma de ligar esses modelos ao controlador físico.

Modelaram-se também estruturas complexas (células), tendo os exemplos sido baseados em dois sistemas reais: sistema NovaFlex e célula Sony.

A utilização dos dois paradigmas - objectos e "*frames*", mostrou que ambos são adequados, não se pode dizer que satisfazem integralmente os requisitos da modelação de sistemas flexíveis de manufactura. Comparando os dois modelos concluiu-se que apresentam vantagens e inconvenientes.

Algum esforço de investigação deve ainda ser colocado no sentido de se desenvolver um paradigma de modelação que se adeque completamente aos sistemas flexíveis de manufactura.

Em termos da criação da plataforma de suporte, apresentou-se o trabalho desenvolvido na concepção e implementação de dois sistemas de manufactura. Deste trabalho importa realçar a dificuldade que representa instalar sistemas num país com firmas mal preparados tecnicamente e, quase sempre, com pouca informação para prestar.

Baseado na NovaFlex, apresentou-se uma proposta para a integração de controladores locais, através da construção de servidores, implementados de acordo com o paradigma cliente-servidor. Estes servidores permitiram a construção de imagens da funcionalidade dos controladores locais, no mundo do supervisor, permitindo assim uma mais fácil ligação dos modelos aos controladores físicos.

O subsistema de montagem, construído em torno do robô BOSCH, da NovaFlex foi modelado para permitir a validação dos conceitos apresentados ao longo da tese, ficando assim criada uma plataforma de suporte a sistemas de supervisão, com modelos dos componentes da célula e os respectivos servidores, que fazem a ligação aos controladores locais.

Para testar a plataforma de suporte desenvolveu-se um pequeno programa de controlo. Este programa foi criado a partir de uma rede de Petri. Dada a adequabilidade das redes de Petri para modelarem os aspectos dinâmicos apresentou-se ainda um método que permite a síntese de sistemas de supervisão, a partir deste tipo de redes.

Em termos de trabalho futuro pretende-se continuar o esforço de integração dos diferentes subsistemas da NovaFlex, de maneira a que num futuro breve se tenha uma plataforma completa. O aprofundamento da síntese de supervisores a partir de redes de Petri é outro ponto de direccionamento do trabalho.

6. Apêndice A - Fotografias da NovaFlex e Produto NovaClock

Figura A.1 - Fotografia da NovaFlex - Subsistema Multi-Robô FAS

Figura A.2 - Fotografia da NovaFlex - Subsistema FMS

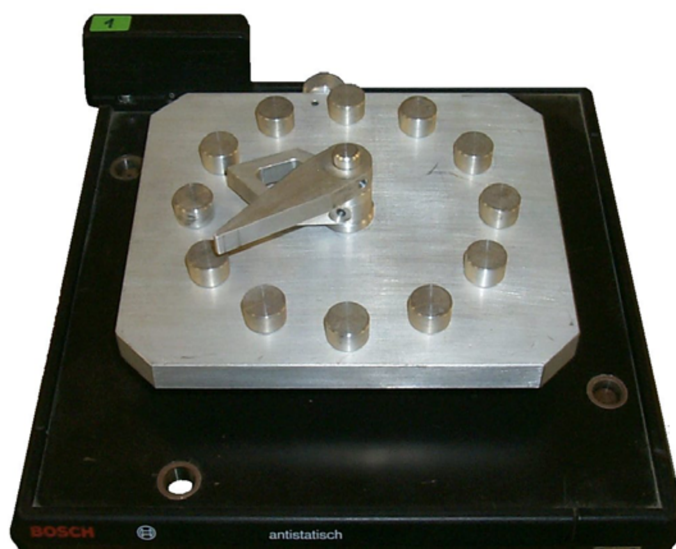


Figura A.3 - Produto NovaClock

7. Apêndice B - Fotografia da Célula SONY e Pêndulo de Cranfield

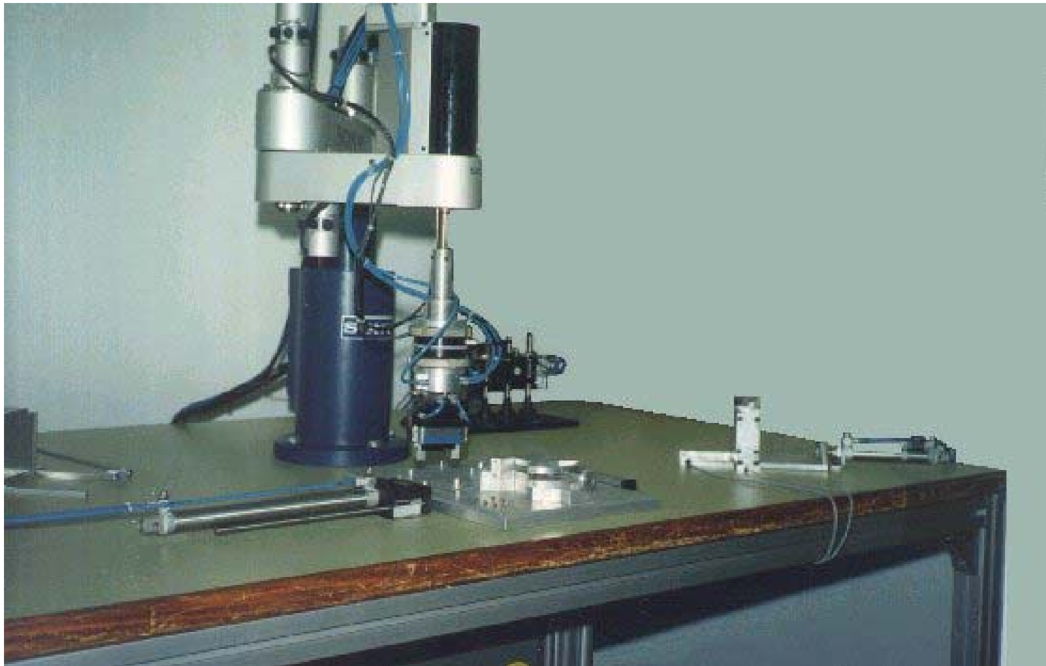


Figura B.1 - Fotografia da Célula SONY

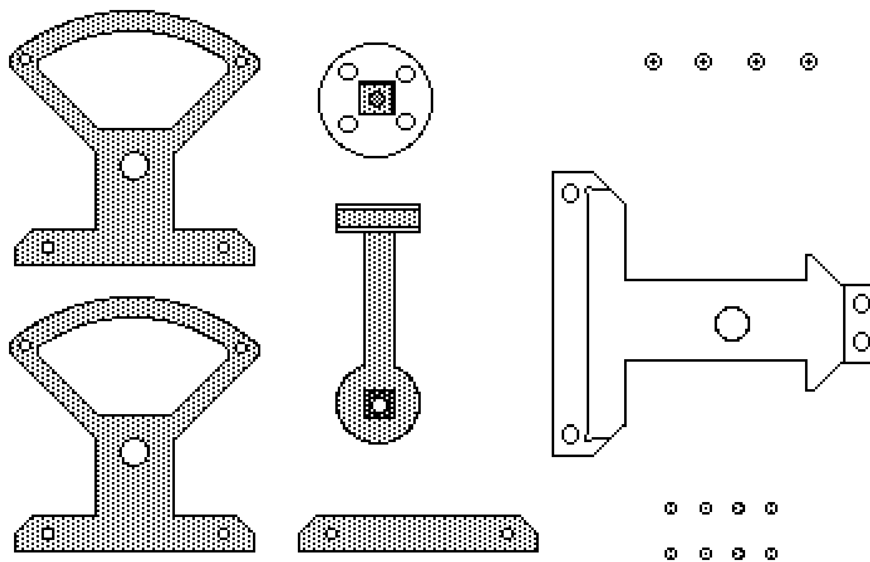


Figura B.2 - Fotografia do Pêndulo de Cranfield

8. Agradecimentos

É impossível chegar ao fim deste trabalho e não realçar as duas pessoas sem as quais e, por razões diferentes, este trabalho teria sido impossível: o meu orientador Prof. Camarinha de Matos e a Célia.

Toda a minha gratidão vai para o Prof. Camarinha de Matos que durante todo este tempo demonstrou para comigo uma atitude de grande interesse, não só no acompanhamento do trabalho, com constantes discussões, incentivos e as tão necessárias chamadas de atenção, mas acima de tudo pela amizade que demonstra, gerindo com um rigor notável a fronteira entre trabalho e amizade. Neste sentido, agradeço especialmente a sua constante disponibilidade para me receber, com prejuízo do seu tempo de descanso e de família, exprimindo, por isso, um voto de agradecimento à sua esposa que pacientemente me aturou durante as minhas visitas de trabalho.

Aos membros do grupo de Sistemas Robóticos e CIM: Hélder Pita, Luís Osório, Ricardo Rabelo e Luís Seabra Lopes, pelas discussões, incentivos e bom ambiente, que tornam o trabalho bastante mais fácil e agradável.

Ao Fernando Moura Pires e Luís Correia do Departamento de Informática, gostaria também de deixar um muito obrigado pela paciência que tiveram, ao ler alguns dos capítulos da tese, e as sugestões apresentadas. Aos meus colegas do DI gostaria também de agradecer, com especial destaque para a Geiza e o Pimentão.

Aos alunos de projectos de fim de curso que tive o privilégio de co-orientar, Sandra Gadanho, Nuno Chalmique Chagas, Florbela Tique Aires, Eduardo Braz e Luís Fernandes, um agradecimento especial pelo trabalho que desenvolveram, dando um contributo inestimável para a elaboração desta tese.

Gostaria ainda de agradecer ao Sr. Joaquim Martins que, além de me proporcionar um conjunto de visitas, bastante importantes, a várias unidades industriais, participou ainda activamente no desenho e implementação da NovaFlex.

Um agradecimento muito especial para o grande companheiro das noites longas, o meu amigo e colega João Carlos Silva, não só pelo trabalho desenvolvido enquanto aluno, que tive o privilégio de co-orientar, e discussões sobre o tema da tese, mas acima de tudo pelo espírito de paz e amizade que se sente ao falar com esta boa alma, felizmente pertencente ao grupo de Sistemas Robóticos e CIM.

Não posso também deixar de expressar um agradecimento especial à minha mãe que com todo o seu carinho e amizade, me tem ajudado a chegar até aqui.

Finalmente, um pensamento profundo para a minha estrela do norte, cujo brilho constante, tanto me tem ajudado e acompanhado. Por isso, gostaria de deixar bem expressa toda a minha gratidão, pela sua constante presença. Obrigado Célia.

9. Bibliografia

1. Barata, J. and L.M. Camarinha-Matos, *Development of a FMS/FAS System - The CRI's Pilot Unit,* in *Studies in Informatics and Control*1994, p. 231-239.

2. Camarinha-Matos, L.M. and F. Sastron. *Information Integration for CIM planning Tools*. in *CAPE'91 IFIP Conference on Computer Applications in Production and Engineering*. 1991. Bordeaux - France: ELSEVIER Publishers.

3. Camarinha-Matos, L.M. and L. Osório. *CIM Information Management System - An Express-based Integration Platform*. in *IFAC Workshop on CIM in Process and Manufacturing Industries*. 1992. Espoo - Finland: CHAPMAN & HALL Publishers.

4. Barata, J., L.M. Camarinha-Matos, and J.F.R. Chavarría, *Modelling, Dynamic Persistence and Active Images for Manufacturing Processes,* in *Studies in Informatics and Control*1994, p. 173-183.

5. Molina, A., *et al. Modelling Manufacturing Resources, Processes and Strategies to Support Concurrent Engineering*. in *First International Conference on CONCURRENT ENGINEERING RESEARCH AND APPLICATIONS*. 1994. Pittsburgh - USA:

6. Lieberherr, K. and C. Xiao, *Formal Foundations for Object-Oriented Data Modeling,* in *IEEE Transactions on Knowledge and Data Engineering*1993, p. 462-478.

7. Agesen, O., S. Frolund, and M.H. Olsen, *Persistent and Shared Objects in Beta,* Master, Aarhus University, 1989

8. Zdonik, S.B., *What Makes Object-Oriented Database Management Systems Different?,* in *Advances in Object-Oriented Database Systems,* A. Dogac, *et al.*, Editors. 1994, Springer-Verlag: Kusadasi. p. 3-26.

9. Blair, G., H. Bowman, and R. Lea, *Basic Concepts I,* in *Object-Oriented Languages Systems and Applications,* G. Blair, *et al.*, Editors. 1991, PITMAN: London - UK. p. 24-42.

10. Khoshafian, S. and R. Abnous, *OBJECT ORIENTATION - Concepts, Languages, Databases, User Interfaces*. 1990, New York - USA: WILEY. 433.

11. Rosen, J.P., *What Orientation Should Ada Objects Take?,* in *Communications of the ACM*1992, p. 71-76.

12. Booch, G., *Object Oriented Design with application*. Series in ADA and Software Engineering, 1991, Redwood - USA: The Benjamin/Cummings. 520.

13. Meyer, B., *Object-Oriented Software Construction*. Series in Computer Science, ed. C.A.R. Hoare. 1988, Prentice Hall International. 534.

14. McIlroy, M.D. *Mass-produced Software Components*. in *Software Engineering Concepts and Techniques (1968 NATO Conf. on Software Engineering)*. 1976.

15. Cardelli, L. and P. Wegner, *On Understanding, Types, Data Abstraction, and Polymorphism,* in *Computing Surveys*1985, p. 471-522.

16. Defense, D.o., *Reference Manual for the Ada Programming Language* - 1983, Department of Defense,

17. Schonberg, E., M. Gerhardt, and C. Hayden, *A Technical Tour of Ada,* in *Communications of the ACM*1992, p. 43-52.

18. Minsky, M., *A Framework for Representing Knowledge,* in *The Psychology of Computer Vision,* P. Winston, Editor. 1975, McGraw-Hill: New York - USA. p. 211-277.

19. Hayes, P.J., *The Logic of Frames,* in *Readings in Artificial Intelligence,* R.J. Brachman and H.J. Levesque, Editors. 1979, Tioga Publishing: Palo Alto - USA. p. 287-295.

20. Green, C. *Application of Theorem Proving to Problem Solving*. in *1st International Joint Conference on Artificial Intelligence*. 1969. Washington - USA: Mitre Corp.

21. Fikes, R. and T. Kehler, *The Role of Frame-Based Representation In reasoning,* in *Communications of the ACM*1985, p. 904-920.

22. Camarinha-Matos, L.M. *Programação Orientada por Objectos no Controlo de Estações Robóticas*. in *5º Congresso Português de Informática*. 1988. Lisboa - Portugal:

23. Camarinha-Matos, L.M., *Sistema de Programação e Controle de Estações Robóticas,* Dissertação de Doutoramento, Universidade Nova de Lisboa - DI, 1989

24. Camarinha-Matos, L.M. *Knowledge Architecture for Flexible Programming of Robotic Cells*. in *20th ISIR - Int. Symp. on Industrial Robots*. 1989. Tokyo - Japan:

25. Camarinha-Matos, L.M. and H. Pinheiro-Pita. *Interactive Planning of Motion and Assembly Operations*. in *IEEE International Workshop on Intelligent Motion Control*. 1990. Istanbul - Turkey: IEEE.

26. Pimentão, J.P., R. Amador, and J.M. Pires. *Introdução à Representação de Conhecimento Através de Frames*. in *2as Jornadas Nacionais de Projecto, Planeamento e Produção Assistidos por Computador*. 1989. Lisboa - Portugal:

27. IntelliCorp, *KEE™ Software Development System User's Manual* - 1986, IntelliCorp, Manual

28. Knowledge-Craft, *Knowledge Craft Reference Manual* - 1987, Carnegie Group, Manual

29. Petri, C.A., *Kommunikation mit Automaten,* PhD dissertation, Univ. Bonn, 1962

30. Nazareth, D.L., *Investigating the Applicability of Petri Nets for Rule Based System Verification,* in *IEEE Transactions on Knowledge and Data Engineering*1993, p. 402-415.

31. Holt, A., *et al.*, *Final Report of the Information System Theory Project* - 1968, Rome Air Development Center, Griffiths Air Force Base, Technical Report

32. Dennis, J. *Modular, Assynchronous Control Structures for a High Performance Processor*. in *Rec. Project MAC Conf. Concurrent Systems and Parallel Computation*. 1970. New York - USA:

33. Hack, M., *Decidability Questions for Petri Nets,* Ph.D. Dissertation, MIT - Dep Elec Eng, 1975

34. Patil, S., *Coordintion of Asynchronous Events,* Ph.D. Dissertation, MIT - Dep Elec Eng, 1970

35. Malo-Tamayo, A., *Use of Petri Nets in Assembly Definition,* in *IEEE System Man & Cybernetics*1994,

36. D'Souza, L.A. and S.K. Khator, *A Survey of Petri Net Applications in Modeling Controls for Automated Manufacturing Systems,* in *Computers in Industry*1994, p. 5-16.

37. Lin, C., *et al.*, *Logical Inference of Horn Clauses in Petri Net Models,* in *IEEE Transactions on Knowledge and Data Engineering*1993, p. 416-425.

38. David, R. *Modeling of Dynamic Systems By Petri Nets*. in *ECC 91 European Control Conference*. 1991. Grenoble - France:

39. Jensen, K., *High Level Petri Nets: Theory and Applications,* in *A High Level Language for Systems Design and Analysis*. 1991, Springer Verlag: p. 44-119.

40. Jensen, K., *Coloured Petri Nets. Petri Nets: Central Models and their Properties,* in *Advances in Petri Nets*1986, p. 248-299.

41. Zhou, M. and F. DiCesare, *Petri Net Synthesis for Discrete Event Control of Manufacturing Systems*. Discrete Event Dynamic Systems, ed. Y.-C. Ho. Vol. 1. 1993, Kluwer Academic Publishers. 233.

42. Colombo, A.W., J. Martinez, and R. Carelli. *Formal Validation of Complex Production Systems Using Coloured Petri Nets*. in *ICRA'94 - International Conference on Robotics and Automation*. 1994. Califórnia - USA:

43. Sifakis, J., *Use of Petri Nets for Performance Evaluation,* in *Measuring, Model and Evaluating Comp Systems,* H. Beilner and E. Gelenbe, Editors. 1977, North-Holland Publishers: p. 75-93.

44. Hillion, H.P. and J.M. Proth, *Performance Evaluation of Job-Shop System using Timed Event-Graphs,* in *IEEE Transactions on Automatic Control*1989,

45. David, R. and H. Alla. *Continuous Petri Nets*. in *8th European Workshop on Appli. and Theory of Petri Nets*. 1987. Zaragoza - Spain:

46. Lopes, L.S., *GOLOG - Um gestor de Objectos em Prolog* - 1993, DEE - Universidade Nova de Lisboa,

47. Fu, K.S., R.C. Gonzalez, and C.S.G. Lee, *ROBOTICS - Control, Sensing, Vision and Intelligence*. CAD/CAM robotics and computer, ed. S. Rao. 1987, Singapore: McGraw-Hill. 580.

48. Camarinha-Matos, L.M., *et al.*, *Towards o Taxonomy of CIM Activities,* in *International Journal of Computer Integrated Manufacturing*1995,

49. Camarinha-Matos, L.M. and H. Pinheiro-Pita. *Intelligent CASE for CIM*. in *ETFA'92 - IEEE International Workshop on Emerging Technologies and Factory Automation*. 1992. Melbourne - Australia:

50. Camarinha-Matos, L.M., H. Pinheiro-Pita, and L. Osório. *Hybrid Programming Paradigms in CIM-CASE*. in *IFIP/IFAC Working Conference on Knowledge Based Hybrid Systems in Engineering and Manufacturing*. 1993. Budapest - Hungary: North-Holland.

51. Camarinha-Matos, L.M. and H. Pinheiro-Pita. *Interactive Planning in CIM-CASE*. in *IEEE Int. Conference on Robotics and Automation*. 1993. Atlanta - USA:

52. Camarinha-Matos, L.M., *et al.*, *Interactive Assembly Task Planning and Execution Supervision,* in *Studies in Informatics and Control*1994, p. 185-193.

53. Zhon, M. and F. Dicesare, *Petri Net Synthesis for Discrete Event Control of Manufacturing Systems*. 1993, Kluwer Academic Publishers.

54. ISO92, *ISO CD 10303-1 Product Data Representation and Exchange - Part 1: Overview and Fundamental Principles* - 1992, ISO TC184/SC4,

55. Arbouy, S., *et al.*, *STEP - Concepts Fondamentaux*. 1994, AFNOR.

56. Silva, J.C.M., *Integração de uma Célula Robótica em Ambiente UNIX* - 1993, DI - Universidade Nova de Lisboa, Relatório Projecto

57. Braz, E. and L. Fernandes, *Gestão e Monitorização do Armazém Automático* - 1994, DEE - Universidade Nova de Lisboa, Relatório Projecto

58. Aires, F., *BOSCHSYS - Sistema de Integração de um Controlador BOSCH em ambiente UNIX* - 1994, DI - Universidade Nova de Lisboa, Relatório Projecto

59. Gadanho, S.C. and N.C. Chagas, *STEAM - Sistema de Transporte e Encaminhamento de Materiais* - 1994, DEE - Universidade Nova de Lisboa, Relatório de Projecto

60. Camarinha-Matos, L., L.S. Lopes, and J. Barata. *Execution Monitoring in Assembly with Learning Activities*. in *1994 IEEE International Conference on Robotics and Automation*. 1994. San Diego - USA: IEEE.

61. BOSCH, *BAPS Programming Language* - 1992, ROBERT BOSCH, Manual de Programação

62. BOSCH, *Programming the ID80/E-SLS.* Manual BOSCH ID soft-PRO 80, 1989. .

63. Barata, M., *Tese de Doutoramento a publicar,* Phd, Universidade Nova de Lisboa, 1995

64. Steiner, J.G., B.C. Neuman, and J.I. Schiller. *Kerberos: An Authentication Service for Open Network Systems*. in *USENIX*. 1988. Dallas - USA:

65. Barata, J. and L.M. Camarinha-Matos. *Dynamic Behaviour Objects in Modelling Manufacturing Processes*. in *CAPE'95 - The Fifth International Conference on Computer Applications in Production and Engineering*. 1995. Beijing - China: