UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CAMPUS CURITIBA

GERÊNCIA DE PESQUISA E PÓS-GRADUAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA E INFORMÁTICA INDUSTRIAL - CPGEI

Roni Fabio Banaszewski

Paradigma Orientado a Notificações: Avanços e Comparações

DISSERTAÇÃO DE MESTRADO

CURITIBA

27 de Março de 2009

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial

---

DISSERTAÇÃO
apresentada a UTFPR
para obtenção do grau de

MESTRE EM CIÊNCIAS

por

RONI FABIO BANASZEWSKI

PARADIGMA ORIENTADO A NOTIFICAÇÕES: AVANÇOS E COMPARAÇÕES

Banca Examinadora:

Presidente e Orientador:
Prof. Dr. Cesar Augusto Tacla **UTFPR**
Co-Orientador:
Prof. Dr. Jean Marcelo Simão **UTFPR**
Examinadores:
Prof. Dr. Paulo Cézar Stadzisz **UTFPR**
Prof. Dr. Fabiano Silva **UFPR**
Prof. Dr. Fabrício Enembreck **PUCPR**

Curitiba, 27 março de 2009

RONI FABIO BANASZEWSKI

PARADIGMA ORIENTADO A NOTIFICAÇÕES: AVANÇOS E COMPARAÇÕES

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI) da Universidade Tecnológica Federal do Paraná (UTFPR), como parte dos requisitos para a obtenção do título de Mestre em Ciências. Área de Concentração: Informática Industrial.

Orientador: Prof. Dr. Cesar Augusto Tacla

Co-Orientador: Prof. Dr. Jean Marcelo Simão

Curitiba

2009

Ficha catalográfica elaborada pela Biblioteca da UTFPR – Campus Curitiba

B212p Banaszewski, Roni Fabio
Paradigma orientado a notificações: avanços e comparações / Roni Fabio Banaszewski. – 2009.
xix, 261 p. : il. ; 30 cm

Orientador: Cesar Augusto Tacla
Co-orientador: Jean Marcelo Simão
Dissertação (Mestrado) – Universidade Tecnológica Federal do Paraná. Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial. Área de Concentração: Informática Industrial, 2009
Bibliografia: p. 249-61

1. Paradigmas (Informática). 2. Paradigma orientado a notificações. 3. Engenharia de software. 4. Mecanismo de inferência por notificações. I. Tacla, Cesar Augusto, orient. II. Simão, Jean Marcelo, co-orient. III. Universidade Tecnológica Federal do Paraná. Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial. Área de Concentração em Informática Industrial. IV. Título.

CDD 621.3

AGRADECIMENTOS

Agradecimentos de ordem pessoal

Quero agradecer principalmente a Deus pela vida, pela saúde e pela perseverança concedida, tais elementos permitiram-me concluir este trabalho. Agradeço também a todas as pessoas que se fizeram presentes, que se preocuparam e me auxiliaram com atos ou pensamentos.

Agradeço especialmente a minha família, a qual merece caras palavras. Obrigado pelo apoio sentimental e material que foi a minha força para concluir mais esta etapa e que me impulsiona a seguir cada vez mais longe para a realização dos meus sonhos. Sei que cada um de vocês se orgulha por eu ter atingido uma etapa que nenhum outro de nós tinha atingido antes, mas este orgulho que sentem por mim converto em uma obrigação de a cada dia ser mais digno de nos representar.

Agradeço por tudo ao meu pai (Seu João) e a minha mãe (Dona Dionísia) especialmente pela vida e a educação concedida e aos meus irmãos pelo apoio e a amizade sempre verdadeira.

Agradecimentos de ordem profissional

Reconheço que agradecer é um ato muito difícil. Todos os que concluíram um trabalho de mestrado sabem que não o fizeram sozinhos. Os resultados destes trabalhos não se devem apenas ao seu autor, mas também aos esforços e conhecimento empregados por outras pessoas que contribuem de alguma forma para a obtenção dos resultados. Desta forma, reconheço que este trabalho não é só meu, mas também dos autores que li, dos professores com quem tive aulas na graduação e pós-graduação e principalmente dos meus orientadores.

Quero agradecer ao meu orientador Prof. Dr. Cesar Augusto Tacla pelos ensinamentos, conselhos de pesquisa e pelo tempo despendido nas leituras dos meus trabalhos. Também, agradeço pela confiança depositada em mim ao me conceder a oportunidade para realizar o mestrado e abrir perspectivas para um potencial doutoramento. Também agradeço ao Prof. Tacla por permitir que o Prof. Dr. Jean Marcelo Simão atuasse como co-orientador oficial neste trabalho de mestrado e exercesse orientações efetivas e conjuntas.

Na verdade, o Prof. Simão desejaria ter orientado oficialmente este trabalho, o qual se refere a uma extensão de sua tese de doutorado. Entretanto, devido a problemas de ordem

sistêmica relativos à sua integração ao programa de pós-graduação, este trabalho apenas pôde ser concretizado na prática com o consentimento do Prof. Tacla. Sem este consentimento, certamente este trabalho não teria nem como ser iniciado.

Neste âmbito, um agradecimento especial faço ao Prof. Dr. Jean Marcelo Simão, por todo o empenho, sabedoria, compreensão e antes de mais nada pela paciência, pelos ensinamentos, pelos direcionamentos na pesquisa e pelas horas despendidas na leitura dos meus trabalhos. Suas sugestões sempre competentes por meio de discussões e correções sempre bem-vindas fizeram com que este trabalho fosse concluído.

Outrossim, agradeço ao Prof. Dr. Paulo Cézar Stadzisz pelos ensinamentos em disciplinas de mestrado, pelos conselhos de pesquisa, pela participação nas bancas de qualificação e defesa final desta dissertação. Ademais, agradeço ainda ao Prof. Stadzisz por ter acompanhado e acreditado em meu trabalho durante o curso de mestrado.

Agradeço igualmente ao Prof. Dr. Fabiano Silva e ao Prof. Dr. Fabrício Enembreck por terem aceitado o convite para compor a banca de defesa final desta dissertação. Tenho a honra de contar com os senhores na composição desta banca.

Por fim, agradeço a todos os colegas que, direta ou indiretamente, contribuíram para a execução deste trabalho e também à CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) pelo apoio financeiro e à UTFPR/CPGEI/LSI (Laboratório de Sistemas Inteligentes) por todo o suporte oferecido durante o desenvolvimento deste trabalho.

Sinceramente,

Roni Fabio Banaszewski

SUMÁRIO

LISTA DE FIGURAS XII
LISTA DE TABELAS XVI
LISTA DE ABREVIATURAS E SIGLAS XVII
RESUMO XIX
ABSTRACT XX
1 INTRODUÇÃO 1
1.1 MOTIVAÇÃO 1
1.1.1 *Softwares* otimizados 4
1.1.2 *Softwares* complexos 4
1.1.3 *Softwares* paralelos e distribuídos 6
1.1.4 Facilidades de implementação 7
1.2 CONTEXTUALIZAÇÃO E OBJETIVO DO TRABALHO 8
1.2.1 As origens do Paradigma Orientado a Notificações – um meta-modelo de controle 8
1.2.2 O surgimento do Paradigma Orientado a Notificações 12
1.2.3 Objetivos 13
1.3 ORGANIZAÇÃO DO TRABALHO 13
2 FUNDAMENTAÇÃO TEÓRICA 15
2.1 PARADIGMA NAS CIÊNCIAS FÍSICAS 15
2.2 PARADIGMA NA CIÊNCIA DA COMPUTAÇÃO 18
2.3 MUDANÇAS DE PARADIGMAS 20
2.4 EVOLUÇÃO DOS PARADIGMAS DE PROGRAMAÇÃO POR MEIO DAS LINGUAGENS DE PROGRAMAÇÃO 24
2.4.1 Primeira geração (1940) 25
2.4.2 Segunda geração (1950) 26
2.4.3 Terceira geração (1954) 27
2.4.4 Quarta geração (1970) 32
2.4.5 Quinta geração (atualidade e futuro) 37
2.5 CLASSIFICAÇÃO DAS LINGUAGENS EM PARADIGMAS DE PROGRAMAÇÃO 38
2.6 CLASSIFICAÇÃO DOS ATUAIS PARADIGMAS DE PROGRAMAÇÃO 39
2.7 PARADIGMAS IMPERATIVOS 41

2.7.1 Paradigma Procedimental 41
2.7.2 Paradigma Orientado a Objetos 42
2.7.2.1 Visão geral 42
2.7.2.2 Programação Orientada a Objetos e Dirigida a Eventos 45
2.7.3 Paradigmas Emergentes 49
2.7.3.1 Paradigma Orientado a Agentes 49
2.7.3.2 Paradigma Orientado a Aspectos 51
2.7.3.3 Paradigma Orientado a Componentes 52
2.7.3.4 Relação dos Paradigmas Emergentes com o Paradigma Orientado a Objetos 54
2.7.4 Reflexões sobre o Paradigma Imperativo 54
2.7.4.1 Reflexão sobre a eficiência 55
2.7.4.2 Reflexão sobre o acoplamento e reuso 57
2.7.4.3 Reflexão sobre as dificuldades de programação 59
2.7.4.4 Reflexão sobre o paralelismo e distribuição 61
2.8 PARADIGMAS DECLARATIVOS 63
2.8.1 Paradigma Funcional 63
2.8.2 Paradigma Lógico 64
2.8.2.1 Visão geral 64
2.8.2.2 Sistemas Especialistas 65
2.8.3 Sistemas Baseados em Regras (SBR) 67
2.8.3.1 Visão geral 67
2.8.3.2 Algoritmo RETE 69
2.8.3.3 TREAT (TREe Associative Temporal redundancy) 74
2.8.3.4 LEAPS (Lazy Evaluation Algorithm for Production Systems) 76
2.8.3.5 HAL (Heuristically-Annotated-Linkage) 78
2.8.3.6 Algoritmos de inferência interpretados e compilados 82
2.8.4 Reflexões sobre o Paradigma Declarativo 83
2.8.4.1 Reflexões sobre as dificuldades de programação 84
2.8.4.2 Reflexão sobre acoplamento e eficiência 85
2.8.4.3 Reflexão sobre o paralelismo e distribuição 86
2.9 CONCLUSÃO 88
3 PARADIGMA ORIENTADO A NOTIFICAÇÕES 89
3.1 CONTEXTUALIZAÇÃO 90
3.2 O MODELO DO PARADIGMA PROPOSTO 91
3.3 A ESSÊNCIA DO MODELO PROPOSTO 92

3.3.1 Objetos colaboradores dos elementos da base de fatos 93
3.3.2 Objetos colaboradores das regras 96
3.4 MECANISMO DE NOTIFICAÇÕES 98
3.4.1 Visão geral 98
3.4.2 Exemplificação 100
3.5 UM EXEMPLO DE APLICAÇÃO DO PARADIGMA ORIENTADO A NOTIFICAÇÕES 102
3.5.1 *Rules* de coordenação 103
3.5.2 *Rules* em conflito 104
3.5.3 Características das *Rules* 105
3.5.4 *Rules* de decisão 106
3.6 RESOLUÇÃO DE CONFLITOS 107
3.6.1 Modelo centralizado de resolução de conflitos 108
3.6.2 Modelo descentralizado de resolução de conflitos (monoprocessado) 109
3.7 REFLEXÕES SOBRE O PARADIGMA ORIENTADO A NOTIFICAÇÕES 110
3.7.1 Reflexão sobre a eficiência 111
3.7.2 Reflexão sobre o acoplamento 112
3.7.3 Reflexão sobre as facilidades de programação 114
3.7.4 Reflexão sobre o paralelismo e distribuição 116
3.8 APLICABILIDADES 120
3.9 CONCLUSÃO 121
4 MATERIALIZAÇÃO E AVANÇOS DO PARADIGMA PROPOSTO 123
4.1 SISTEMA DE CONTROLE DE UM ROBÔ-MOTORISTA NO TRÂNSITO 124
4.2 MATERIALIZAÇÃO EM LINGUAGEM IMPERATIVA 125
4.3 VISÃO GERAL DO *FRAMEWORK* EM C++ 127
4.3.1 Estrutura da classe FBE 128
4.3.2 Estrutura da classe *Rule* 131
4.3.3 Reflexões 133
4.4 PARTICULARIDADES DO *FRAMEWORK* EM C++ 133
4.4.1 Implementação do *Attribute* 134
4.4.1.1 Notificações otimizadas 134
4.4.1.2 Re-notificações 135
4.4.2 Implementação do *Premise* 136
4.4.3 Implementação da *Condition* 139

4.4.4 Implementação da *Rule* 141
4.4.5 Reflexões 143
4.5 MATERIALIZAÇÃO DE JUNÇÕES DOS ATUAIS PARADIGMAS 144
4.6 CONCLUSÃO 146
5 ESTUDOS COMPARATIVOS 147
5.1 CONTEXTUALIZAÇÃO 148
5.2 ESTUDOS COMPARATIVOS SOBRE A PLATAFORMA DOS COMPUTADORES PESSOAIS 149
5.2.1 Aplicação Mira ao Alvo 149
5.2.2 Considerações sobre os estudos comparativos 152
5.2.3 Estudo comparativo entre as versões do *framework* do PON 153
5.2.4 Paradigma Orientado a Notificação versus Paradigma Imperativo 156
5.2.4.1 Primeiro cenário 157
5.2.4.2 Segundo cenário 163
5.2.4.3 Reflexões sobre as implementações 166
5.2.4.4 Reflexões sobre os resultados 172
5.2.5 Paradigma Orientado a Notificação versus Paradigma Declarativo 174
5.2.5.1 Estudos comparativos práticos 176
5.2.5.2 Estudo comparativo teórico 186
5.3 ESTUDO COMPARATIVO SOBRE PLATAFORMA EMBARCADA 195
5.3.1 Sistema de Condicionamento de Ar 197
5.3.1.1 Conceitos gerais 197
5.3.1.2 Conceitos específicos 200
5.3.2 Estudos comparativos práticos 202
5.3.2.1 Considerações sobre os estudos práticos 203
5.3.2.2 Primeiro cenário 205
5.3.2.3 Segundo cenário 206
5.3.2.4 Reflexões 208
5.4 CONCLUSÃO 209
6 CONCLUSÃO E TRABALHOS FUTUROS 211
6.1 CONCLUSÃO 211
6.2 TRABALHOS FUTUROS 213
6.2.1 Melhoramentos na facilidade de programação 213
6.2.2 Melhoramentos no tocante à Engenharia de Software 214
6.2.3 Melhoramentos na eficiência de execução 215

6.2.4 Estudo da aplicabilidade em ambientes multiprocessados 216
6.2.5 Estudo da aplicabilidade na Inteligência Artificial 217
6.2.6 Estudo da aplicabilidade na Computação Ubíqua 219
APÊNDICE A – MODELO DE RESOLUÇÃO DE CONFLITOS DESCENTRALIZADO PARA AMBIENTES MULTIPROCESSADOS 221
A.1 MODELO DESCENTRALIZADO PARA AMBIENTES MULTIPROCESSADOS ... 222
A.2 MECANISMO DE ESCALONAMENTO DE REGRAS 223
A.3 REFLEXÕES 226
APÊNDICE B – INICIATIVAS DE JUNÇÃO DOS CONCEITOS DOS ATUAIS PARADIGMAS 227
B.1 CLIPS++ 227
B.2 RETE++ 228
B.3 ILOG RULES 229
B.4 R++ 230
APÊNDICE C – REGRAS DE FORMAÇÃO OU FORMATION RULES 233
APÊNDICE D – CÓDIGOS-FONTE RELATIVOS AOS ESTUDOS COMPARATIVOS 235
D.1 PLATAFORMA DOS COMPUTADORES PESSOAIS 236
D.1.1 Paradigma Orientado a Notificações versus Paradigma Imperativo 236
D.1.1.1 Primeiro cenário 236
D.1.1.2 Segundo cenário 238
D.1.2 Paradigma Orientado a Notificações versus Paradigma Declarativo 240
D.1.2.1 Primeiro cenário 240
D.1.2.2 Segundo cenário 242
D.2 PLATAFORMA EMBARCADA 244
D.2.1 Paradigma Orientado a Notificações versus Paradigma Imperativo 244
D.2.1.1 Primeiro e segundo cenário 244
REFERÊNCIAS BIBLIOGRÁFICAS 249

LISTA DE FIGURAS

Figura 1: Pseudocódigo do Paradigma Imperativo 3
Figura 2: Integração da entidade de manufatura Kuka386 a um sistema computacional (Simão, 2005) 9
Figura 3: Exemplo de uma regra em PON 10
Figura 4: Colaboração por notificações das entidades do meta-modelo 11
Figura 5: Ilusão de ótica 21
Figura 6: Coexistência entre paradigmas (RICCIOLI, 1651) 23
Figura 7: Evolução entre paradigmas 23
Figura 8: Linha evolutiva das linguagens de programação, baseada em (BROOKSHEAR, 2006) 25
Figura 9: Exemplo de código em linguagem imperativa 29
Figura 10: Mecanismo interno de execução do LISP (baseado em (QUEINNEC, 1996) 30
Figura 11: Exemplo do fatorial na programação declarativa e imperativa 31
Figura 12: Exemplo de código em C++ 33
Figura 13: Código em PROLOG e a busca por profundidade com retrocesso (SCOTT, 2000) 35
Figura 14: Evolução das linguagens em relação aos paradigmas de programação, baseado em (BROOKSHEAR, 2006) 39
Figura 15: Classificação dos atuais paradigmas 40
Figura 16: Simulação de uma classe do Paradigma Orientado a Objetos com uma linguagem do Paradigma Procedimental 44
Figura 17: Invocação explícita e seqüencial de métodos 46
Figura 18: Processo de detecção de eventos (FAISON, 2006) 46
Figura 19: Pseudocódigo da Programação Dirigida a Eventos 48
Figura 20: Composição de componentes (CRNKOVIC e LARSSON, 2002) 52
Figura 21: Exemplo de redundância temporal e estrutural 56
Figura 22: Acoplamento entre objetos 57
Figura 23: Acoplamentos no escopo de classe e escopo de método 58
Figura 24: Cadeia de referência entre classes 58
Figura 25: Replicação de código 61
Figura 26: Tela do *Shell* CLIPS 66
Figura 27: Arquitetura interna de um Sistema Baseado em Regras 68

Figura 28: Estrutura do algoritmo RETE 71
Figura 29: Estrutura do algoritmo TREAT 75
Figura 30: Estrutura do LEAPS 78
Figura 31: Arquitetura de um Sistema Baseado em Regras que infere sobre o HAL 80
Figura 32: Estrutura do HAL para problemas combinatórios 80
Figura 33: Relação entre o paradigma proposto e os atuais paradigmas 90
Figura 34: Representação parcial da célula de manufatura 94
Figura 35: Estrutura do *FBE* 94
Figura 36: Representação de uma *Rule* 96
Figura 37: Estrutura da *Rule* 96
Figura 38: Relação entre os objetos principais e colaboradores 98
Figura 39: Exemplificação do mecanismo de notificações 100
Figura 40: Célula de manufatura completa simulada na ferramenta ANALYTICE II 103
Figura 41: Plano de produção para os produtos V1 e V2 104
Figura 42: Regras entrelaçadas 105
Figura 43: Regras para controlar os recursos 106
Figura 44: Regras para sinalizar a inserção e remoção de peças 107
Figura 45: Modelo centralizado de resolução de conflitos 109
Figura 46: Modelo descentralizado de resolução de conflitos 110
Figura 47: Solução do PON para as redundâncias temporais e estruturais 112
Figura 48: Acoplamento entre objetos no Paradigma Imperativo e entre *FBEs* e *Rules* no Paradigma Orientado a Notificações 113
Figura 49: Exemplificando a redução de acoplamentos 114
Figura 50: Exemplo da criação de uma *Rule* em C++ 115
Figura 51: Ferramenta amigável para a composição de *Rules* 115
Figura 52: Comunicação distribuída com o PON 119
Figura 53: Cenário do trânsito 125
Figura 54: Estrutura do *framework* 127
Figura 55: Instanciação do *framework* 128
Figura 56: Estrutura estendida dos componentes colaboradores dos FBEs 129
Figura 57: Inicialização dos *FBEs* e *Rules* em C++ 130
Figura 58: Estrutura estendida dos componentes colaboradores das *Rules* 133
Figura 59: Notificações baseadas em lista encadeada e tabela *hash* 134
Figura 60: Definição de *Premise Exclusive* em código C++ 138

Figura 61: Esquematização da *Premise* Exclusiva 138
Figura 62: Exemplo de uma expressão causal complexa em C++ 139
Figura 63: Representação padrão de uma expressão causal complexa em PON 140
Figura 64: Representação esquemática de uma regra complexa com sub-condições 140
Figura 65: Representação de uma expressão causal complexa com sub-condições 141
Figura 66: Estrutura da classe *Rule* 142
Figura 67: Código exemplo da derivação de uma *Rule* em C++ 142
Figura 68: Código exemplo da derivação de uma *Rule* em Java 143
Figura 69: Cenário similar ao tradicional jogo do mira ao alvo 151
Figura 70: Representação das *Rules* na versão antiga e nova do *framework* do PON 154
Figura 71: Resultados referentes ao primeiro experimento entre as versões do *framework* do PON 155
Figura 72: Resultados referentes ao segundo experimento entre as versões do *framework* do PON 155
Figura 73: Expressões causais referentes ao primeiro cenário do estudo comparativo com o PI 158
Figura 74: Resultados do experimento referente ao primeiro cenário do estudo comparativo com o PI com nenhuma expressão causal verdadeira 159
Figura 75: Analogia dos mecanismos de execução do imperativo e de notificação a um sistema de encanamento 160
Figura 76: Resultados do experimento referente ao primeiro cenário do estudo comparativo com o PI com uma expressão causal verdadeira 161
Figura 77: Resultados do experimento referente ao primeiro cenário do estudo comparativo com o PI com variações porcentuais de expressões causais verdadeiras 162
Figura 78: Representação do segundo cenário do estudo comparativo com o Paradigma Imperativo 163
Figura 79: Expressões causais referentes ao segundo cenário do estudo comparativo com o PI 164
Figura 80: Resultados do experimento referente ao segundo cenário do estudo comparativo com o Paradigma Imperativo com variações porcentuais de expressões causais verdadeiras 165
Figura 81: Representação de uma expressão causal imperativa em linguagem Assembly 167
Figura 82: Representação de uma expressão causal imperativa referenciando estrutura de dados vetores em linguagem Assembly 168

Figura 83: Experimento com as expressões causais com acesso direto aos estados dos atributos 169
Figura 84: Representação hipotética da relação entre os objetos colaboradores por meio do código gerado por um compilador 171
Figura 85: Primeiro cenário do estudo comparativo com o Paradigma Declarativo 180
Figura 86: Representação das regras para o primeiro cenário do estudo comparativo com o Paradigma Declarativo 181
Figura 87: Resultados do experimento em relação ao primeiro cenário do estudo comparativo com o Paradigma Declarativo 181
Figura 88: Segundo cenário do estudo comparativo com o Paradigma Declarativo 183
Figura 89: Regras do segundo cenário do estudo comparativo com o Paradigma Declarativo 183
Figura 90: Resultados dos experimentos para o segundo cenário do estudo comparativo com o Paradigma Declarativo 184
Figura 91: Representação esquemática da função assintótica do mecanismo de notificações 193
Figura 92: Placa EAT55 da eSysTech 196
Figura 93: Sistema de ar condicionado 198
Figura 94: Classes do sistema de condicionamento de ar 200
Figura 95: Representação da única expressão causal aprovada no estudo comparativo sobre a plataforma embarcada 202
Figura 96: Comportamento do PON em ambos os cenário do estudo comparativo com o PON e o POO na plataforma embarcada 205
Figura 97: Resultados do experimento sobre a plataforma embarcada sobre o primeiro cenário 206
Figura 98: Resultados do experimento sobre a plataforma embarcada sobre o segundo cenário 207
Figura 99: Experimento com as expressões causais com acesso direto aos estados dos atributos sobre o primeiro e segundo cenário 208
Figura 100: Representação de conflito em ambiente multiprocessado 222
Figura 101: Modelo de escalonamento 224
Figura 102: Representação de uma *Formation Rule* 234

LISTA DE TABELAS

Tabela 1: Exemplo de código em linguagem de máquina (BROOKSHEAR, 2006)..... 26
Tabela 2: Exemplo de código em Assembly (BROOKSHEAR, 2006)..... 27
Tabela 3: Exemplo de código em LISP e nas linguagens convencionais 30
Tabela 4: Popularidade entre os paradigmas (TIOBE, 2008) 45
Tabela 5: Estados da bomba de calor..... 199
Tabela 6: Estados da entrada de ar para a bomba de calor no estado COOLING 199

LISTA DE ABREVIATURAS E SIGLAS

AGV	(*Auto Guided Vehicles*) – Veículo Auto-Guiado
ATAL	(*Agent Theories, Architectures, and Languages*) – Teoria de Agentes, Arquiteturas e Linguagens
BIOS	(*Basic Input/Output System*) – Sistema Básico de Entrada e Saída
CLIPS	(*C Language Integrated Production System*) – Sistema de Produção Integrado à Linguagem C
CODASYL	(*Conference on Data Systems Language*) – Conferência sobre Linguagens para Sistemas de Dados
COOL	(*C Object-Oriented Language*) – Linguagem Orientada a Objetos em C
CORBA	(*Common Object Request Broker Architecture*)
DCOM	(*Distributed Component Object Model*)
DEC	(*Digital Electronic Computer*) – Computador Eletrônico Digital
DJGPP	(*DJ's GNU Programming Platform*)
DPMI	(*DOS Protected Mode Interface*)
ESYSTECH	(*Embedded Systems Technologies*) – Tecnologias de Sistemas Embarcados, Empresa Brasileira de Eletrônica
FORTRAN	(*FORmula TRANslation*) – Tradução de Fórmulas
FR	(*Formulation Rule*) – Regra de Formação
GCC	(*GNU Compiler Collection*)
GNU	(*GNU Is Not Unix*)
GoF	(*Gang of Four*) – Gangue dos Quatro
GUI	(*Graphical User Interface*) – Interface Gráfica do Usuário
HAL	(*Heuristically-Annotated-Linkage*) – Algoritmo de Ligações Anotadas Usando Heurísticas
IA	Inteligência Artificial
IDF	(*Intel Developer Forum*) – Fórum de Desenvolvedores da Intel
LCD	(*Liquid Crystal Display*) – Tela de Cristal Liquido
LEAPS	(*Lazy Evaluation Algorithm for Production Systems*) – Algoritmo de Avaliação Tardia para Sistemas de Produção
LEX	(*Lexicographic-Sort*) – Classificação Lexicográfica
LISP	(*LISt Processing*) – Processamento de Listas
LSI	Laboratório de Sistemas Inteligentes, antes chamado de LSIP

LSIP	Laboratório de Sistemas Inteligentes de Produção
MEA	(*Means-Ends-Analysis*) – Análise de Meio e Fim
MS-DOS	*Microsoft - Disk Operating System)*
NASA	(*National Aeronautics and Space Administration*) – Administração Nacional da Aeronáutica e do Espaço
PD	Paradigma Declarativo
PI	Paradigma Imperativo
POA	Paradigma Orientado a Agentes
POAS	Paradigma Orientado a Aspectos
POC	Paradigma Orientado a Componentes
POO	Paradigma Orientado a Objetos
PON	Paradigma Orientado a Notificações
POSA	(*Pattern-Oriented Software Architecture*) – Arquitetura de *Software* Orientada a Padrões
PP	Paradigma Procedimental
PROLOG	(*PROgramming in LOGic*) – Programação em Lógica
PS/2	(*Personal System/2*) – Sistema Pessoal/2
RMI	(*Remote Method Invocation*) – Invocação Remota de Métodos
RTC	(*Real-Time Clock*) – Relógio de Tempo-Real
SBR	Sistemas Baseados em Regras
SE	Sistemas Especialistas
SRAM	(*Static Random Access Memory*) – Memória Estática de Acesso Aleatório
STL	(*Standard Template Library*) – Biblioteca de Padrão de Templates
SysML	(*Systems Modeling Language*) – Linguagem de Modelagem de Sistemas
TD	*Timestamp* Dominante
UML	(*Unified Modeling Language*) – Linguagem de Modelagem Unificada
USB	(*Universal Serial Bus*) – Barramento Serial Universal
XCON	(*eXpert CONfigurer*) – Configurador Inteligente

RESUMO

Os atuais paradigmas de programação de *software*, mais precisamente o Paradigma Imperativo (PI) e o Paradigma Declarativo (PD), apresentam deficiências que afetam: (a) o desempenho das aplicações; (b) a facilidade de programação no PI ou as funcionalidades na programação no PD; e (c) a obtenção de "desacoplamento" (ou acoplamento mínimo) entre os módulos de *software*, o que dificulta seus reaproveitamentos bem como o uso de multiprocessamento.

Na verdade, o PI e o PD são similares ao serem baseados em buscas ou no percorrer sobre entidades passivas, que consistem em dados (e.g. fatos ou estados de variáveis ou de atributos de outras entidades) e comandos de decisão (e.g. expressões causais como *se-então* ou regras). Nestes, as buscas afetam o desempenho das aplicações por gerar redundâncias de processamento e dificultam o alcance de uma dependência mínima dos módulos por gerar acoplamento implícito entre eles.

Entretanto, o PI e o PD se diferem em termos de facilidades na programação. É difícil programar com o PI, uma vez que é preciso manipular diretamente os comandos das linguagens e organizá-los de tal maneira para formar o fluxo de execução dos programas. Felizmente, o PD poupa o desenvolvedor destas particularidades, mas para isto ele perde as funcionalidades existentes no PI como acesso a *hardware* e certas otimizações algorítmicas.

Com o objetivo de prover uma solução para este conjunto de deficiências, foi desenvolvido o Paradigma Orientado a Notificações (PON) derivado de uma teoria de controle e inferência precedente. O PON se inspira nos conceitos do PI (e.g. objetos) e do PD (e.g base de fatos e regras) oferecendo melhores qualidades do que estes paradigmas. Basicamente, o PON usa objetos para tratar de fatos e regras na forma de composições de outros objetos menores. Todos estes objetos apresentam características comportamentais de certa autonomia, independência, reatividade e colaboração por meio de notificações pontuais. Estas características são voltadas à realização participativa das funcionalidades do *software* por esses objetos.

Em suma, este trabalho apresenta o PON classificando-o como um paradigma efetivo, disserta sobre as qualidades e vantagens dele e a quais contextos se aplicam e principalmente, o compara com os paradigmas vigentes. Estas comparações se dão por meio de explicações e estudos práticos e teóricos, onde a eficiência de execução é salientada. O trabalho conclui sobre as vantagens e pertinências do PON, bem como abre perspectivas de pesquisa sobre ele, onde o multiprocessamento é um exemplo.

ABSTRACT

The current software programming paradigms, normally the Imperative Programming (IP) and Declarative Programming (DP), present deficiencies that affect: (a) the software performance; (b) the programming easiness in the IP case or the programming functionalities in the DP case; and (c) the achievement of “decoupling” (or minimal coupling) between the software modules, which complicates their reuse as well as their applications to multiprocessing.

In fact, IP and DP present similarities once they are based on searches upon passive entities that consist in data (e.g. facts or states of variable/attributes of other entities) and decision commands (e.g. causal expressions as if-then or rules). In these paradigms, the searches affect the software performance by generating processing redundancies and complicate the achievement of a minimal dependency between the modules by implicitly coupling them.

However, IP and DP are different in terms of programming easiness. It is difficult programming with the IP once the developer should directly handle the language commands and organize them to compose the software execution flow. Fortunately, the DP liberate the developer of these particularities. However, it does not offer the same functionalities of the IP such as the hardware access and certain algorithmic optimization.

In order to provide a solution to this set of the drawbacks, the Notification Oriented Paradigm (NOP) was proposed from a precedent control and inference theory. NOP is inspired on the IP concepts (e.g. objects) and the DP concepts (e.g. fact base and rules) and offers better qualities than these paradigms. Basically, NOP uses objects to deal with facts and rules, which in fact are composition of smaller objects. All these objects present features such as autonomy, independence, reactivity, and collaboration by means of punctual notifications. These features are related to the participative execution of the software functionalities by means of these objects.

In short, this work presents and classifies NOP as an effective paradigm, discusses about its qualities and discusses advantages, and discusses about the appropriate contexts that it is applicable, and, mainly, presents the comparisons with the current paradigms. These comparisons occur by means of explanations as well as by practical and theoretical studies, where the execution performance is evidenced. The work concludes about the PON advantages and pertinences, and still open perspectives to new researches, where the multiprocessing is an example.

CAPÍTULO 1

INTRODUÇÃO

Esta dissertação de mestrado tem como objeto de estudo um novo paradigma de programação de computadores intitulado Paradigma Orientado a Notificações (PON). O trabalho disserta sobre as qualidades do PON e o compara com alguns dos paradigmas vigentes por meio de experimentos, cálculos e explicações.

De forma sucinta, o PON foi proposto inicialmente por Jean Marcelo Simão, na forma de uma solução de controle discreto e de inferência, evoluindo posteriormente para a forma de paradigma de programação (SIMÃO, 2001, 2005; SIMÃO e STADZISZ, 2002, 2008, 2009a, 2009b; SIMÃO, STADZISZ e KÜNZLE, 2003)[1].

Neste capítulo, a seção 1.1 detalha os fatores motivadores que justificam a proposta do PON, a seção 1.2 explica sucintamente o PON e detalha os objetivos pretendidos com a realização deste trabalho. Por fim, a seção 1.3 apresenta a organização dos capítulos que compõem esta dissertação de mestrado.

1.1 MOTIVAÇÃO

Nas últimas décadas, muitas pesquisas contribuíram significativamente para a evolução tecnológica. Uma das maiores contribuições foi o advento dos computadores eletrônicos no início da década de 40. Desde então, os computadores vêm evoluindo incessantemente. Os computadores evoluíram de grandes e lentas estações de trabalho para pequenos e velozes computadores pessoais, como *desktops* e os *notebooks*. Os atuais computadores comportam processadores de alto desempenho, os quais são centenas ou milhares de vezes mais velozes do que os primeiros processadores (BROOKSHEAR, 2006).

Neste âmbito, segundo as afirmações feitas em 1965 por Gordon Moore, as quais fundamentaram a Lei de Moore, a capacidade de processamento tende ao crescimento exponencial (MOORE, 1965). Segundo ele, a quantidade de transistores alojada em uma

[1] Salienta-se que o PON é atualmente objeto de pedido de patente junto ao INPI (Instituto Nacional de Propriedade Industrial) sob número provisório 015080004262 via Agência de Inovação da UTFPR (SIMÃO e STADZISZ, 2008). Assim sendo, a utilização do PON se submete ao respeito dos direitos relativos a esta potencial patente.

placa de silício duplica a cada 24 meses, contribuindo para o aumento da capacidade de processamento dos computadores (MOORE, 1965).

A Lei de Moore se manteve predominante durante as últimas quatro décadas, tal qual o avanço constante da tecnologia dos processadores. Não obstante ao avanço tecnológico decorrente, a supremacia desta lei durante todos estes anos, vem afetando os hábitos dos programadores de computadores, desestimulando-os a criar programas ou *software* mais eficientes em termos de recursos computacionais. Entretanto, é a qualidade de *software*, inclusive em termos de economia de processamento, que define o uso efetivo e correto da capacidade de processamento disponível.

Neste contexto, além dos maus hábitos exercidos pelos programadores, há outros fatores que influenciam a qualidade do *software* gerado, como o emprego ou não de métodos de engenharia de *software* e a linguagem de programação escolhida. Além da linguagem de programação em si, algo que influencia a eficiência de processamento e a qualidade em geral do *software* é o próprio paradigma que rege a linguagem.

Ao bem da verdade, em geral, os atuais paradigmas de programação não contribuem para a construção de programas eficientes. Sucintamente, do ponto de vista do controle de processamento, os atuais paradigmas são classificados em Paradigma Imperativo (PI) e Paradigma Declarativo (PD).

No PI, o programador expressa por meio de uma linguagem de comandos seqüenciais "o que o sistema deve fazer", "como o sistema deve fazer" e "em qual ordem o sistema deve fazer" (ROY e HARIDI, 2004). No PD, por sua vez, o programador expressa por meio de uma linguagem de "regras" e até mesmo de "funções" somente "o que o sistema deve fazer", se abstendo das particularidades de implementação dos algoritmos, sendo a utilização de 'algoritmos genéricos' um habito em grande parte deste paradigma (ROY e HARIDI, 2004).

Embora estes paradigmas difiram na forma pela qual os programas são concebidos, eles se assemelham na forma pela qual os programas são executados. Nestes, as expressões causais (e.g. comandos *se-então*) e os dados (e.g. variáveis) são tratados como entidades passivas, as quais são relacionadas por meio de uma pesquisa estabelecida pelo fluxo de execução dos programas. Em geral, este processo causa considerável desperdício de processamento (SIMÃO e STADZISZ, 2009b).

Por exemplo, na Figura 1, o pseudocódigo expressa um "programa" fundamentado no PI, cuja execução ocorre por meio de um laço de repetição que permite pesquisar e avaliar os estados das variáveis. Neste pseudocódigo, as expressões causais são percorridas seqüencialmente, passivamente e repetidamente de forma a avaliar os estados das variáveis às

quais se referem. No entanto, muitas destas expressões causais não são necessariamente aprovadas, pois as variáveis referenciadas podem não apresentar os estados esperados para tal.

```
enquanto (verdadeiro)
início
  se(objeto1.atributo = 1) e (objeto2.atributo = 1) e (objeto3.atributo = 1) então
    objeto1.metodo();
    objeto2.metodo();
    objeto3.metodo();
  fim-se

  se(objeto1.atributo = 2) e (objeto2.atributo = 2) e (objeto3.atributo = 2) então
    objeto1.metodo();
    objeto2.metodo();
    objeto3.metodo();
  fim-se
  ...
  se(objeto1.atributo = N) e (objeto2.atributo = N) e (objeto3.atributo = N) então
    objeto1.metodo();
    objeto2.metodo();
    objeto3.metodo();
  fim-se
fim
```

Figura 1: Pseudocódigo do Paradigma Imperativo

Este modo de execução gera desperdício de processamento, pois há muitas avaliações lógicas que consomem recursos de processamento sem contribuir significativamente para a execução do programa. Como resultado, o uso do processador para processar estas avaliações desnecessárias (i.e. expressões causais não aprovadas) atrasa a execução das instruções significativas, impedindo que o programa apresente melhor desempenho (SIMÃO & STADZISZ, 2008).

Certamente, o algoritmo ilustrativo apresentado pode ser otimizado, sendo que um meio evidente para tal seria o uso do condicional *senão*. Porém, este condicional não foi utilizado para permitir melhor elucidação das redundâncias e principalmente porque, na prática, estas redundâncias não podem ser evitadas quando as expressões causais se encontram dispersas. A dispersão se dá, por exemplo, quando duas ou mais expressões causais se encontram separadas por outros comandos que não se enquadram no escopo de um *senão* ou mesmo quando estas são definidas no escopo de diferentes funções ou módulos.

Ainda neste âmbito da melhoria dos algoritmos, otimizações também têm sido aplicadas sobre “algoritmos genéricos” do PD que, em uma explicação simplista, são abstrações sobre o PI. Entretanto, o fator relevante desta discussão é que os atuais paradigmas não impelem otimizações efetivas e fáceis dos programas na atividade de programação de computadores, tanto quanto o presente avanço tecnológico dos processadores não impele tal otimização.

1.1.1 *Softwares* otimizados

O descaso com a eficiência do *software*, tanto por parte dos programadores quanto por parte dos atuais paradigmas, pode até ser aceitável na implementação de programas usuais de pequeno a médio porte que executam em arquiteturas atualizadas segundo os princípios da Lei de Moore. Nestas arquiteturas, a ineficiência não é percebida devido à relativa baixa quantidade de instruções a serem executadas diante da alta capacidade de processamento dos atuais processadores.

Não obstante, este descaso com a eficiência pode causar maiores efeitos na construção de outros tipos de programas, como certos programas de sistemas embarcados (BANASZEWSKI, SIMÃO, TACLA e STADZISZ, 2007). Nestes sistemas, a capacidade do *hardware* é normalmente insuficiente para "maquiar" as deficiências do *software*.

Os sistemas embarcados, particularmente, usam pequenos programas de computador projetados para executar uma ou poucas funções específicas em um *hardware* específico (BARR, 1999). Geralmente, este tipo de sistema é basicamente constituído por componentes meramente suficientes para realizar este conjunto relativamente pequeno de tarefas específicas, com processadores de baixa capacidade de processamento. Isto se deve a questões mercadológicas como diminuição de custos e mesmo tamanho físico que inviabilizam o uso de um *hardware* mais sofisticado (STADZISZ e RENAUX, 2007; WOLF, 2007).

Neste contexto, salienta-se que os sistemas embarcados devem ser desenvolvidos com respeito às restrições impostas e, ainda, apresentar um desempenho adequado. Assim, a prática de desenvolvimento destes sistemas deve enfatizar, por necessidade evidente, a eficiência (WOLF, 2007; HEATH, 2003).

Na verdade, da mesma forma que no *software* embarcado, o descaso com a eficiência também deve ser desencorajado na construção de *software* em geral, mesmo que estes executem sobre plataformas de *hardware* com processadores potentes. Isto se deve por diversos motivos, principalmente para diminuir o tempo de resposta ao usuário e para evitar atualizações desnecessárias de *hardware*.

1.1.2 *Softwares* complexos

No âmbito dos *softwares* complexos, particularmente nos monoprocessados, a eficiência é ainda mais importante para a sua execução apropriada ou até mesmo para a sua

viabilidade. Um exemplo de *software* complexo, dentre tantos existentes, é um sistema operacional moderno (e.g. *Windows Vista* ou *Linux Ubuntu*) que executa conjuntamente e controla outras aplicações, como editores de texto e planilhas de cálculos. Outro exemplo de *software* complexo são os jogos de computadores que apresentam dificuldades para executar em ambientes monoprocessados, sendo aconselhado o bi-processamento (*dual-core*)[2].

De fato, mesmo com a evolução exponencial da tecnologia dos processadores, em alguns casos a capacidade de processamento ainda é insuficiente para suportar a demanda por processamento não otimizado em *software* complexo e monoprocessado (OLIVEIRA e STEWART, 2006). Além disso, a necessidade de construção de sistemas computacionais mais complexos cresce em passos mais largos do que o avanço da capacidade de processamento (ABBAS, 2004).

A necessidade crescente por *software* mais complexo (e mais barato) se deve, principalmente, ao aumento da complexidade no modo de vida das pessoas e das instituições no mundo atual, envolto na era tecnológica. Esta complexidade instiga a aplicação cada vez mais freqüente de soluções computacionais aos problemas emergentes como aplicativos cada vez mais "inteligentes", carros "inteligentes", casas "inteligentes", dentre outros (KAISLER, 2005; LOKE, 2006).

Desta forma, para suportar esta necessidade sem otimizações, seria necessário que a Lei de Moore permanecesse em vigor ainda por muitos anos. No entanto, a evolução da tecnologia dos processadores segundo tal lei parece estar próxima do seu fim. Segundo o próprio Moore, em uma entrevista concedida na comemoração de 10 anos do *Intel Developer Forum* (IDF) em São Francisco no ano de 2007, o fim da lei poderá ocorrer em no máximo quinze anos (MOORE, 2007).

A continuação da lei está sendo ameaçada pela limitação física do processo litográfico, usado na confecção de dispositivos semicondutores. Segundo Moore, em breve, será impossível alinhar os átomos de silício de forma a possibilitar o controle do fluxo eletrônico (MOORE, 2007).

No entanto, estudos científicos buscam alternativas à tecnologia de transistores para manter a Lei de Moore válida por muitas outras décadas. A aposta dos cientistas são as tecnologias relacionadas aos nanotubos de carbono e a computação quântica. Porém, estas tecnologias ainda não foram difundidas comercialmente, uma vez que as pesquisas

[2] Exemplos de jogos *dual-core* são o Falcon 4.0 e o Supreme Commander (BONANNI, 1999; GAS POWERED GAMES, 2007).

relacionadas ainda precisam avançar para apresentar de forma concreta os reais benefícios destas tecnologias (KAYE, LAFLAMME e MOSCA, 2007; O'CONNELL, 2006).

Enquanto estas tecnologias não são apresentadas efetivamente como a solução à permanência da Lei de Moore, a viabilidade da construção de certos *softwares* mais complexos pode estar justamente na programação mais eficiente. Para isto, os hábitos dos programadores devem ser remodelados a fim de que se tire maior proveito da capacidade de processamento dos computadores, da mesma forma como já ocorre na programação de sistemas embarcados. Entretanto, o ideal seria que novos paradigmas fossem propostos de forma a incentivar a construção de *software* mais eficiente sem esforços adicionais ao programador diferentemente do que se observa atualmente na construção de *software* embarcado.

1.1.3 *Softwares* paralelos e distribuídos

Mesmo que aquelas novas tecnologias venham a surgir para substituir a tecnologia dos atuais processadores e esses hábitos e paradigmas voltados à otimização sejam concretizados, ainda assim não seriam suficientes para certas aplicações com altíssima demanda de processamento em termos de computação monoprocessada.

Como exemplos destas aplicações estão alguns sistemas de tempo real para controle de usinas nucleares (DÍAZ, GARRIDO, ROMERO *et al*, 2007) e alguns sistemas de controle de manufatura (LAW e MCCOMAS, 1999; OLIVEIRA e STEWART, 2006). Assim, uma alternativa efetiva para suportar este tipo de demanda por processamento seria o emprego da computação paralela ou da computação distribuída (HUGHES e HUGHES, 2003).

A computação paralela e a computação distribuída propõem a subdivisão da execução do *software* em tarefas. Essencialmente, uma tarefa é um conjunto de instruções com características de execução concorrente em relação às demais tarefas, as quais são distribuídas por vários processadores.

Na computação dita paralela, os processadores encontram-se em um mesmo computador ou ambiente computacional coeso, enquanto que na computação distribuída, os processadores encontram-se em nós computacionais conectados por rede. Na verdade, estes modelos computacionais têm sido motivados pela produção em larga escala de processadores, o que causa a redução dos seus preços, e mesmo pela evolução tecnológica relacionada às redes de computadores, no caso da computação distribuída (TANENBAUM e STEEN, 2006).

No entanto, tanto as linguagens como os paradigmas de programação atuais também não apresentam facilidades necessárias para atuarem efetivamente na computação paralela e distribuída (e.g. forte acoplamento entre as partes do código) e nem o aproveitamento adequado para atuar nestes ambientes (e.g. desperdício de processamento e comunicação) (HUGHES, 2003; SIMÃO e STADZISZ, 2008).

Neste âmbito, com o intuito de prover aos programadores os reais benefícios da computação paralela e distribuída, novas soluções de programação estão sendo desenvolvidas que, entretanto, não são suficientes por simplesmente se tratarem de extensões aos atuais paradigmas[3] (SIMÃO e STADZISZ, 2008).

1.1.4 Facilidades de implementação

Em suma, a construção de código eficiente e distribuível não deve ser negligenciada frente ao crescimento exponencial da capacidade de processamento, uma vez que o aumento da complexidade dos sistemas é um fato certo e a continuação da validade dos princípios da Lei de Moore é um fato duvidoso (MOORE, 2007). Ainda e ademais, esta preocupação com a eficiência de execução é importante para evitar a atualização constante de *hardware* ou mesmo distribuição que pode ser inviável (e.g. financeiramente) ou, pelo menos, causar um desperdício (e.g. financeiro).

No entanto, a construção de *software* mais eficiente e com facilidades de distribuição não deveria afetar a produtividade dos programadores. Pelo contrário, a construção de *software* deve ser cada vez mais facilitada, devido justamente ao aumento da quantidade de problemas complexos a serem resolvidos computacionalmente, os quais demandam maiores esforços cognitivos para serem construídos.

Desta forma, a prática de programação baseada na implementação de códigos eficientes e com facilidades de distribuição não deve exigir grandes esforços dos programadores, diferentemente do que ocorre na construção de *software* embarcado (no caso de eficiência) e de *software* multiprocessado (no caso da distribuição) usando os conceitos e tecnologias relativas aos atuais paradigmas.

---

[3] A título de exemplo, algumas destas soluções que auxiliam na distribuição de software são: (a) compiladores inteligentes que detectam possível paralelismo entre partes do código (BANERJEE, 1995); (b) *middlewares* como o CORBA (AHMED, 1998) e o RMI (*Remote Method Invocation*) (REILLY e REILLY, 2002) que abstraem funcionalidades de distribuição; e ainda (c) a proposta de novas linguagens de programação ou extensões das linguagens atuais para atuar nestes ambientes, como a Presto (BERSHAD, LAZOWSKA e LEVY, 1988) e a CC+ (WILSON e LU, 1996).

Em relação à construção de *software* embarcado, a programação é tediosa por exigir vários refinamentos do código para torná-lo mais eficiente (BARR, 1999). Em relação a construção de *software* multiprocessado, a programação é cautelosa e sujeita a erro, principalmente porque a responsabilidade de tratar as tarefas de sincronismo e comunicação entre as partes do *software* e até mesmo do controle de acesso aos dados do programa recaem diretamente sobre o programador (HUGHES e HUGHES, 2003).

Assim sendo, surge a necessidade de que novos paradigmas de programação sejam propostos para oferecer novas formas de conceber *software*, provendo maiores facilidades do que os paradigmas atuais. Neste âmbito, também é necessário que estes novos paradigmas de programação sejam propostos de maneira tal a oferecer mecanismos que aumentem a eficiência de execução e a distributividade do *software* de forma transparente ao programador a fim de suportar, por exemplo, a demanda por *software* com grandes necessidades de processamento.

Portanto, de acordo com o que foi exposto, a área de programação clama por novas propostas que permitam, sem um esforço adicional: (a) gerar códigos que executem eficientemente; (b) tornar mais simples a tarefa de desenvolvimento de *software*; e ainda (c) facilitar o uso e oferecer maiores benefícios da computação paralela e distribuída.

Neste contexto, a presente dissertação avalia o Paradigma Orientado a Notificações (PON) como uma solução para parte destes problemas (i.e. eficiência e facilidades de desenvolvimento) e um caminho de pesquisa para outra parte destes problemas (i.e. facilidades de distribuição).

1.2 CONTEXTUALIZAÇÃO E OBJETIVO DO TRABALHO

1.2.1 As origens do Paradigma Orientado a Notificações – um meta-modelo de controle

O Paradigma Orientado a Notificações (PON) foi concebido baseado na dissertação de mestrado e na tese de doutorado de Jean Marcelo Simão (SIMÃO, 2001; SIMÃO, 2005). O objetivo da dissertação e principalmente da tese foi propor novos mecanismos de controle que suprissem as necessidades relacionadas com os sistemas modernos de produção, como o tratamento das variações acentuadas de produção e mesmo a personalização em massa, que clamam por aperfeiçoamentos relativos à agilidade na organização da produção (SIMÃO, 2005).

Neste sentido, Simão propôs uma abordagem de controle que permite, de forma eficiente e intuitiva, organizar as colaborações entre entidades de manufatura (e.g. recursos ou equipamentos) a fim de alcançar agilidade na produção. Esta abordagem refere-se a um meta-modelo (genérico) de controle discreto que foi aplicado à simulação de sistemas de manufatura ditos inteligentes. Esta simulação ocorreu sobre a ferramenta de projeto e de simulação de sistemas de manufatura chamada ANALYTICE II (SIMÃO, 2005)[4].

Nestes sistemas "inteligentes", as entidades de manufatura são integradas a sistemas computacionais "comuns" (e.g. *software* de controle) por meio de "recursos virtuais" (i.e. *drivers* avançados). Estes permitem o acesso a dados e serviços pelos sistemas computacionais através de uma rede de comunicação de dados (SIMÃO, 2005).

A título de exemplo, a Figura 2 ilustra a entidade de manufatura *Kuka386* (i.e. robô de transporte de peças) integrada como um componente de um sistema computacional de controle por meio do Kuka386-virtual (i.e. um *smart-driver*). Esta integração se dá por meio da rede de comunicação de dados, onde os *feedbacks* (e.g. comandos) entre o sistema computacional e o equipamento real são intermediados pelo recurso virtual ou *driver* deste equipamento.

Na verdade, todo e qualquer recurso virtual expressa os estados ou valores do respectivo equipamento por meio de entidades chamadas de atributos, bem como disponibiliza seus serviços por meio de entidades chamadas de métodos. Assim sendo, todos os recursos-virtuais apresentam a mesma forma de *feedback* para com um determinado sistema computacional.

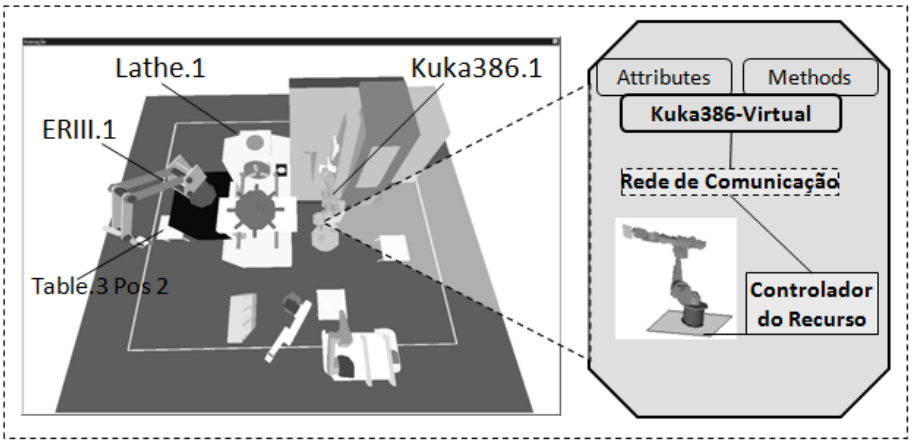


Figura 2: Integração da entidade de manufatura Kuka386 a um sistema computacional (Simão, 2005)

[4] O projeto do ANALYTICE II foi desenvolvido por gerações de pesquisadores do Laboratório de Sistemas Inteligentes de Produção (LSIP) da UTFPR (KOSCIANSKI, ROSINHA, STADZISZ e KÜNZLE, 1999; SIMÃO, 2005).

No domínio de manufatura, o meta-modelo em questão permite compor *software* de controle onde a representação das relações causais de controle se dá por meio de regras causais sobre os atributos e métodos dos recursos-virtuais. Um exemplo de uma regra na forma de conhecimento causal é apresentado na Figura 3. A semântica desta regra refere-se ao controle das relações entre os equipamentos *Lathe.1* (i.e. um torno mecânico), *ERIII.1* (i.e. um robô de transporte de peças) e *Table.3* (i.e. uma mesa para armazenamento temporário de peças), os quais compõem parte da célula de manufatura simulada no ANALYTICE II e apresentada na figura precedente.

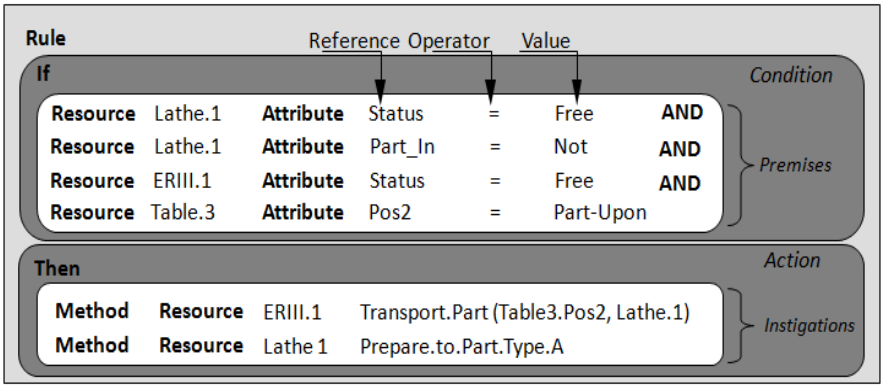


Figura 3: Exemplo de uma regra em PON

Em linhas gerais, uma regra consiste em uma condição e uma ação, onde a condição representa a avaliação causal de uma regra enquanto uma ação consiste no conjunto de instruções/comandos executáveis de uma regra. No exemplo, a condição da regra verifica se os recursos *Lathe.1* e *ERIII.1* estão livres, se o recurso *Lathe.1* não tem peça dentro de si e se há algum produto sobre o recurso *Table.3*. Se estes estados forem constatados, a ação da regra faz com que o robô *ERIII.1* transporte o respectivo produto para ser manipulado pelo torno *Lathe.1*.

No meta-modelo, cada regra causal é computacionalmente representada por um conjunto de entidades relacionadas, as quais podem ser criadas automaticamente a partir do conhecimento expresso em um ambiente de programação (i.e. um *wizard*). Dentre estas entidades, o cerne é a entidade chamada *Rule* (Regra).

Uma *Rule* é decomposta em uma entidade *Condition* (Condição) e uma entidade *Action* (Ação). De maneira similar, uma *Condition* é decomposta em uma ou mais entidades *Premises* (Premissas) e uma *Action* é decomposta em uma ou mais entidades *Instigations* (Instigações). Ainda, conforme dito, cada recurso-inteligente (*Resource*) também é decomposto em entidades menores, os *Attributes* (Atributos) e os *Methods* (Métodos).

Todas estas entidades colaboram por meio de uma abordagem ímpar de inferência baseada em notificações, a fim de ativar as regras pertinentes para execução (SIMÃO e STADZISZ, 2008). Esta abordagem é explicitada pelo esquema apresentado na Figura 4. Nesta abordagem, cada recurso (*Resource*) notifica o seu conhecimento factual por meio de capacidades reativas incorporadas às suas entidades *Attributes* às demais entidades envolvidas.

Sucintamente, a cada mudança no estado de um *Attribute*, ele próprio notifica imediatamente uma ou um conjunto de entidades *Premises* relacionadas para que estas reavaliem os seus estados lógicos, comparando o valor notificado com outro valor (uma constante ou um valor notificado por outro *Attribute*) usando um operador lógico. Se o valor lógico da entidade *Premise* se alterar, esta notifica uma ou um conjunto de entidades *Conditions* conectadas para que seus estados lógicos sejam reavaliados.

Deste modo, cada entidade *Condition* notificada reavalia o seu estado lógico de acordo com o valor recém notificado pela *Premise* em questão e os valores notificados previamente pelas demais *Premises* conectadas. Assim, quando todas as entidades *Premises* que compõem uma entidade *Condition* apresentam o estado lógico verdadeiro, a entidade *Condition* é satisfeita, decorrendo na aprovação da sua respectiva *Rule* para a execução. Com isto, a entidade *Action* conexa a esta *Rule* é executada, podendo invocar serviços (*Methods*) nos recursos por intermédio das entidades *Instigations* (SIMÃO, 2005).

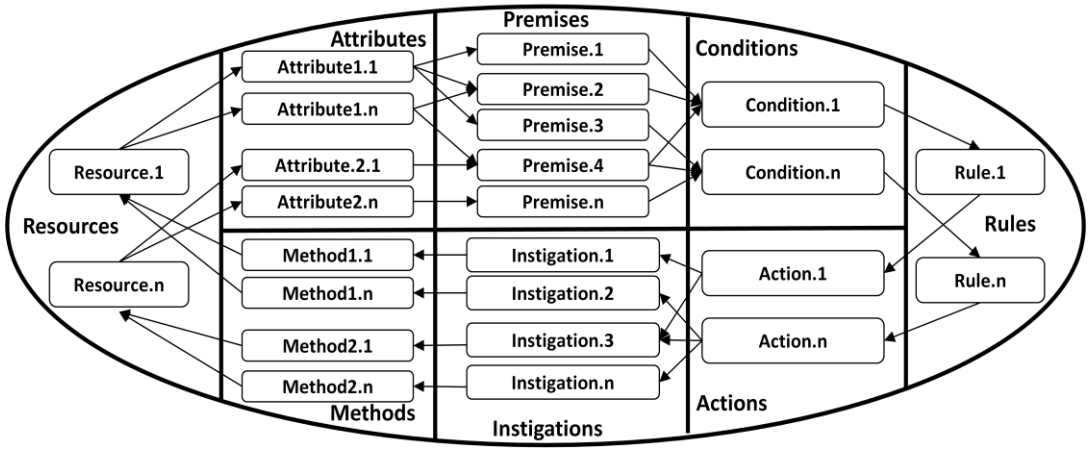


Figura 4: Colaboração por notificações das entidades do meta-modelo

Por meio deste mecanismo de inferência, o meta-modelo provê uma solução efetiva para compor e executar *software* de controle no domínio de sistemas de manufatura modernos. Este fato foi confirmado pela implementação e análise do meta-modelo sobre diversas perspectivas no simulador ANALYTICE II (SIMÃO, 2005; SIMÃO, STADZISZ, BANASZEWSKI e TACLA, 2008). Nesta análise, o meta-modelo cumpriu as expectativas e

os resultados estão apresentados em (SIMÃO, 2001, 2005; SIMÃO e STADZISZ, 2002; SIMÃO, STADZISZ e MOREL, 2006; SIMÃO, STADZISZ, BANASZEWSKI e TACLA, 2008).

1.2.2 O surgimento do Paradigma Orientado a Notificações

Incentivado pelos resultados conseguidos e pelo fato dos constituintes do meta-modelo serem genéricos, o autor deste vislumbrou a possibilidade de aplicação do meta-modelo em domínios diferentes daquele de produção. Assim sendo, ele propôs o meta-modelo como uma solução geral de controle discreto, bem como uma solução geral de inferência (SIMÃO e STADZISZ, 2002, 2008; SIMÃO, FABRO, STADZISZ *et al*, 2003; SIMÃO, STADZISZ e KÜNZLE, 2003).

Posteriormente, o referido autor percebeu que o meta-modelo poderia ser utilizado como um modelo para guiar o programador na concepção de programas, ou seja, como um novo paradigma de programação, o qual foi denominado de Paradigma Orientado a Notificações (PON) (SIMÃO e STADZISZ, 2008). A pertinência desta constatação se dá principalmente porque o meta-modelo apresenta qualidades inexistentes nos atuais paradigmas de programação.

No meta-modelo, os "dados" (e.g. *Attributes* e mesmo *Methods* dos *Resources* ou *Elements*) e os "comandos" (e.g. *Premises*, *Conditions* e *Rules*) são entidades independentes que apresentam características de autonomia e cooperação para realizar as avaliações lógicas. Neste modelo, os "dados" ou *Attributes* são autônomos para decidirem o momento correto no qual as expressões causais pertinentes devem ser avaliadas.

Os *Attributes* comunicam os seus estados por meio de notificações pontuais às partes de uma expressão causal, as *Premises*. Por sua vez, as *Premises* cooperam, por meio de notificações pontuais, com as expressões causais propriamente ditas (i.e. as *Conditions*) a fim de determinar os seus estados lógicos. Esta abordagem baseada em notificações evita pesquisas e, conseqüentemente, a ocorrência de avaliações desnecessárias.

Com a aplicabilidade destes conceitos na forma de um novo paradigma de programação, os programas passam a ser concebidos para alcançar melhor desempenho de execução, maior desacoplamento entre as suas partes e ainda prováveis maiores facilidades e beneficios ao atuar em ambientes multiprocessados (i.e. computação paralela e distribuída). Além do mais, todas estas qualidades são providas naturalmente e transparentemente ao

programador, o qual se atem basicamente em representar o conhecimento do programa em forma de regras, resultando em maiores facilidades de programação.

No âmbito deste trabalho de mestrado, alguns trabalhos publicados e outros visando publicação (BANASZEWSKI, SIMÃO, TACLA e STADZISZ, 2007; BANASZEWSKI, 2008; LUCCA, SIMÃO, BANASZEWSKI, STADZISZ e TACLA, 2008) vêm apresentando e discutindo o PON como um paradigma de programação em ascensão. Estes trabalhos mostram a viabilidade do PON frente aos principais paradigmas da atualidade pelo fato deste oferecer certas qualidades na programação local e, provavelmente, paralela e distribuída.

1.2.3 Objetivos

Esta dissertação, baseada nas publicações citadas e em novos esforços de pesquisa, reapresenta o PON e disserta sobre as suas qualidades e sobre os benefícios atingíveis com o seu uso. Com isto, esta dissertação pretende explorar o PON como um novo paradigma de programação, procurando responder as indagações sobre as vantagens efetivas do PON em relação aos demais paradigmas e em quais contextos elas se fazem mais úteis.

Neste sentido, a fim de melhor elucidar as vantagens do PON, esta dissertação particularmente apresenta experimentos e cálculos comparativos entre instâncias criadas com o PON e com outros paradigmas. Nestes experimentos, as instâncias do PON são implementadas sobre um *framework* que materializa avanços realizados sobre o meta-modelo de controle proposto inicialmente por Simão (SIMÃO, 2005). Estes avanços também são apresentados no escopo desta dissertação.

Quanto aos experimentos, mais precisamente, estes comparam o PON em termos de desempenho de execução com instâncias do PI e do PD. Dentre outros, um dos fatores motivadores para a realização destes experimentos é a ausência de comparações efetivas do PON para com os paradigmas vigentes.

1.3 ORGANIZAÇÃO DO TRABALHO

Para apresentar o PON como um paradigma emergente e descrever as suas vantagens em relação aos demais paradigmas, esta dissertação de mestrado está dividida em seis capítulos.

No capítulo 2 são apresentados os conceitos fundamentais que formam a base conceitual utilizada para estruturar este trabalho. O capítulo introduz o termo paradigma nas ciências físicas e na ciência da computação e também apresenta uma visão geral dos paradigmas atuais na computação, relatando as suas principais deficiências.

No capítulo 3, os conceitos do PON são apresentados de forma detalhada. O capítulo apresenta a essência do PON, a qual consiste em suas entidades elementares e as colaborações destas por meio de notificações. Ainda, este capítulo apresenta as soluções propostas em PON para sanar as principais deficiências dos atuais paradigmas.

No capítulo 4, o PON é apresentado sobre a perspectiva da materialização de seus conceitos em uma linguagem de programação. Esta materialização consiste em uma nova versão que agrega avanços sobre o meta-modelo proposto inicialmente por Simão (SIMÃO, 2005). Ainda, este capítulo discute brevemente sobre algumas iniciativas de programação que materializam junções dos conceitos do PI e do PD com o intuito de obter maiores benefícios na programação. Porém, estas iniciativas continuam deficientes, simplesmente porque são baseados nos atuais paradigmas.

No capítulo 5, o PON é comparado com os atuais paradigmas por meio de estudos prático-experimentais e teóricos sobre duas plataformas, a plataforma dos computadores pessoais e uma plataforma embarcada. Estas comparações se dão principalmente com os paradigmas que serviram de inspiração para a concepção do PON, mais precisamente o Paradigma Orientado a Objetos (POO) do PI e o Paradigma Lógico (PL) do PD, especialmente sobre os Sistemas Baseados em Regras (SBR) do PL. Em suma, a essência do capítulo é apresentar alguns estudos comparativos a fim de confirmar as vantagens do PON perante estes paradigmas.

No capítulo 6, as conclusões sobre o trabalho desenvolvido são apresentadas assim como as perspectivas dos possíveis desdobramentos em trabalhos futuros.

CAPÍTULO 2

FUNDAMENTAÇÃO TEÓRICA

O objetivo deste capítulo é introduzir o termo paradigma e apresentar uma visão geral dos paradigmas atuais na computação, com ênfase ao Paradigma Orientado a Objetos e ao Paradigma Lógico, salientando deste, os Sistemas Baseados em Regras. Estes paradigmas são enfatizados porque o PON reaproveita alguns de seus princípios, evoluindo-os para resolver as suas principais deficiências.

A seção 2.1 apresenta uma definição precisa do termo paradigma na ciência física; a seção 2.2 apresenta a definição do mesmo termo na ciência da computação; a seção 2.3 contextualiza as mudanças de paradigmas na ciência física e na computação e aponta algumas contradições; a seção 2.4 apresenta os principais fatos que levaram a propostas de novos paradigmas da computação por meio de uma breve história da evolução das linguagens de programação.

Ainda, a seção 2.5 classifica as linguagens de programação em paradigmas; a seção 2.6 classifica os principais paradigmas de programação em dois mais amplos, o Paradigma Imperativo e o Paradigma Declarativo; a seção 2.7 apresenta o Paradigma Imperativo e relata sobre as suas principais deficiências; a seção 2.8 apresenta o Paradigma Declarativo e também descreve sobre as suas principais deficiências; por fim, a seção 2.9 conclui o capítulo.

A título de informação, caso o leitor apresente conhecimento sobre uma ou várias seções deste capítulo, poderá não lê-las ou postergar a leitura sem prejuízo ao entendimento da proposta que é apresentada a partir do capítulo 3. Não obstante, seria interessante que o leitor dedicasse atenção às seções 2.7.4 e 2.8.4 que fazem reflexões sobre os Paradigmas Imperativos e Declarativos.

2.1 PARADIGMA NAS CIÊNCIAS FÍSICAS

O termo Paradigma (do grego *Parádeigma*) é definido pelo dicionário inglês Oxford (OXFORD, 1989) com o significado de um típico exemplo ou um padrão. O dicionário Aurélio (FERREIRA, 2006) repete a definição inglesa e ainda apresenta uma definição formalizada na visão do célebre historiador da ciência Thomas Kuhn.

Segundo a definição encontrada no Dicionário Aurélio, Kuhn considera paradigmas como as “realizações científicas que geram modelos que, por um período mais ou menos

longo e de modo mais ou menos explícito, orientam o desenvolvimento posterior das pesquisas exclusivamente na busca da solução para os problemas por elas suscitados" (FERREIRA, 2006).

Esta definição foi transcrita em 1962 na primeira edição da obra *"The Structure of Scientific Revolutions"* (KUHN, 1962), na qual Kuhn contribui para alterar a noção que se tinha sobre o progresso científico através de uma análise sobre a história da ciência.

Apesar desta definição ser a pretendida por Kuhn para o termo paradigma, a mesma não foi claramente representada em sua obra devido ao estilo impreciso do autor. Tal fato levou a diferentes interpretações para o termo. Estas interpretações já eram observadas logo após a publicação da obra de Kuhn, quando o termo paradigma começou a ser utilizado em vários textos da ciência, nos quais raramente a definição pretendida era empregada (MASTERMAN, 1970).

Em 1965, no Seminário Internacional sobre Filosofia da Ciência em Londres, Kuhn recebeu várias críticas, principalmente sobre as definições imprecisas do termo paradigma. A crítica mais relevante foi argüida por Margareth Masterman (MASTERMAN, 1970). Masterman argumentou que o termo paradigma foi definido em 21 formas diferentes na obra de Kuhn, porém, guardando alguma similaridade entre elas. Pela análise das similaridades entre estas definições, Masterman classificou-as em três distintas categorias, reduzindo deste modo o leque de interpretações diferentes ao termo em questão para Paradigma Metafísico, Paradigma Sociológico e Paradigma Exemplar (MASTERMAN, 1970). Eis o detalhamento desses:

- **Paradigma Metafísico:** é representado por uma crença profunda em um modelo da ciência dentro de uma área particular de conhecimento. A comunidade que crê nesta ciência é caracterizada por apresentar forte fidelidade aos conceitos defendidos, mesmo quando eles são criticados ou contestados por outros modelos. Esta definição faz perceber a dificuldade de se impor um novo paradigma que substitua o anterior.
- **Paradigma Sociológico:** é representado por fatores que mantêm unida uma determinada comunidade de praticantes de uma ciência, referenciada como comunidade científica por Kuhn. Esses fatores incluem crenças, valores e técnicas, e são entendidos como o interesse comum de um grupo sobre um conhecimento particular.

- **Paradigma Exemplar:** representa o significado para o termo paradigma segundo a real pretensão de Kuhn. É este sentido para o termo paradigma definido no dicionário Aurélio.

Esta classificação apresentada por Masterman instigou Kuhn a definir o termo paradigma com maior precisão e clareza conceitual. Para isto, Kuhn escreveu uma autocrítica no posfácio incluso na segunda edição da sua célebre obra (KUHN, 1970). No posfácio, Kuhn reconheceu sua imprecisão em relação à definição do termo paradigma, referindo à classificação de Masterman.

Segundo Kuhn, a maioria das interpretações atribuídas ao termo deve-se apenas a "incongruências estilísticas". Desta forma, dentre as 3 categorias apresentadas por Masterman, Kuhn reconhece somente duas destas: uma mais genérica e outra mais específica e precisa.

Na definição genérica, Kuhn reconhece o Paradigma Sociológico classificado por Masterman. Segundo as palavras de Kuhn, "um paradigma é aquilo que os membros de uma comunidade (científica) partilham e, inversamente, uma comunidade (científica) consiste em homens que partilham um paradigma" (KUHN, 1970)[5]. Aquilo que é partilhado é chamado por Kuhn de matriz disciplinar. Disciplinar porque a comunidade que segue uma mesma disciplina compartilha os mesmos princípios e matriz porque é composto de vários elementos (KUHN, 1970). Entre estes elementos estão os paradigmas Metafísicos e Exemplar.

Na definição mais específica, Kuhn reconhece o Paradigma Exemplar classificado por Masterman, o qual se refere basicamente a um modelo seguido por uma comunidade (científica). Neste sentido, um modelo consiste no conhecimento adquirido por uma comunidade (científica) sobre um determinado domínio, no qual a comunidade se baseia para expandir o seu conhecimento ou entendimento sobre este domínio. Os modelos são constituídos de métodos e técnicas concebidas a partir de trabalhos realizados pela comunidade (científica) a fim de guiar os participantes (e.g. cientistas) na solução de novos problemas, os quais podem ser resolvidos com a ajuda de ferramentas (e.g. telescópio no caso de modelos astronômicos). Entre todas as definições encontradas em sua obra, Kuhn considera a definição do Paradigma Exemplar a de uso mais apropriado (KUHN, 1970).

[5] Os parênteses em torno da palavra "científica" são inclusões próprias do autor desta dissertação.

2.2 PARADIGMA NA CIÊNCIA DA COMPUTAÇÃO

Na Ciência da Computação, apesar de ser uma ciência moderna e baseada principalmente na criatividade do homem ao invés de constatações físicas sobre elementos do mundo, ela se mostra fiel às definições do termo paradigma para Kuhn.

Nesta ciência moderna, o termo paradigma é empregado como a maneira de compreender um problema do mundo real para transformar este problema em uma solução computacional. Isto se dá, baseando-se em um conjunto de conceitos e técnicas empregadas com o auxílio de ferramentas de projeto e programação, sendo a programação o foco deste discurso[6].

O Paradigma Exemplar se adéqua perfeitamente ao contexto de programação, no sentido de que uma comunidade científica da computação segue um modelo que representa uma forma particular de enxergar os problemas do mundo real a serem tratados computacionalmente.

Geralmente, o modelo do paradigma está disponível para o programador através de uma linguagem de programação. Neste sentido, a linguagem de programação seria a ferramenta necessária para que o modelo suporte soluções concretas para os problemas do mundo real. As linguagens de programação são os meios de tornar o paradigma aplicável, assim como são as ferramentas das ciências físicas (i.e. telescópios e equações matemáticas na ciência astronômica). Devido à forte relação entre o termo paradigma e as linguagens de programação, os paradigmas são referenciados na ciência da computação como paradigmas de programação (KAISLER, 2005).

O significado de paradigma de programação não se fundamenta apenas no Paradigma Exemplar. Apesar de Kuhn ter apontado, em seu posfácio, o Paradigma Exemplar como a definição preferida, ele não refutou o Paradigma Sociológico e mesmo o Metafísico, considerando esse como um conceito global do termo e este apenas como elemento desse (KUHN, 1970). Entretanto, o fato importante é que eles continuam sendo definições válidas.

Neste sentido, as definições do Paradigma Sociológico e Metafísico também são usadas na ciência da computação para designar um paradigma de programação. De fato, o termo paradigma de programação pode ser definido como o Paradigma Sociológico, pois um

[6] Ao bem da verdade, é comum que as ferramentas de projeto dos programas, em um contexto de engenharia de *software*, sejam concordantes com os paradigmas que regem as linguagens de programação atuais.

grupo de desenvolvedores deve compartilhar uma mesma forma de ver e pensar sobre os problemas para permitir colaboração e coesão em suas ações.

Neste âmbito, há projetos de *software* que demandam conhecimentos heterogêneos em relação às técnicas e paradigmas dentro de um mesmo grupo. Exemplo: os projetos de construção de *web sites*, onde linguagens de programação tradicionais (e.g. Java, do Paradigma Imperativo) são usadas em conjunto com linguagens ditas de *script* (e.g. HTML, do Paradigma Declarativo).

Nestes projetos, é possível observar uma formação de subgrupos formados pelos integrantes com conhecimentos semelhantes em uma linguagem ou paradigma de programação. Isto leva a confirmação de que na computação, como na ciência de Kuhn, um paradigma se caracteriza pelos conceitos compartilhados por um grupo ou comunidade científica que se mantém unidas pelo que acreditam.

Ademais, um paradigma de programação também pode ser entendido como o Paradigma Metafísico devido à grande devoção apresentada por um grupo de programadores pela linguagem de programação que adotam, e consecutivamente, pelo paradigma que esta linguagem apóia.

Além das definições de Kuhn para o termo paradigma, há outras definições que podem ser usadas para esclarecer o significado da palavra paradigma na computação. Entre estas definições, encontra-se a elaborada pela célebre autora Marilyn Ferguson, comumente referenciada como a mestra da Nova Era por causa da sua obra “A Conspiração Aquariana”, também conhecida como “A Bíblia da Nova Era” (FERGUSON, 1980).

Apesar de Ferguson não ter a intenção de definir o termo paradigma no contexto da computação, ela o fez de uma maneira que se aproxima bastante ao seu real significado nesta área da ciência. Ferguson descreve o termo paradigma como uma forma de estruturar o pensamento, sendo um esquema usado para a compreensão de certos aspectos da realidade (FERGUSON, 1980).

Ao interpretar esta definição para o contexto da computação, entende-se como um paradigma a forma pela qual um determinado problema é percebido (a realidade de Ferguson) e a maneira usada para estruturar o pensamento na intenção de conceber uma solução para este problema por meio de uma linguagem de programação.

Apesar destas definições elaboradas por Kuhn e Ferguson serem voltadas às ciências físicas, elas permitem a interpretação adequada do termo à ciência da computação. Porém, para esclarecer o termo paradigma de forma menos interpretativa e mais fiel à ciência da

computação, recorre-se às definições de autores que expressam a real definição do termo de maneira própria a esta área.

Segundo David Watt, o termo paradigma de programação consiste na seleção de conceitos chaves da programação (e.g. tipos de dados, variáveis, escopo, abstração, concorrência e controle), usados de maneira conjunta para formar um estilo de programação (WATT, 2004). Segundo Peter Van Roy, um paradigma de programação é um sistema formal que define como a programação é realizada. Cada paradigma tem o seu próprio conjunto de técnicas para programação e forma de estruturar o pensamento na concepção dos programas (ROY e HARIDI, 2004).

2.3 MUDANÇAS DE PARADIGMAS

Tanto nas ciências físicas quanto na ciência da computação ocorrem mudanças na forma de perceber as realidades, tal fenômeno é conhecido como mudança de paradigma. Este termo foi introduzido por Kuhn para explicar a evolução das ciências físicas através do tempo e pode ser emprestado para a ciência da computação para referir às mudanças entre os paradigmas de programação.

Kuhn defendeu que os grandes progressos da ciência não resultam de mecanismos de continuidade, ou seja, incrementos e melhorias cumulativas, como acreditavam os cientistas daquela época, mas sim de mecanismos de ruptura (i.e. mudança de paradigma). Mas a ruptura não é tão simples de ocorrer devido à resistência a mudanças pelos membros de uma comunidade científica (KUHN, 1962).

A resistência é resultado da dificuldade cognitiva humana de perceber o mundo por uma perspectiva diferente daquela que se está acostumado. A perspectiva corrente é referida como paradigma dominante, pois domina a mente da pessoa. A nova perspectiva é referida como paradigma emergente, pois tende a substituir o paradigma dominante (KUHN, 1970).

Esta inflexibilidade cognitiva humana pode ser exemplificada pela análise cuidadosa da Figura 5, onde é possível encontrar duas representações diferentes. Porém, dependendo da forma pela qual a pessoa analisa esta figura, ela somente consegue identificar uma representação. Esta representação exemplifica o paradigma dominante. Desta forma, a pessoa pode visualizar prontamente a imagem de uma velha e somente com algum esforço conseguir visualizar a imagem de uma bela moça, ou em ordem inversa.

Figura 5: Ilusão de ótica

Em suma, o fato é que algumas pessoas que vêm o mundo de uma determinada maneira são muitas vezes incapazes de enxergar de outra ou demandam muitos esforços para isto. Assim, há pessoas que crêem insistentemente nos conceitos de um paradigma dominante e, mesmo que este paradigma seja totalmente refutado por outro emergente, elas continuam resistentes a mudar as suas formas de pensar (KUHN, 1970).

Por causa desta forte crença, é normal um paradigma se manter por muitos anos no pensamento de uma comunidade científica dificultando a ruptura. Isto se deve, em partes, à proteção persistente destes princípios pelos membros de uma comunidade. Segundo Kuhn, a crença de uma comunidade científica nos conceitos de um paradigma faz com que os seus membros somente realizem pesquisas sobre os temas certamente tratáveis pelo paradigma, raramente há pesquisas com a intenção de refutar o paradigma corrente (KUHN, 1970). Mesmo quando o paradigma se mostra insuficiente para responder uma questão considerada tratável, o paradigma mantém os seus valores e a questão não respondida é ignorada, alegando que a falha não se refere aos conceitos do paradigma, mas a quem não soube aplicá-lo corretamente (KUHN, 1970).

Porém, a mudança de paradigma se torna mais evidente na medida em que novos instrumentos da ciência (e.g. ferramentas como telescópios ou computadores mais potentes) são inventados, o que possibilita maiores avanços nos estudos sobre o modelo do paradigma (KUHN, 1970). Com estes avanços, novas questões são levantadas e questões não respondidas podem se tornar cada vez mais freqüentes. Estes avanços podem apontar as deficiências do paradigma dominante e abrir espaço para que haja a ruptura do paradigma, com a proposta de um novo. Porém, mesmo diante da crise de um paradigma, a resistência à mudança ainda persiste entre os membros (KUHN, 1970).

A resistência é maior na ciência física do que na ciência da computação, pois ocorre de maneira mais drástica. Segundo Kuhn, as mudanças de paradigmas ocorrem nas ciências

físicas somente quando um paradigma emergente substitui totalmente o paradigma dominante, tal fenômeno gera muitos conflitos entre as comunidades discordantes (KUHN, 1970). Na verdade, não se trata necessariamente de uma mudança total, mas sim substancial como argumenta o próprio Kuhn em um segundo momento.

Como exemplo, pode-se considerar a ciência astronômica, aonde durante muitos anos acreditou-se na veracidade do paradigma geocêntrico, segundo o qual o Sol giraria em torno da Terra. Todos os cálculos matemáticos realizados sobre os movimentos dos planetas confirmavam que o paradigma geocêntrico era realmente correto, isto ocorria por causa das limitações dos problemas levantados. No entanto, em um certo momento, Copérnico (1473-1543) e outros astrônomos começaram a impor o paradigma heliocêntrico, argüindo que a Terra girava em torno do Sol e não o contrário. Mesmo assim, ainda por muitos anos o paradigma geocêntrico resistiu e seus constatadores sofriam com a rejeição social, acusações de heresia e, até mesmo, perda da própria vida. Só depois de muito conflito, a mudança de paradigma ocorreu com a substituição do paradigma geocêntrico pelo paradigma heliocêntrico (KUHN, 1970).

Na ciência da computação, a resistência à mudança de paradigmas é menos drástica do que na ciência física, pois ambas as ciências diferem em relação ao “mundo” que fundamenta o modelo. Nas ciências físicas, um modelo é formado por constatações físicas sobre o mundo, enquanto na ciência da computação (até então) o modelo é principalmente fruto da criatividade humana, a qual só depende dos padrões usados pelo homem para enxergar os problemas computacionais de diferentes perspectivas.

Desta forma, enquanto nas ciências físicas é inadmissível a coexistência de dois ou mais modelos diferentes em um mesmo tempo, na ciência da computação este fato é totalmente aceitável. Isto ocorre, porque só o homem pode governar a sua própria mente, assim não se pode obrigar um programador a pensar conforme uma dada perspectiva imposta por um paradigma. Na verdade, esta coexistência é até incentivada na ciência da computação, uma vez que, por causa do aumento da complexidade dos problemas a serem resolvidos computacionalmente, é uma vantagem contar com diferentes formas de enxergar um problema (ROY e HARIDI, 2004).

Esta diferença entre as mudanças de paradigma na ciência física e na ciência da computação pode ser observada pela interpretação da gravura apresentada na Figura 6. Esta obra de Giovanni Battista Riccioli, chamada *Almagestum Novum* (RICCIOLI, 1651) representa a musa da astronomia, Urânia, pesando dois paradigmas rivais, o paradigma geocêntrico e o heliocêntrico. Esta representação é contestada fortemente por Kuhn, pois ele

afirma que dois paradigmas não podem ser pesados, pois o paradigma emergente surge para refutar totalmente o paradigma anterior, apresentando visões totalmente opostas do mundo (KUHN, 1970). Porém, na ciência da computação é admissível a coexistência de diferentes paradigmas e seus usos são pesados de acordo com as características de cada problema computacional a ser resolvido (ROY e HARIDI, 2004).

Figura 6: Coexistência entre paradigmas (RICCIOLI, 1651)

Apesar de Kuhn contestar a co-existência entre os paradigmas nas ciências físicas, muitos paradigmas apresentam alguns conceitos em comum. Isto se deve porque os paradigmas propostos quase sempre mantêm alguns conceitos dos paradigmas anteriores, assim como ilustra a Figura 7. Este reaproveitamento de conceitos ocorre porque nem sempre uma comunidade científica está totalmente errada em relação aos seus estudos. Na verdade, este fato é até mesmo confirmado pelas próprias palavras de Kuhn, "... toda vez que o paradigma do qual derivam deixa de funcionar efetivamente, pelo menos parte dessas realizações sempre demonstra ser permanente" (KUHN, 1970).

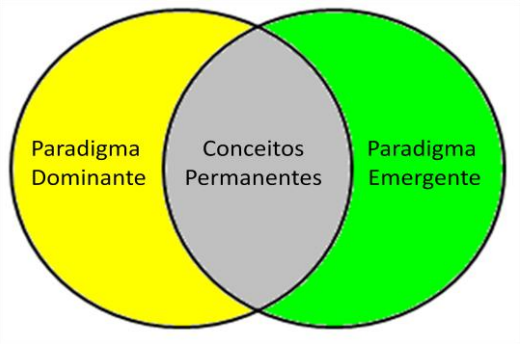


Figura 7: Evolução entre paradigmas

Na ciência da computação, este reaproveitamento de conceitos entre os paradigmas ocorre da mesma forma. Geralmente, os paradigmas propostos (emergentes) reaproveitam características ou, ao menos, alguns elementos de outros paradigmas. Isto ocorre porque o

autor do novo paradigma certamente apresenta conhecimentos sobre alguns outros paradigmas, o que influencia a elaboração do seu modelo. Esta influência se mostra benéfica, uma vez que o contato do autor com os paradigmas dominantes permite-o manter as melhores características e refutar as ruins, propondo melhorias relativas a estas últimas.

Neste âmbito, para entender como os paradigmas de programação evoluíram através do tempo (“mudança de paradigmas”), é necessário descrever a evolução das linguagens de programação. Isto se faz necessário porque as linguagens estão intimamente relacionadas aos paradigmas, uma vez que as linguagens materializam os conceitos dos paradigmas de programação.

2.4 EVOLUÇÃO DOS PARADIGMAS DE PROGRAMAÇÃO POR MEIO DAS LINGUAGENS DE PROGRAMAÇÃO

As linguagens de programação evoluem através do tempo. Linguagens já existentes evoluem agregando novas características e novas linguagens surgem apresentando características completamente diferentes das habituais de acordo com os conceitos de um novo paradigma de programação (BERGIN e GIBSON, 1996).

Conforme os ensinamentos de Kuhn, a ciência só evolui significativamente quando um novo paradigma é proposto, o mesmo se repete no domínio da programação. As linguagens até evoluem através de evoluções incrementais, com melhoramentos ou propostas de novas linguagens sobre os princípios de um paradigma, mas estes melhoramentos não alteram de forma significativa a forma de conceber e/ou executar programas. As melhorias de maior impacto ocorrem quando novos paradigmas de programação são propostos e as linguagens os materializam (KAISLER, 2005).

As melhorias incrementais (de menor impacto) e mesmo as de maior impacto têm ocorrido, em geral, com o objetivo de aumentar o nível de abstração na programação. Neste sentido, segundo a lista das linguagens de programação criada por Kinnersley (2008), cerca de 2500 linguagens de programação já foram propostas contribuindo de alguma forma para melhorar a atividade de programação de sistemas computacionais, cada qual com diferentes impactos.

As propostas destas linguagens podem ser enquadradas em cinco diferentes gerações, onde cada geração contribuiu ou ainda contribui significativamente para o aumento do nível

de abstração. A evolução das linguagens conforme as cinco gerações está representada na Figura 8.

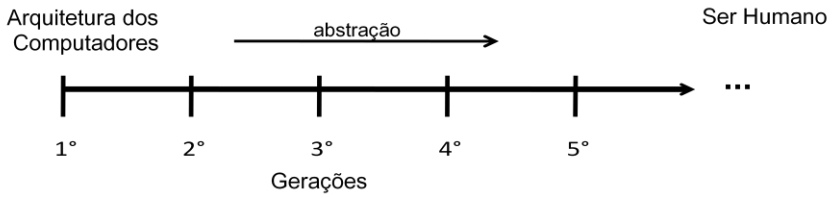


Figura 8: Linha evolutiva das linguagens de programação, baseada em (BROOKSHEAR, 2006)

A linha caracteriza a evolução das linguagens durante as últimas décadas no sentido de alterar gradualmente a arte de programar a partir de termos da arquitetura de máquina para uma forma mais compreensível ou intuitiva ao ser humano (BROOKSHEAR, 2006).

Assim, para compreender a evolução das linguagens de programação e, conseqüentemente, dos paradigmas de programação, as próximas seções contemplam, de forma sucinta, o surgimento das linguagens de programação que contribuíram significativamente por materializar novas formas de conceber programas. As subseções estão estruturadas de acordo com as cinco gerações.

2.4.1 Primeira geração (1940)

Em meados de 1940, no início do advento dos computadores eletrônicos, a forma de programação era bastante precária, tornando-se uma atividade tediosa e bastante árdua. Os programadores precisavam compreender os detalhes da arquitetura do computador e adaptar a forma de pensar nos problemas computacionais de acordo com o processo de execução das instruções pelos computadores.

Basicamente, um computador funciona recuperando as instruções de sua memória, interpretando estas instruções para finalmente executá-las nos seus meios de processamento (HENNESSY e PATTERSON, 2007). Nesta geração, as instruções eram escritas pelos programadores com menção íntima a esta seqüência de passos de execução por meio da mesma linguagem compreendida pelos computadores, ou seja, a linguagem de máquina. A linguagem de máquina pode ser considerada como a primeira linguagem de programação, na qual a programação ocorre de maneira imperativa pela manipulação seqüencial de dados na memória do computador.

A linguagem de máquina é constituída por um conjunto de códigos binários ou octais (evoluindo mais tarde para hexadecimais) usados para manter controle sobre um processador. Através destes códigos é possível realizar operações acessando diretamente as instruções de um processador (BROOKSHEAR, 2006). Na verdade, este tipo de programação ocorre na computação atual, entretanto, ela é geralmente ocultada por camadas de linguagem de mais alto nível.

O código apresentado na primeira coluna da Tabela 1, mesmo que em versão mais recente, ilustra a precariedade desta linguagem para fins de programação. Neste trecho com 5 linhas de código escritas em notação hexadecimal, o programador acessa os conteúdos da memória endereçados por 6C e 6D para somar estes valores e armazenar o resultado no endereço de memória 6E (BROOKSHEAR, 2006).

Tabela 1: Exemplo de código em linguagem de máquina (BROOKSHEAR, 2006)

156C	Carrega no registrador 5 o conteúdo do endereço de memória 6C
166D	Carrega no registrador 6 o conteúdo do endereço de memória 6D
5056	Soma os valores dos registradores 5 e 6 e põe o resultado no registrador 0
306E	Move o valor do registrador 0 para o endereço de memória 6E
C000	Finaliza o programa

A programação por meio da manipulação direta das instruções é tendente a erro e improdutiva devido à inadequação da mente humana em manipular centenas de cadeias de bits, gerando muita insatisfação nos programadores. Desta forma, os programadores aspiravam por propostas de novas linguagens de programação que provessem maiores facilidades e que permitissem construir programas de maior complexidade.

2.4.2 Segunda geração (1950)

No início da década de 1950, devido à insatisfação dos programadores com a linguagem de máquina, a linguagem Assembly foi proposta. A Assembly facilitou a programação propondo o uso de abreviaturas de palavras no idioma inglês como mnemônicos para representar as instruções de um processador em linguagem de máquina (PIROGOV, 2005). A representação do código em Assembly, em uma versão “moderna”, é apresentado na Tabela 2 para executar a mesma operação apresentada na Tabela 1.

Tabela 2: Exemplo de código em Assembly (BROOKSHEAR, 2006)

LD R5, varA	Carrega no registrador 5 o conteúdo da variável varA
LD R6, varB	Carrega no registrador 6 o conteúdo da variável varB
ADDI R0, R5 R6	Soma os valores dos registradores 5 e 6 e põe o resultado no registrador 0
ST R0, varTotal	Move o valor do registrador 0 e armazena na variável varTotal
HLT	Finaliza o programa

Como pode-se observar, o grau de correspondência entre os mnemônicos e as instruções em linguagem de máquina é de um para um, os quais eram traduzidos para o código de máquina por meio do Assembler ou "montador", que vem a ser o precursor dos atuais compiladores (HYDE, 2003).

A tentativa de aproximação das linguagens de programação ao dialeto humano (usando mnemônicos) com a criação da linguagem Assembly foi um avanço em relação às linguagens de máquina, mesmo se atualmente ela é classificada como de baixo-nível. Assembly é uma linguagem de baixo-nível porque apresenta um meio-termo entre a precariedade das linguagens de máquina e a sofisticação das linguagens de programação atuais, as quais são classificadas como de alto-nível (ROY e HARIDI, 2004).

2.4.3 Terceira geração (1954)

A terceira geração foi marcada pela proposta das linguagens de alto-nível, que oferecem comandos de manipulação mais amigáveis e de forma que não seja necessário um conhecimento aprofundado da arquitetura dos computadores. Com isto, mudou-se a forma de programar e deu-se início ao melhoramento contínuo em busca de novas formas de resolver problemas computacionais, principalmente em busca daquela que seja mais próxima a cognição humana (BROOKSHEAR, 2006).

Em 1954, com o propósito de oferecer uma linguagem com maior abstração do que a linguagem Assembly, John Backus criou a linguagem FORTRAN (*FORmula TRANslation* - Tradução de Fórmulas) para a IBM (BACKUS, BEEBER e GOLDBERG, 1957). Na mesma época, surgiram outras linguagens (e.g. COBOL e ALGOL) com este mesmo propósito, o de obter maior nível de abstração. Estas linguagens deram início à proposta das linguagens imperativas propriamente ditas, sendo que as linguagens de máquina e Assembly são consideradas como linguagens imperativas puras, devido às suas proximidades com o *hardware*.

As linguagens imperativas propriamente ditas compartilhavam a mesma essência das linguagens imperativas puras. Elas manipulavam dados armazenados na memória do computador por meio de uma seqüência única de comandos (e.g. comandos de atribuição e condicionais). Posteriormente, esta seqüencialidade foi contornada pelos programadores por meio do uso do comando de desvio *goto* (SCHILDT, 1997).

O comando *goto* foi empregado na implementação de laços de repetição ou mesmo na execução de uma determinada linha de código fora do fluxo de execução estabelecido. Porém, o uso abusivo deste comando tornava difícil a leitura do código por causa dos desvios na seqüencialidade, sendo o código comumente e coloquialmente referenciado como "código macarronada" (*spaghetti code*) (DIJKSTRA, 1968).

Devido a este problema, estas linguagens são classificadas como Linguagens Não-Estruturadas. A solução para esse problema das Linguagens Não-Estruturadas surgiu em meados da década de 60 com a proposta das Linguagens Estruturadas como PASCAL, ADA, BASIC e C. Estas contribuíram com maior organização na estrutura dos programas por meio do emprego de estruturas de repetição, estruturas de decisão e funções voltadas à modularização do código. Isto permitiu maior produtividade, qualidade e facilidades de manutenção (BROOKSHEAR, 2006).

Entretanto, mesmo se as Linguagens Estruturadas apresentam melhores mecanismos de abstração e codificação do que as Não-Estruturadas, essencialmente ambas se referem a uma camada de abstração sobre a linguagem de máquina ou Assembly. Esta abstração foi proporcionada pelo advento dos compiladores, os quais têm a função de traduzir um comando destas linguagens para vários das linguagens de máquina ou Assembly, com a vantagem de prover certa portabilidade entre diferentes arquiteturas de processadores (BERGIN e GIBSON, 1996)[7].

Este aumento no nível de abstração pode ser observado através da análise comparativa do trecho de código apresentado na Figura 9. Este trecho corresponde a uma expressão de atribuição dentro de um bloco *se-então* implementado na linguagem C, onde esta única expressão corresponde às várias instruções apresentadas na Tabela 1 e Tabela 2.

[7] Neste âmbito, mais tarde surgiram os interpretadores que permitiram traduzir programas de certas linguagens para uma máquina virtual que consiste em uma abstração e simulação em *software* dos recursos reais do *hardware*.

```
1...
2if((a > 0) && (b > 0))
3{
4  total = a + b;
5}
6...
```

Figura 9: Exemplo de código em linguagem imperativa

As linguagens de alto-nível, sobretudo as estruturadas, tornaram a programação mais intuitiva e simples em relação às tecnologias anteriores. Entretanto, apesar das melhorias, a essência destas linguagens continua a mesma das anteriores, ou seja, o programador continua a organizar as expressões de forma seqüencial e se preocupando com questões intrínsecas à implementação, tal como o gerenciamento da memória e manipulação direta das variáveis (BROOKSHEAR, 2006). Isto gera alguns inconvenientes como a maior probabilidade de erros na programação e menor produtividade ao forçar o programador a lidar com as particularidades de implementação.

Em 1960, em paralelo às melhorias das linguagens convencionais (Estruturadas e Não-Estruturadas) surgia a linguagem LISP (abreviação de *LISt Processing* – Processamento de Listas) proposta por John McCarthy (MCCARTHY, 1960). LISP foi proposta como uma solução inovadora na programação, a qual permitiu melhoras significativas na abstração em relação às linguagens convencionais (e.g. recursividades, condicionais, *garbage collector*). Mais tarde, surgiram outras linguagens com os mesmos princípios de LISP, como a Miranda, Haskell, Sisal, pH e ML (BROOKSHEAR, 2006).

LISP e as linguagens similares permitiram programar de forma independente de certas particularidades algorítmicas (e.g. atribuições, declaração de variáveis e gerenciamento de memória por ponteiros), dando início à proposta das linguagens ditas declarativas. As linguagens declarativas oferecem maiores facilidades na programação, principalmente porque o programador se concentra mais no problema computacional e menos nos detalhes da implementação.

Neste âmbito, LISP é essencialmente uma linguagem para representação de uma certa classe de expressões simbólicas na forma de funções recursivas (MCCARTHY, 1960). Um programa em LISP é constituído basicamente de “funções” e dados (símbolos e números) que são armazenados e manipulados em listas encadeadas (SCOTT, 2000).

Na verdade, uma função em LISP é composta de dois elementos, o nome da função e os seus argumentos, os quais podem ser dados ou mesmo outras funções (GRAHAM, 2003). Uma função é delimitada por um parêntese de abertura e outro de fechamento, assim como apresentado na expressão à esquerda na Tabela 3.

Tabela 3: Exemplo de código em LISP e nas linguagens convencionais

```
1 ...
2 (+ (+ C D)(+ E F))
3 ...
```

```
1 ...
2 A = C + D;
3 B = E + F;
4 total = A + B;
5 ...
```

Esta expressão apresenta três funções aninhadas, aonde a função mais externa recebe duas funções como parâmetros a fim de somar os seus resultados. Neste exemplo, as funções internas somam separadamente os valores de *C* e *D* e de *E* e *F*. A mesma computação é apresentada na linguagem convencional C, à direita.

LISP se difere das linguagens convencionais (imperativas) em relação ao mecanismo de execução. As linguagens convencionais executam seqüencialmente atribuindo temporariamente o resultado de uma expressão a uma variável. Em LISP, não há explicitamente o conceito de atribuição de valores a variáveis, em vez disso, as funções internas quando avaliadas, imediatamente instigam a execução da função que as invocam por meio dos seus resultados (GRAHAM, 2003).

Visando preciosismo de explicação, esta forma particular de execução é apresentada na Figura 10. Esta figura representa a estrutura interna de execução do programa da Tabela 3, a qual ocorre no percorrer de listas encadeadas. Nesta solução, cada função é representada por uma lista particular, onde cada nó pode apontar para um dado (*C*, *D*, *E* ou *F*), outra função (+) ou mesmo para um endereço nulo (Ø).

Na figura, a lista do meio corresponde à função mais externa da expressão, na qual os seus dois argumentos são representados por referências a outras duas listas, as lista internas. Deste modo, as listas internas podem executar seqüencialmente ou mesmo de forma concorrente para retornar um valor que instigará a avaliação da função mais externa.

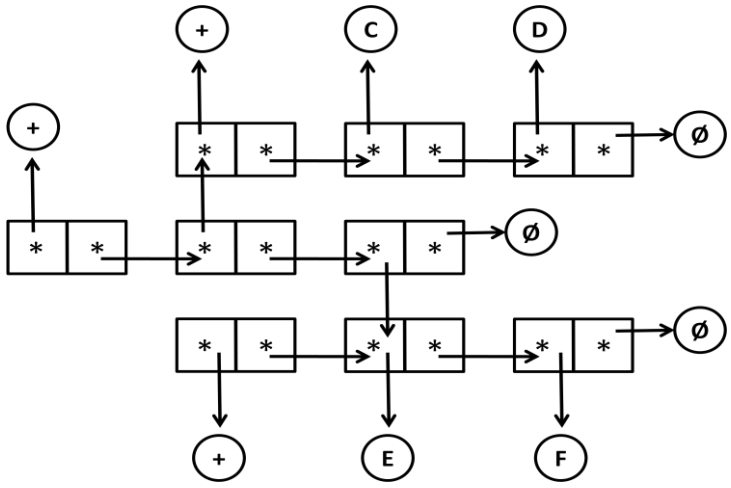


Figura 10: Mecanismo interno de execução do LISP (baseado em (QUEINNEC, 1996)

Ademais, em LISP, os laços de repetição, geralmente se dão por meio de funções recursivas aliadas a um critério de parada onde o programador não define explicitamente o laço. Isto é exemplificado de maneira mais particular com o programa de cálculo de fatorial em LISP (na forma declarativa) e em C (na forma imperativa), conforme descreve os trechos de código apresentados na Figura 11.

```
1 //Paradigma Declarativo - Exemplo em LISP
2 (defun fatorial (n) (if (<= n 1) 1 (* n (fatorial (- n 1)))))
3
4 //Paradigma Imperativo - Exemplo em C
5 int fatorial ( int n )
6 {
7     int resposta;
8     resposta = 1;
9
10    for ( int t = 1; t <= n; t++ )
11    {
12       resposta = resposta * t ;
13    }
14
15    return resposta;
16 }
```

Figura 11: Exemplo do fatorial na programação declarativa e imperativa

Como observado, o programa em LISP é mais elegante em termos de implementação, uma vez que adota o mecanismo de recursão ao invés de laços de repetição. No entanto, a programação mais elegante nem sempre é a mais indicada em situações onde desempenho é um fator importante.

Na recursão, uma função define algo em termos de si mesma, não necessitando guardar estados em variáveis locais ou mesmo alterar valores. Diferentemente, as funções que implementam laços de repetição devem necessariamente guardar estados em variáveis locais, devendo alterar estes estados a cada ciclo. Entretanto, apesar destas diferenças, a recursão e laços de repetição compartilham o mesmo propósito, ou seja, qualquer iteração implementada com laços de repetição pode ser reimplementada com recursão e vice-versa (SCOTT, 2000).

Na verdade, a recursão é mais natural à programação declarativa do que à programação imperativa. De forma oposta, os laços de repetição são mais naturais à programação imperativa, sendo que algumas linguagens declarativas são desprovidas deste mecanismo. Geralmente, mesmo que as atuais linguagens imperativas suportem ambos os mecanismos, os programadores destas linguagens usualmente adotam laços de repetição ao invés de recursão. No entanto, apesar da recursão se apresentar como uma solução mais elegante do que os laços de repetição, dependendo do *hardware* no qual o programa executa, ela pode apresentar desempenho inferior aos laços de repetição (SCOTT, 2000).

2.4.4 Quarta geração (1970)

Esta geração representa as linguagens de programação que ofereceram maior proximidade da forma de estruturar os problemas computacionais à cognição humana. Ela se refere basicamente a evoluções nas linguagens imperativas e declarativas.

Em 1967, no Centro de Computação Norueguês de Oslo, Ole-Johan Dahl e Kristen Nygaard projetaram a linguagem imperativa SIMULA (DAHL e NYGAARD, 1967) para permitir a construção de sistemas computacionais a partir da "simulação" do mundo real por meio de "objetos" computacionais (i.e. módulos coesos de *software*), uma vez que o ser humano percebe o mundo em termos de objetos (NEWELL e SIMON, 1972).

A linguagem SIMULA foi fundamentada nas Linguagens Estruturadas, principalmente na ALGOL, reutilizando os conceitos de blocos de comandos, funções, variáveis e atribuindo maior organização e coesão a estes conceitos através de entidades chamadas classes. Classes são moldes usados para a construção de objetos computacionais, chamados simplesmente de objetos (POO e KIONG, 2008). As classes são geralmente manipuladas pelos programadores por meio de uma linguagem de programação enquanto os objetos são entidades executáveis, sendo manipulados pelo computador. Em suma, um objeto é representado por meio de uma classe, onde os estados são tratados por atributos da classe (i.e. "variáveis") e os comportamentos por métodos da classe (i.e. "funções").

Naquela época, a proposta destes conceitos não apresentou grande repercussão, pois a precariedade da proposta clamava por alguns ajustes. Uma das precariedades dizia respeito ao forte acoplamento entre os objetos, onde qualquer objeto poderia acessar diretamente as variáveis dos outros (BERGIN e GIBSON, 1996).

Em 1970, o grupo de pesquisa coordenado por Alan Kay da Xerox PARC lançou a linguagem Smalltalk (KAY, 1996). Smalltalk estendeu os conceitos do SIMULA, fazendo ressurgir os conceitos propostos anteriormente com algumas melhorias, principalmente com a inclusão das propriedades de encapsulamento (ver seção 2.7.2) e variáveis de classe (i.e. variáveis *static*). Com isto, Smalltalk teve maior repercussão do que a linguagem SIMULA, contribuindo significativamente para alterar a forma de visualizar os problemas computacionais e evoluir as linguagens imperativas.

Mais tarde, outras linguagens de programação foram propostas a fim de materializar estes mesmos conceitos. Entre estas linguagens, estão a Object Pascal, C++ e Java, as quais ainda vigoram, sendo que estas representam as entidades do mundo real ou virtual por meio

de objetos. Para melhor elucidar estes conceitos, um exemplo do uso de objetos é apresentado na Figura 12, no trecho de código implementado na linguagem C++.

Neste, nas linhas 2, 3 e 4, há a declaração de quatro objetos, dois da classe A (i.e. a1 e a2), um da classe B (i.e. b1) e um da classe C (i.e. c1), respectivamente. Nas linhas 6, 7 e 8, os objetos da classe A e B têm os seus estados inicializados por meio da atribuição de um valor numérico. Como observado, o estado do objeto da classe C não foi inicializado, isto se explica porque o valor do seu estado depende da execução das expressões causais declaradas no escopo do método *main* (linhas 12 e 17), o método principal desta linguagem[8]. Basicamente, estas expressões causais atestam se os estados dos objetos da classe A e B apresentam o mesmo valor numérico. Caso isto seja confirmado, o estado do objeto da classe A é atribuído ao objeto da classe C.

```
1 ...
2 A a1, a2;
3 B b1;
4 C c1;
5
6 a1.setAtributo(1);
7 a2.setAtributo(2);
8 b1.setAtributo(2);
9 ...
10 int main()
11 {
12   if (a1.getAtributo() == b1.getAtributo())
13   {
14     c1.getAtributo() = a1.getAtributo();
15   }
16
17   if (a2.getAtributo() == b1.getAtributo())
18   {
19     c1.getAtributo() = a2.getAtributo();
20   }
21 }
22 ...
```

Figura 12: Exemplo de código em C++

Em paralelo a melhoria das linguagens imperativas, alguns estudos eram realizados na construção de uma nova linguagem declarativa, a linguagem PROLOG (*PROgrammation in LOGic*). Esta linguagem permitiria realizar deduções lógicas baseadas em texto escrito em linguagem natural (BERGIN e GIBSON, 1996).

No início de 1970, mais precisamente em 1972, Alain Colmerauer da Universidade de Aix-Marseille contribuiu com a construção da base da linguagem PROLOG. Ele projetou um sistema que compreendia e respondia, com falso ou verdadeiro, a perguntas elaboradas em linguagem natural, particularmente no idioma francês. Em 1974, a PROLOG foi

[8] O método *main* consiste no ponto de início de execução de um programa em C++. Se este método for omitido, um programa é impossibilitado de executar.

definitivamente concebida por meio da união dos trabalhos de Colmerauer com os de Robert Kowalski, da Universidade de Edimburgo, o qual realizava estudos sobre a dedução lógica automática (prova automática de teoremas) (COLMERAUNER e ROUSSEL, 1992).

PROLOG permite ao programador compor a solução computacional da mesma forma que ele organiza o seu conhecimento para raciocinar, ou seja, por meio de relações lógicas sobre coisas ou situações que ocorrem no mundo real (NEWELL e SIMON, 1972).

Em PROLOG, as relações lógicas são representadas por meio de regras. Uma regra consiste basicamente em uma expressão causal *se-então* (i.e. *if-else*), onde a parte *if* da regra é chamada de condição e a parte *then* é chamada de conclusão. Os objetos ou situações que ocorrem no sistema (representado ou idealizado) são computacionalmente chamados de elementos da base de fatos, sendo que o conjunto destes elementos constitui uma base de fatos. Os elementos da base de fatos representam pedaços úteis de informação de um sistema, sendo os seus estados chamados de fatos (FRIEDMAN-HILL, 2003).

Esta relação entre elementos da base de fatos e regras é mais bem compreendida através do exemplo apresentado na Figura 13. Basicamente, este exemplo corresponde a uma versão declarativa do código imperativo apresentado na Figura 12. Nesta versão declarativa, as mesmas entidades do exemplo imperativo (i.e. objetos e expressões causais) são representadas por uma única regra e três instâncias de elementos da base de fatos, correspondentes às classes A, B e C. Como no exemplo anterior, cada um destes elementos da base de fatos apresenta um único estado ou fato na forma de um valor numérico. Entretanto, um elemento da base de fatos em PROLOG não só representa estados, este pode representar também relações entre estados (situações). Por exemplo, *maior (B, A)*, o elemento da base de fatos chamado “maior” afirma o fato de que B é maior do que A.

No exemplo da Figura 13, a única regra realiza uma simples dedução lógica sobre os fatos. Nesta notação, a condição está à direita de “:-” e a conclusão está à esquerda de “:-”. A condição consiste em uma conjunção entre duas avaliações lógicas (separadas pela vírgula), as quais são chamadas de premissas da condição. As respectivas premissas atestam se duas instâncias quaisquer de elementos da base de fatos do tipo *A* e *B* apresentam o mesmo estado (fato) numérico *Y*. Se ambas as premissas forem satisfeitas, conclui-se que há um elemento da base de fatos do tipo *C* que apresenta o mesmo fato *Y* de *A* e *B*.

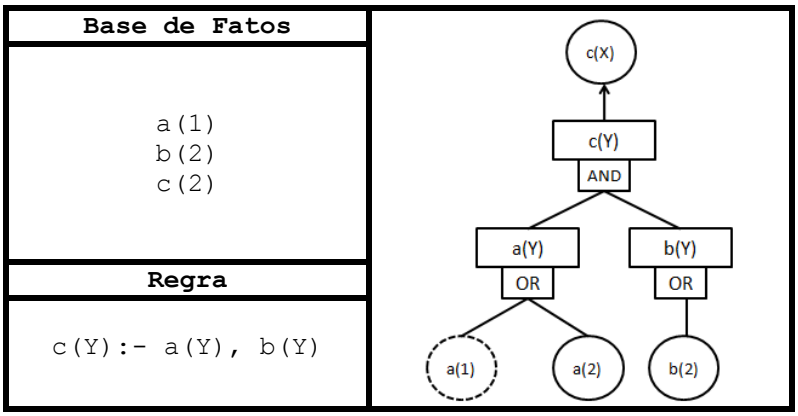


Figura 13: Código em PROLOG e a busca por profundidade com retrocesso (SCOTT, 2000)

Para realizar esta conclusão, PROLOG realiza um processo de busca exaustiva sobre os elementos da base de fatos e regras representados em forma da estrutura de dados de uma árvore, como ilustrado na Figura 13. A árvore consiste de níveis alternados de conjunções (AND) e disjunções (OR). A conjunção interliga as premissas de uma condição enquanto a disjunção interliga as instâncias de elementos da base de fatos que podem aprovar cada premissa.

PROLOG adota a busca por profundidade com retrocesso (*backtracking*), a qual é realizada sobre a estratégia de encadeamento para trás (*backward chainning*). Nesta estratégia, a busca se inicia com o conhecimento de um objetivo, uma incógnita ou indagação que geralmente é materializada na conclusão de uma regra, e prossegue com a satisfação das premissas subjacentes. Esta estratégia se difere da estratégia de encadeamento para frente (*forward chaining*), na qual a busca se inicia sem o conhecimento prévio do objetivo. Nesta, as premissas devem ser satisfeitas para derivar o objetivo ou a conclusão de uma regra.

Neste âmbito, visando maior preciosismo na explicação, pode-se considerar a busca por profundidade (com retrocesso) apresentada na Figura 13. Primeiramente, estabelece-se uma incógnita, por exemplo, encontrar o valor da variável *X* para *c(X)*. Esta incógnita é representada da seguinte maneira na execução do programa em PROLOG:

?- c(X)

Após esta definição, a busca ocorre em profundidade, da esquerda para a direita, sobre os elementos da base de fatos e a única regra do exemplo (somente sobre os elementos da base de fatos da conclusão) na tentativa de encontrar um elemento que satisfaça esta incógnita. Neste processo, a busca encontra o elemento da base de fatos *c(Y)* da única regra do exemplo:

c(Y):- a(Y), b(Y)

Neste caso, *Y* é atribuído a *X*. Como *Y* é uma variável que não armazena nenhum valor concreto, o processo de busca por *X* continua por meio da prova de *c(Y)*. A partir deste ponto, a incógnita a ser descoberta passa a ser *c(Y)* ao invés de *c(X)*.

O mecanismo interno do PROLOG percorre o nó *c(Y)*, passando a considerar a premissa mais à esquerda *a(Y)* como a próxima incógnita. Nesta fase, primeiramente tenta satisfazer a premissa atribuindo o valor do elemento da base de fatos *a(1)* ao *Y*, ou seja, *Y = 1*.

Após a validação da premissa *a(Y)*, considera-se a premissa *b(Y)* como a próxima incógnita. Desta forma, atribui-se o valor do elemento da base de fatos *b(2)* a *Y*, ou seja, *Y = 2*. Neste ponto, a condição da regra não é aprovada, pois *a(Y)* é diferente de *b(Y)*, quando ocorre o mecanismo do *backtracking*.

O *backtracking* retorna à incógnita anterior *a(Y)* para que ela seja avaliada com um outro valor. Assim, a partir do elemento da base de fatos *a(2)*, atribui-se o valor 2 ao *Y*, passando para a prova da incógnita *b(2)*. Desta vez, a condição é satisfeita e conclui-se que *c(Y)* é igual a *c(2)* e consecutivamente *c(X)* igual a *c(2)*.

Por meio deste exemplo prático, percebe-se que os fatos dos elementos da base de fatos são correlacionados automaticamente pelo mecanismo interno desta linguagem, demonstrando as facilidades de programação com a forma declarativa. Neste exemplo, observou-se que as regras são genéricas, as quais se referem às classes ao invés de objetos. Assim, apenas a declaração de uma regra é suficiente para tratar de qualquer quantidade de objetos da classe A e B.

No exemplo imperativo, apresentado na Figura 12, as mesmas facilidades da programação declarativa não são encontradas. No imperativo, as expressões causais devem ser definidas especialmente para cada conjunto de estados que se deseja correlacionar. Desta forma, conforme o exemplo do código imperativo, os esforços na programação aumentam à medida que a quantidade de objetos das classes A e B aumenta, pois os estados destes objetos devem ser correlacionados explicitamente pelo programador em cada uma das expressões causais.

Desta forma, constata-se por meio dos exemplos apresentados na Figura 12 e na Figura 13 que a programação declarativa demanda menos esforços de programação do que a programação imperativa. Entretanto, o preço a ser pago por esta facilidade pode estar no desempenho das aplicações, uma vez que as linguagens declarativas demandam buscas para

correlacionar as expressões causais e os estados em tempo de execução por meio de algoritmos assaz genéricos e estruturas de dados caras computacionalmente.

Além do mais, o programador que adota linguagens imperativas potencialmente se beneficia de maior flexibilidade na programação, principalmente para otimizar o código em termos de acesso a *hardware* e otimizações algorítmicas pontuais, a despeito das dificuldades que isto pode trazer.

2.4.5 Quinta geração (atualidade e futuro)

A Quinta Geração, diferentemente das anteriores, apresenta linguagens que não contribuem diretamente para aumentar o nível de abstração na programação. Na verdade, esta geração não é de senso comum na literatura. Não obstante, o autor desta dissertação considera esta geração como o atual estado da arte e a tendência da evolução das linguagens de programação.

Nesta geração, parte dos conceitos propostos nas linguagens até a quarta geração é selecionada e mesclada em novas linguagens de programação, por exemplo, no *framework* JESS, os elementos da base de fatos são representados como objetos implementados em Java para serem manipulados por regras (FRIEDMAN-HILL, 2003). Também nesta geração, novos conceitos são propostos e materializados sobre linguagens de programação já existentes, por exemplo, o *framework* JADE permite a implementação de agentes de *software* sobre a linguagem Java (BELLIFEMINE, CAIRE e GREENWOOD, 2007). Tais conceitos referem-se aos atuais paradigmas de programação, o imperativo e declarativo (de uma forma geral).

Nesta geração, o autor deste trabalho caracteriza as linguagens pelo suporte dos conceitos de mais de um paradigma. Sendo assim, elas são classificadas como linguagens multi-paradigmas. Entre estas linguagens, estão as linguagens próprias de um único paradigma (e.g. C++ e Java) que são estendidas para materializar novos paradigmas, bem como as linguagens multi-paradigmas propriamente ditas, as quais são projetadas especialmente para suportar paradigmas diferentes, como as linguagens Oz (ROY, 2005) e Scala (ODERSKY, ALTHERR, CREMET *et al*, 2004).

As linguagens multi-paradigmas são as mais adequadas diante do aumento da complexidade dos problemas computacionais, os quais geralmente são compostos de outros sub-problemas. Estas linguagens permitem empregar diferentes formas de resolver cada

problema, cabendo ao programador a escolha daquela que melhor se adapta ao problema. Este fato é confirmado em (ROY e HARIDI, 2004). Segundo Roy e Haridi (2004), nenhum paradigma é uma panacéia quando usado sozinho, alguns paradigmas se adaptam bem para alguns problemas e menos para outros.

Outra vantagem das linguagens multi-paradigmas é a menor resistência a mudanças pelos programadores aos conceitos de uma nova proposta de um paradigma. Este fato foi constatado em (MEYER, 1998), onde foi observada a melhor aceitação de novos paradigmas pelos programadores quando estes são materializados como extensões das linguagens de programação mais populares. Com isto, toda a experiência e a familiaridade com os comandos de uma linguagem são mantidas, facilitando a adoção dos conceitos de um novo paradigma. A constatação deste fato pode incentivar a proposta de novos paradigmas de programação.

2.5 CLASSIFICAÇÃO DAS LINGUAGENS EM PARADIGMAS DE PROGRAMAÇÃO

Cada uma das gerações na evolução das linguagens de programação contribuiu de alguma maneira para melhorar a forma de conceber os programas, evoluindo ou propondo diferentes paradigmas de programação. A evolução das linguagens em relação aos paradigmas de programação é ilustrada em ordem cronológica na Figura 14.

As linguagens em destaque referem-se àquelas que primeiro concretizaram os conceitos de um determinado paradigma (como visto na seção 2.4) e as demais, às evoluções incrementais sobre um mesmo paradigma. As linguagens multi-paradigmas não são explicitadas nesta figura, uma vez que estas apenas apresentam a união dos conceitos dos atuais paradigmas.

Atualmente, com a concordância de vários autores da área (WATT, 2004; SCOTT, 2000; BROOKSHEAR, 2006), há a coexistência de quatro principais paradigmas de programação, os quais são referenciados neste trabalho como Paradigmas Dominantes. Entre os Paradigmas Dominantes estão o Paradigma Procedimental (PP), o Paradigma Funcional (PF), o Paradigma Orientado a Objetos (POO) e o Paradigma Lógico (PL).

Além dos Paradigmas Dominantes, há ainda outros paradigmas que ainda não receberam o mesmo grau de importância. Estes paradigmas se referem a propostas recentes que contribuem com novas formas de estruturar o pensamento do programador na concepção de programas. Estes paradigmas apresentam grandes expectativas devido às suas qualidades.

Deste modo, são referenciados como Paradigmas Emergentes. Entre os Paradigmas Emergentes estão o Paradigma Orientado a Agentes, o Paradigma Orientado a Aspectos e o Paradigma Orientado a Componentes.

Os Paradigmas Emergentes são geralmente materializados sobre as linguagens próprias dos Paradigmas Dominantes, como C++ e Java, para facilitar a aceitação dos conceitos propostos. Apesar de alguns destes paradigmas apresentarem linguagens próprias, este trabalho considera que os seus conceitos são mais comumente aplicados sobre as linguagens próprias dos Paradigmas Dominantes. Desta forma, a Figura 14 ilustra a classificação das linguagens de programação em paradigmas, não fazendo menção às linguagens dos Paradigmas Emergentes.

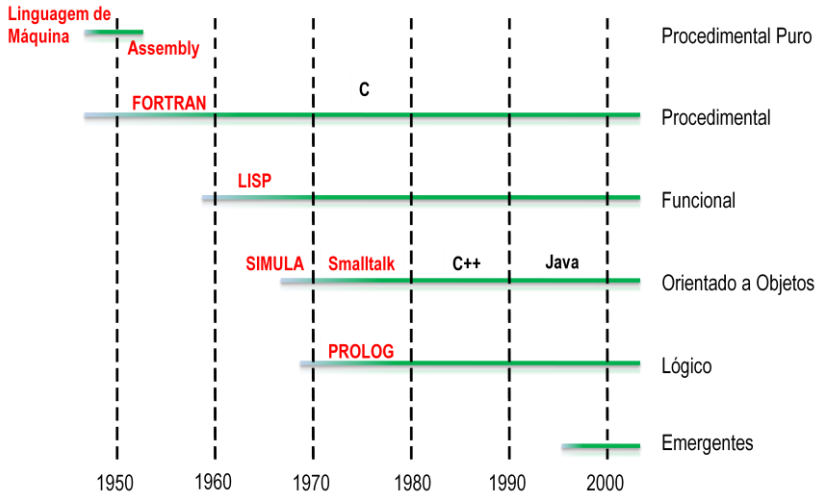


Figura 14: Evolução das linguagens em relação aos paradigmas de programação, baseado em (BROOKSHEAR, 2006)

2.6 CLASSIFICAÇÃO DOS ATUAIS PARADIGMAS DE PROGRAMAÇÃO

No âmbito deste trabalho, em concordância com os autores da literatura, os atuais paradigmas de programação (Dominantes e Emergentes) são considerados como subconjuntos de dois paradigmas maiores, o Paradigma Imperativo (PI) e o Paradigma Declarativo (PD), assim como ilustra a Figura 15. Desta forma, no escopo deste trabalho, mesmo entendendo que o PI e o PD são classificações dos paradigmas propriamente ditos, os termos PI e PD são empregados comumente no texto a fim de se referir aos subconjuntos, orientando-se por suas características comuns.

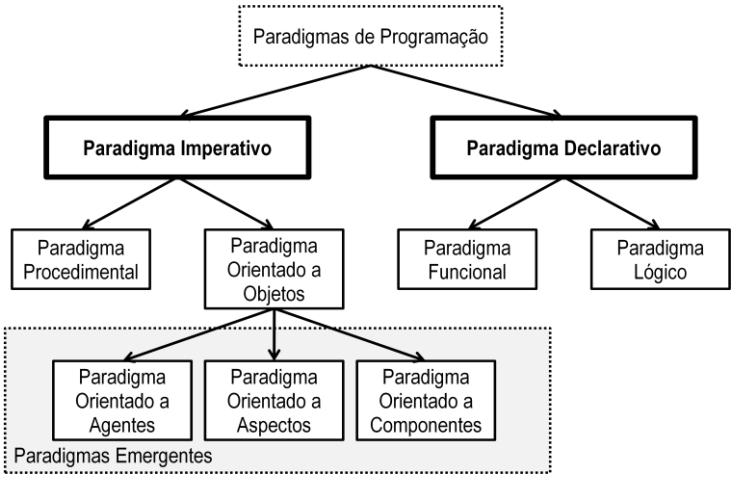


Figura 15: Classificação dos atuais paradigmas

Essencialmente, o Paradigma Imperativo é constituído pelo PP e POO, os quais são caracterizados pela interação mais próxima do programador para com o computador. O PI também é caracterizado pela forma seqüencial na qual as instruções são processadas pelo *hardware* do computador. Neste ponto, o PP e o POO são similares, a diferença está essencialmente na representatividade e na organização das instruções.

O Paradigma Declarativo, por sua vez, é constituído essencialmente pelo PF e PL, os quais são caracterizados pela interação mais simplificada do programador com o computador por meio de mecanismos que escondem as particularidades de implementação. No Paradigma Declarativo, o programador se concentra mais na organização do conhecimento sobre a resolução do problema computacional do que na implementação do mesmo. Este é um ponto positivo do PD em relação ao PI. Porém, para oferecer estas facilidades o PD perde em velocidade de execução para o PI e em certas flexibilidades, principalmente em relação a otimizações algorítmicas e facilidades de acesso ao *hardware* (SCOTT, 2000).

A classificação dos paradigmas em PD e PI ainda gera controvérsias entre alguns autores da literatura, principalmente em relação à classificação do POO (conceitos materializados em linguagem como Smalltalk, C++ e Java) como um PI. Alguns autores, como Friedman-Hill (2003), classificam o POO efetivamente como PI devido à flexibilidade oferecida e pela íntima relação com o PP (conceitos materializados nas linguagens estruturadas e não-estruturadas), enquanto outros autores como Giarratano e Riley (1993), classificam POO como um tipo do PD devido à maior abstração oferecida por este paradigma.

Devido à falta de consenso entre estes autores, o presente trabalho classifica os paradigmas de acordo com o ponto de vista do seu autor. Este trabalho considera o POO como uma evolução melhorada dos princípios utilizados nas linguagens estruturadas do PP, pois os programadores escrevem o código do programa de forma similar, se diferindo,

basicamente, na forma pela qual o código é estruturado e abstraído. Em suma, o POO apresenta mecanismos mais ricos de abstração e estrutura, mas o código é elaborado de maneira efetivamente imperativa. Desta forma, o POO não poderia ser classificado como um PD.

De um ponto de vista parecido, os Paradigmas Emergentes também são classificados como PI. Nesta classificação, considerou-se que estes paradigmas são materializados com maior freqüência sobre os conceitos do POO. No entanto, esta classificação é particular ao contexto deste trabalho, pois há soluções destes paradigmas relacionadas a outros Paradigmas Dominantes.

2.7 PARADIGMAS IMPERATIVOS

Esta seção apresenta os Paradigmas Imperativos relevantes (i.e. PP e POO) com maiores detalhes e relata sobre as suas principais deficiências. Conforme dito anteriormente, estes paradigmas são caracterizados pela relação mais próxima (quanto à imperatividade) à seqüência pela qual as instruções são processadas pelos computadores.

Dentre os dominantes, o PP foi o primeiro paradigma de programação proposto, evoluindo posteriormente para formas mais avançadas como o POO. Atualmente, o POO é o paradigma mais popular, o que motiva que paradigmas emergentes disponibilizem seus conceitos sobre ele.

Neste âmbito, esta seção descreve brevemente o PP, o POO e três Paradigmas Emergentes e aponta as deficiências comuns entre estes paradigmas, as quais se apresentam porque estes compartilham os mesmos princípios do (supra) Paradigma Imperativo.

2.7.1 Paradigma Procedimental

O Paradigma Procedimental (PP) define um estilo de programação baseado na organização seqüencial de variáveis e comandos. As variáveis representam os estados das entidades (abstratas ou relativas ao mundo real) e os comandos executam ações sobre estes estados (BROOKSHEAR, 2006), sendo as variáveis e comandos organizadas em funções e procedimentos, permitindo alcançar grau significativo de modularidade no código.

O Paradigma Procedimental foi o paradigma mais utilizado até a década de 90. Por causa desta supremacia duradoura, muitos dos sistemas comerciais atuais foram implementados com linguagens procedimentais e ainda tantos outros continuam sendo desenvolvidos nestas linguagens (KING, 2008). Inclusive, esta forma procedimental de pensar permitiu gerar as primeiras técnicas de análise de *software* antes de programá-lo efetivamente, dando assim, fundações à chamada Engenharia de *Software*.

O Paradigma Procedimental ainda vigora pelos motivos de considerável modularização, certa simplicidade de programação e relativa eficiência oferecida nos programas decorrentes, uma vez que a forma de codificação dos comandos condiz com a forma pela qual os mesmos são executados pela máquina. Este fato facilita a geração de executáveis enxutos e eficientes. Outro fator que contribui para a permanência deste paradigma é a experiência adquirida há anos pelos programadores neste tipo de linguagem. Entretanto, subseqüentemente e mais recentemente, o PP perdeu espaço considerável para o POO. Alguns exemplos de linguagens baseadas no Paradigma Procedimental são FORTRAN, COBOL, ALGOL, BASIC, PASCAL e C.

2.7.2 Paradigma Orientado a Objetos

2.7.2.1 Visão geral

A mente humana compreende a maioria dos problemas do mundo real e mesmo abstrato por meio da interação entre os objetos que o compõem. Pelo fato de muitos dos problemas solucionáveis computacionalmente estarem intrinsecamente relacionados ao mundo composto por objetos, o Paradigma Orientado a Objetos (POO) propõem a concepção de sistemas computacionais por meio da composição e relacionamentos entre entidades computacionais chamadas de objetos (POO e KIONG, 2008). Os objetos do POO abstraem os objetos do mundo, apresentando somente as características pertinentes para a implementação das soluções computacionais.

Objetos são compreendidos como elementos ativos (e.g. pessoa e pássaro) ou inativos (e.g. mesa e porta) que compõem o mundo real, o abstrato e o abstrato-informático (e.g. arquivo, tabela e botão de uma interface gráfica). Estes objetos são munidos de estados e comportamentos que agem sobre seus próprios estados (e.g. a *pessoa* X estava *sentada* e agora está *em pé*) ou agem sobre o comportamento de outros objetos (e.g. *pessoa* X abriu a

porta Y). Dois ou mais objetos podem apresentar os mesmos estados e comportamentos, parecendo idênticos, mas eles se diferem pela existência da uma identidade (POO e KIONG, 2008).

Mesmo que todos os objetos sejam diferentes em termos de estados e comportamentos, alguns deles podem ainda apresentar características em comum, tal como o compartilhamento da estrutura do objeto. A definição da estrutura do objeto é definida por meio de uma classe. Desta forma, classes e objetos apesar de serem conceitos facilmente confundidos, divergem na forma que são utilizados (POO e KIONG, 2008).

Além do conceito de classes e objetos, o POO oferece outros conceitos ou propriedades que auxiliam na construção de programas, como o encapsulamento, a herança e o polimorfismo (POO e KIONG, 2008):

- **Encapsulamento:** o encapsulamento é o ocultamento de uma seleção de atributos e métodos dos objetos para o acesso externo. Esta propriedade evita que certas partes de um objeto se tornem desnecessariamente acessíveis a outros.
- **Herança:** a herança permite reusar a estrutura de classes já definidas em classes derivadas, contribuindo para reduzir a quantidade de código a ser elaborado pelo programador. A herança é uma solução típica para prover o reuso.
- **Polimorfismo:** o polimorfismo refere-se à definição de métodos que aparentemente se apresentam similares, mas que apresentam comportamento diferente segundo o subtipo de cada objeto.

Com a proposta destas propriedades, o POO se difere do PP na questão organizacional, ou seja, na separação de atributos e métodos em uma estrutura modular. Na verdade, por meio das estruturas e comandos de uma linguagem do PP, como a linguagem C, é possível simular a estrutura e comportamento de um objeto do POO, similarmente como ocorre em uma linguagem típica do POO, como a linguagem C++. Mesmo assim, esta similaridade não afeta a independência entre os paradigmas.

Este fato pode ser constatado nas duas versões (em C e em C++) de um programa que calcula a idade de pessoas, conforme ilustra a Figura 16. Mais especificamente, à esquerda na figura é apresentada a versão do código em C, enquanto que à direita está a versão em C++. A versão em C foi implementada com o uso da entidade estrutura (i.e. *struct*), a qual consiste em uma coleção de estados (i.e. variáveis) a fim de representar uma entidade mais complexa ou mais abstrata. Apesar de permitir representar entidades e estados complexos do mundo real e abstrato, uma estrutura não suporta a definição de funções. Esta característica, entre outras

(e.g. encapsulamento, herança, polimorfismo), diferencia esta entidade modular em relação às classes do POO. Assim, uma classe também se apresenta como uma coleção, mas com a capacidade de organizar dados e também funções (LAFORE, 2002).

Desta forma, na versão em C, uma função (e.g. *calculaIdade*) é declarada fora do escopo da estrutura, devendo receber o endereço da respectiva estrutura a fim de atuar sobre os seus estados. Diferentemente, na versão em C++, as funções de uma classe (chamadas de métodos) acessam os seus estados (chamados de atributos) sem a necessidade destes serem passados como parâmetros.

```c
#include <stdio.h>

struct Pessoa
{
    int diaNascimento;
    int mesNascimento;
    int anoNascimento;
    int idade;
};

...

int CalculaIdade ( struct Pessoa p, int ano )
{
     return ano - p.ano;
}

...

int main()
{
   struct Pessoa Einstein, Newton;

   Einstein.dia  = 14;
   Einstein.mes  = 3;
   Einstein.ano  = 1879;

   Newton.dia    = 4;
   Newton.mes    = 1;
   Newton.ano    = 1643;

   Einstein.idade = calculaIdade ( Einstein, 2009 );
   Newton.idade  = calculaIdade ( Newton, 2009 );

   printf("Idade de Einstein: %d \n", Einstein.idade);
   printf("Idade de Newton: %d  \n", Newton.idade);

   getchar();
   return 0;
}
```

```c
#include <stdio.h>

class Pessoa
{
  private:
    int diaNascimento, mesNascimento, anoNascimento;
    int idade;

    // Esta é uma função construtora.
    // Uma construtora inicializa variáveis
    Pessoa (int diaNasc, int mesNasc, int anoNasc)
    {
        diaNascimento   = diaNasc;
        mesNascimento   = mesNasc;
        anoNascimento   = anoNasc;
    }

    // A função está dentro da classe ou estrutura
    void calculaIdade(int anoCorrente)
    {
        return  anoCorrente - anoNascimento;
    }

    // Função que retorna a idade da Pessoa
    // devido a propriedade de encapsulamento
    int getIdade()
    {
         return idade;
    }
};

int main()
{
    Pessoa Einstein (14, 3, 1879);
    Pessoa Newton (4, 1, 1643);

    Einstein.calculaIdade (2009);
    Newton.calculaIdade (2009);

    printf("Idade de Einstein: %d \n", Einstein.getIdade());
    printf("Idade de Newton: %d  \n", Newton.getIdade());

    getchar();
    return 0;
}
```

Figura 16: Simulação de uma classe do Paradigma Orientado a Objetos com uma linguagem do Paradigma Procedimental

Na verdade, apesar do PP se confundir com o POO no exemplo explicitado, estes ainda se diferem na essência da programação. No PP, a essência da programação é decompor o programa em uma coleção de variáveis, estrutura de dados e funções, enquanto que em POO a decomposição ocorre em termos mais abstratos, ou seja, em classes e no emprego de propriedades como as supra explanadas.

Desta forma, o POO se torna um paradigma independente do PP, sendo considerado atualmente o paradigma mais utilizado na construção de sistemas computacionais, segundo a

pesquisa TIOBE (TIOBE, 2008). Exemplos de linguagens que materializam este paradigma são a SIMULA, Smalltalk, C++, JAVA e C#.

A pesquisa TIOBE, baseada no fluxo mensal de pesquisas realizadas por meio das máquinas de buscas mais conhecidas (e.g. Google, MSN, Yahoo, YouTube), apresenta estatísticas sobre a popularidade das linguagens e paradigmas de programação perante a comunidade da computação. A pesquisa do mês de outubro de 2008 é ilustrada na Tabela 4.

Segundo as estatísticas apresentadas na Tabela 4, o POO é o paradigma mais popular da atualidade, atingindo quase 58% de interesse pela comunidade. Segundo a mesma figura, esta popularidade é tendente ao crescimento, pois ela apresenta o crescimento de 4% da popularidade do POO em relação ao mesmo período do ano anterior. Ao analisar os demais paradigmas, verifica-se a queda de interesse em seus conceitos no mesmo período.

Tabela 4: Popularidade entre os paradigmas (TIOBE, 2008)

Categoria	**Popularidade em Outubro de 2008**	**Variação para Outubro de 2007**
Paradigma Orientado a Objetos	57.6%	+4.0%
Paradigma Procedimental	39.6%	-3.1%
Paradigma Funcional	1.9%	-0.1%
Paradigma Lógico	0.9%	-0.8%

Ademais, a programação orientada a objetos é suportada atualmente por um conjunto de linguagens e processos de modelagem, fiéis ao POO, que antecedem a atividade de programação no chamado ciclo de engenharia de *software*. Dentre elas, a linguagem mais popular é a UML (*Unified Modeling Language* – Linguagem de Modelagem Unificada) (PENDER, 2003) e o processo de desenvolvimento mais popular é o Processo Unificado (*Unified Process*) (KRUCHTEN, 2003).

2.7.2.2 Programação Orientada a Objetos e Dirigida a Eventos

Apesar do POO se diferenciar estruturalmente do PP, ambos compartilham o mesmo mecanismo de execução. Em ambos os paradigmas, os comandos são avaliados seqüencialmente. Por exemplo, o código em C++ e o diagrama de seqüência da UML na Figura 17 ilustram o fluxo de execução do objeto A que invoca seqüencialmente quatro métodos do objeto B.

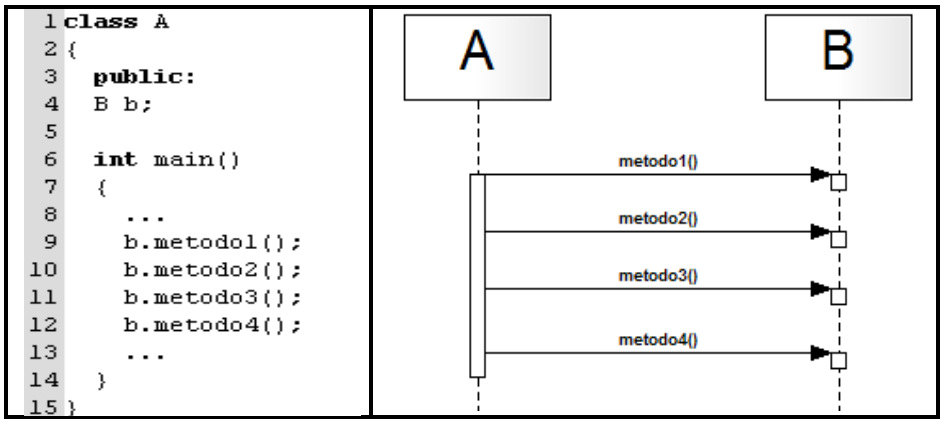


Figura 17: Invocação explícita e seqüencial de métodos

Neste exemplo, os métodos são invocados explicitamente de acordo com a seqüência de execução. No entanto, alternativamente, os métodos também poderiam ser invocados por meio da ocorrência de um evento, desrespeitando esta seqüência de execução (BROOKSHEAR, 2006). Este estilo de programação é chamado de Programação Dirigida a Eventos (PDE) e é freqüentemente vista como um subtipo do POO.

Na PDE, os objetos interagem guiados pela ocorrência de um evento. Um evento consiste em uma condição detectável que pode instigar a execução de um método. O objeto que detecta a ocorrência do evento é chamado de objeto transmissor, pois este transmitirá as informações do evento detectado para os objetos interessados, os quais são chamados de objetos receptores (FERG, 2006).

A relação entre os objetos transmissores e receptores podem ocorrer de forma direta ou indireta, conforme ilustra a Figura 18. À esquerda da figura, um objeto transmissor detecta a ocorrência de um evento e notifica este evento diretamente a um ou vários objetos receptores interessados[9]. À direita na mesma figura, um objeto transmissor detecta a ocorrência de um evento e notifica este evento indiretamente a um ou vários objetos receptores por intermédio do objeto *Dispatcher*[10].

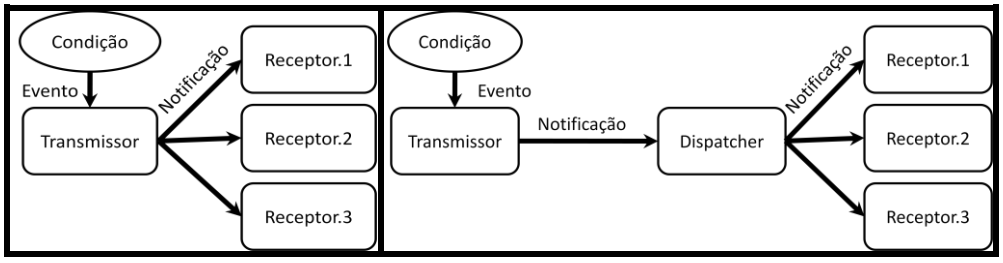


Figura 18: Processo de detecção de eventos (FAISON, 2006)

[9] Do ponto de vista da Engenharia de *Software*, esta solução é implementada no padrão de *software Observer* (GAMMA, HELM, JOHNSON e VLISSIDES, 1995).

[10] Do ponto de vista da Engenharia de *Software*, esta solução é implementada no padrão de *software Handler* (FERG, 2006).

Em ambos os modelos, um objeto transmissor não é conectado explicitamente aos receptores (por chamadas explicitas de métodos), estas conexões são definidas em tempo de execução (FAISON, 2006). Desta forma, os objetos transmissores não conhecem os objetos receptores em tempo de compilação. Sendo assim, a PDE oferece alto grau de desacoplamento (FAISON, 2006).

Na PDE, o objeto transmissor notifica a ocorrência do evento para os objetos receptores por meio da passagem de um *token*, o qual pode ser representado por uma mensagem formada por uma cadeia de caracteres ou um objeto com dados, ambas as formas de representação são usadas para guardar as devidas informações sobre o evento. Entretanto, a última representação é a mais comum (FERG, 2006).

Esta comunicação se inicia no instante em que o objeto transmissor detecta a ocorrência de um evento (por meio de uma condição). Na seqüência, o objeto transmissor compõe um *token* e envia-o diretamente aos objetos receptores (no modelo apresentado à esquerda da Figura 18) ou indiretamente a estes por meio de um objeto *Dispatcher* (no modelo apresentado à direita da Figura 18). Neste último modelo, o *Dispatcher* pode oferecer um *buffer* temporário de armazenamento aos *tokens* para controlar o fluxo de notificações.

O modelo que emprega o *Dispatcher* é mais complexo e mesmo assim, o mais aplicado (FERG, 2006). Este é comum no tratamento de eventos gerados pela manipulação de componentes (e.g. botão) de uma GUI (*Graphical User Interface*), no tratamento passivo das interrupções de *hardware* pelos sistemas operacionais e até mesmo na computação distribuída (e.g. servidor *web*) (FAISON, 2006).

Neste modelo, o *Dispatcher* extrai cada *token* do *buffer*, se houver e, analisa a informação contida neste para determinar o tipo de evento que ocorreu. Esta análise ocorre por meio de um laço de repetição finito. De acordo com o tipo de evento, o *Dispatcher* identifica os objetos receptores interessados e repassa o *token* para estes (FERG, 2006). Este processo é expresso no pseudocódigo apresentado na Figura 19.

```
enquanto (verdadeiro)
início
  pega um evento do fluxo de entrada ou do buffer temporário
  se (tipo de evento = condição de término)
  início
    sai do laço de repetição
  fim

  se (tipo do evento = ...)
  início
    chama o método do objeto receptor passando o token como argumento
  fim

  senão se (tipo do evento = ...)
  início
    chama o método do objeto receptor passando o token como argumento
  fim

  senão
  início
    trata do tipo desconhecido de evento ou simplesmente o ignora
  fim
fim
```

Figura 19: Pseudocódigo da Programação Dirigida a Eventos

Como se pode constatar pela análise do pseudocódigo, a seqüência pela qual os métodos dos objetos receptores são invocados é imprevisível, pois as chamadas dependem da ocorrência de um evento. Com isto, a seqüencialidade explícita de execução das instruções pode ser "quebrada".

Esta forma reativa de execução de métodos é semelhante ao que ocorre no âmbito da Programação Baseada em Frames (PBF). Um frame consiste em uma estrutura de dados que é utilizada para representar objetos ou uma classe de objetos relacionados. Esta relação se dá por meio de uma hierarquia taxonômica, formando um sistema de frames (KARP, 1992).

Sucintamente, um frame é composto por uma ou mais entidades chamadas *slots*, os quais representam atributos para armazenar valores primitivos ou mesmo referências para outros frames. Os *slots* apresentam capacidades reativas providas pelas entidades integradas chamadas *facets*. As *facets* permitem reações a eventos ocorridos nos *slots* (e.g. eventos de adição, remoção ou alteração de valor), permitindo que expressões causais sejam avaliadas para verificar restrições (e.g. assegurar somente atribuições de valores permitidos) ou mesmo que funções ou métodos sejam invocados para a execução de ações (e.g. cálculo do valor do *slot*, difusão de atualizações para outros valores de *slots* ou acesso a um banco de dados) (KARP, 1992).

Apesar da PBF se apresentar similar à PDE (na forma representativa e também nas capacidades reativas), ambas se diferem em relação à forma em que executam. Normalmente, um sistema de frames é construído sobre uma máquina de inferência que permite deduções sobre as relações entre os frames e seus estados, sendo que esta prática não é comum na PDE.

2.7.3 Paradigmas Emergentes

Esta seção descreve brevemente segundo a literatura, a essência de três principais Paradigmas Emergentes: o Paradigma Orientado a Agentes, o Paradigma Orientado a Aspectos e o Paradigma Orientado a Componentes.

2.7.3.1 Paradigma Orientado a Agentes

Segundo Russell e Norvig (2002), o Paradigma Orientado a Agentes (POA) permite construir programas empregando entidades de *software* chamadas agentes. Estes habitam em um ambiente real (e.g. representando robôs) ou virtual (e.g. embutidos em um *game*) e percebem mudanças no ambiente por meio de sensores e agem sobre o ambiente de acordo com as mudanças percebidas (RUSSELL e NORVIG, 2002).

Há várias definições de agentes, sendo muito citadas as de Jennings *et al.*(1999) e Ferber (1995) que são complementares. Segundo Jennings *et al.* (1999), um agente é uma entidade informática, situada em um ambiente, capaz de executar ações de maneira flexível e autônoma a fim de atingir objetivos estabelecidos no momento de sua concepção. Esta definição introduz várias propriedades, explicadas abaixo com base nos dois autores citados:

- **Situado:** um agente está imerso num ambiente sendo capaz de perceber o que acontece neste ambiente e de executar ações que o modificam. Ferber (1995) diz que a percepção do ambiente é limitada e que o agente possui apenas uma representação parcial do ambiente ou mesmo não possui representação.
- **Autonomia:** um agente deve ser capaz de agir sem intervenção humana e deve ser capaz de aprender a partir de suas próprias experiências. Além disso, deve ser capaz de controlar seus estados e ações. Ferber (1995) diz que um agente pode eventualmente se reproduzir (clonar).
- **Flexibilidade:** um agente é sensível às mudanças do ambiente e deve responder (se adaptar) a elas.
- **Proatividade:** um agente é capaz de exibir um comportamento proativo guiado por seus objetivos e, portanto, de tomar a iniciativa para executar ações.

Enfim, um agente é uma entidade social capaz de interagir com outros agentes e com usuários a fim de solucionar seus próprios problemas e de cooperar na realização dos objetivos do outros.

Usualmente, os agentes são implementados sobre os conceitos do POO. Porém, um agente não se confunde com um objeto, pois conceitualmente um agente apresenta propriedades mais complexas do que um objeto. Apesar de agentes apresentarem encapsulamento, herança e passagem de mensagens, uma das diferenças fundamentais é que um agente atua de forma proativa, tendo objetivos individuais ou coletivos. Diferentemente de um agente, um objeto atua passivamente até que um de seus métodos seja invocado. Em outras palavras, um objeto aguarda que outro objeto lhe indique o que fazer enquanto que o agente decide por si só quando e o que deve ser feito.

Os agentes se apresentam sob formas distintas, há pelo menos duas formas de classificá-los, *a)* móveis e estáticos quanto à mobilidade e *b)* reativos e cognitivos/deliberativos quanto ao mecanismo de raciocínio. Em relação à mobilidade, os agentes estáticos executam em uma única máquina ao passo que agentes móveis são capazes de se deslocarem pelos nós de uma rede de comunicação. A mobilidade visa reduzir o tráfego de dados na rede e diminuir o processamento local. O agente utiliza os recursos da máquina visitada para lá processar a informação economizando recursos da máquina de origem e tráfego de dados.

A segunda forma de classificação, reativos e cognitivos/deliberativos, diz respeito ao modelo de raciocínio dos agentes. Agentes cognitivos ou deliberativos possuem um modelo explícito e simbólico do mundo e, portanto, são capazes de raciocinar, planificar ações e negociar com outros agentes a fim de coordenar suas ações (WOOLDRIDGE e JENNINGS, 1995; NWANA, 1996). Um dos modelos mais conhecidos para agentes cognitivos é o *BDI (Beliefs, Desires and Intentions)* desenvolvido por (RAO E GEORGEFF, 1995). *Beliefs* são as crenças que o agente possui sobre o estado do ambiente que podem corresponder à realidade ou não. *Desires* são os diferentes estados que o agente pode vir a perseguir ou as coisas que ele deseja realizar (WEISS, 1999). *Intention* é um objetivo ou estado-alvo que o agente se comprometeu a alcançar e para tal empenhou recursos. Os agentes reativos não possuem um modelo simbólico do ambiente e funcionam no modo estímulo-resposta. O comportamento de um agente reativo pode ser guiado por uma necessidade interna (e.g. para satisfazer uma necessidade) ou por um estímulo proveniente do ambiente (FERBER, 1995; BROOKS, 1999).

De fato, o POA apresenta características ímpares, as quais provem maior potencialidade na construção de aplicações computacionais.

2.7.3.2 Paradigma Orientado a Aspectos

Segundo Laddad (2003), o Paradigma Orientado a Aspectos (POAc) permite separar os conceitos principais dos conceitos secundários de um código típico orientado a objetos, encapsulando os conceitos secundários em uma unidade modular chamada de aspecto.

Por exemplo, em um sistema bancário, a funcionalidade de *logging* é extremamente necessária na implementação das operações bancárias, como o saque, depósito e transferência de dinheiro. Neste contexto, a funcionalidade de *logging* é considerada um conceito secundário em relação a estas operações que são as principais do sistema (MILES, 2004).

Geralmente, nas implementações com o POO, o código da funcionalidade de *logging* é intercalado e replicado no código das operações bancárias ou então, ele é centralizado em um objeto (e.g. objeto da classe *Logging*) e as chamadas para os métodos deste objeto são também intercaladas no código das operações bancárias. Esta prática acaba misturando os conceitos principais e secundários, prejudicando a leitura do código ou mesmo dificultando a manutenção do código (e.g. alteração da assinatura do método invocado deve ser realizada em vários pontos distintos) (MILES, 2004).

Para resolver este impasse, o POAc propõe o encapsulamento das funcionalidades secundárias em forma de aspectos e permite que estes aspectos sejam invocados implicitamente (sem declaração de chamada de métodos no código) pelas funcionalidades principais. Na verdade, os aspectos são invocados por eventos que ocorrem sobre os métodos que encapsulam as funcionalidades principais. Os aspectos podem ser executados antes ou depois da chamada de um método ou construtor, ou mesmo na execução de um método ou construtor. Ainda, um aspecto pode ser executado durante a execução de outro aspecto (MILES, 2004).

No exemplo do sistema bancário empregando o POAc, ao invocar a operação de saque (representada por um método de um objeto), as funcionalidades de *logging* como outras funcionalidades secundárias (e.g. autorização de acesso e as demais disposições de segurança) são invocadas de maneira implícita, não estando definidas no código deste método. Para isto, as chamadas para estas funcionalidades são pré-configuradas pelo programador, em um arquivo separado do código. Segundo defensores do POAc, isto gera certos benefícios, como o maior incentivo ao reuso e maiores facilidades de entendimento ou alteração do código (MILES, 2004).

2.7.3.3 Paradigma Orientado a Componentes

Segundo D'Souza e Wills (1998), o Paradigma Orientado a Componentes (POC) é um estilo de programação baseado no reuso e combinação de componentes. Ainda, segundo as definições de D'Souza e Wills (1998), um componente consiste em uma parte de *software* reusável que pode ser desenvolvida independentemente para que seja combinada com outras partes a fim de construir uma unidade maior.

Deste modo, grande parte de um sistema pode ser construída apenas pela combinação de componentes, os quais também podem ser compostos por outros componentes menores e assim consecutivamente, até que a decomposição alcance a unidade de um objeto. Outrossim, a parte do sistema criada por meio da combinação destes componentes pode ser considerada como um componente unitário e ser usada para compor aplicações maiores. Segundo estas explicitações, percebe-se que em POC, os sistemas são construídos por meio da combinação entre componentes, assim como apresentado na Figura 20 (CRNKOVIC e LARSSON, 2002).

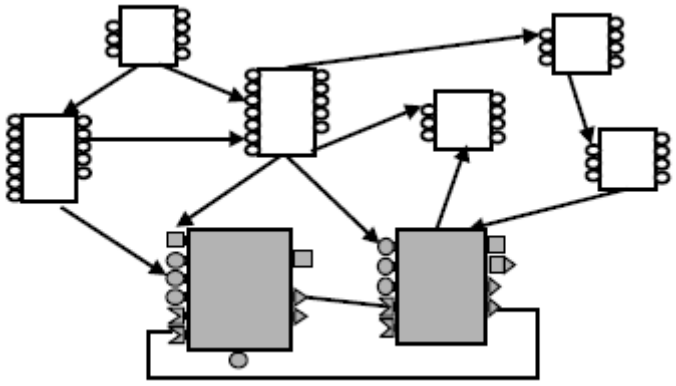


Figura 20: Composição de componentes (CRNKOVIC e LARSSON, 2002)

Conforme a figura, os componentes se relacionam para realizar o fluxo de execução do programa. Estas relações se dão por meio de invocações de serviços (representado pelas setas direcionadas) oferecidos por estes componentes. No entanto, estes serviços não são acessados diretamente, estes são acessados por meio de entidades descritoras destes serviços, as interfaces. O uso das interfaces é uma característica particular dos componentes, permitindo que a definição/descrição dos serviços seja completamente separada da implementação dos mesmos, principalmente para evitar que o código da implementação seja alterado indevidamente (CRNKOVIC e LARSSON, 2002).

Por exemplo, um *widget* (e.g. botão, campo de texto) é um componente que é utilizado sem o programador precisar ter acesso ao seu código-fonte. O programador apenas precisa conhecer os serviços disponibilizados por este *widget*, os quais estão expressos na interface

deste componente. Neste caso particular, um *widget* pode ser representado por um único objeto, o que confunde os conceitos do POC com os do POO (GAO, TSAO e WU, 2003).

No entanto, um componente pode ser muito mais completo do que um simples objeto. Geralmente, um componente é composto de vários objetos ou mesmo de outras entidades natas de outros paradigmas (e.g. funções), uma vez que a heterogeneidade é uma característica particular dos componentes. Assim, os componentes podem ser implementados em linguagens diferentes e mesmo assim apresentar capacidades de cooperação. Este fato se deve, principalmente, porque as interfaces dos componentes são independentes de suas implementações (GAO, TSAO e WU, 2003).

Ainda, devido a esta independência, um componente pode ser integrado em uma aplicação em forma de binários, ou seja, em forma de arquivo compilado (sem o código-fonte). Neste caso, apenas é necessário que a interface deste executável esteja disponível para a integração. Com isso, os componentes não compilados ou as classes em definição poderão invocar os serviços deste componente. Estas invocações poderão ocorrer por chamadas de métodos ou por meio de mecanismos mais sofisticados de comunicação, assim como pelas tecnologias derivadas do *XML* (CRNKOVIC e LARSSON, 2002).

Estas diferenças demonstram que o POC apresenta qualidades não existentes no POO puro, o qual se atém a composição de aplicações por meio de objetos. Ainda, o POC se diferencia na perspectiva de programação. O POC permite compor um programa pensando em entidades de maior grau de abstração do que objetos, isto reduz a responsabilidade do programador na programação. No POO, a criação de um programa é mais árdua, pois muitas vezes, o programador é incentivado a manipular os comandos das linguagens a fim de compor uma funcionalidade.

Desta forma, com o POC, a programação se torna muito mais produtiva, uma vez que a responsabilidade do programador é reduzida quando se reutiliza funcionalidades pré-fabricadas. Esta facilidade é maior, quando a manipulação de componentes é auxiliada por ferramentas de desenvolvimento (e.g. NetBeans da Sun, Visual Age da IBM, JBuilder da Borland) por meio de qualidades interativas e visuais (GAO, TSAO e WU, 2003). Estas facilidades, se aliadas com maiores ofertas de componentes, incentivará cada vez mais o emprego do POC.

2.7.3.4 Relação dos Paradigmas Emergentes com o Paradigma Orientado a Objetos

Apesar destes paradigmas proverem conceitos completamente diferentes, eles apresentam algumas características em comum. Estes paradigmas são geralmente materializados em linguagens dos Paradigmas Dominantes, principalmente sobre as linguagens do Paradigma Orientado a Objetos. Ainda, o POAc e o POC são fortemente fundamentados no POO, pois certos problemas deste paradigma serviram como inspiração para que eles fossem propostos. Desta forma, POAc e POC permitem que os conceitos do POO continuem sendo empregados, porém, com alguns aperfeiçoamentos suficientes para alterar a forma de visualizar os problemas computacionais.

A relação entre as linguagens e Paradigmas Dominantes e o POA se difere levemente das relações para com o POAc e POC. A proposta do POA não foi fundamentada diretamente nas deficiências dos Paradigmas Dominantes. Desta forma, o POA apresenta relação com estes paradigmas por necessidade de aceitação, frente à forte resistência a mudanças apresentada pela comunidade da computação.

Este fato foi constatado no fórum do ATAL (*Agent Theories, Architectures, and Languages*) de 1998 (MEYER, 1998). Este fórum abordou a independência dos conceitos do POA dos demais paradigmas de programação. A conclusão deste fórum foi a inviabilidade de propor linguagens próprias do POA, principalmente porque os programadores apresentam resistência em adotar estas linguagens.

Neste fórum, foi constatado que os programadores estão completamente habituados com os conceitos das linguagens convencionais e toda a flexibilidade proporcionada por estas, se tornando desinteressados em aprender uma nova linguagem. A alternativa encontrada para tornar o POA mais acessível à comunidade da computação foi implementar os seus conceitos em forma de *middleware/framework* para ser facilmente integrado nas linguagens dos Paradigmas Dominantes, principalmente nas linguagens do POO, como por exemplo, o Jade sobre o Java (MEYER, 1998).

2.7.4 Reflexões sobre o Paradigma Imperativo

Após a apresentação sucinta dos paradigmas considerados subtipos do PI, esta seção apresenta as principais deficiências dos mesmos. Para isto, a subseção 2.7.4.1 discute sobre as deficiências relacionadas à eficiência e a subseção 2.7.4.2 refere-se à deficiência do

acoplamento entre as partes de um programa imperativo e mesmo à dificuldade de reuso destas partes devido ao acoplamento. A subseção 2.7.4.3 descreve as dificuldades em programar com o PI e por fim, a subseção 2.7.4.4 discute sobre as questões relacionadas à programação paralela e distribuída.

2.7.4.1 Reflexão sobre a eficiência

Como constatado pelo estudo dos paradigmas subtipos do PI, a essência do PI consiste na organização seqüencial do *software* (ou de cada entidade de *software*) por meio de comandos em uma linguagem procedimental ou orientada a objetos, os quais são executados seqüencialmente pelo mecanismo interno destas linguagens.

Basicamente, este mecanismo consiste em "buscas" no percorrer sobre entidades passivas, as quais correspondem aos dados (e.g. variáveis, vetores e listas) e aos comandos de decisão (e.g. *se-então* e *escolha-de-casos*).

Devido à seqüencialidade da busca e a passividade dos buscados nas linguagens do PI, as linhas de código se tornam interdependentes e há problemas de redundância na execução dos programas. Neste âmbito, as expressões causais são avaliadas passivamente ocasionando as chamadas redundâncias temporais e estruturais. Estes problemas podem afetar o desempenho dos programas desenvolvidos.

A redundância temporal consiste na avaliação desnecessária e repetida de expressões causais na presença de estados (de atributos/variáveis) já avaliados e inalterados. A redundância estrutural ocorre quando o conhecimento sobre um valor Booleano de uma expressão lógica (e.g. comparação de uma variável com uma constante) não é compartilhado entre outras expressões causais pertinentes, causando reavaliações desnecessárias.

As redundâncias temporais e estruturais são observadas no trecho de código apresentado na Figura 21. Neste, três expressões causais (*se-então*) atestam os estados *x* e *y* dos objetos A e B, a fim de mudar o estado *y* de B quando forem aprovadas.

```
1 ...
2 A->setX(false);
3 B->setX(false);
4 B->setY(true);
5
6 while(B->getY() == true){
7   if(A->getX == true)
8   {
9     B->setY(true);
10  }
11  if((B->getX() == true) && (B->getY() == false))
12  {
13    B->setY(true);
14  }
15  if((B->getX() == true) && (B->getY() == true))
16  {
17    B->setY(false);
18  }
19 }
20 ...
```

Figura 21: Exemplo de redundância temporal e estrutural

Certamente, este código ilustrativo poderia ser mais bem otimizado com o uso de "aninhamentos", principalmente na avaliação da premissa (*B->getX() == true*). Porém, os "aninhamentos" não foram utilizados a fim de favorecer a apresentação das redundâncias que ocorrem naturalmente no PI. Ademais, na prática, estas redundâncias são uma constante cada vez que as expressões causais se encontram dispersas. Exemplos objetivos de tais dispersões são as expressões causais separadas por outros comandos ou expressões causais em diferentes métodos ou objetos.

A redundância temporal é observada na avaliação repetida da primeira expressão causal (linha 7) a cada ciclo interativo, mesmo não apresentando variações em relação às avaliações anteriores. A redundância estrutural é observada na avaliação da primeira premissa das duas últimas expressões causais (linhas 11 e 15). Esta premissa é avaliada duas vezes em um mesmo ciclo, sem alterações no seu valor lógico.

No exemplo apresentado, os efeitos das redundâncias podem ser imperceptíveis em termos de desempenho. No entanto, em sistemas compostos por muitas expressões de decisão, que são avaliadas redundantemente, os efeitos no desempenho da aplicação podem ser inaceitáveis (SIMÃO e STADZISZ 2008).

Para evitar a redundância temporal, é preciso que o mecanismo interno destas linguagens avalie as expressões causais somente após a mudança do estado de uma variável/atributo. Para evitar a redundância estrutural, é necessário que os resultados das avaliações lógicas comuns a várias expressões causais sejam compartilhados entre estas expressões, a fim de prevenir avaliações redundantes.

2.7.4.2 Reflexão sobre o acoplamento e reuso

A interdependência entre os dados e comandos no imperativo devido à seqüencialidade da busca e a passividade dos buscados, além de ocasionar avaliações redundantes também ocasionam o acoplamento entre as partes dos programas.

No POO, de forma particular, o acoplamento também pode ser ocasionado por meio dos relacionamentos entre os objetos, uma vez que o funcionamento de um programa neste paradigma é totalmente dependente da relação entre os objetos. No POO, um programa executa por meio da oferta e procura de serviços (métodos) que é gerenciada por eles próprios e concretizada através dos relacionamentos, gerando uma espécie de "rede de comunicações" entre os objetos.

A representação desta "rede de comunicações" é ilustrada na Figura 22, na qual os objetos são representados pelos círculos e os relacionamentos são ilustrados pelas setas. Nesta ilustração, todos os objetos se apresentam inter-relacionados, direta ou indiretamente.

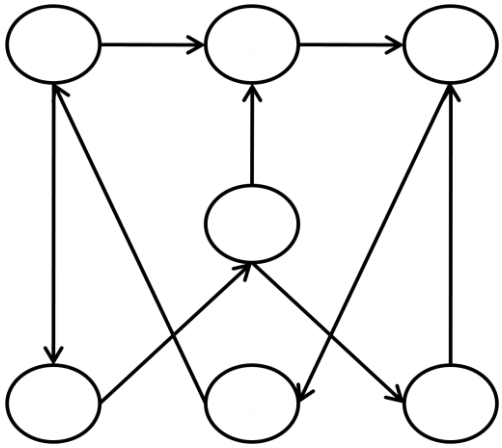


Figura 22: Acoplamento entre objetos

Este inter-relacionamento entre os objetos ocorre naturalmente segundo as diretivas do modelo de programação do POO. Por exemplo, primeiramente o programador define as classes a serem implementadas, para que, na seqüência, o mesmo defina os respectivos atributos e métodos. Neste momento, se os atributos consistirem apenas em tipos primitivos e os métodos forem implementados para somente manipular estes atributos, então, não há oportunidade de uma aplicação se tornar fortemente acoplada.

Entretanto, na realidade, este cenário não ocorre. Normalmente, as classes apresentam atributos que guardam referência para outras classes, acoplando estas classes. Algumas formas de acoplamento podem ser vistos no trecho de código apresentado na Figura 23, o qual se refere à estrutura da classe hipotética *A*. Esta classe apresenta acoplamento com as classes *B* e *C*, as quais ocorrem por referência no escopo da classe (linha 4) e no escopo de um

método, quando se recebe uma referência por parâmetro (linha 7 e 8) ou esta referência é declarada como variável local no método (FAISON, 2006).

```
1 public class A
2 {
3   private:
4   B* b;
5
6   public:
7   void setB(B* b);
8   void manipulateC(C* c);
9 }
```

Figura 23: Acoplamentos no escopo de classe e escopo de método

O acoplamento da classe *A* com as demais pode gerar dificuldades de reuso desta em outros programas, principalmente porque se fará necessário importar também, as suas classes acopladas. Conforme o trecho de código, as implementações das classes *B* e *C* são intrínsecas a implementação da classe *A*. Assim sendo, em princípio, estas três classes devem ser reusadas em conjunto. Porém, se as classes acopladas (i.e. *B* e *C*) também referenciarem outras classes, formando uma cadeia de referência entre classes (Figura 24), o reuso desta simples e única classe (i.e. *A*) pode se tornar uma tarefa ainda mais difícil.

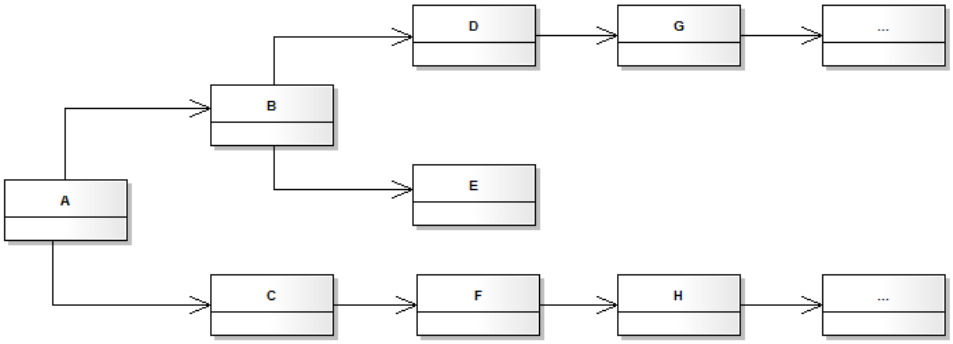


Figura 24: Cadeia de referência entre classes

Assim, para que a classe *A* seja reutilizada, é necessário importar toda esta cadeia de relações, se tornando, muitas vezes, uma tarefa impraticável quando não se tem acesso ao código das classes para efetuar as devidas alterações, pois muitos serviços não são pertinentes a nova aplicabilidade e poderiam ser omitidos. A fim de tentar minimizar este problema típico do POO, engenheiros de *software* propuseram várias soluções em forma de padrões de *software*. Apesar de estas serem efetivas para o que se propõem, as mesmas exigem maiores esforços para serem aplicadas pelo programador, não provendo o reuso de forma natural e transparente como desejado.

Em suma, o POO não é totalmente claro em seus objetivos como paradigma. Primeiramente, este propõe a estruturação do código por meio de classes e propriedades como a herança e encapsulamento, objetivando entre outras vantagens, maior capacidade de reuso

do que na programação procedimental. Entretanto, o que se constata na programação POO é que apenas o uso destas propriedades não é suficiente para se atingir alto nível de reuso, principalmente porque a essência do paradigma impõe alguns empecilhos.

Na verdade, a essência da execução de um programa implementado sobre os conceitos do paradigma, pela comunicação entre os objetos, incentiva o acoplamento. A existência de altos níveis de acoplamento reduz as capacidades de reuso de partes de um programa.

Como prova desta falta de clareza, está a proposta de várias soluções em forma de padrões de *software* a fim de compartilhar técnicas sofisticadas para se atingir maior reuso ou mesmo o emprego de diagramas UML, como o diagrama de seqüência e comunicação, que facilitam a identificação dos relacionamentos entre os objetos por meio das chamadas de métodos para reduzir o acoplamento entre as partes. O emprego destas técnicas mostra a insuficiência deste paradigma em atingir o reuso de forma natural, uma vez que a responsabilidade de se construir um programa desacoplado recai totalmente sobre o programador.

2.7.4.3 Reflexão sobre as dificuldades de programação

Além dos problemas relacionados ao acoplamento, o PI também apresenta problemas relacionados à composição dos programas. Geralmente, nos programas criados com os conceitos do PI, o código que envolve a lógica da aplicação se encontra disperso entre os comandos e expressões de controle da linguagem, tornando difícil a leitura e entendimento do código, além de desviar a atenção do programador do que realmente é importante.

Naturalmente, os programadores apresentam dificuldades na manipulação das linguagens imperativas, principalmente pela necessidade de gerenciamento de memória por meio de ponteiros e pelas sintaxes pouco intuitivas, como ocorre na representação de atribuições de valores por meio do símbolo “=” e de igualdade por meio do símbolo “==”. A manipulação direta destas sintaxes pouco intuitivas e complexas das linguagens imperativas na composição da lógica da aplicação torna a programação tendente a erro. A solução seria disponibilizar componentes de abstração (e.g. comandos declarativos) ou ferramentas amigáveis (*wizard*) que permitissem inserir a lógica da aplicação de uma forma transparente às particularidades do código imperativo.

Entretanto, apesar destas particularidades, a maior dificuldade na programação imperativa se encontra nos direcionamentos do paradigma materializado. Apesar do POO

oferecer avanços estruturais, como constatado na seção 2.4.4, este paradigma não avançou em termos de semântica. O código OO é de difícil compreensão, não só pela sintaxe, mas pela organização da lógica da aplicação.

Devido aos relacionamentos/entrelaçamentos entre os objetos, se torna difícil compreender uma funcionalidade analisando apenas uma classe. Geralmente, é preciso considerar as demais classes relacionadas. Esta dificuldade se deve à dispersão da lógica da aplicação por métodos de vários objetos coligados, o que leva à necessidade de seguir o fluxo de chamada de método para entender toda a lógica.

Na manutenção de um programa, o problema continua. Devido aos relacionamentos entre os objetos, muitas vezes, a alteração de uma parcela do conhecimento em algum ponto do programa pode resultar na necessidade de alteração em outros, se tornando uma reação em cadeia em termos de alterações.

Assim, a inclusão de uma simples expressão causal sobre a detecção de um novo estado se torna uma tarefa difícil em um programa implementado com o POO. Primeiramente, deve-se decidir o ponto em que esta expressão será inserida entre os emaranhados de comunicação e segundo, a seqüencialidade de execução desta expressão deve ser levada em conta, ou seja, a expressão deve ser incluída em um ponto em que não venha afetar na avaliação das demais expressões e que esta apresente oportunidade de ser avaliada.

Ainda, devido a seqüencialidade, a mesma expressão causal pode precisar ser incluída repetidamente em várias partes do código, onde há mudança de estados que possam afetar o seu estado lógico. Este fato ocorre em algumas aplicações que demandam ações imediatas (e.g. games e de controle), quando a expressão causal deve ser avaliada prontamente após as alterações nos estados que podem afetá-la. Um exemplo deste caso ocorre no trecho de código na Figura 25a, no qual após a mudança de estado de um atributo (linhas 2 e 8), as respectivas expressões causais são prontamente avaliadas.

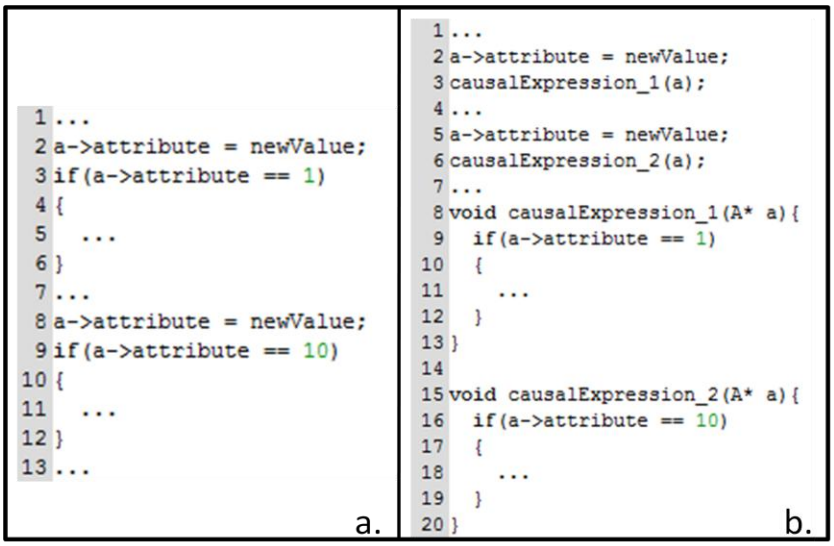


Figura 25: Replicação de código

Uma alternativa para evitar esta repetição de código seria a inclusão desta expressão dentro de um método, e invocá-lo no ponto em que a expressão deve ser avaliada, como ilustra o trecho de código na Figura 25b. Esta alternativa evita a repetição de código, a qual é prejudicial em termos de manutenção por causa da necessidade de atualização do código em cada ponto replicado.

Entretanto, apesar desta centralização da expressão causal sobre o escopo de um método, o que centraliza a modificação do código em um único ponto, o problema ainda persiste. Com a criação deste método, o programador deve percorrer a lógica dispersa e identificar os pontos necessários (baseado na seqüencialidade da aplicação) para a chamada deste método. Assim, a cada ponto em que ocorre uma mudança de estado, a chamada de método deve ser invocada explicitamente.

Muitas vezes, o programador pode esquecer-se de incluir esta chamada. Nestes casos, o programa pode se apresentar inconsistente. O ideal seria que a expressão causal fosse avaliada automaticamente após a mudança de estado do atributo, facilitando o trabalho do programador e reduzido a chance de ocorrência de erros na programação.

2.7.4.4 Reflexão sobre o paralelismo e distribuição

A seqüencialidade de execução do imperativo e a passividade dos seus elementos que levam à interdependência entre os dados e comandos de um programa, além de tornar difícil a leitura do código também tornam difícil a partição do mesmo. Com isso, o PI dificulta a

obtenção dos reais benefícios da computação paralela e distribuída (SIMÃO e STADZISZ, 2008).

Por exemplo, estas linguagens não permitem a partição e distribuição do código em entidades como atributos, métodos e expressões causais. Normalmente, a partição ocorre em termos de entidades modulares, principalmente em termos de objetos. No entanto, mesmo que os comandos sejam separados em classes/objetos, a dificuldade ainda permanece.

Esta dificuldade se deve por causa dos múltiplos relacionamentos entre os objetos (como visto na seção 2.7.4.2), que leva a um forte acoplamento entre as partes (FAISON, 2006). O forte acoplamento gera dificuldades na partição do sistema em conjuntos independentes de objetos, levando muitas vezes, ao uso excessivo de mecanismos de sincronização entre as chamadas de métodos para manter consistência entre os dados (HUGHES e HUGHES, 2003).

Além do mais, o emprego destes mecanismos de sincronização para prevenir o acesso concorrente aos atributos e métodos deve ser controlado explicitamente pelo programador. Este precisa definir explicitamente por meio de comandos de sincronização quais os métodos que necessitam de um controle mais rigoroso para evitar inconsistências nos dados referenciados. Esta prática dificulta a programação multiprocessada com o POO, gerando muitas falhas de programação devido à responsabilidade suplementar atribuída ao programador.

Este forte acoplamento é observado também no trecho de código em C++ apresentado na Figura 21. Estas expressões são executadas de forma interdependente, o que inviabiliza a separação destas para execução em diferentes nós de processamento. Mesmo que estas expressões fossem alocadas a *threads* ou processos para execução em paralelo, o uso de mecanismos de sincronização afetaria o desempenho da aplicação. Além disso, em uma aplicação com grandes quantidades de expressões causais, esta prática se tornaria inviável devido à freqüente troca de contextos entre os *threads* (TANENBAUM, 2001).

Desta forma, após a apresentação destas deficiências percebe-se que o Paradigma Imperativo clama por melhorias em sua forma de estruturar e executar programas. Uma alternativa para prover parte destas melhorias pode ser a adoção de alguns conceitos do Paradigma Declarativo.

2.8 PARADIGMAS DECLARATIVOS

Esta seção apresenta os Paradigmas Declarativos. Esta seção descreve brevemente o Paradigma Funcional e enfatiza o Paradigma Lógico, principalmente em relação à aplicabilidade dos conceitos do Paradigma Lógico na construção dos Sistemas Especialistas e consecutivamente dos Sistemas Baseados em Regras. Ainda, esta seção também apresenta as deficiências do Paradigma Declarativo.

2.8.1 Paradigma Funcional

O Paradigma Funcional (PF) se difere largamente dos Paradigmas Imperativos. Ambos os paradigmas são inspirados em trabalhos largamente independentes, de dois ilustres pesquisadores da década de 30, Alonzo Church e Alan Turing. Entretanto, mesmo se tratando de trabalhos independentes, eles apresentavam objetivos idênticos. O objetivo de Church e Turing era definir formalmente um modelo efetivo para que máquinas de cálculo (i.e. computadores) executassem algoritmos.

Alan Turing, baseado em autômatos, propôs um modelo de execução seqüencial que armazenava dados temporariamente em células de uma "fita" (TURING, 1936). Este modelo ficou conhecido como a Máquina de Turing, a qual inspirou a concepção do PP. Da mesma forma pela qual Turing acessava e modificava valores em uma fita, no PP e mesmo no POO, variáveis também são acessadas e alteradas.

Alonzo Church, baseado nos conceitos de funções parametrizadas, propôs o modelo chamado Cálculo Lambda (este nome se deve ao uso do símbolo λ para representar parâmetros em funções matemáticas). O Cálculo Lambda inspirou o surgimento do PF.

A essência do Paradigma Funcional é a construção de programas a partir da manipulação de dados por meio de funções. Funções podem invocar outras funções ou mesmo serem passadas como argumentos para outras funções (SCOTT, 2000).

Uma vantagem deste paradigma é permitir a programação em mais alto nível, como demonstrou o exemplo na seção 2.4.3. Como exemplos de algumas linguagens baseadas no Paradigma Funcional estão LISP, Miranda, Haskell, Sisal, pH e ML.

2.8.2 Paradigma Lógico

2.8.2.1 Visão geral

Apesar dos paradigmas apresentados anteriormente se basearem em teorias diferentes, eles acabam compartilhando a mesma forma de execução, ou seja, em todos eles, um programa ou mesmo uma função, recebe dados como entrada, realiza computações e gera uma saída. Este processo é definido como um mapeamento da entrada para a saída do programa (WATT, 2004).

O Paradigma Lógico (PL) diferencia-se dos demais paradigmas neste aspecto. O PL implementa relações entre dados ao invés desse tipo de mapeamento citado, uma vez que, além das variáveis, os "comandos" causais também são representados na forma de dados. Esta relação se dá pela busca realizada por meio de um mecanismo de inferência sobre as definições de comandos ou expressões causais (i.e. regras) e o conjunto de dados (i.e. base de fatos). Este processo de busca foi relatado no exemplo em PROLOG na seção 2.4.4.

Para melhor e mais formalmente elucidar a explanação do conceito de relação entre dados (i.e. elementos da base de fatos e regras), considere dois conjuntos *S* e *T*. Há uma relação *r* entre *S* e *T* se, para cada *x* em *S* e *y* em *T*, *r(x,y)* é verdadeiro ou falso (WATT, 2004). Uma relação se diferencia do mapeamento de acordo com a correspondência entre pares de valores dos conjuntos.

Em uma relação entre *S* e *T*, um valor em *x* de *S* pode ser relacionado a vários valores de *y* em *T*. Por exemplo, em uma relação de igualdade ("=") entre os elementos destes conjuntos, para qualquer par de *s* e *t*, *s = t* pode ser verdadeiro ou falso.

No mapeamento, cada valor *x* em *S* é mapeado para exatamente um único valor *y* em *T*. Por exemplo, a mesma expressão de igualdade, se implementada em uma função *f* (i.e. *f(x,y)*), dado *x* e *y*, *f* retornará apenas o valor lógico correspondente a este par de valores.

Assim, a cardinalidade na relação é geralmente muitos-para-muitos enquanto a cardinalidade no mapeamento é muitos-para-um (WATT, 2004). Desta forma, a essência do PL se fundamenta no modelo relacional. Por meio deste modelo, um mecanismo de inferência realiza buscas por valores (fatos) que satisfaçam certa relação (regra) a fim de provar um teorema (objetivo) (SCOTT, 2000).[11]

[11] Uma outra forma de inferência sobre o PL ocorre no âmbito de sistemas de frames (KARP, 1992)

2.8.2.2 Sistemas Especialistas

Os conceitos do PL podem ser aplicados na construção de Sistemas Especialistas (SE). Um SE é um programa de computador usado para capturar conhecimento em um domínio específico e que atua por meio de procedimentos de inferência para resolver problemas requerentes de conhecimento humano para resolvê-los (FEIGENBAUM, 1982).

Os SE são constituídos basicamente de elementos da base de fatos e regras, os mesmos conceitos elementares das linguagens do PL, como o PROLOG. No entanto, mesmo PROLOG oferecendo os componentes necessários para implementar SE, esta linguagem não é amplamente empregada para este fim. Ao invés de usar linguagens do PL para construir SE, geralmente se usam linguagens concebidas especialmente para a construção de SE (GIARRATANO e RILEY, 1993).

Este fato ocorre devido à aplicabilidade dos SEs na captura e organização do conhecimento, necessitando de linguagens e ferramentas mais versáteis do que o PROLOG para facilitar este processo. PROLOG é inviável para este objetivo, principalmente por causa de sua sintaxe atípica, como ilustrado na Figura 13, e também pelas limitações da lógica de primeira ordem e pela falta de suporte a diferentes algoritmos de busca em sua forma pura.

Atualmente, o PROLOG só admite o algoritmo *backtracking* sobre a estratégia de busca *Backward Chaining* (encadeamento para trás) de forma natural, sendo que o algoritmo *Forward Chaining* pode ser integrado ao PROLOG, mas para isto, o mesmo deve ser implementado pelo programador (BLACKBURN, BOS, e STRIEGNITZ, 2006).

Desta forma, ferramentas próprias para a construção de SE foram elaboradas para prover as funcionalidades necessárias, assim como a estratégia de busca *Forward Chaining* e novos algoritmos de inferência. Atualmente, há várias opções de ferramentas para o desenvolvimento de SE, as quais são chamadas de *shells* (FRIEDMAN-HILL, 2003).

Um *shell* consiste em um conjunto formado por uma linguagem, pela arquitetura de um SE pré-implementada, por um mecanismo de inferência eficiente e outras funcionalidades úteis para facilitar o desenvolvimento, tais como um editor gráfico ou textual, um gerenciador de arquivos, um depurador e mesmo um gerador de código quando uma interface gráfica amigável é utilizada (GIARRATANO e RILEY, 1993). Neste contexto, a Figura 26 ilustra a tela principal do *shell* CLIPS (i.e. ferramenta desenvolvida pela NASA) (GIARRATANO e RILEY, 1993), a qual consiste em um conjunto de sub-janelas que auxiliam o programador na tarefa de criar Sistemas Especialistas.

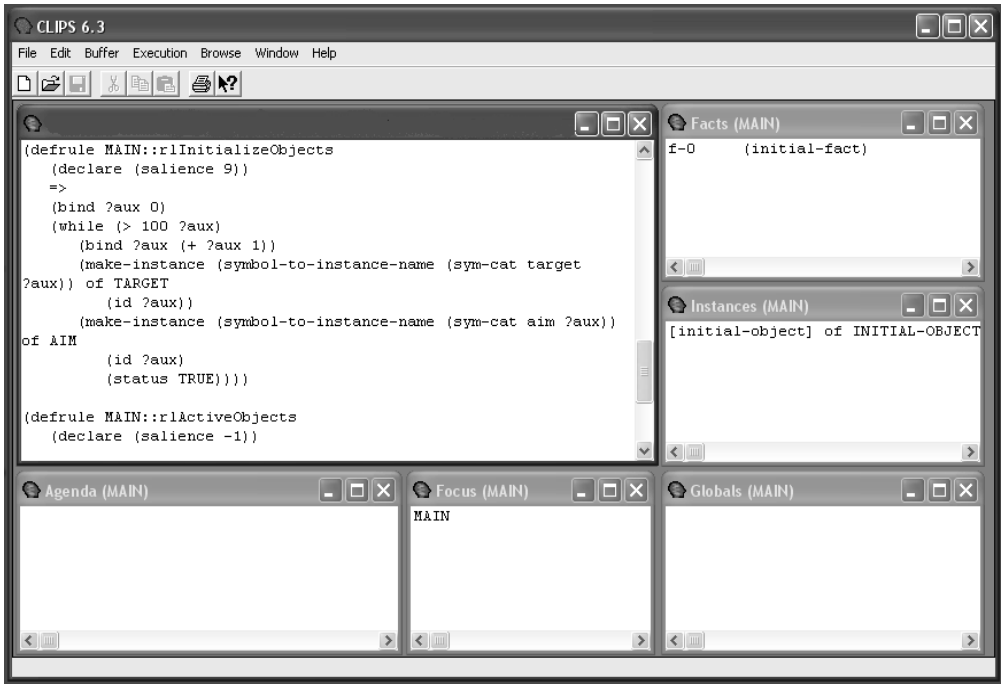


Figura 26: Tela do *Shell* CLIPS

O primeiro *shell* surgiu a partir do sistema especialista MYCIN (DAVIS, BUCHANAN e SHORTLIFFE, 1977). MYCIN era usado para o diagnóstico de infecções por bactérias no sangue. Este sistema oferecia todas as funcionalidades necessárias para compor as regras e elementos da base de fatos para o domínio de medicina. Porém, estas entidades eram fortemente integradas com o resto do sistema, impossibilitando o reuso da estrutura do *shell* para a criação de SE para outros domínios. Este fato foi observado e, em 1979 foi proposto o EMYCIN (Empty MYCIN) (MELLE, 1979). Basicamente, o EMYCIN consiste na arquitetura comum dos SEs, ou seja, o MYCIN sem a base de conhecimento pertinente à medicina.

O uso dos *shells* incentivou a criação de SE para diferentes domínios, como exemplo estão os sistemas de monitoramento de processos industriais, sistemas de roteamento de chamadas telefônicas e até mesmo controles simulados de robôs (FRIEDMAN-HILL, 2003). Deste modo, observa-se que os conceitos dos SE foram aplicados a atividades diferentes (e.g. monitoramento e controle) da vislumbrada inicialmente que seriam essencialmente atividades envolvendo a captura do conhecimento de um especialista em um determinado domínio ou profissão particular. Devido a esta diversidade de aplicação, o conjunto de todos estes

sistemas (i.e. esta generalização dos conceitos dos SE) que fazem uso da relação entre elementos da base de fatos e regras é chamado de Sistemas Baseados em Regras[12].

2.8.3 Sistemas Baseados em Regras (SBR)

2.8.3.1 Visão geral

Os SBR bem como os SE foram inspirados nos trabalhos de Newell e Simon (1972), na intenção de simular o raciocínio humano por meio de um mecanismo de inferência que relaciona fatos (i.e. estados dos elementos da base de fatos) e regras. Eles consideraram os elementos da base de fatos e as regras como "entidades", as quais são instigadas a partir de percepções (NEWELL e SIMON, 1972).

Newell e Simon (1972) demonstraram que muitos dos problemas que o ser humano resolve poderiam ser expressos em termos de regras. Eles observaram que o cérebro é estimulado a partir de entradas sensórias (i.e. fatos), as quais são armazenadas em entidades chamadas "elementos da base de fatos". Os elementos da base de fatos são usados para armazenar temporariamente o conhecimento necessário para a resolução dos problemas. Estes estímulos (fatos) ativam algumas "regras" a fim de produzir uma resposta apropriada a um dado problema (NEWELL e SIMON, 1972).

O mecanismo que ativa as regras por meio dos fatos, ou seja, que simula o raciocínio humano foi chamado por eles de Processador Cognitivo. O Processador Cognitivo age procurando por regras que são ativadas pelos estímulos sensitivos. Se várias regras forem ativadas ao mesmo tempo, o Processador Cognitivo realiza uma resolução de conflito para decidir qual regra tem maior prioridade para ser executada (NEWELL e SIMON, 1972).

A partir das observações de Newell e Simon (1972), foi elaborada a arquitetura dos SBR. Um SBR consiste basicamente de uma memória para armazenar as regras (Base de Regras), uma memória para armazenar os fatos e conseqüentemente os elementos da base de fatos (Base de Fatos) e ainda do Processador Cognitivo para inferir sobre estas duas bases (Máquina de Inferência). Estes componentes formam a arquitetura dos SBR, como apresentado na Figura 27 (FRIEDMAN-HILL, 2003).

---

[12] Devido à forte relação dos Sistemas Baseados em Regras com o paradigma proposto, a essência e as particularidades destes sistemas serão abordadas em uma seção desvencilhada do Paradigma Lógico.

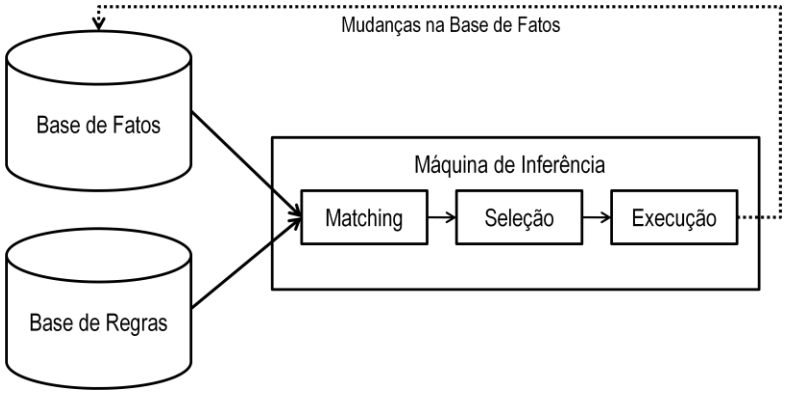


Figura 27: Arquitetura interna de um Sistema Baseado em Regras

A Base de Fatos é composta de um conjunto de elementos da base de fatos que guardam os estados (fatos) do sistema. A Base de Regras contém um conjunto de regras para representar o conhecimento ou decisões de controle no sistema. A Máquina de Inferência (MI) é o modulo de raciocínio do SBR responsável por comparar regras e fatos durante os ciclos de inferência, gerando novos fatos. A Base de Fatos e a Base de Regras são separadas do mecanismo de inferência, permitindo que a MI seja reutilizada em outras aplicações.

Um ciclo de inferência consiste em três fases distintas, conhecidas como as fases de *matching* (casamento), seleção e execução:

- A fase de *matching* compara os fatos em relação às regras para ativá-las, colocando-as de forma desordenada em um repositório chamado de conjunto de conflito (i.e. *conflict set*).
- A fase de *seleção* ordena as regras conforme alguma estratégia de conflito, como na estratégia baseada na prioridade das regras ou em estratégias relativas à recentidade dos fatos que ativaram as regras (e.g. estratégias LEX e MEA) (FRIEDMAN-HILL, 2003), para formar o conjunto ordenado de regras chamado *Agenda*.
- A fase de *execução* seleciona a primeira regra da *Agenda* e executa a sua ação. Nesta execução, a regra pode inserir novos elementos na Base de Fatos ou mesmo invocar algum serviço externo (e.g. funções de alguma outra entidade de *software*).

Entre as fases de um ciclo de inferência, a fase de *matching* é a que mais interfere no desempenho dos SBR. O desempenho é mais afetado na implementação dos SBR desacompanhados de um mecanismo de inferência inteligente, como ocorria na implementação dos primeiros SBR. Nestes SBR, o processo de ativação de regras para

execução era bastante demorado devido à comparação exaustiva entre regras e fatos por meio de um mecanismo de inferência ineficiente.

Mais precisamente, na fase de *matching*, cada regra (da Base de Regras) era avaliada contra os estados de cada elemento da base de fatos, resultando na aprovação de algumas regras para o conjunto de conflito. Na fase de *seleção*, as regras eram ordenadas e uma delas era selecionada para execução. Na fase de *execução*, a execução de uma regra geralmente afetava (como ainda afeta) o estado de pelo menos algum elemento da base de fatos, o que repercutia na desaprovação ou mesmo na aprovação de novas regras ao conjunto de conflito. Assim, a cada ciclo, todas as regras do conjunto de conflito eram descartadas e processo de *matching* (demorado) era reiniciado.

Segundo Daniel Miranker, a fase de *matching* nestes primeiros SBR corresponde aproximadamente a 90% do tempo de execução dos SBR (MIRANKER e LOFASO, 1991). A ineficiência destes sistemas ocorre por causa da avaliação redundante entre fatos e regras, uma vez que muitos dos testes realizados em cada ciclo de *matching* apresentam os mesmos resultados dos ciclos anteriores, gerando desperdício de processamento.

Para evitar este problema, algumas soluções foram propostas a fim de guardar os estados já avaliados em ciclos anteriores. Assim, as comparações são realizadas somente das regras restritivamente contra os estados dos elementos da base de fatos atualizados recentemente. Estas soluções são chamadas de algoritmos de inferência incrementais.

Exemplos destes tipos de algoritmos são o RETE (FORGY, 1982), o TREAT (MIRANKER, 1987), o LEAPS (MIRANKER, BRANT, LOFASO e GADBOIS, 1990) e o HAL (LEE e CHENG, 2002). Estes algoritmos são de suma importância para viabilizar a execução dos SBR. Sem o emprego destas soluções eficientes, os ciclos de inferência demorariam muito tempo para serem concluídos, inviabilizando o uso dos SBR em muitos casos (SIMÃO, 2005). Devido a esta importância e pertinência (a este trabalho), esses algoritmos serão apresentados com maiores detalhes nas próximas seções, com ênfase ao RETE, que é o algoritmo de maior impacto industrial (FRIEDMAN-HILL, 2003).

2.8.3.2 Algoritmo RETE

O algoritmo RETE, que significa rede em latim (pronunciado *ree-tee*), tem sido usado em *shells* de SBR para melhorar o desempenho da fase de *matching*. Atualmente, o RETE é o

algoritmo de inferência de maior impacto industrial, sendo aplicado, pelo menos, nos seguintes *shells*: OPS5, ART, CLIPS, RuleWorks, ILOG Rules e JESS.

O algoritmo RETE foi desenvolvido em 1982 por Charles Forgy fundamentado em uma rede de nós interconectados. Cada rede é usada para representar uma regra, mas usualmente as redes representativas de várias regras se fundem pelo compartilhamento de alguns nós, aparentando se tratar de uma única rede. Em uma rede, cada nó representa um ou mais testes encontrados na parte condicional de uma dada regra (FRIEDMAN-HILL, 2003).

O algoritmo RETE entra em funcionamento sobre a ocorrência de três eventos na Base de Fatos: o evento de inserção, de modificação ou de remoção de um elemento da base de fatos. Na inserção, o elemento da base de fatos se torna algo chamado *α-token* que percorre os nós da rede para a ativação de uma regra. Na remoção, o elemento da base de fatos também se torna um *α-token* que percorre os nós da rede para a desativação de uma regra. Na modificação de um elemento da base de fatos, há a construção de dois *α-tokens*, um para remover as informações dos nós da rede e o outro para inserir as novas informações relativas ao elemento.

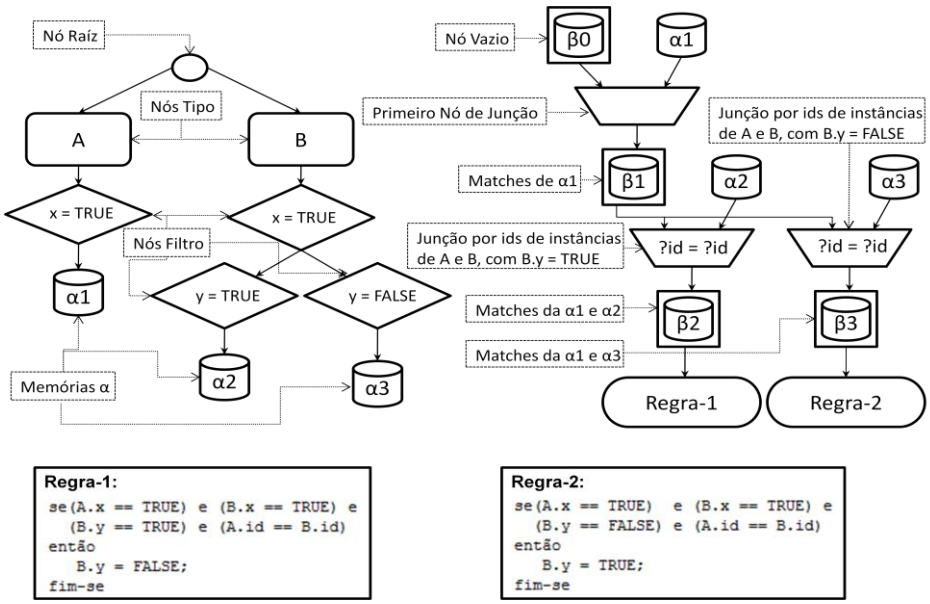
2.8.3.2.1 Estrutura

O RETE implementa um processo de busca otimizado, manipulando elementos da base de fatos e regras sobre a estrutura de redes ou grafos. Mais precisamente, o RETE guarda informação sobre avaliações anteriores das regras para evitar avaliações repetidas e ainda avalia as regras somente quando a Base de Fatos é atualizada, ou seja, quando um elemento da base de fatos é inserido/modificado/removido da Base de Fatos (FORGY, 1982). Com isto, o RETE resolve o problema das redundâncias temporais do PI. Esta solução se faz possível porque o RETE é composto de duas sub-redes chamadas rede alfa (*α-network*) e rede beta (*β-network*), as quais evitam muito processamento desnecessário.

A *α-network* é composta por certos nós sendo cada qual responsável pela avaliação lógica de fatos. A *β-network*, por sua vez, é composta por outros nós, sendo cada qual responsável pela correlação entre fatos. Se as avaliações e correlações forem satisfeitas para uma regra, a mesma é aprovada sendo, portanto, inserida no conjunto de conflito (DOORENBOS, 1995).

O algoritmo RETE pode ser mais bem compreendido por meio da análise da sua estrutura. A estrutura do RETE é ilustrada em forma de rede na Figura 28, que representa a *α-*

network (à esquerda) e a *β-network* (à direita). Na verdade, esta rede é composta por duas sub-redes, referente às regras Regra-1 e Regra-2, devido ao compartilhamento de um nó entre estas duas sub-redes. As mesmas regras são apresentadas em forma textual na mesma figura no formato das expressões causais *se-então*.



Regra-1:

```
se(A.x == TRUE) e (B.x == TRUE) e
  (B.y == TRUE) e (A.id == B.id)
então
   B.y = FALSE;
fim-se
```

Regra-2:

```
se(A.x == TRUE)  e (B.x == TRUE) e
  (B.y == FALSE) e (A.id == B.id)
então
   B.y = TRUE;
fim-se
```

Figura 28: Estrutura do algoritmo RETE

Pela análise da rede, pode se constatar que a estrutura do RETE é composta por sete diferentes tipos de nós:

- **Nó Raiz:** representa o ponto de entrada dos elementos da base de fatos na rede. Na verdade, o elemento inserido na rede é transformado em outro elemento chamado *α-token*:
  - O *α-token* tem a função de representar o elemento da base de fatos e o tipo de evento ao qual ele corresponde, ou seja, ele pode se referir a um evento de inserção ou remoção na Base de Fatos. O *α-token* encapsula o elemento da base de fatos inserido e por meio de uma *tag* sinaliza os eventos pertinentes (i.e. inserção ou remoção) (DOORENBOS, 1995).
  - A sintaxe do *α-token* é a seguinte: (tag FATO-ID). A variável *tag* pode assumir dois valores, "+" ou "-", quais representam respectivamente a inserção ou remoção de elementos na Base de Fatos. A variável *FATO-ID*, chamada de *Time Tag*, é um identificador atribuído ao *token* quando o mesmo é criado ou modificado. Na modificação de um elemento da base de fatos, apesar de apenas o valor ser modificado, o respectivo *α-token* entra na rede assumindo um novo *Tag Time*.
- **Nó Tipo:** analisa o tipo/classe das instâncias dos elementos da base de fatos.

- **Nó Filtro:** representa um simples teste condicional (i.e. uma premissa). Ele filtra os *α-tokens*, armazenando-os em um nó de memória, o Nó Memória-α.
- **Memória-α (α-memory):** armazena temporariamente os *α-tokens* que passaram pelo processo de filtragem.
- **Nós de Junção:** correlaciona instâncias de duas *α-memories* distintas. O Nó de Junção é o nó mais complexo da rede e por isto precisa de uma explicação mais detalhada:
  - À direita na Figura 28, o Nó de Junção (i.e. nó em forma de trapézio mais à esquerda) referente à correlação entre os *ids* de A e B recebe dados pelas suas duas entradas (entrada à direita e entrada à esquerda) e gera uma saída.
  - A entrada à esquerda recebe um *α-token* de um nó *α-memory* superior (e.g. nó *α2*) enquanto a entrada à direita recebe uma referência para um conjunto de *tokens*, os *β-tokens*, localizados em outra memória, chamada *β-memory* (e.g. nó *β1*). Um *β-token* é o resultado da saída da computação de um Nó de Junção.
  - Desta forma, se ambas as entradas forem satisfeitas, o Nó de Junção é ativado para execução. Em execução, o Nó de Junção realiza testes de consistência entre o *α-token* e os *β-tokens* e guarda temporariamente o resultado no *β-token*, o qual é armazenado no *β-memory* seguinte (nó *β2*).
  - O *β-token* é mais complexo do que um *α-token*, pois aquele é composto por uma lista de *α-tokens* que satisfizeram os testes de consistência.
- **Memória-β (β-memory):** armazena *β-tokens* recebidos da saída do Nó de Junção, os quais podem alimentar a entrada de outro Nó de Junção posicionado de acordo com o fluxo de seqüência na rede. Estes *β-tokens* representam os fatos avaliados que satisfizeram parcialmente a condição de uma regra.
- **Nó Regra:** apresenta função similar ao *β-memory*. Porém, o Nó Regra armazena um *β-token* que satisfez completamente a condição de uma regra. Quando o *token* alcança este nó, a regra representada é ativada.

2.8.3.2.2 Funcionamento

O funcionamento do algoritmo RETE pode ser melhor compreendido por meio de um exemplo prático. Para isto será adotada a mesma representação da Figura 28. Para explicar a

propagação de *tokens* pela rede, dois elementos da base de fatos (A e B) são supostamente inseridos na Base de Fatos a fim de ativar a regra Regra-1 para o conjunto de conflito.

+ FATO-1: [A (x TRUE) (id 1)] e

+ FATO-2: [B (x TRUE) (y TRUE) (id 1)]

Primeiramente, o *α-token* (+FATO-1) é inserido na rede. A inserção ocorre através do Nó Raiz para que o *α-token* seja propagado pela *α-network*. Após a inserção, o *α-token* passa pelo teste de tipo de classe para determinar se ele realmente guarda um elemento da base de fatos do tipo *A*. Conseqüentemente, o (+FATO-1) é direcionado ao Nó Filtro localizado de acordo com o fluxo de seqüência. Este nó verifica o estado do atributo *x* de *A*. Se o estado for TRUE, então (+FATO-1) é armazenado em *α-memory* (α1), que é o nó subseqüente. Esta ação ativa a *β-network*.

A *β-network* (Figura 28 à direita) é ativada com a execução do primeiro Nó de Junção. Apesar da ativação aparente de uma única entrada do Nó de Junção pelo nó *α-memory* (*α1*), a execução ocorre devido ao uso do nó auxiliar *β-memory* (*β0*). *β0* é usado como entrada à esquerda do primeiro Nó de Junção, este armazena um *β-token* vazio que simplifica a implementação da rede[13]. Segundo a versão representada na Figura 28, o resultado da junção entre *β0-memory* e *α1-memory* reflete os mesmos valores do *α1-memory*, os quais são armazenados na seguinte *β-memory* (*β1*) (GIARRATANO e RILEY, 1993).

Ao observar a Figura 28, nota-se um aperfeiçoamento do algoritmo para garantir melhor desempenho. O nó *β1-memory* é compartilhado por dois Nós de Junção para poupar espaço em memória ao evitar a replicação de dados e principalmente para evitar avaliações redundantes da mesma premissa. Este compartilhamento resolve o problema das redundâncias estruturais do PI.

Mesmo compartilhando dados, nenhum Nó de Junção foi ainda ativado para execução. Em ambos, o (+FATO-1) ativou somente a entrada à esquerda de cada Nó de Junção. Para ativar completamente o nó, é necessário que as memórias *α2* ou *α3* sejam atualizadas para ativar a entrada à direita. Esta atualização ocorrerá com a inserção do (+FATO-2) (GIARRATANO e RILEY, 1993).

O (+FATO-2) é inserido na *α-network* com a intenção de ativar a regra *Regra-1*. Primeiramente, o (+FATO-2) passa pelo teste de tipo que verifica se este elemento é do tipo *B*. Com isto, (+FATO-2) é enviado para o ramo do Nó Filtro. Este ramo é complexo, pois

[13] Este uso do nó auxiliar *β-memory* não é generalizado pois algumas implementações do algoritmo RETE, como o OPS5 (BROWNSTON, FARRELL, KANT e MARTIN, 1985), usam duas referências para *α-memories* como entradas para o primeiro Nó de Junção da *β-network*.

comporta três Nós Filtros. O primeiro Nó Filtro *(B.x == TRUE)* avalia o (+FATO-2) similarmente como ocorreu com o (+FATO-1). Porém, o *token* não é prontamente armazenado em um nó *α-memory*, devido à existência de outros dois Nós Filtros suplementares, sendo necessário passar por um deles[14] (GIARRATANO e RILEY, 1993).

Após a avaliação do primeiro Nó Filtro, o (sub) Nó Filtro mais à esquerda é ativado *(B.y == TRUE)*, já que o (+FATO-2) guarda uma instância do elemento da base de fatos *B* com o estado *B.y* como TRUE. Assim sendo, a avaliação neste Nó Filtro é satisfeita, resultando na ativação da memória *α2*.

Por fim, a ativação da memória *α2* ativa a entrada à direita do Nó de Junção correspondente. Desta forma, o Nó de Junção correlaciona os *β-token(s)* na *β1-memory* (i.e. (+FATO-1)) com os *α-token(s)* na *α2-memory* (i.e. (+FATO-2)), gerando um outro *β-token* como saída. Esta saída ativa o *β2-memory* seguinte, que finalmente ativa a *Regra-1*, representada pelo Nó Regra, para o conjunto de conflito. Na fase de execução, a *Regra-1* usa o *β-token* que a ativou para referenciar os elementos da base de fatos em sua ação.

Além da inserção, o RETE oferece outras duas maneiras de atuar sobre a rede: removendo e modificando um *token*. Para remover um *token* da rede (representado por um sinal de subtração (-)) é adotado um processo similar ao de inserção. Porém, neste caso, o *token* deve percorrer todos os *α*- e *β*-memories para remover o *token* alvo. Para modificar o estado de um elemento da base de fatos é necessário que o *token* percorra a rede por duas vezes consecutivas. Na primeira vez, o respectivo *token* alvo é removido e na seqüência, o mesmo é inserido novamente com um valor modificado. O *token* modificado entra na rede assumindo um novo *Time Tag*.

2.8.3.3 TREAT (TREe Associative Temporal redundancy)

O algoritmo TREAT foi proposto por Miranker na intenção de melhorar o desempenho do algoritmo RETE (MIRANKER, 1987, 1989). Para isto, TREAT reutiliza a estrutura do RETE, atribuindo uma sutil alteração no modo em que a *β-network* é executada.

---

[14] Este atraso no armazenamento da informação é mais um aperfeiçoamento do algoritmo. Por outro lado, o uso de uma *α-memory* compartilhada por mais de um Nó Filtro pode se tornar um problema, principalmente quando há vários Nós Filtros em um mesmo nível. Em implementações mais simples, o *token* é avaliado em cada Nó Filtro afetando o desempenho. Porém, em implementações mais sofisticadas, uma Tabela *hash* é empregada para permitir acesso direto ao respectivo Nó Filtro por meio do valor armazenado no *token* (DOORENBOS, 1995).

Na execução desta rede, o TREAT evita guardar estados temporários nas *β-memories*, seguindo o princípio de que o custo de reavaliação das junções entre os elementos das *α-memories* é menor do que o custo de manter os estados nas *β-memories* (MCDERMOTT, NEWELL e MOORE, 1978).

A Figura 29 apresenta a estrutura de uma regra em TREAT, onde cada regra é representada por uma rede. Esta estrutura é similar a estrutura de uma regra em RETE (Figura 28), as quais se diferem em relação à inexistência das *β-memories*. Devido à similaridade de suas estruturas, o funcionamento do algoritmo TREAT também é similar ao funcionamento do RETE (MIRANKER, 1989).

Por exemplo, a rede ilustrada na Figura 29 apresenta a estrutura do algoritmo TREAT para a aprovação da Regra-1 (ao conjunto de conflito). Para que esta aprovação ocorra, considerar-se-á que a *α1-memory* já contém o *token* [A (x TRUE) (id 1)] e que o *token* [B (x TRUE) (id 1)] será inserido na rede para aprovação da Regra-1.

Desta forma, o *token* [B (x TRUE) (id 1)] entra na rede por meio do Nó Raiz e uma Tabela *Hash* é pesquisada a fim de identificar o Nó de Tipo a ser percorrido por este *token*, neste caso é o Nó de Tipo B. Com isto, a expressão causal referente ao Nó de Filtro subjacente, a qual atesta se o estado do atributo *x* de B é verdadeiro (B.x == TRUE), é avaliada e satisfeita, resultando no armazenamento de uma referência para este elemento na *α2-memory*.

Na seqüência, considerando que todas as *α-memories* da regra (*α1-memory* e *α2-memory*) contém elementos, o teste de consistência em relação aos *ids* de A e B é executado por meio de uma busca exaustiva, com a combinação dos valores de todos os elementos das respectivas *α-memories*. Este processo resulta na aprovação da Regra-1 ao conjunto de conflitos.

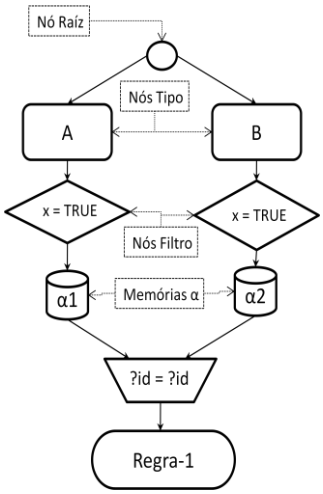


Figura 29: Estrutura do algoritmo TREAT

Apesar da pesquisa exaustiva e redundante, segundo os estudos apresentados em (MIRANKER, 1987), TREAT apresenta melhor desempenho do que o RETE. A melhora no desempenho se deve, principalmente, à diminuição das operações de gerenciamento de memória na inserção e remoção de elementos da base de fatos (MIRANKER, 1987). Por exemplo, nas remoções de elementos da base de fatos, o algoritmo TREAT apenas remove a referência (i.e. ponteiro) para o elemento armazenado nas *α-memories* e também remove as referências para as regras no conjunto de conflito, as quais são conexas a este elemento. Diferentemente, o RETE realiza buscas pelos *tokens* replicados em várias *β-memories* a fim de reprovar as regras no conjunto de conflitos. No entanto, apesar dos avanços no desempenho, o RETE continua sendo o algoritmo mais aplicado comercialmente (FRIEDMAN-HILL, 2003).

2.8.3.4 LEAPS (Lazy Evaluation Algorithm for Production Systems)

O algoritmo LEAPS foi proposto também por Miranker com a intenção de prover melhorias ao desempenho dos algoritmos de inferência, tal como o TREAT e o RETE (MIRANKER, BRANT, LOFASO e GADBOIS, 1990). O LEAPS exibe estrutura de rede similar ao RETE e ao TREAT, como apresentado na Figura 30, se apresentando como uma evolução desses. Porém, LEAPS se difere na forma pela qual as regras são avaliadas e executadas.

Nos algoritmos TREAT e RETE, a inserção de um elemento da base de fatos pode acarretar a aprovação de várias regras para o conjunto de conflito, no entanto, a execução da primeira regra aprovada pode resultar na desaprovação das demais, afetando o desempenho destes algoritmos, uma vez que as regras foram aprovadas e não foram executadas. Isto se deve porque estes algoritmos realizam a avaliação antecipada (*eager evaluation*) das regras para a execução (MIRANKER, BRANT, LOFASO e GADBOIS, 1990).

O LEAPS implementa a avaliação tardia (*lazy evaluation*) das regras. Em LEAPS, após a constatação da aprovação de uma regra, ao invés de ela ser adicionada ao conjunto de conflito, a regra é imediatamente executada, adiando as avaliações das demais, evitando assim aprovações desnecessárias (MIRANKER, BRANT, LOFASO e GADBOIS, 1990).

A essência do algoritmo LEASP consiste em buscas para a aprovação das regras sustentada por uma pilha de referência para os elementos da base de fatos. Esta pilha guarda referência para os últimos elementos inseridos na base de fatos. Os elementos da base de fatos

são identificados por números identificadores chamados de *timestamps*, os quais variam seqüencialmente com a inserção ou modificação dos elementos. O elemento da base de fatos referenciado pelo topo da pilha é chamado de *Timestamp* Dominante (TD). Este consiste ao último elemento inserido na rede e com o maior valor de *timestamp*.

O processo de aprovação de uma regra em LEAPS pode ser observado no exemplo da Figura 30. Nesta, as memórias $\alpha 1$ e $\alpha 2$ são representadas por duas tabelas, onde as colunas correspondem respectivamente aos valores do *timestamp* (ts) e aos atributos *x* e *id* das classes A e B. Cada uma destas tabelas guardam referência para dois elementos da base de fatos. Na mesma figura, a pilha de referencia aos elementos da base de fatos contém duas colunas, as quais correspondem às memórias $\alpha 1$ e $\alpha 2$, respectivamente. Nesta pilha, o TD apresenta *timestamp* 4 (*ts(4)*) (MIRANKER, BRANT, LOFASO e GADBOIS, 1990).

Desta forma, para que a Regra-1 seja aprovada, a referência para o elemento da base de fatos com *ts(4)* é extraída do topo da pilha. Este elemento está armazenado na *α1-memory*. Em seguida, a busca para avaliação entre o elemento da base de fatos com ts(4) e os elementos da *α2-memory* é inicializada. Primeiramente, o elemento da base de fatos com ts(4) é comparado com o elemento com ts(3), não havendo aprovação. Posteriormente, o elemento da base de fatos com ts(4) é comparado com o elemento com ts(2), neste caso, os *ids* são iguais e a regra é aprovada (Miranker, Brant, Lofaso, & Gadbois, 1990).

Em LEAPS, a regra não é enviada para o conjunto de conflito como ocorre em RETE e TREAT. Diferentemente, a busca de novas aprovações pára e a regra recém aprovada é executada imediatamente (MIRANKER, BRANT, LOFASO e GADBOIS, 1990).

No entanto, se nenhuma regra houvesse sido aprovada com o elemento da base de fatos com ts(4), a referência para o elemento com ts(3) seria extraído da pilha e uma nova busca se iniciaria a fim de aprovar novas regras. Mas de qualquer forma, após a finalização da busca com o elemento da base de fatos com ts(4), o elemento com ts(3) será extraído da pilha e a busca continuará (MIRANKER, BRANT, LOFASO e GADBOIS, 1990).

Porém, o elemento com ts(3) não será comparado com o elemento com ts(4), uma vez que a busca só ocorre em relação aos elementos da base de fatos da *α-memory* que apresentam *timestamp* menor do que o *timestamp* do TD. Neste caso, como o TD possui ts(3), a avaliação somente será realizada em relação ao elemento da base de fatos com ts(1). Assim, a busca continuará nesta direção até a pilha se tornar vazia, quando a aplicação termina a sua execução (MIRANKER, BRANT, LOFASO e GADBOIS, 1990).

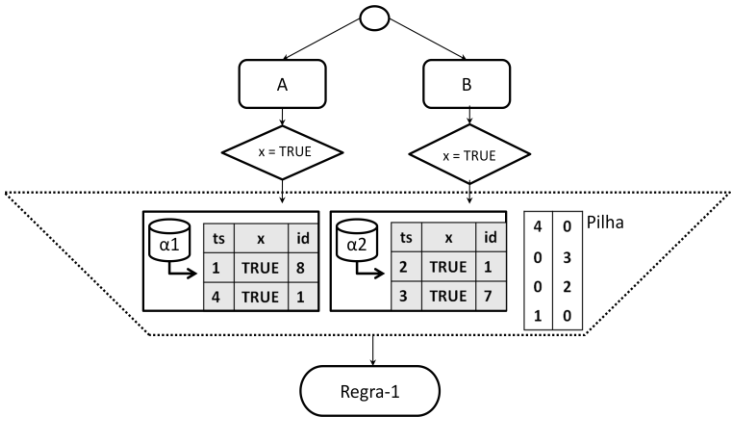


Figura 30: Estrutura do LEAPS

De fato, a proposta do algoritmo LEAPS é pertinente para prover melhoramentos no desempenho dos algoritmos RETE e TREAT, como pode ser constatado em (MIRANKER, BRANT, LOFASO e GADBOIS, 1990). No entanto, esta solução continua apresentando buscas sobre os elementos passivos, principalmente no co-relacionamento dos fatos.

2.8.3.5 HAL (Heuristically-Annotated-Linkage)

O algoritmo HAL (LEE e CHENG, 2002) apresenta uma solução bem diferente do RETE, TREAT e LEAPS. HAL evita as buscas pelas α-memories para satisfazer os relacionamentos entre os fatos (testes de consistência entre fatos). Ao invés disto, o HAL utiliza o conhecimento heurístico embutido nas regras e nos elementos da base de fatos para que estas relações sejam satisfeitas (LEE e CHENG, 2002).

A cooperação entre estes elementos ocorrem por meio de uma única rede global ao invés de várias redes locais (ou uma rede por regra) como ocorre nos demais algoritmos. Esta rede é composta por nós de alto nível (classes), de baixo nível (regras) e, quando necessário, nós de nível intermediário (i.e. nós de junção que realizam testes de consistência) (LEE e CHENG, 2002).

A construção da rede ocorre pela ligação bidirecional entre classes e regras ou classes e nós intermediários. As classes que não apresentam atributos envolvidos em testes de consistência são ligadas diretamente às regras, recebendo destas a responsabilidade de controlar as avaliações lógicas sobre a premissa que referencia seus atributos. Assim, esta deve notificar a regra somente quando o estado lógico da premissa for alterado.

As classes que apresentam atributos envolvidos em testes de consistência são ligadas indiretamente às regras por meio dos nós intermediários. Nesta esquematização, uma

premissa que correlaciona os fatos de duas ou mais classes é registrada a um nó intermediário, o qual é conectado com as respectivas classes (LEE e CHENG, 2002). Neste caso, o próprio nó intermediário é responsável por controlar as avaliações lógicas para a premissa pertinente, devendo atualizar o estado lógico da regra conectada somente quando o estado lógico da premissa for alterado.

Devido a esta flexibilidade proporcionada, uma aplicação que não apresenta características combinatórias pode ser implementada sem o emprego de nós intermediários. Com isto, evita-se o processamento gasto para co-relacionar os fatos. Esta mesma flexibilidade não existe nos algoritmos descritos nas subseções precedentes.

Neste âmbito, a Figura 31 apresenta a estrutura do algoritmo HAL aplicada a problemas que não envolvem características combinatórias. Nesta figura, a estrutura representativa do algoritmo está envolta de componentes estruturais do SBR: a base de fatos, uma base de regras (representada pelo conjunto de regras em forma de símbolos circulares) e uma máquina de inferência, a qual executa o algoritmo.

Nesta figura, um nó Classe (i.e. representado pelo símbolo retangular com o seu nome) representa todas as instâncias de elementos da base de fatos do seu tipo (i.e. representadas pelos símbolos circulares conectados às entidades Classes). Apesar deste nó ser denominado Classe, este é implementado na forma de um objeto com características genéricas. Da mesma forma, um nó Regra (i.e. representado pelo símbolo retangular com o seu nome) também é implementado na forma de um objeto com características genéricas para atuar sobre todas as instâncias de regras (i.e. representadas pelos símbolos circulares conectados às entidades Regras) que compartilham a sua estrutura. Estes objetos genéricos se comunicam a fim de representar as relações entre as suas instâncias mais específicas, conforme ilustra a figura. Esta comunicação é gerenciada pala máquina de inferência, uma vez que estes nós não apresentam características autônomas suficientes para coordenarem as suas ações.

Por exemplo, supondo que um determinado elemento da base de fatos foi inserido na base de fatos, isto gera um evento que ativa a execução da máquina de inferência. Primeiramente, esta direciona o elemento inserido para a sua respectiva classe por meio da identificação do tipo de classe que este elemento pertence. Feito isso, o estado do elemento é então avaliado sobre as premissas registradas em sua classe. Se alguma destas premissas for satisfeita, o mecanismo de inferência envia esta informação para a regra genérica (conectada à classe) para que esta atualize o estado lógico de uma de suas instâncias gerenciadas. Se alguma instância for satisfeita, a mesma é aprovada para o conjunto de conflito.

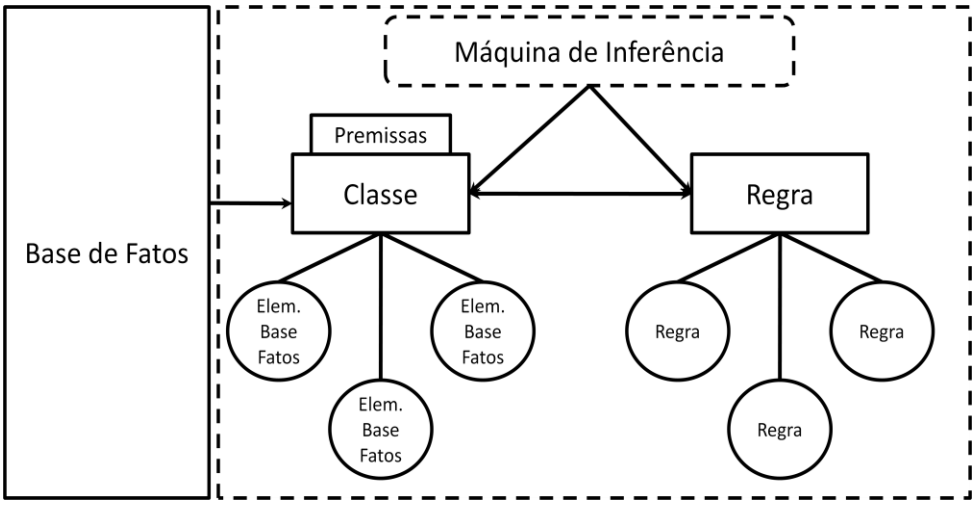


Figura 31: Arquitetura de um Sistema Baseado em Regras que infere sobre o HAL

A Figura 32 apresenta um exemplo que expressa o funcionamento deste algoritmo para problemas combinatórios. Nesta, uma premissa da Regra-1 relaciona os fatos das classes A e B pelos seus *ids*. Sendo assim, estas estão ligadas indiretamente à regra, uma vez que o nó intermediário é responsável por representar esta premissa (LEE e CHENG, 2002).

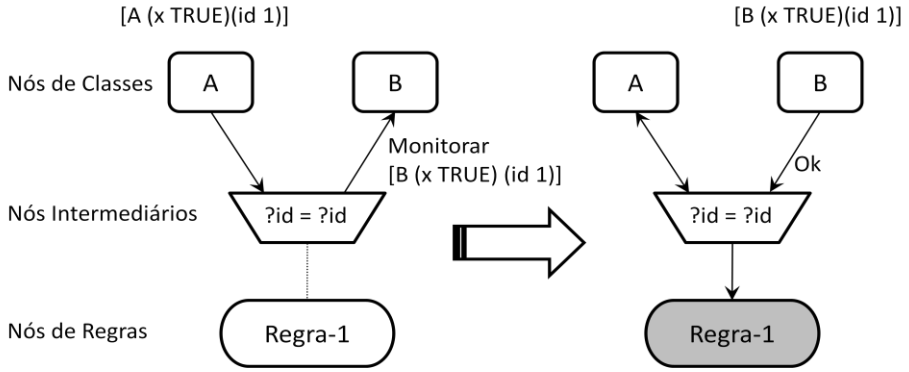


Figura 32: Estrutura do HAL para problemas combinatórios

Para que a Regra-1 seja aprovada, os testes de consistência entre os fatos das classes A e B devem ser realizados. No entanto, as buscas comparativas entre estes fatos são evitadas, uma vez que as classes conhecem (a priori) as premissas e regras pertinentes bem como os valores que resultarão na satisfação delas, evitando avaliações desnecessárias (LEE e CHENG, 2002).

Por exemplo, considerando a inserção do elemento da base de fatos [A (x TRUE) (id 1)] na Base de Fatos, a máquina de inferência verifica que a classe do elemento (i.e. a classe A) está relacionada a um Nó Intermediário devido ao teste de consistência pelos *ids*. Na seqüência, a máquina de inferência instiga a classe A para esta enviar uma notificação ao Nó Intermediário com os valores de seus atributos (i.e. fatos) (LEE e CHENG, 2002). Por sua vez, o Nó Intermediário extrai o valor do *id* e notifica a classe B para monitorar a chegada de um elemento com o *id* igual a 1.

Com a inserção do elemento da base de fatos [B (x TRUE) (id 1)] e a conseqüente atribuição deste elemento para a classe B, o mecanismo de inferência instiga a comparação dos fatos deste elemento com os valores monitorados, que resulta na identificação da chegada do elemento monitorado. Assim, o mecanismo de inferência instiga a notificação imediata do Nó Intermediário sobre este evento. Por sua vez, o Nó Intermediário recebe a notificação, constata a mudança do estado lógico da premissa pertinente e notifica a Regra-1 sobre a validação desta premissa. Por fim, a regra recebe a notificação, atualiza o seu estado e constata a sua aprovação (LEE e CHENG, 2002).

De fato, HAL apresenta uma solução bastante eficiente ao evitar, aparentemente, buscas sobre os elementos passivos. Neste algoritmo, as relações de inferência acontecem de maneira bastante pontual ao relacionar os elementos que são verdadeiramente pertinentes. Entretanto, apesar de se apresentar como uma solução eficiente, este algoritmo também apresenta algumas deficiências.

Em geral, as deficiências estão relacionadas à passividade dos seus elementos (i.e. classes e regras) e principalmente à estrutura genérica pela qual as regras tratam os elementos da base de fatos, ou seja, em termos de classes ao invés de uma forma mais específica, como objetos. A passividade dos elementos faz com que toda a reatividade do programa esteja alocada na Base de Fatos, a qual faz despertar o mecanismo de inferência deste algoritmo. Com isto, os elementos da base de fatos se tornam concentrados em uma única base monolítica e são manipulados nesta base por um mecanismo também monolítico, dificultando a execução distribuída das regras.

Além disso, a estrutura genérica dos elementos da base de fatos também dificulta a distribuição devido ao acoplamento, uma vez que uma classe concentra a responsabilidade de tratar de todos os objetos do seu tipo, centralizando este processo. Mesmo as interações entre as instâncias de elementos da base de fatos e de regras ocorrem por intermédio destas estruturas mais genéricas.

Ainda, por causa da generalidade das classes e mesmo das regras, o algoritmo HAL pode apresentar alguns sinais de buscas. Por exemplo, as buscas ocorrem devido ao monitoramento de um fato por uma classe, ou seja, quando a Base de Fatos delega um elemento da base de fatos à sua respectiva classe esta deve comparar o estado do elemento com os estados monitorados. Este processo comparativo pode resultar em avaliações desnecessárias, podendo afetar o desempenho do algoritmo quando uma classe monitora muitos dados.

2.8.3.6 Algoritmos de inferência interpretados e compilados

Apesar dos algoritmos de inferência apresentados, como o TREAT, LEAPS e HAL, apresentarem maior eficiência do que o RETE, atualmente, os *shells* mais populares como o JESS, OPS5, CLIPS, ILOG *Rules* e o *RuleWorks*, adotam o RETE como algoritmo de inferência.

No entanto, o fato de que a maioria dos *shells* implementam o mesmo algoritmo de inferência não significa que todos eles apresentam as mesmas capacidades. Os *shells* podem ser diferenciados em relação à oferta de funcionalidades, disponibilidade de suporte e preço. Neste âmbito, os *shells* mais sofisticados são os comerciais, como o JESS e o ILOG *Rules*. Ainda, os *shells* são diferenciados também por meio da forma que executam.

Os *shells* podem executar de forma interpretada ou compilada. No interpretado, os comandos do programa são transformados em código de máquina por demanda, enquanto que no compilado, o programa inteiro é transformado de uma única vez em código de máquina. Exemplos de *shells* interpretados e compilados são respectivamente o *CLIPS* (GIARRATANO e RILEY, 1993) e o *RuleWorks* (DIGITAL, 2008).

O *shell* CLIPS (*C Language Integrated Production System*) foi desenvolvido pela NASA utilizando a linguagem de programação C. O CLIPS é constituído de um editor de código para a sua linguagem, cuja sintaxe é baseada na linguagem LISP, e de janelas auxiliares para acompanhar o estado da execução dos programas, conforme ilustrou a Figura 26.

O CLIPS apresenta algumas características satisfatórias, como alta portabilidade, boa documentação, nenhum custo para uso e suporte a alguns conceitos das linguagens do POO. Estes conceitos são providos pela linguagem COOL (*C Object-Oriented Language*), integrada ao CLIPS. Porém, a COOL não apresenta a mesma flexibilidade e a capacidade das linguagens imperativas usuais como C++ e Java (GIARRATANO e RILEY, 1993).

O *shell RuleWorks* foi desenvolvido pela DIGITAL (DIGITAL, 2008). Ele foi escrito em C e, como o CLIPS, a sintaxe é similar ao da linguagem LISP. Apesar das semelhanças, *RuleWorks* gera aplicações mais eficientes do que o CLIPS, principalmente porque elas são compiladas.

Em *RuleWorks*, o processo de compilação ocorre por meio da tradução dos fatos e regras, implementados na linguagem do *RuleWorks*, para a linguagem C. Isto permite que o

programador inclua alguns trechos de código imperativo no código traduzido, para, enfim, compilar o código por meio de qualquer compilador C/C++ compatível.

No entanto, apesar de ser uma vantagem, a inclusão de código imperativo ao código traduzido pode gerar um código de difícil entendimento ou mesmo gerar erros de execução. Isto se deve, principalmente, por causa da ilegibilidade do código gerado automaticamente pelo tradutor e o emprego de diferentes sintaxes (código C e código do *RuleWorks*) em um mesmo programa.

Pelo fato do *RuleWorks* ser compilado, ele se diferencia das soluções interpretadas pela eficiência das aplicações geradas, pela flexibilidade na codificação e também na forma pela qual o RETE é implementado. *RuleWorks*, como qualquer outro *shell* que gera código compilado, implementa o RETE diferentemente da representação em rede apresentada na Figura 28 (DOORENBOS, 1995).

A versão compilada do algoritmo RETE substitui a representação em forma de arcos e nós da rede por um conjunto de funções específicas e otimizadas, ou seja, cada entrada dos nós é implementada em uma função particular (DOORENBOS, 1995). Como resultado, o código se torna maior do que o usado na representação tradicional, isto se deve ao grande número de funções especializadas que são geradas (DOORENBOS, 1995).

Geralmente, a versão interpretada implementa a rede tradicionalmente, ou seja, da mesma forma como ilustrado na Figura 28. Os nós da rede são representados explicitamente por módulos de dados que são percorridos por *tokens*. Cada tipo de nó (e.g. Nó Filtro e Nó de Junção), tem sua(s) entrada(s) implementadas por uma função genérica ao invés de uma especializada (DOORENBOS, 1995).

No entanto, geralmente as versões compiladas são preferíveis em relação às interpretadas, principalmente porque a eficiência é uma característica fundamental e necessária nos SBR.

2.8.4 Reflexões sobre o Paradigma Declarativo

Como visto na seção 2.7.4, o Paradigma Imperativo apresenta algumas deficiências que clamam por melhorias. Essencialmente, estas deficiências se referem à eficiência de execução, ao acoplamento entre as partes do código e às dificuldades de programação em ambientes monoprocessados e multiprocessados.

O Paradigma Declarativo se propõe a resolver algumas destas deficiências, principalmente a relacionada à dificuldade de programação em ambientes monoprocessados. Porém, a maioria destas deficiências ainda se repete por causa da sua própria natureza (i.e. buscas e elementos passivos). As seguintes subseções refletem sobre estas soluções, sendo que a subseção 2.8.4.1 discute sobre a dificuldade de programação, a subseção 2.8.4.2 discute sobre os acoplamentos e eficiência e por fim, a subseção 2.8.4.3 discute sobre as dificuldades na programação multiprocessada (i.e. paralela e distribuída).

2.8.4.1 Reflexões sobre as dificuldades de programação

As linguagens funcionais e lógicas previnem o programador da interação direta com os comandos das linguagens imperativas. Por exemplo, os comandos da linguagem LISP e PROLOG encapsulam os comandos imperativos, evitando o seu manuseio direto, enquanto os *shells* dos SBR permitem a composição da lógica da aplicação por meio de regras em ferramentas amigáveis, normalmente, evitando contato com as particularidades das linguagens imperativas.

No entanto, apesar destas soluções facilitarem a programação, elas não apresentam a mesma flexibilidade das linguagens imperativas. O uso destas soluções pode limitar a criatividade do programador e também o acesso aos componentes de *hardware* ou ainda impossibilitar a construção de códigos mais eficientes, uma vez que os comandos declarativos são implementados para uso genérico.

Uma solução para este problema seria unir as vantagens das linguagens imperativas e declarativas como já ocorre, por exemplo, em CLIPS++ (SBR/CLIPS e imperativo/C++) (OBERMEYER e MIRANKER, 1994) e ILOG Rules (SBR/ILOG Rules e Imperativo/C++ (ALBERT, 1994)). Com isto, os desenvolvedores poderiam usar características desejáveis de ambos os paradigmas de acordo com as suas necessidades.

No entanto, apesar desta união entre conceitos se apresentar desejável, ela não é amplamente utilizada devido a razões técnicas e culturais. Normalmente, os programadores adotam somente as linguagens imperativas para desenvolver as suas aplicações, principalmente as linguagens OO.

Este fato se deve principalmente à experiência adquirida nestas linguagens ou mesmo à resistência ou tempo indisponível para aprender uma nova abordagem. Além disto, este tipo

de programação resulta em código de difícil entendimento devido à mistura de sintaxes de linguagens de diferentes paradigmas.

Assim sendo, o ideal seria que ambos os paradigmas fossem apresentados em uma única linguagem por meio de uma sintaxe comum. Com isto, o programador poderia "pesar" o emprego do paradigma de acordo com o tipo de problema computacional. Desta forma, para abordar problemas que demandam maior flexibilidade, o programador poderia preferir o uso dos conceitos do PI, enquanto que, para abordar problemas onde a flexibilidade é menos importante do que as facilidades de programação, o PD poderia ser preferível.

2.8.4.2 Reflexão sobre acoplamento e eficiência

A reflexão sobre o PD e mesmo o PI vai certamente além da questão das facilidades de programação, incluindo, dentre outras, a questão do acoplamento. Mesmo se o PI e o PD apresentam diferenças evidentes em relação aos seus estilos de programação, estes combinam em relação às suas deficiências relativas ao acoplamento. Nestes paradigmas, os mecanismos internos de execução são baseados em buscas sobre elementos passivos, podendo gerar dependências entre os seus componentes.

No Paradigma Funcional, as buscas ocorrem por meio da iteração pelas listas que armazenam o programa (vide seção 2.4.3). No Paradigma Lógico, implementado na linguagem PROLOG, a busca é exaustiva ocorrendo em profundidade com retorno através de um grafo formado por componentes de regras e elementos da base de fatos (vide seção 2.4.4). Nos SBR puros, a busca também é exaustiva sobre a comparação cíclica de cada fato a cada regra. Mesmo nos SBR atuais, os quais são implementados com algoritmos de inferência inteligentes, a busca também existe. Por exemplo, no RETE, um *token* percorre a rede a fim de remover os respectivos *β-tokens* das *β-memories* relacionadas (vide seção 2.8.3.2) ou então, nos testes de consistência realizados pelos Nós de Junção.

No entanto, apesar de apresentar buscas, este algoritmo provê uma solução inteligente aos problemas das redundâncias do PI. Este resolve a redundância temporal ao memorizar estados dos fatos já avaliados em outros ciclos de inferência e resolve a redundância estrutural ao compartilhar estados dos fatos entre diferentes expressões causais. Além do RETE, o HAL também apresenta uma solução inteligente para este mesmo problema. No HAL, as avaliações desnecessárias são evitadas ao evitar ao máximo as buscas pelos estados dos fatos. Neste, as expressões causais são avaliadas apenas quando os elementos da base de fatos assumem o

estado pertinente para a expressão causal. Neste caso, os elementos da base de fatos notificam as expressões para que estas sejam avaliadas.

Ademais, no PF e no PL desprovido destes algoritmos de inferência, as redundâncias continuam sendo um problema. No PF, a aplicação de laços de repetição nas funções ou mesmo a recursividade aplicada sobre as funções acarreta em ciclos de repetição de uma mesma parcela de código, a qual pode conter expressões causais e, por conseguinte, avaliações desnecessárias sobre estas. No PL, a ocorrência de buscas por retorno expressa bem a existência de processamento desnecessário, uma vez que as buscas ocorrem "cegamente" em relação aos fatos. O mesmo é constatado na implementação dos SBR puros. Em suma, o problema da redundância ocorre porque todos estes paradigmas têm seus modelos de execução baseados em buscas sobre entidades passivas.

2.8.4.3 Reflexão sobre o paralelismo e distribuição

As execuções baseadas em buscas afetam o desempenho das aplicações baseadas nos atuais paradigmas, porém, nem sempre, apesar da interdependência entre os dados e comandos, este mecanismo de execução acarreta no acoplamento entre as partes de um programa. No PF, por exemplo, apesar de se constatar redundâncias na execução das aplicações, as partes de um programa não são fortemente acopladas devido às características naturais do paradigma (i.e. evita atribuição de valores). Desta forma, o PF apresenta facilidades de atuação na computação paralela, uma vez que a execução das funções argumentos pode ocorrer de forma independente (MITCHELL, 2003).

No Paradigma Lógico propriamente dito (i.e. materializado na linguagem PROLOG), a computação paralela é complicada, principalmente porque os fatos (dados) e regras (comandos) são armazenados em uma única base e o mecanismo de execução (inferência) é monolítico, ou seja, ele centraliza todo o processo de inferência. Certamente, há algumas soluções que viabilizam a execução deste paradigma de forma paralela (HERMENEGILDO, 1985; CORNEY e KIBLER, 1981; QUINN, 1987), porém, os ganhos em desempenho não são tão altos devido aos problemas de sincronização e comunicação entre as partes (i.e. premissas) da regra, como expresso em (LAU e YAN, 1991).

Nos SBR, apesar de serem compostos por entidades modulares com responsabilidades distintas e bem definidas, ainda o processo de inferência é monolítico devido a unicidade da Máquina de Inferência. Neste caso, mesmo se a base de conhecimento fosse decomposta em

sub-bases de conhecimento e distribuída em diferentes nós de processamento, ainda haveria um acoplamento no sentido que a MI é responsável por buscar pelos estados dos objetos nas sub-bases de conhecimento e correlacioná-los como detalhado em (SIMÃO, 2005).

Mesmo em uma arquitetura onde cada nó de processamento contém uma MI particular, as sub-bases de conhecimento deveriam ser concebidas com máxima prudência em relação ao desacoplamento entre os seus elementos, a fim de evitar ao máximo a comunicação (para fim de sincronização) entre as MI.

Apesar das dificuldades, algumas soluções de SBR paralelos e distribuídos foram propostas (WILSCY e PARAMESWARAN, 1992; KELLY e SEVIORA, 1989). Comumente, estas soluções empregam o algoritmo RETE. Algumas soluções paralelizam a fase de *matching* dos SBR (KALP, 1988; GUPTA, 1986; STOLFO, 1984) por meio da execução paralela dos nós da rede. Outras paralelizam a fase de *execução* (ISHIDA, 1990; MALDOVAN, 1986), quando as regras são executadas de forma paralela após a resolução de conflito.

O *shell* UMASS oferece a paralelização de ambas as fases (NEIMAN, 1992). No entanto, ambas as formas de paralelização são difíceis de serem alcançadas. Estas demandam excessivos comandos de sincronização, principalmente para acessar os *tokens* armazenados nas memórias da rede.

Uma alternativa para evitar o excesso de sincronismo em relação o acesso às memórias das regras, seria separar as regras em redes independentes e distribuí-las entre os nós processadores. Desta forma, cada rede apresentaria sua própria cópia dos elementos da memória. No entanto, esta alternativa resultaria em redundância de dados e redundância nas operações em relação a atualizações sobre estes dados, as quais precisariam ocorrer em cada memória para manter consistência entre os dados.

Além disto, a rede continuaria sendo percorrida de forma seqüencial e a máxima capacidade da paralelização não seria atingida, pois dependendo do fluxo de execução, as execuções poderiam sobrecarregar um processador enquanto outros poderiam se encontrar ociosos.

Por este fato, a paralelização em termos de nós da rede é a mais adequada do que em termos de rede (NEIMAN, 1992). Todavia, mesmo nesta condição, a paralelização ainda não é completa. Por exemplo, a fase de *matching* deve terminar antes da resolução de conflito ser executada para que as regras sejam selecionadas para a execução em paralelo. Desta forma, enquanto a fase de *matching* e a fase de *execução* estão sendo executadas, a fase de seleção

aguarda os seus términos. Em suma, a distribuição com as atuais tecnologias do PD não é nada evidente.

2.9 CONCLUSÃO

Este capítulo introduziu o termo paradigma nas ciências físicas e na ciência da computação como também apresentou a ocorrência de mudanças de paradigmas nestas ciências. Ainda, este capítulo apresentou os atuais paradigmas de programação, como também as suas classificações em Paradigma Imperativo e Paradigma Declarativo. Em relação a estes paradigmas, este capítulo relatou concisamente sobre as suas principais deficiências.

Deste modo, pode-se concluir que devido às estas deficiências que acabam afetando a eficiência de execução (redundâncias temporal e estrutural no Paradigma Imperativo), facilidades de programação (no Paradigma Imperativo) e flexibilidade de programação (no Paradigma Declarativo) e a aplicação efetiva na computação paralela e distribuída, os atuais paradigmas não apresentam soluções apropriadas para a computação monoprocessada e muito menos para a computação multiprocessada.

Neste âmbito, surge a proposta de um novo paradigma de programação chamado Paradigma Orientado a Notificações. Este propõe soluções efetivas para as deficiências apresentadas e se apresenta como um paradigma com grande potencial de aplicabilidade.

CAPÍTULO 3

PARADIGMA ORIENTADO A NOTIFICAÇÕES

O objetivo deste capítulo é apresentar o Paradigma Orientado a Notificações (PON). Para isto, a seção 3.1 contextualiza o PON como um novo paradigma de programação. Subseqüentemente, a seção 3.2 introduz o modelo do paradigma proposto.

Por sua vez, a seção 3.3 apresenta os componentes elementares do PON que essencialmente são os objetos notificantes para tratar dos elementos da base de fatos e regras. Na verdade, os conceitos de objetos e elementos da base de fatos/regras serviram de inspiração à concepção dos componentes elementares do PON, os quais apresentam evoluções estruturais e comportamentais desses por meio de certas capacidades de colaboração baseada em notificações pontuais.

A seção 3.4 apresenta justamente o chamado mecanismo de notificações do PON. Estruturalmente, o PON apresenta os objetos para tratar elementos da base de fatos e regras na forma de composições de outros objetos menores. Todos estes objetos apresentam características comportamentais de certa autonomia, independência, reatividade e colaboração por notificações pontuais. Estes componentes colaboram formando este mecanismo ímpar de inferência, denominado mecanismo de notificações, que viabiliza (dentre outros benefícios) a execução monoprocessada e mesmo multiprocessada de forma otimizada.

A seção 3.5 apresenta um exemplo de sistema sobre o qual os conceitos do PON foram realmente aplicados e cuja aplicação se mostrou bastante efetiva (SIMÃO, 2005). Mais precisamente, este exemplo se refere a um controle das relações entre um conjunto de equipamentos industriais que atuam em uma célula de manufatura simulada para realizarem a execução de diferentes planos de produção. Esta simulação ocorreu sobre a ferramenta de projeto e de simulação de sistemas de manufatura chamada ANALYTICE II (SIMÃO, 2005).

A seção 3.6 aborda a questão de resolução de conflitos neste paradigma proposto. Ao bem da verdade, deve-se adotar um mecanismo suplementar de resolução de conflitos em PON a fim de organizar a execução das regras conflituosas (sejam elas locais, paralelas ou distribuídas).

De fato, o PON surge como um novo paradigma de programação com o objetivo de sanar as principais deficiências dos atuais paradigmas. A seção 3.7 discute as soluções propostas para estas deficiências. Por sua vez, a seção 3.8 descreve sobre os tipos de

problemas mais provavelmente apropriados ao PON e, por fim, a seção 3.9 conclui sobre o capítulo.

3.1 CONTEXTUALIZAÇÃO

O Paradigma Orientado a Notificações (PON) apresenta uma solução inspirada em partes do Paradigma Imperativo (PI) e do Paradigma Declarativo (PD). Na verdade, o PON é um paradigma emergente que se inspira e evolui particularmente de dois sub-paradigmas do PI e PD. De forma mais estrita, o PON se inspira no Paradigma Orientado a Objetos (POO) e no Paradigma Lógico (PL), com ênfase nos conceitos dos Sistemas Baseados em Regras (SBR), os quais oferecem (até então) modelos de programação com maior proximidade à cognição humana.

O PON reaproveita os principais conceitos naturais do POO, como a abstração em forma de classes/objetos, a reatividade da programação dirigida a eventos e a flexibilidade na programação. O PON também reaproveita certos conceitos próprios dos SBR, como a representação do conhecimento em termos de regras e as facilidades da programação declarativa. A relação do PON com os paradigmas que inspiraram a sua solução está ilustrada na Figura 33.

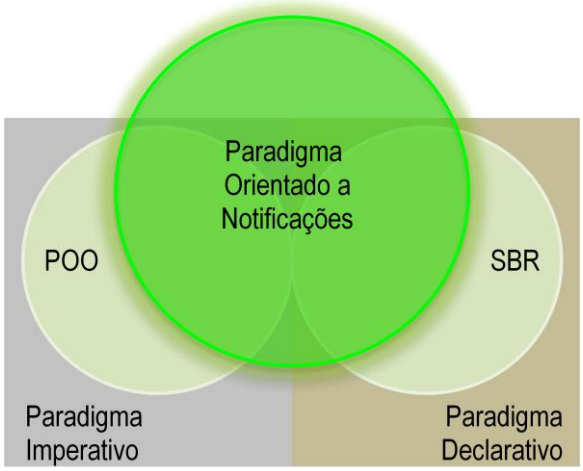


Figura 33: Relação entre o paradigma proposto e os atuais paradigmas

Na figura, devido ao reaproveitamento dos principais conceitos do POO e dos SBR, o PON inclusive apresenta-se como uma intersecção entre estes dois estilos de programação. Desta forma, esta representação explicita que o modelo proposto provê a possibilidade de uso (de partes) de ambos os estilos de programação em seu modelo de programação. Assim, o programador terá a liberdade de empregar alguns princípios do declarativo para se beneficiar

das facilidades de programação ao mesmo tempo em que poderá usar os princípios do imperativo para manter a flexibilidade na programação.

Apesar do PON se inspirar nesses estilos de programação, a sua solução é considerada ímpar, não consistindo em um mero incremento aos conceitos existentes. Na verdade, as suas contribuições são suficientes para resolver os problemas destes paradigmas e estabelecer efetivamente um novo paradigma de programação.

De forma similar, como visto na seção 2.4.4, o POO se baseou fortemente nos princípios do Paradigma Procedimental, provendo uma solução estrutural e organizacional para as deficiências deste paradigma. Por meio desta evolução e inovação de conceitos, o POO se apresentou como um novo e efetivo paradigma de programação.

Em concordância com esta passagem na história evolutiva da programação, não é uma condição obrigatória e necessária que se apresente conceitos totalmente e absolutamente inovadores para que um novo paradigma de programação seja proposto. Os conceitos pré-existentes podem ser melhorados ou evoluídos para apresentar uma nova perspectiva na concepção de programas, sendo considerado um novo paradigma de programação devido a esta evolução.

Segundo as palavras de Kuhn, "uma nova teoria nunca ou quase nunca é um mero incremento do que já é conhecido. Sua assimilação requer a reconstrução da teoria precedente e a reavaliação dos fatos anteriores" (KUHN, 1970). Em assentimento às palavras de Kuhn, o PON é apresentado neste capítulo como um novo paradigma de programação, que devido as suas qualidades, acredita-se muito em sua aceitação e popularidade diante à comunidade da ciência da computação (BANASZEWSKI, SIMÃO, TACLA e STADZISZ, 2007).

3.2 O MODELO DO PARADIGMA PROPOSTO

O paradigma proposto consiste em um modelo que guia a mente do programador na concepção de programas por meio da evolução dos componentes elementares dos SBR, como os elementos da base de fatos e as regras. Todavia, estes componentes têm as suas estruturas e comportamentos evoluídos em relação aos SBR para que possam atuar efetivamente como parte de um mecanismo de inferência com características ímpares.

Estes componentes foram incorporados ao modelo proposto devido à estreita relação análoga com a estruturação e funcionamento natural do raciocínio humano, principalmente quando realizado sobre problemas de decisão (NEWELL e SIMON, 1972). Segundo as

constatações de Newell e Simon (1972), a mente humana percebe o mundo como um conjunto de estados de elementos (os fatos) e realiza ações sobre ou motivada por estes elementos e seus estados por meio da verificação de sentenças causais (as regras).

No mundo cibernético, os elementos e os fatos percebidos pela mente humana podem ser representados naturalmente por meio de objetos do POO. Estes elementos podem apresentar fatos pelo estado de seus atributos e executar ações pela execução de seus métodos. Devido a esta compatibilidade de representação, o PON adotou esta entidade estrutural do POO e, ademais, evoluiu-a para representar os elementos da base de fatos em uma forma não passiva. Essencialmente, o PON incorpora novas características a esta representação, como autonomia e capacidades colaborativas conforme detalhado adiante.

De forma similar à representação dos elementos da base de fatos, as regras também são representadas na forma de objetos não passivos. Na verdade, uma regra é representada por uma estrutura composta, a qual é formada por um conjunto pré-determinado de objetos autônomos e independentes com capacidades de colaboração também por notificação. Entretanto, estas "inteligências" destes objetos são criadas de maneira automatizada e transparente ao "programador".

Desta forma, unindo e evoluindo os conceitos de SBR (elementos da base de fatos e regras) com os de POO (objetos), o programador passará a conceber os programas de forma natural, correspondente àquela empregada (costumeiramente) na tomada de decisão sobre os problemas do seu cotidiano. Entretanto, a similaridade entre estes modelos se encontra apenas e apropriadamente na representação do conhecimento, diferenciando-se na forma pela qual o mecanismo de raciocínio é instigado e como o conhecimento é organizado. Estas semelhanças e dessemelhanças entre os modelos em questão são consideradas nas seções seguintes e são exemplificadas sobre o domínio do *software* de controle de uma célula de manufatura simulada.

3.3 A ESSÊNCIA DO MODELO PROPOSTO

Atualmente, os conceitos do PON estão materializados sobre uma linguagem de programação, a linguagem C++, por meio da qual os elementos da base de fatos e as regras são representados como objetos. No entanto, estes objetos apresentam qualidades que os diferem dos objetos típicos da OO. Por exemplo, os objetos elementos da base de fatos apresentam características de autonomia, reação e colaboração por notificações. Estes

detectam e reagem a mudanças em seus estados contribuindo para que os objetos regras conexos sejam avaliados.

Todavia, os objetos elementos da base de fatos se relacionam com os objetos regras de forma indireta. Na verdade, os objetos elementos da base de fatos contribuem para a avaliação dos objetos regras por intermédio de outros objetos, os objetos colaboradores.

Os objetos regras, com o auxílio de seus objetos colaboradores, gerenciam as suas próprias avaliações lógicas e decidem o momento de suas execuções. Estas avaliações são realizadas por meio de um mecanismo de inferência ímpar, baseado em notificações, dos objetos colaboradores. Este mecanismo permite que as avaliações lógicas ocorram eficientemente e independentemente uma das outras.

Esta seção tem como objetivo apresentar os objetos colaboradores dos objetos elementos da base de fatos e dos objetos regras, descrevendo as suas principais responsabilidades e colaboração por notificações. A seção 3.3.1 apresenta os objetos colaboradores dos elementos da base de fatos e a seção 3.3.2 apresenta os objetos colaboradores das regras.

3.3.1 Objetos colaboradores dos elementos da base de fatos

Em PON, um elemento da base de fatos é representado como uma subclasse da classe *FBE* (i.e. *Fact Base Element*). A classe *FBE* é usada para descrever estados e serviços de entidades reais ou cibernéticas em um problema computacional. Como exemplo, o conceito de *FBE* já foi aplicado na representação computacional de equipamentos virtuais sobre o domínio de uma célula de manufatura simulada[15]. Esta foi implementada sobre a ferramenta de projeto e de simulação de sistemas de manufatura chamada ANALYTICE II[16] (SIMÃO, 2005).

A Figura 34 apresenta três equipamentos virtuais que foram efetivamente representados por objetos *FBEs*. De forma mais precisa, estes equipamentos em questão são o *Puma560.1* (i.e. robô de transporte de produtos), o *Store3x3.1* (i.e. armazém com nove posições para armazenar produtos) e o *Table2P.1* (i.e. mesa com duas posições usada para

[15] Estes equipamentos são ditos “inteligentes” por estarem integrados ao sistema computacional simulado.

[16] Mais precisamente, estes equipamentos são representados computacionalmente por recursos virtuais que se comunicam com os equipamentos “simulados” por meio de uma rede de comunicação de dados no ANALYTICE II (SIMÃO, 2005).

armazenar produtos). Estes mesmos equipamentos estão apresentados na forma de subclasses de *FBE* em notação UML, à direita da mesma figura. Essencialmente, cada *Equipamento-FBE* representa os estados e serviços por meio de atributos e métodos.

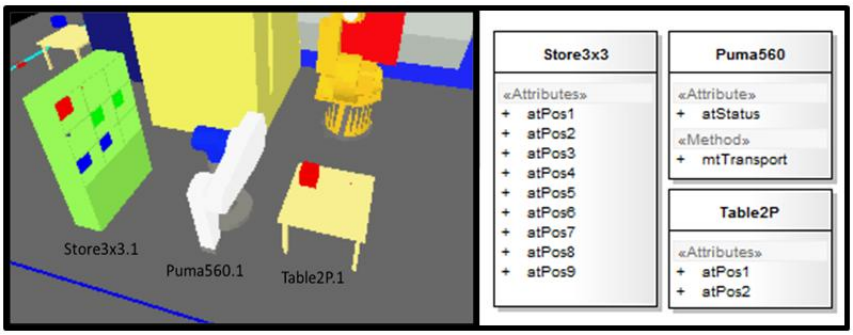


Figura 34: Representação parcial da célula de manufatura

Os atributos e métodos de um *FBE* (e.g. de um *Equipamento-FBE*) consistem nos objetos colaboradores desta instância. Deste modo, a representação de atributos e métodos na forma de objetos colaboradores se difere da representação padrão do POO, principalmente porque nesta os atributos e métodos são tratados como meras entidades passivas.

A Figura 35 apresenta a estrutura de um objeto *FBE*. Basicamente, um objeto *FBE* guarda referências aos objetos colaboradores representativos dos atributos e métodos. Assim, o objeto *FBE* pode referenciar vários objetos Atributos, os quais são representados como instâncias da classe *Attribute*. Da mesma forma, o objeto *FBE* pode referenciar vários objetos Métodos, estes são representados como instâncias da classe *Method*.

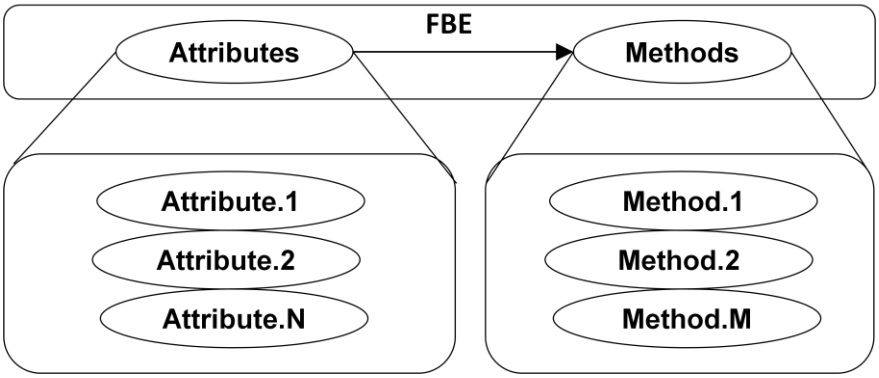


Figura 35: Estrutura do *FBE*

Particularmente, os objetos *Attributes* têm como responsabilidade reagir a mudanças em seus estados notificando as condições dos objetos regras. Na verdade, cada *Attribute* notifica cada mudança de estado ocorrido sobre si para os componentes colaboradores pertinentes das condições dos objetos regras conexos, os quais avaliam este estado e atualizam o estado lógico das suas respectivas regras.

Em relação aos objetos *Methods*, cada qual encapsula a funcionalidade de um método. Um objeto *Method* pode ser aplicado de duas formas diferentes. Ele pode ser usado para:

- guardar referência para a chamada de um método padrão de um objeto do POO,
- ou ainda para incorporar em seu próprio escopo a implementação da funcionalidade de um método.

Basicamente, a responsabilidade de um *Method* é receber notificações ou instigações de objetos colaboradores da ação de um objeto regra, os quais implicam na execução do seu método referência ou então da funcionalidade implementada diretamente em seu escopo.

A implementação dos atributos e métodos em forma de objetos colaboradores pode ser considerada como uma evolução da estrutura de classes da OO. Com isto, os objetos apresentam maior independência de suas partes e ainda podem agregar maiores capacidades para atuar de forma colaborativa com os demais objetos do modelo.

A idealização desta representação foi motivada por dois fatores principais. O primeiro fator se deve à necessidade de uma solução prática e implícita para incorporar características reativas aos objetos comuns. Incentivado por esta limitação do POO (falta de reatividade implícita nos atributos) e conseqüentemente de suas linguagens como a C++, optou-se por encapsular os tipos de dados primitivos (e.g. Booleano, inteiros, ponto-flutuante, caractere e cadeia de caracteres) em forma de objetos, incorporando a estes as características reativas faltantes.

O outro fator motivador se deve a necessidade de maior desacoplamento entre as partes dos objetos. Com a possibilidade de particionar os objetos em partes menores, como em termos de atributos e métodos, o programador pode se beneficiar das reais vantagens do uso da computação paralela e distribuída. Por exemplo, atributos e métodos podem ser separados dos objetos para executarem paralelamente em diferentes nós de processamento caso seja necessário.

Assim, com estes melhoramentos sobre a estrutura de um objeto, o PON se mostra mais efetivo do que o POO, uma vez que promove a atualização imediata dos estados lógicos de uma regra a partir da reatividade dos estados pertinentes. Ainda, o PON apresenta maior descentralização na execução das regras do que o PL/SBR e POO, principalmente pela possibilidade de receber notificações de atributos ou agir sobre os métodos de forma independente e desacoplada conforme detalhado na próxima subseção.

3.3.2 Objetos colaboradores das regras

Em PON, uma regra causal é tratada como uma instância da classe *Rule*. A Figura 36 ilustra uma regra causal que representa a essência de uma dada *Rule*. Esta *Rule* controla uma dada relação entre equipamentos (de uma célula de manufatura) apresentados à esquerda na Figura 34. Estes equipamentos são representados na forma de classes (derivadas da classe *FBE*) à direita na mesma figura com os seus respectivos *Attributes* e *Methods*.

Na *Rule* em questão, há três premissas que verificam se (a) o recurso *Puma560.1* se apresenta livre, (b) se o recurso *Table2P.1* não apresenta um produto sobre a posição 1 e (c) se o recurso *Store3x3.1* apresenta um produto na posição 1. Se estes estados forem constatados, a ação da *Rule* faz com que o robô *Puma560.1* transporte o produto da posição 1 do armazém *Store3x3.1* para a posição 1 da mesa *Table2P.1*.

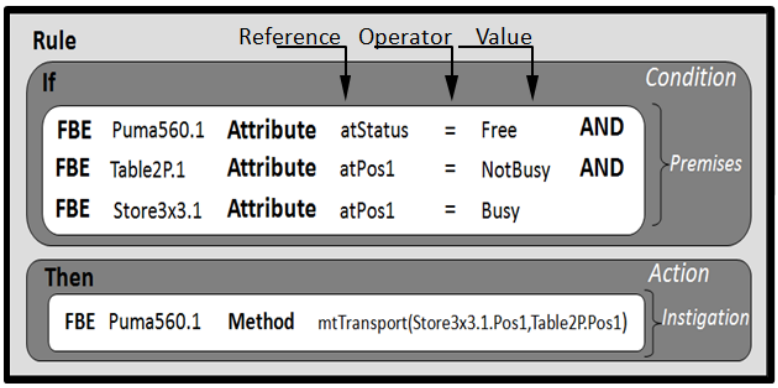


Figura 36: Representação de uma *Rule*

A representação desta *Rule* na forma de uma causal permite expor que cada *Rule* é composta por certos objetos colaboradores. A relação da *Rule* com estes objetos é ilustrada na Figura 37. Um objeto *Rule* é constituído diretamente por um objeto *Condition* e um objeto *Action*. Na verdade, os objetos *Condition* e *Action* cooperam a fim de realizar o conhecimento causal da regra.

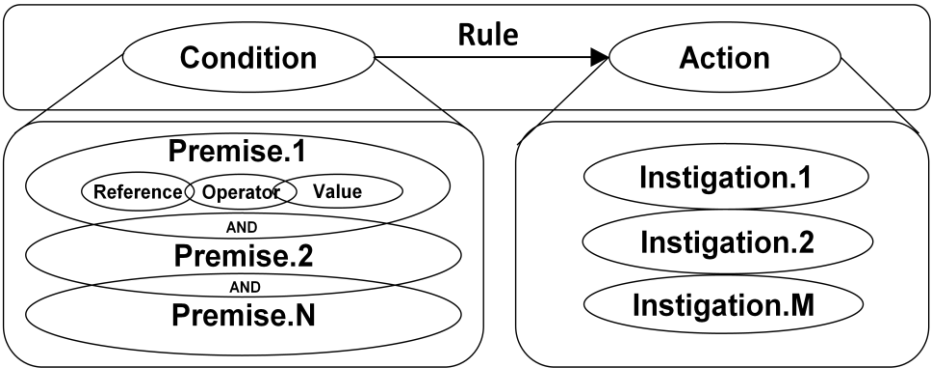


Figura 37: Estrutura da *Rule*

Ainda, um objeto *Rule* é também constituído (de certa forma) por objetos *Premises* e *Instigations*. Na verdade, os objetos *Instigations* se relacionam de forma indireta com os objetos *Rules* por intermédio do objeto *Action*. Um objeto *Action* pode apresentar referências para um ou vários objetos *Instigations*. Basicamente, os objetos *Instigations* constituem o corpo dos objetos *Action*.

No instante da execução de uma *Rule*, o objeto *Action* invoca os serviços de cada *Instigation* conexo a ele, de forma seqüencial ou concorrente conforme se deseje. A responsabilidade de um objeto *Instigation* é invocar os serviços dos objetos *Methods* (de forma local ou remota), provendo os parâmetros corretos quando necessário. Por meio dos objetos *Instigations*, o objeto *Rule* altera os estados dos objetos *FBEs*.

Os objetos *Premises* também se relacionam com os objetos *Rules* por intermédio de outros objetos, neste caso os objetos *Conditions*. Um objeto *Condition* apresenta conexão com um ou mais objetos *Premises*. Cada objeto *Premise* consiste em um teste lógico sobre os valores de um ou dois objetos *Attributes* de um ou dois *FBEs*. Deste modo, conforme ilustra a Figura 37, um objeto *Premise* é composto de três elementos principais: *Reference*, *Operator* e *Value*.

O elemento *Reference* guarda referência a um objeto *Attribute*, o qual notifica o objeto *Premise* sobre a mudança de seu estado. O elemento *Operator* corresponde a um operador lógico, o qual é usado para realizar comparações entre o valor do *Reference* e o valor do elemento *Value*, que pode consistir em uma simples constante ou uma referência para outro objeto *Attribute*.

No âmbito dos objetos *Conditions*, os objetos *Premises* são relacionados entre si por meio de operações de conjunção (operador AND) e disjunção (operador OR) a fim de estabelecer o valor lógico da condição (*Condition*). Estas operações são realizadas devido à notificação de valores lógicos dos objetos *Premises* para os objetos *Conditions*.

Na verdade, em PON, uma *Rule* é aprovada para execução simplesmente por meio da cooperação por notificações entre os seus objetos colaboradores, mais precisamente pela cooperação entre os objetos *Attributes*, *Premises* e *Conditions* conforme é explicado na seção que segue.

3.4 MECANISMO DE NOTIFICAÇÕES

3.4.1 Visão geral

O mecanismo de notificações consiste no mecanismo interno de execução das instâncias do paradigma proposto, o qual determina o fluxo de execução das aplicações. Por meio deste mecanismo, as responsabilidades de um programa são divididas entre os objetos do modelo, os quais cooperam por meio de notificações, informando uns aos outros as parcelas de suas contribuições a fim de formar o fluxo de execução do programa.

As relações pelas quais os objetos colaboram é apresentada no diagrama de classe da UML na Figura 38. Neste, os objetos das classes *Rule* e *FBE* se apresentam em extremidades opostas e se relacionam com o auxílio dos objetos colaboradores conforme as conexões modeladas[17].

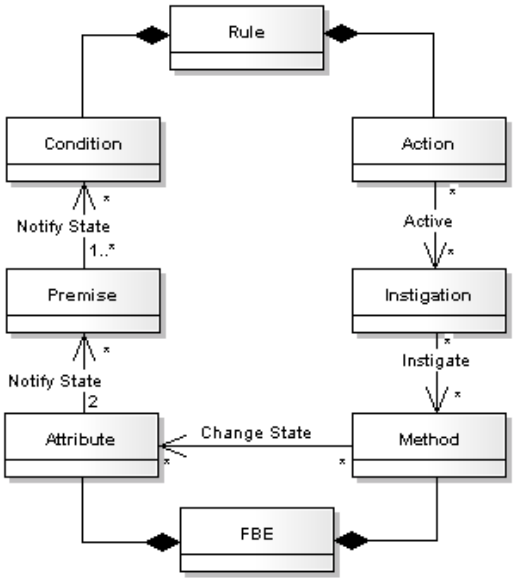


Figura 38: Relação entre os objetos principais e colaboradores

Estas conexões são estabelecidas em tempo de execução na medida em que os objetos são criados. Por exemplo, na criação de um objeto *Premise* pelo menos um objeto *Attribute* é considerado como o seu *Reference*. Uma vez que um *Attribute* é referenciado em uma *Premise*, o *Attribute* considera automaticamente esta *Premise* como sendo interessada em

[17] Nesta ilustração, podem-se constatar a modelagem de dois fluxos opostos de notificações: o fluxo ativo e o passivo em relação aos objetos da extremidade. Por exemplo, o fluxo de notificações originado no *FBE* é ativo em relação ao *FBE* e passivo em relação ao *Rule*. De maneira oposta, o fluxo de notificações originado no *Rule* é ativo em relação à *Rule* e passivo em relação ao *FBE*. Estes fluxos são formados pela cooperação entre os objetos colaboradores destas extremidades.

receber notificações sobre o seu estado. Assim, o *Attribute* identifica todas as *Premises* interessadas e notifica-as quando os seus estados mudam.

Similarmente, quando uma *Premise* é conectada a uma *Condition*, a *Premise* considera automaticamente esta *Condition* como interessada em receber notificações sobre o seu estado. Assim, a *Premise* identifica todas as *Conditions* interessadas e notifica-as quando os seus estados mudam. Deste modo, após as devidas conexões serem estabelecidas, estes objetos estão aptos a se comunicarem por meio de notificações.

Essencialmente, a cooperação entre estes objetos se inicia a cada mudança no estado de um objeto *Attribute*. Devido à ocorrência deste evento, o próprio *Attribute* notifica imediatamente uma ou um conjunto de *Premises* relacionadas para que estas reavaliem os seus estados lógicos. Deste modo, uma *Premise* realiza um cálculo lógico a cada momento em que recebe notificações de um *Attribute*, comparando o elemento *Reference* com o *Value*, usando o elemento *Operator*. Se o valor lógico da *Premise* se altera, a *Premise* colabora com a avaliação lógica de uma ou de um conjunto de *Condition*s conectadas, que ocorre por meio da notificação sobre a mudança de seu estado lógico.

Conseqüentemente, cada *Condition* notificada avalia o seu valor lógico de acordo com as notificações da *Premise* e com o operador lógico (de conjunção ou disjunção) utilizado. Assim, no caso de uma conjunção, quando todas as *Premises* que integram uma *Condition* são satisfeitas (em estado verdadeiro), a *Condition* também é satisfeita, resultando na aprovação da sua respectiva *Rule* para a execução.

Na verdade, o momento exato da execução de uma *Rule* é determinado após a resolução de conflito entre esta e as demais regras aprovadas, se houver conflitos. Um conflito ocorre quando duas ou mais regras referenciam um mesmo recurso e demandam exclusividade de acesso a este recurso. Deste modo, as regras concorrem para adquirir acesso exclusivo a este recurso, sendo que somente uma destas regras em conflito pode executar por vez, a qual obteve o acesso exclusivo.

Com os conflitos solucionados, a respectiva *Rule* está pronta para executar o conteúdo da sua *Action*. Uma *Action* é conectada a um ou vários *Instigations*. Os *Instigations* colaboram com as atividades das *Action*s, acionando a execução de algum serviço de um objeto *FBE* por meio dos seus objetos *Methods*. Geralmente, as chamadas para os *Methods* mudam os estados dos *Attributes* e o ciclo de notificação recomeça.

Com esta explanação, percebe-se que objetos colaboradores se apresentam desacoplados ou “minimamente acoplados conforme o ponto de vista empregado” devido à comunicação realizada por meio de notificações. Estas características comportamentais (i.e.

comunicações por notificações) e estruturais (i.e. desacoplamento) favorecem a aplicação do mecanismo de notificações para ambientes multiprocessados, pois apenas se faz necessário que um objeto notificante conheça o endereço do objeto notificado para que uma notificação ocorra. Além do mais, as notificações são pontuais e necessárias devido à mudança de estado do *Attribute*. Tal fato colabora para reduzir as comunicações entre os nós de processamento e para otimizar o processamento em cada nó (em termos de avaliações causais).

3.4.2 Exemplificação

O mecanismo de notificação pode ser mais bem compreendido por meio de um simples exemplo prático. Este exemplo exibe as colaborações entre os objetos colaboradores na aprovação da mesma *Rule* apresentada na Figura 36, a qual controla as relações entre os equipamentos virtuais *Puma560.1*, *Table2P.1* e *Store3x3.1*. Estes são instâncias das classes *FBE* apresentadas na Figura 34 à direita da representação parcial da célula de manufatura em questão.

Ao considerar estes equipamentos virtuais como *FBE*, os atributos destes são expressos por meio dos objetos *Attributes* e os serviços destes são disponibilizados por meio dos objetos *Methods*. Desta forma, a alteração de um estado de um equipamento (i.e. alteração de estado de um *Attribute*) ativa um fluxo de notificações que pode instigar a execução de um serviço (i.e. de um *Method*) em um dado equipamento. Esta colaboração por notificações pode ser verificada de maneira mais concreta pela interação dos elementos na aprovação da respectiva *Rule*. Esta colaboração por notificações é ilustrada na Figura 39.

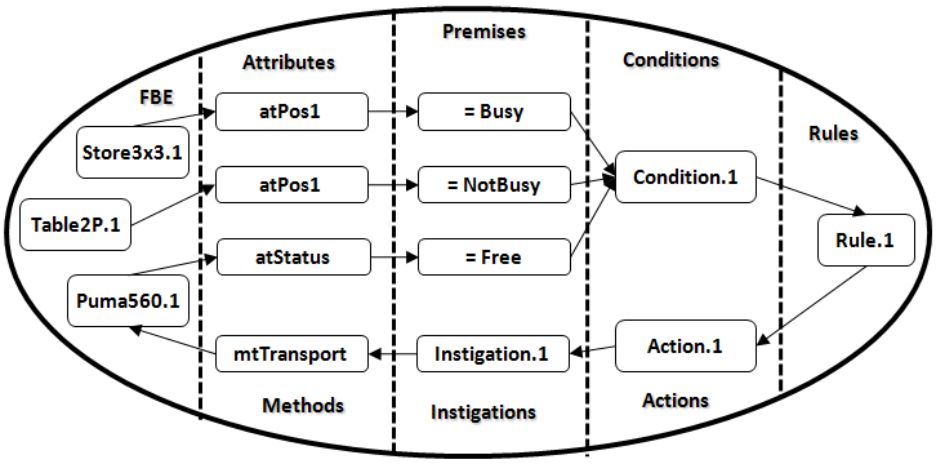


Figura 39: Exemplificação do mecanismo de notificações

Pela análise desta figura, pode-se perceber com clareza a essência do PON. Nesta, os objetos estão dispostos de forma desacoplada, mesmo na relação entre o *FBE* e *Rule* com os seus objetos colaboradores. Particularmente, cada objeto se comporta como uma unidade independente e autônoma. Entretanto, quando eles colaboram por meio do mecanismo de notificação, estes se harmonizam como uma única entidade devido à convergência para a execução de uma tarefa em comum.

A cooperação através do mecanismo de notificação é ativada com a mudança de qualquer estado encapsulado por um *Attribute*. Deste modo, no instante em que o robô inteligente *Puma560.1* se apresenta livre, esta informação é enviada para o seu *Attribute atStatus*. Este reage ao evento notificando as *Premises* pertinentes. Na representação acima, o respectivo *Attribute* está conectado a apenas uma *Premise* por motivos de simplificação.

Na seqüência, a *Premise* notificada compara o valor notificado de *atStatus* (*Reference*) com o valor do elemento *Value* usando o elemento *Operator*. Se diferenças entre os valores for constatado, significa que o estado lógico da *Premise* mudou desde a última avaliação[18]. Em relação ao exemplo, a *Premise* notificada é satisfeita pelo estado atual do *Attribute atStatus*, gerando uma notificação à *Condition.1*.

A *Condition.1* representa uma conjunção entre as três *Premises* conectadas, sendo que esta realiza o seu teste lógico de acordo com os estados lógicos notificados por estas *Premises*. Uma *Condition* realiza o seu teste lógico por meio da comparação entre dois valores, os quais se referem respectivamente ao número de *Premises* conectadas e um contador numérico que denota a quantidade destas *Premises* que estão com os seus estados lógicos verdadeiros. Assim, se a *Condition.1* representasse uma disjunção, esta seria considerada aprovada quando o contador numérico apresentasse valor maior ou igual a 1.

Particularmente, com a mudança do estado lógico da *Premise* em questão (*atStatus* == *FREE*) para verdadeiro, esta notifica a *Condition.1* para que esta incremente o seu contador de *Premises* verdadeiras e atualize o seu estado lógico em relação a este incremento. Assim, a *Condition.1* compara o contador numérico (valor 1) com a quantidade de *Premises* conectadas a ela (valor 3) para decidir sobre a aprovação da *Rule.1*. Nesta avaliação, a *Condition.1* ainda não foi satisfeita.

Para que a *Condition.1* seja satisfeita é necessário que as outras duas *Premises* a notifiquem sobre os seus estados lógicos verdadeiros, as quais se referem ao *Attribute atPos1*

[18] Em relação à primeira notificação, esta ocorre no instante em que um *Attribute* é conectado com uma *Premise*. Neste caso, a *Premise* é avaliada de acordo com o estado inicial deste *Attribute* podendo constatar a mudança de seu estado lógico para verdadeiro.

do *FBE Table2P.1* e ao *Attribute atPos1* do *FBE Store3x3.1*. Assim, considerando que a mesa *Table2P.1* apresenta a posição 1 desocupada, o seu *Attribute atPos1* notifica a *Premise* conectada (*atPos1 == NotBusy*) que realiza o seu teste lógico e instiga o fluxo de notificações até a *Condition.1*. Do mesmo modo, quando um produto é inserido na posição 1 do armazém *Store3x3.1*, o *Attribute atPos1* notifica a respectiva *Premise* (*atPos1 == Busy*), que também realiza o seu teste lógico e notifica a *Condition-1*. Esta, por sua vez, incrementa o seu contador numérico para estas duas *Premises* e realiza o seu cálculo lógico.

Assim, após a *Condition.1* ser atualizada pelas três *Premises* em questão, a própria constata a mudança de seu estado lógico para verdadeiro, pois agora todas as *Premises* conectadas também apresentam o estado lógico verdadeiro. Por fim, a *Condition.1* notifica a *Rule.1* sobre a sua aprovação.

Com a aprovação, a *Rule.1* executa o conteúdo da sua respectiva *Action* por meio de outro fluxo de notificações composto pelos objetos *Instigations* e *Methods* respectivos. Este fluxo segue na direção do *FBE Puma560.1* para instigar este a transportar o produto armazenado na posição 1 do armazém *Store3x3.1* para a posição 1 da mesa *Table2P.1*.

Outrossim, a *Rule.1* pode ser executada imediatamente após a aprovação ou então ela pode ser executada após a resolução de conflitos com outras regras, supondo que hajam tais regras conflituosas.

3.5 UM EXEMPLO DE APLICAÇÃO DO PARADIGMA ORIENTADO A NOTIFICAÇÕES

As seções anteriores apresentaram os conceitos de *FBEs*, *Rules* e de seus respectivos objetos colaboradores e como estes objetos colaboram por meio do mecanismo de notificações. Para isto, se fez uso da colaboração entre apenas três equipamentos virtuais (representados por *FBEs*) pertencentes a uma célula de manufatura, tendo uma dada colaboração controlada por apenas uma *Rule*.

No entanto, os conceitos do PON são aplicáveis a situações mais complexas do que a apresentada. Em (SIMÃO, 2005), os conceitos do PON foram aplicados para controlar relações que envolvem maior quantidade de equipamentos virtuais que exercem atividades controladas por uma maior quantidade de regras.

Para demonstrar esta aplicabilidade, esta seção aborda a mesma célula de manufatura em questão, mas em sua forma completa, apresentando as colaborações entre vários

equipamentos virtuais controladas por várias regras. Esta célula na sua forma completa é apresentada na Figura 40, a qual é adotada para a produção de dois produtos virtuais (V1 e V2). Esta exemplificação é descrita conforme está expresso em (SIMÃO, 2005).

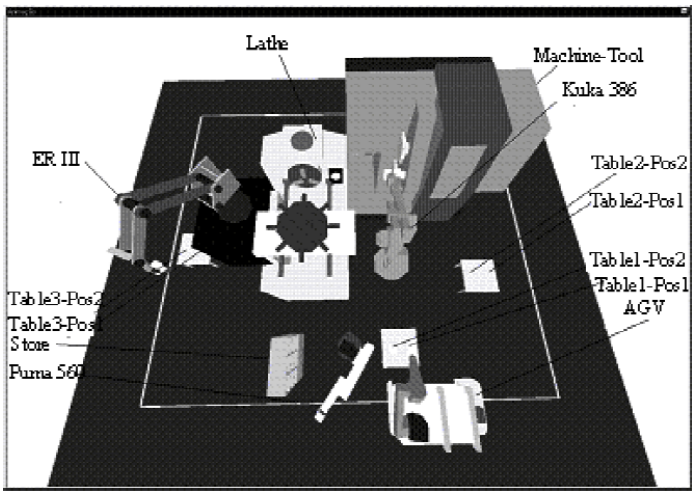


Figura 40: Célula de manufatura completa simulada na ferramenta ANALYTICE II

O controle das interações entre estes equipamentos virtuais é composto basicamente por *Rules*, as quais se subdividem em *Rules* para decisão e *Rules* para coordenação. Particularmente, as *Rules* para decisão definem quando os produtos podem ser inseridos (i.e. na forma de peças brutas) ou retirados da respectiva célula (i.e. quando estiverem devidamente processados). As *Rules* de coordenação, por sua vez, definem os momentos apropriados para a cooperação entre os equipamentos virtuais com o objetivo de concluir os passos de produção.

3.5.1 *Rules* de coordenação

Um conjunto de *Rules* foi criado para coordenar a cooperação entre os equipamentos virtuais a fim de possibilitar a produção dos produtos V1 e V2. As *Rules* foram elaboradas visando à cooperação entre um conjunto de recursos que permitem realizar as operações definidas nos planos de processo para cada tipo de produto.

Os planos de processo para os produtos V1 e V2 são definidos na Figura 41. Estes descrevem o processo guiado pelo armazenamento de um produto nas posições de armazenamento do armazém *Store3x3.1* e das três *Tables* (i.e. *Table2P.1*, *Table2P.2* e *Table2P.3*) até serem manipulados pelos respectivos equipamentos de fabricação de produtos, isto é, *Machine-Tool* (Máquina Ferramenta) para o produto V1 e *Lathe* (torno) para o produto V2.

Cada *Rule* de coordenação é responsável pela cooperação entre os equipamentos de fabricação de produtos e os equipamentos de transporte (i.e. *Puma560*, *Kuka386*, *AGV* (*Auto Guided Vehicles*)) para que estes transportem uma peça de um local de armazenamento para outro ou diretamente para um equipamento de fabricação de produtos.

V1: { < Store [pos 1, 2, 3, 4, 5, or 6] > < Table 1 [pos 2] > < Machine-Tool > < Table 2 [pos 1 or 2] > }

V2: { < Store [pos 7, 8, or 9] > < Table 1 [pos 2] > < Table 3 [pos 1] > < Lathe > < Table 3 [pos 2] > }

Figura 41: Plano de produção para os produtos V1 e V2

Com relação a estes planos de produção, a Figura 42 apresenta um conjunto de *Rules* que controlam o transporte de produtos do armazém *Store3x3.1* para a posição 1 ou 2 da mesa *Table2P.1*, sendo que a posição na respectiva mesa depende dos direcionamentos expressos na *Rule* aprovada. Estas *Rules* permitem a execução dos primeiros passos dos planos de processo para os produtos V1 e V2 ao transportar os produtos nesta seqüência. Complementarmente, a Figura 43 apresenta outro exemplo de *Rules*, as quais habilitam os passos subseqüentes definidos nesses planos de processo. Nestas figuras, as *Rules* se apresentam auto-explicativas por estarem expressas na forma de expressões causais[19].

3.5.2 *Rules* em conflito

Na Figura 42, todas as nove *Rules* usam a *Premise* "*Puma560.1 Status = Free*". As mesmas se apresentam em conflito devido a restrições físicas do recurso conflitante *Puma560.1*, uma vez que somente uma *Rule* aprovada pode efetivamente instigar o serviço de transporte deste equipamento. Para evitar este conflito, as nove *Rules* foram concebidas de forma inter-bloqueada, ou seja, assegurando por verificações suplementares que apenas uma destas *Rules* é realmente aprovada e consecutivamente executada por vez.

[19] As *Rules* e os objetos colaboradores são apresentados na forma de expressões causais para facilitar o entendimento das relações entre os equipamentos. Certamente, esses podem ser apresentados na forma de diagramas em UML, uma vez que estes são implementados como objetos.

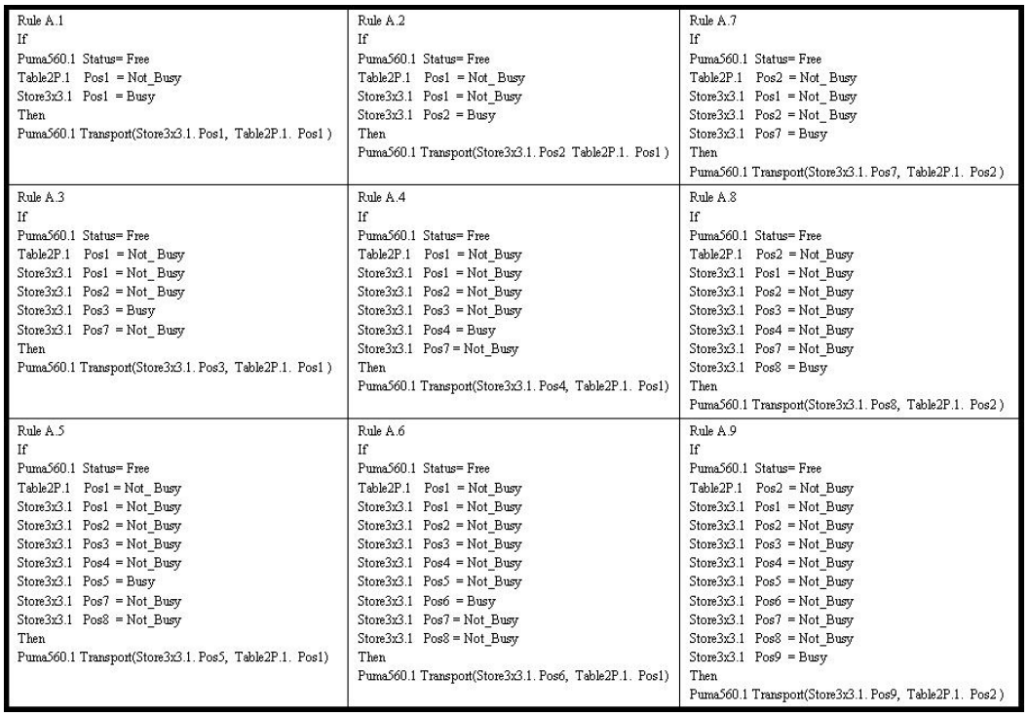


Figura 42: Regras entrelaçadas

Um mecanismo de resolução de conflitos pode ser desenvolvido para evitar automaticamente o compartilhamento de um recurso conflitante, sem a necessidade da concepção de *Rules* inter-bloqueadas. Este mecanismo identificaria automaticamente os conflitos entre as *Rules* e concederia a prioridade de execução para apenas uma destas *Rules*.

Ainda, esta forma automatizada de resolução de conflitos facilitaria a criação das *Rules*. Por exemplo, conforme a Figura 40, a *RuleA.9* precisaria de somente três premissas ao invés de onze. As outras oito *Premises* que avaliam se as posições do armazém *Store3x3.1* estão desocupadas não seriam mais necessárias. Alguns mecanismos de resolução de conflitos foram criados e serão discutidos mais apropriadamente na seção 3.6.

3.5.3 Características das *Rules*

O conjunto de *Rules* apresentado na Figura 42 também permite a demonstração de algumas propriedades do paradigma proposto. Por exemplo, este conjunto permite a demonstração do compartilhamento da colaboração entre alguns objetos (tal como os objetos *Premises* e *Instigations*) e as qualidades do mecanismo de notificações.

Portanto, este conjunto de *Rules* permite a percepção de que as *Rules* apresentam similaridades estruturais. De fato, as *Rules* são muito similares, com a única diferença sendo o conhecimento de cada *Premise* (i.e. os elementos *Reference*, *Operator* e *Value*) e o

conhecimento de cada *Instigation* conectado (o *Method* referenciado), tal como o número de *Premises* e *Instigations* conectados.

Estas entidades similares facilitam a compreensão e a interação dentro do sistema. Neste âmbito, a Figura 43 apresenta outro conjunto similar de *Rules*. Na verdade, este segundo conjunto de *Rules* similares permite a execução dos passos de produção faltantes. Além do mais, este conjunto de *Rules* também provê confirmação da adequabilidade das características relacionadas ao compartilhamento de colaboração entre os objetos, bem como o mecanismo de notificações.

Rule B If KUKA386.1 Status= Free Puma560.1 Status= Free Machine-Tool.1 Status= Free Table2P.1 Pos1 = Busy Then KUKA386.1 Transport(Table2P.1 Pos1, Machine-Tool.1) Machine-Tool.1 Prepare_to_Receive(PartV1)	Rule D If AGV-Robot.1 Status= Free Puma560.1 Status= Free Table2P.3 Pos1 = Not_Busy Table2P.1 Pos2 = Busy Then AGV-Robot.1 Transport(Table2P.1 Pos2, Table2P.3 Pos1)
Rule C1 If KUKA386.1 Status= Free Machine-Tool.1 Status= Part_Finished Table2P.2 Pos1 = Not_Busy Then KUKA386.1 Transport(Machine-Tool.1, Table2P.2 Pos1)	Rule E If ERIII.1 Status= Free Lathe.1 Status= Free Table2P.3 Pos1 = Busy Then ERIII.1 Transport(Table2P.3. Pos1, Lathe.1) Lathe.1 Prepare_to_Receive(PartV2)
Rule C2 If KUKA386.1 Status= Free Machine-Tool.1 Status= Part_Finished Table2P.2 Pos1 = Busy Table2P.2 Pos2 = Not_Busy Then KUKA386.1 Transport(Machine-Tool.1, Table2P.2 Pos2)	Rule F If ERIII.1 Status= Free Lathe.1 Status= Part_Finished Table2P.3 Pos2 = Not_Busy Then ERIII.1 Transport(Lathe.1, Table2P.3 Pos2)

Figura 43: Regras para controlar os recursos

3.5.4 *Rules* de decisão

Nesta instância de controle é assumido que o recurso *Store3x3.1* é preenchido com peças para a produção dos produtos V1 e V2 sempre que o *Store3x3.1* se apresenta completamente vazio. Assim, a *Rule.A0* na Figura 44 foi criada para sinalizar a necessidade de novas peças no armazém. Normalmente, outras *Rules* que não se apresentam no escopo deste exemplo (em um sistema mais amplo) podem considerar os estados dos *Attributes* que representam as posições para decidir quando recarregar o armazém *Store3x3.1*.

Similarmente, foi assumido que as peças sobre a posição 2 (*Pos2*) da mesa *Table2P.3* e sobre as posições 1 ou 2 (*Pos1* ou *Pos2*) da mesa *Table2P.2* devem ser removidas da célula

de manufatura. O dado momento para que estas peças sejam removidas é determinado pelas *Rule1.1*, *Rule1.2* e *Rule1.3*, também apresentada na Figura 44.

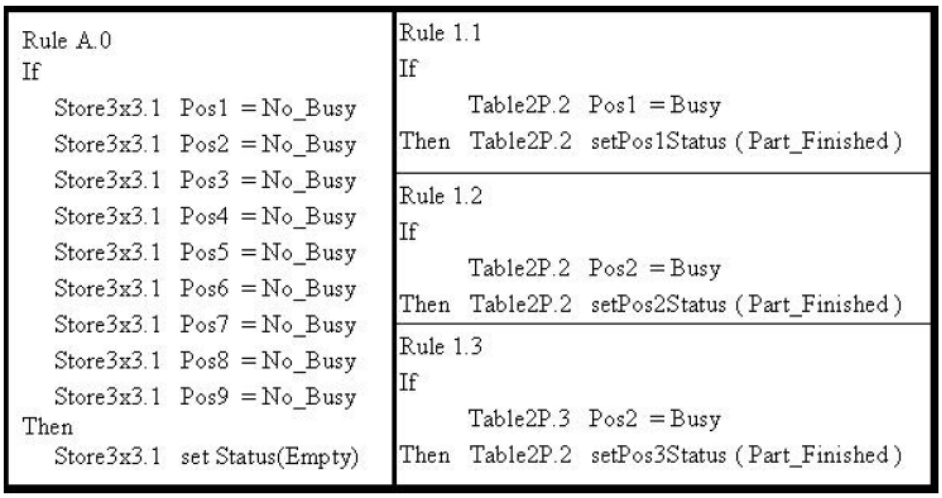


Figura 44: Regras para sinalizar a inserção e remoção de peças

As *Rules* apresentadas nesta seção avaliam e mudam somente *Attributes* dentro de um equipamento particular. Portanto, cada *Rule* pode ser agregada em um respectivo recurso ou *FBE* (mais genericamente). Da mesma forma, todas as *Rules* apresentadas nesta seção podem ser agregadas em um recurso (ou *FBE*) relacionado com a célula de manufatura em questão.

3.6 RESOLUÇÃO DE CONFLITOS

De fato, conforme foi expresso na Figura 42, as *Rules* inter-bloqueadas concorrem pelo acesso a um equipamento conflitante (i.e. *Puma560.1*). No entanto, estas *Rules* se apresentam complexas pelo fato de que a responsabilidade de identificar e solucionar os conflitos recai sobre elas próprias segundo as diretrizes definidas pelo programador. Deste modo, a definição de uma *Rule* inter-bloqueada se torna tediosa, tendente a erro e mesmo cara em termos de processamento devido ao aumento da quantidade de *Premises* a serem avaliadas.

Neste âmbito, esta seção apresenta soluções de resolução de conflitos que automatizam este processo de identificação e tratamento de conflitos entre as *Rules*, evitando todas as inconveniências relacionadas à definição de *Rules* inter-bloqueadas.

Basicamente, a resolução de conflito consiste na organização das execuções de regras aprovadas segundo alguma estratégia pré-estabelecida (FRIEDMAN-HILL, 2003). Estas estratégias podem variar para alcançar o fluxo de execução pretendido pelo programador tanto em ambientes monoprocessados quanto em ambientes multiprocessados.

Em um ambiente monoprocessado, a resolução de conflitos ocorre para estabelecer a ordem de execução das regras, onde apenas uma regra pode executar por vez. Em um ambiente multiprocessado, a resolução de conflitos ocorre para evitar o acesso concorrente a um recurso referenciado por várias regras a fim de manter a consistência do programa.

Nos ambientes monoprocessados, esta dissertação propõe dois modelos diferentes de resolução de conflitos ao PON. No primeiro deles, a resolução de conflitos ocorre por meio de um controlador centralizado, de forma similar como ocorre nos SBR. Neste, o controlador adota alguma estratégia de resolução de conflitos para escalonar as *Rules* em termos da ordem de aprovação ou então em termos de suas prioridades de execução. No segundo modelo, a resolução de conflitos ocorre por meio de um controle descentralizado. Neste modelo, a resolução de conflitos ocorre pela própria cooperação entre os objetos participantes do mecanismo de notificações. A subseção 3.6.1 descreve o modelo centralizado e a subseção 3.6.2 descreve o modelo descentralizado.

Em ambientes multiprocessados (paralelos ou distribuídos), mesmo as *Rules* apresentando características de independência uma das outras, os conflitos ainda podem ocorrer. Assim sendo, o PON apresenta a opção de dois modelos inteligentes de resolução de conflitos para manter a consistência dos dados nestes ambientes indeterminísticos. Estes modelos também são descentralizados, os quais são executados autonomamente por meio da colaboração dos objetos participantes do mecanismo de notificações. Estes modelos serão abordados no Apêndice A, uma vez que a experimentação apresentada na dissertação diz respeito a ambientes monoprocessados.

3.6.1 Modelo centralizado de resolução de conflitos

O modelo centralizado emprega uma entidade concentradora provida de uma estrutura de dados linear (e.g. pilha, fila ou lista), chamada de conjunto de conflitos. Esta estrutura guarda referências para as *Rules* aprovadas, conforme ilustra a Figura 45. Esta estrutura recebe as *Rules* na ordem em que elas são aprovadas, podendo reorganizá-las de acordo com os preceitos da estratégia utilizada para que estas sejam executadas.

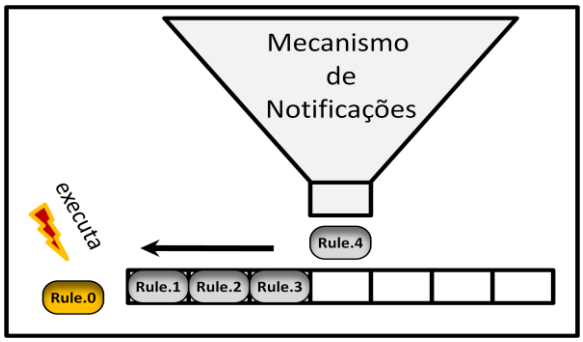


Figura 45: Modelo centralizado de resolução de conflitos

Em certas estratégias, a inserção seqüencial das *Rules* aprovadas ao conjunto de conflito e a ordem em que esta estrutura é percorrida, e.g. FIFO (*First-In First-Out*) ou FILO (*First-In Last-Out*) são suficientes para organizar a execução das *Rules*. Em outras, após a inserção das *Rules*, o conjunto de conflito deve ser reorganizado para corresponder a uma política pré-estabelecida mais complexa.

Atualmente, o PON permite adotar algumas estratégias disponíveis no contexto dos SBR para este modelo, uma vez que sua resolução de conflitos ocorre de maneira similar. Entre elas, estão as estratégias: (a) BREADTH que se baseia no escalonamento FIFO, (b) a DEPTH que se baseia no escalonamento FILO e (c) a PRIORITY que organiza as *Rules* de acordo com as prioridades definidas nas mesmas. Estas estratégias permitem maior flexibilidade na definição do fluxo de execução dos programas, podendo definir uma que esteja mais de acordo com as características do problema abordado.

Estas estratégias, pelo fato de serem baseadas em um mecanismo monolítico de resolução de conflitos são apenas apropriadas ao uso em ambientes monoprocessados.

3.6.2 Modelo descentralizado de resolução de conflitos (monoprocessado)

No modelo descentralizado não há o conceito de conjunto de conflito. Desta forma não há a ordenação das execuções das *Rules* como constatado no modelo centralizado. Entretanto, ao evitar o uso do conjunto de conflito, este modelo se limita em suas estratégias suportadas[20].

Uma estratégia adequada ao monoprocessado se refere ao imediatismo da execução das *Rules*. Neste, uma *Rule* deve ser executada no mesmo instante em que é aprovada, conforme ilustra a Figura 46. Esta estratégia imita o mecanismo de execução das expressões

[20] Mesmo sendo as estratégias do modo centralizado passíveis de implementação no modelo descentralizado, estas demandariam maior grau de colaboração entre os participantes da solução, o que poderia inviabilizar esta prática em termos de desempenho.

causais (*if-then*) do imperativo. No imperativo, após a satisfação da expressão condicional, o seu conteúdo é executado imediatamente.

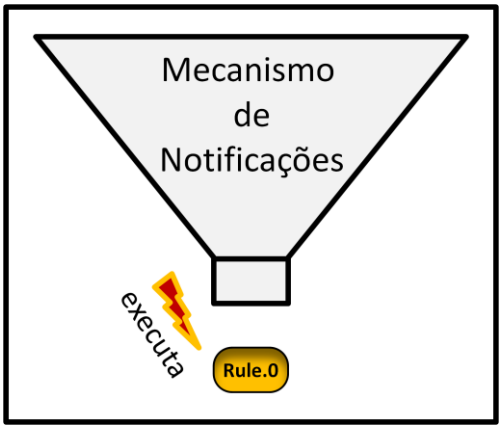


Figura 46: Modelo descentralizado de resolução de conflitos

Esta estratégia apresenta grande eficiência, principalmente porque as aprovações desnecessárias, comuns no modelo centralizado, são evitadas. As aprovações desnecessárias ocorrem quando um conjunto de *Rules* é aprovado ao conjunto de conflito e logo após a execução de uma destas *Rules*, o restante ou parte destas é desaprovada, devendo ser removidas do conjunto de conflito. Com a execução imediata este problema não ocorre, porque toda vez que uma *Rule* é aprovada ela é imediatamente executada.

Devido a este fator e motivado pela forte relação do PON também com as linguagens imperativas, esta estratégia e consecutivamente o modelo descentralizado foram adotados como padrão ("*default*") no emprego dos conceitos do PON.

3.7 REFLEXÕES SOBRE O PARADIGMA ORIENTADO A NOTIFICAÇÕES

Conforme descrito nas seções precedentes, o PON adota e evolui as entidades elementos da base de fatos e regras permitindo que estas sejam decompostas em entidades menores e que estas colaborem por meio de notificações precisas e pontuais. Estas qualidades do PON formam as soluções para as principais deficiências dos atuais paradigmas de programação. As subseções seguintes descrevem sobre as soluções do PON para estas deficiências.

A subseção 3.6.1 descreve sobre a solução para a ineficiência causada pelas buscas nos atuais paradigmas, a subseção 3.6.2 descreve sobre a solução para se obter maior desacoplamento entre os objetos do POO e consecutivamente aumentar o nível de reusabilidade dos objetos. A subseção 3.6.3 descreve sobre as facilidades de programação

providas por PON e como ela pode ser ainda mais fácil com o uso de uma ferramenta *Wizard* e, por fim, a subseção 3.6.4 descreve sobre as facilidades na programação multiprocessada, as quais se devem às qualidades de colaboração e de independência dos objetos que formam o mecanismo de notificações.

3.7.1 Reflexão sobre a eficiência

Uma das principais deficiências dos atuais paradigmas está relacionada aos seus mecanismos de execução, que se apresentam similares ao serem baseados em buscas ou no percorrer sobre elementos passivos. Estas buscas, quando excessivas, afetam o desempenho das aplicações podendo inviabilizar a implementação de certos sistemas sobre um dado *hardware* disponível, clamando por atualizações. No Paradigma Imperativo, em especial, as buscas ocorrem sobre os dados e expressões causais causando os problemas das redundâncias temporais e estruturais.

A redundância temporal é solucionada em PON ao evitar buscas sobre os elementos passivos, uma vez que certos dados (i.e. os *Attributes*) apresentam sensibilidade na detecção de mudanças de estados e apresentam capacidades para notificar pontualmente somente as partes de uma expressão causal afetada por esta mudança (i.e. as *Premises* pertinentes), impedindo que as demais partes e mesmo as demais expressões causais sejam avaliadas ou reavaliadas desnecessariamente. Ainda, o evitar destas avaliações é suplementado pelo fato de que as expressões causais (i.e. *Conditions*) e mesmo as partes (i.e. *Premises*) guardam lembranças de seus estados, contribuindo para eliminar totalmente as redundâncias temporais.

A redundância estrutural é solucionada em PON quando uma *Premise* é compartilhada com duas ou mais expressões causais (i.e. *Conditions*). Esta solução está esquematizada na Figura 47 para as *Rules* inter-bloqueadas apresentadas na Figura 42. Conforme o esquema, quando o robô *Puma560.1* se torna livre, o *Attribute atStatus* notifica pontualmente a única *Premise* conectada. Por sua vez, esta avalia o seu estado lógico uma única vez para este estado e compartilha o resultado lógico com as nove *Rules* relacionadas, eliminando as redundâncias estruturais.

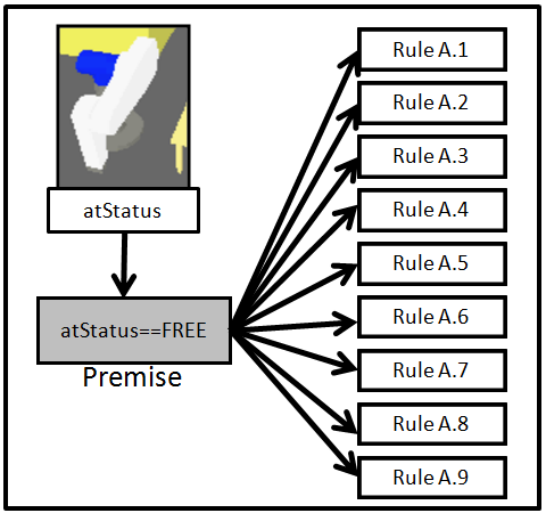


Figura 47: Solução do PON para as redundâncias temporais e estruturais

No PD, as buscas também consistem em uma das principais deficiências. Entre os PDs, o PL é o mais afetado devido às suas qualidades dedutivas. Mesmo no contexto dos SBR atuais que adotam algoritmos de inferência sofisticados como o RETE e seus derivados, as buscas ainda continuam e podem afetar o desempenho das aplicações. Entretanto, uma exceção são os SBR que inferem sobre o algoritmo HAL, uma vez que este reduz significativamente as buscas.

Diferentemente destas soluções, as buscas são totalmente eliminadas em PON, uma vez que as entidades *FBE* apresentam qualidades superiores aos simples elementos da base de fatos dos SBR. Em PON, não há a necessidade de buscas para encontrar as *Rules* relacionadas a um estado modificado, uma vez que as entidades *FBE* (que agrupam estes estados) são conectadas às *Rules* por meio dos objetos colaboradores. Estes permitem que as *Rules* interessadas na mudança de um estado sejam notificadas pontualmente, sem buscas pelas mesmas.

Em suma, o PON elimina as buscas tão típicas no PI e no PD por meio da colaboração de seus objetos que se comunicam por notificações precisas e pontuais. O fato de eliminar as buscas por dados e mesmo expressões causais melhora o desempenho das aplicações como será apresentado no Capítulo 0, no qual instâncias implementadas com o PON serão comparadas com instâncias implementadas sobre os atuais paradigmas.

3.7.2 Reflexão sobre o acoplamento

O acoplamento é outra deficiência sanada pelo PON. Esta deficiência é natural ao Paradigma Imperativo (PI) enquanto que no Paradigma Declarativo (PD) esta deficiência é

menos evidente devido às características modulares das soluções (i.e. base de fatos, base de regras e máquina de inferência). Entretanto, as entidades elementares do PD (e.g. elementos da base de fatos, regras, funções) apresentam características de desacoplamento tão somente quando consideradas individualmente, pois os seus relacionamentos por meio do mecanismo interno de execução os tornam entidades interdependentes conforme discutido previamente.

Neste âmbito, o PON oferece uma solução que permite programar aplicações menos acopladas. Ao programar no estilo orientado a notificações, o programador automaticamente cria programas "desacoplados" (i.e. minimamente acoplados) e com alta capacidade de reuso dos objetos, principalmente dos objetos *FBE*.

Os *FBE* apresentam maiores capacidades de reuso do que os objetos típicos do POO, uma vez que os relacionamentos de um elemento da base de fatos com outros ou objetos comuns do POO, são naturalmente reduzidos. Estes relacionamentos são reduzidos porque diferentemente do que ocorre no POO, não são os próprios objetos (*FBE*) que guiam as suas próprias colaborações.

Em PON, os relacionamentos entre os *FBE* são guiados, à medida do possível, pelas chamadas de métodos inseridas na estrutura de uma *Rule*. Com isto, os objetos não apresentam a mesma necessidade de guardar referência para cada outro que se necessita de um serviço, uma vez que a *Rule* é responsável pelas invocações destes serviços.

Como resultado, o esquema à direita na Figura 48 ilustra, hipoteticamente, a redução dos relacionamentos entre os objetos em relação à figura da esquerda, a qual reapresenta as relações de acoplamento entre os objetos do POO. Esta redução ocorre pelo uso de *Rules*. À direita na mesma figura, uma *Rule* é representada pelo círculo central com borda mais intensa.

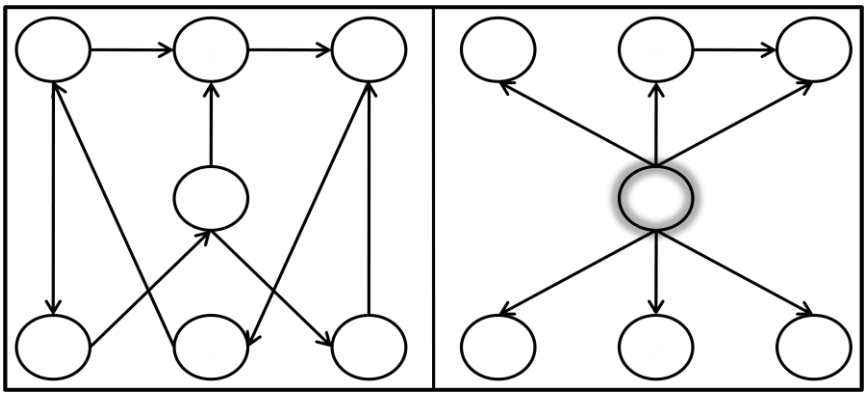


Figura 48: Acoplamento entre objetos no Paradigma Imperativo e entre *FBEs* e *Rules* no Paradigma Orientado a Notificações

Ademais, mesmo se o gerenciamento das relações entre os *FBEs* ocorre nas *Rules*, isto não é um indício de que as *Rules* estão fortemente acopladas com os *FBEs*, principalmente porque uma *Rule* apenas é conectada (indiretamente) a um *FBE* no início do tempo de

execução. Da mesma forma, a implementação de um *FBE* não guarda referência explícita aos objetos colaboradores de uma *Rule* (i.e. *Premises*), as mesmas também são estabelecidas no início do tempo de execução a partir dos parâmetros passados na concepção das *Rules*.

Com isto, a título de exemplo, a implementação de uma classe em PON não apresenta o mesmo problema de acoplamento expresso pelo trecho de código apresentado à esquerda na Figura 49. Em PON, como ilustra o esquema à direita na Figura 49, o *FBE Puma560* não faz referência direta às classes *Store3x3* e *Table2P*, estas referências são realizadas por meio da respectiva regra (i.e. *rule*) representada na mesma figura. Assim, a classe *Puma560* pode ser facilmente reusada em outra aplicação, precisando para isto apenas compor novas *Rules*, as quais podem relacionar os objetos desta classe com outros objetos, não precisando ser necessariamente instâncias de *Store3x3* e *Table2P*.

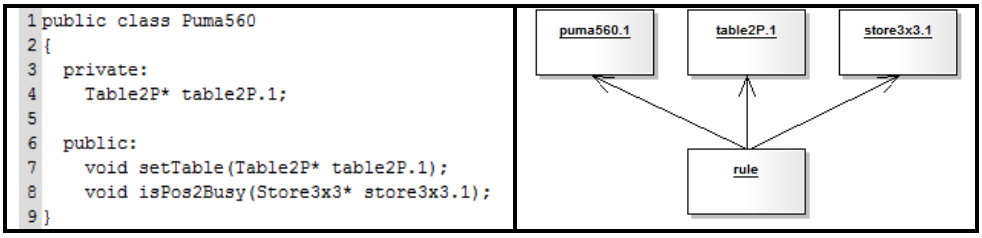


Figura 49: Exemplificando a redução de acoplamentos

Em suma, o emprego das *Rules* do PON para guiar os relacionamentos entre os objetos surge como uma solução natural para reduzir a quantidade de acoplamentos em uma aplicação.

3.7.3 Reflexão sobre as facilidades de programação

Ao se inspirar em conceitos do declarativo, uma qualidade intrínseca aos conceitos do PON é a oferta de facilidades na programação. Atualmente, os conceitos do PON são materializados em uma linguagem imperativa, mais precisamente na linguagem C++, agregando, entretanto, maiores facilidades de programação, as quais são similares àquelas ofertadas na programação declarativa, como será visto no Capítulo 0.

A programação em PON se torna mais fácil, principalmente porque ela é intuitiva à forma cognitiva humana. Além de permitir a representação do conhecimento na forma natural ao ser humano, o PON também visa alcançar esforços mínimos na programação em termos de escrita de código. Como exemplo deste compromisso, a criação dos objetos colaboradores e a conexão dos mesmos são realizadas de forma transparente ao programador, bastando que este apenas se concentre na concepção da *Rule* em si conforme exemplifica o código abaixo que

traz a criação de objetos PON por meio de um *framework* elaborado em C++ (detalhado no Capítulo 4).

```
1 Rule* rlRule = new Rule(Condition::CONJUNCTION);
2 rlRule->addPremise(puma560.1->atStatus, Boolean::TRUE, Premise::EQUAL);
3 rlRule->addPremise(table2P.1->atPos1,   Boolean::TRUE, Premise::EQUAL);
4 rlRule->addPremise(store3x3.1->atPos1,  Boolean::TRUE, Premise::EQUAL);
5 rlRule->addInstigation(mtTransport);
6 rlRule->end();
```

Figura 50: Exemplo da criação de uma *Rule* em C++

Neste âmbito, a fim de tornar a programação efetivamente fácil em PON, está sendo desenvolvida uma ferramenta *Wizard* que permite ao programador compor os *FBEs* e *Rules* em alto nível. Atualmente, uma versão preliminar desta ferramenta já foi desenvolvida. Esta ferramenta permite compor *Rules* por meio de uma interface gráfica, permitindo total transparência em relação ao código fonte. Esta ferramenta foi implementada e testada sobre o domínio de manufatura, sendo aplicada para compor as *Rules* sobre os equipamentos virtuais simulados no ANALYTICE II (SIMÃO, 2005; LUCCA, SIMÃO, BANASZEWSKI *et al*, 2008).

A Figura 51 representa a interface gráfica da ferramenta. Esta permite a composição de *Rules* em forma mais declarativa. Assim, por meio da manipulação dos campos de edição da ferramenta, o programador pode definir os parâmetros necessários para a formação de uma *Rule*. Maiores detalhes sobre esta implementação estão disponibilizados em (LUCCA, SIMÃO, BANASZEWSKI *et al*, 2008).

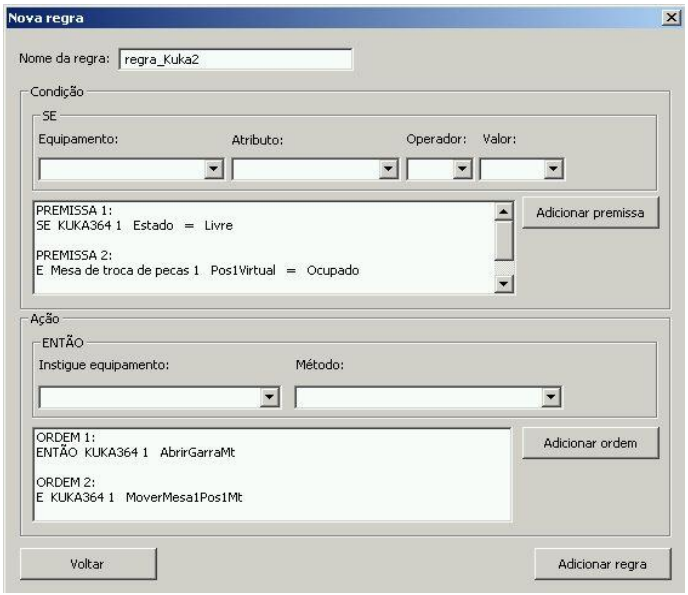


Figura 51: Ferramenta amigável para a composição de *Rules*

Essencialmente, a programação em PON consiste em conceber os relacionamentos entre os objetos *FBEs* por meio das *Rules*, o programador somente precisa compor as *Rules*

sem necessariamente se preocupar em como elas vão interagir para que o fluxo de execução seja estabelecido, principalmente se uma ferramenta *Wizard* for utilizada.

Com a organização do conhecimento do programa em termos de *Rules*, o entendimento da semântica de um programa em PON se torna mais fácil do que no PI. Geralmente, as *Rules* são definidas em um mesmo arquivo, apresentando uma visão geral do funcionamento do programa, diferentemente do PI onde o conhecimento está disperso pelos emaranhados de chamadas de métodos.

Também, uma *Rule* é independente da seqüência em que é definida, podendo ser definida em qualquer ordem ou a qualquer momento em que venha ser necessária, sem afetar (fortemente) a execução das demais expressões. Assim, o programador pode se concentrar na concepção de uma única *Rule* por vez, de forma assaz independente das demais, uma vez que os possíveis conflitos serão resolvidos pelos modelos de resolução de conflito. Esta facilidade demonstra a propriedade de escalabilidade do PON, bem superior às linguagens imperativas.

Ainda, uma *Rule* pode ser criada até mesmo em tempo de execução, uma vez que uma regra é instanciada em forma de objetos (a partir de um dado *Wizard*). Ainda neste âmbito, aproveitando a sua representação, uma *Rule* pode ter sua expressividade incrementada em tempo de execução com, por exemplo, a inclusão de novas *Premises* ou novos *Instigations*.

Em suma, os conceitos do PON surgem para tornar a programação mais simples. Além de permitir a estruturação do conhecimento em forma natural ao ser humano, proporciona a composição deste conhecimento de forma simplificada e transparente das particularidades do código imperativo. No mais, quando estas particularidades se fazem necessárias, o PON possibilita o seu uso, porém o faz de forma a evitar a ocorrência de certos problemas natos a este tipo de programação, como o acoplamento, dispersão do conhecimento e execuções desnecessárias.

3.7.4 Reflexão sobre o paralelismo e distribuição

Além de facilitar a composição do conhecimento, o PON também pode permitir que o programador obtenha os reais benefícios da computação paralela e distribuída. Estes benefícios se devem a organização adequada da estrutura e comportamento dos componentes do PON, que são alcançáveis de forma transparente pelo programador. Ainda, as características declarativas do PON podem ser adotadas para poupar o programador das particularidades que envolvem a computação multiprocessada.

Os componentes do PON (i.e. *FBE*, *Rules* e suas decomposições) são estruturados e organizados para favorecer a execução paralela. Estes apresentam qualidades de independência e desacoplamento, podendo ser alocados independentemente a diferentes nós de processamento. Também, estes componentes apresentam comportamento que incentivam a execução nestes ambientes principalmente porque estes se comunicam ativamente e pontualmente por meio de notificações (SIMÃO, 2005).

Em PON, os componentes cooperam independentemente de suas localidades. Neste paradigma, um componente pode notificar o seu estado para outro componente indiferentemente se este está localizado na mesma região da memória, no mesmo computador ou na mesma sub-rede. Por exemplo, um componente notificante (e.g. um *Attribute*) pode estar executando em um computador enquanto que o componente a ser notificado (e.g. *Premise*) pode estar executando no mesmo computador ou em outro.

Para que a notificação ocorra sobre os componentes, o componente notificante somente precisa conhecer o endereço na memória ou em outro computador onde o componente a ser notificado está alocado. Desta forma, subentende-se que os componentes que compõem uma *Rule* ou um *FBE* podem estar distribuídos entre nós de processamento permitindo que estes sejam executados paralelamente, cooperando por meio de notificações.

Assim sendo, o PON permite que vários fluxos de notificações ocorram de forma paralela. Por exemplo, em um mesmo objeto *FBE*, dois objetos *Attributes* podem ser alterados paralelamente por diferentes *threads* ou processos de execução, iniciando assim dois diferentes fluxos de notificação em direção às *Rules* pertinentes. Estes fluxos são independentes, mesmo sendo iniciados sobre os estados de um único objeto. Ainda, quando os *Attributes* se referem a diferentes objetos *FBEs*, a independência pode ser ainda maior, uma vez que há maiores chances destes notificarem *Rules* distintas.

Da mesma forma, dois objetos *Methods* de um mesmo objeto *FBE* podem executar de forma paralela, uma vez que as *Rules* que invocam estes métodos passam os parâmetros (i.e. os *Attributes* referenciados pelas *Premises* da *Rule*) com a exclusividade de acesso já assegurada por meio do mecanismo de resolução de conflitos. Em PON, a própria resolução de conflito assegura que os métodos em execução não apresentam interferência em relação ao acesso aos *Attributes* pertinentes. Com isto, o programador se abstém da implementação de políticas de sincronização entre as partes do código[21].

[21] Diferentemente do PON, estas mesmas facilidades não ocorrem na programação (multiprocessada) com o POO. Geralmente, o acesso concorrente aos atributos e métodos de um objeto é dificultado,

Em PON, devido à fundamentação nos conceitos do PD, o programador é poupado das tarefas entediantes relacionadas à computação multiprocessada, uma vez que toda a sincronização entre as partes do código ocorre de forma transparente ao programador. Deste modo, o programador constrói sistemas multiprocessados com esforço similar àquele empregado na construção de aplicações monoprocessadas, ou seja, concentrando-se mais na composição do conhecimento da aplicação e menos nas questões de implementação. Com isto, subentende-se que as características declarativas podem reduzir as falhas causadas por programadores no desenvolvimento de aplicações multiprocessadas.

No entanto, além das falhas causadas pelo programador, as falhas também ocorrem devido a problemas relacionados à infra-estrutura que suporta a computação multiprocessada. Na verdade, estas falhas são mais comuns na computação distribuída do que na computação paralela por causa de problemas de comunicação entre os nós remotos de uma rede[22].

Em PON, ao evitar as redundâncias temporais e estruturais, as comunicações remotas entre uma expressão causal (*Rule*) e os estados (*Attributes* dos *FBEs*) a serem avaliados são reduzidas. Com a redução das comunicações entre os nós, reduz-se consecutivamente, as falhas decorrentes da comunicação. Em PON, devido à independência e ao desacoplamento entre as partes de uma *Rule*, estas partes podem ser distribuídas por meio de uma política de balanceamento de carga a fim de reduzir ainda mais as comunicações entre os nós de processamento. Este esquema pode ser visualizado na Figura 52.

---

principalmente porque os comandos (i.e. código de um método) e dados (i.e. variáveis) são fortemente relacionados e a execução destes comandos em relação aos dados ocorre de forma descoordenada, ou seja, sem o auxílio de um mecanismo inteligente como o de resolução de conflitos do PON. Por isso, o POO adota mecanismos de sincronização mais caros em termos de processamento (e.g. semáforo e monitores) para manter a normalidade de execução dos programas, os quais são definidos explicitamente pelo programador, podendo gerar muitas falhas de programação devido à responsabilidade suplementar atribuída ao programador.

O fato é que as falhas na programação multiprocessada são fáceis de cometer e difíceis de serem detectadas, pois demanda muita atenção por parte do programador em relação às múltiplas linhas de execução. Quando estas falhas não são corrigidas, alguns prejuízos expressivos podem ocorrer em tempo de execução. A título de exemplo, uma situação trágica ocasionada por uma falha na programação concorrente ocorreu entre os anos de 1985 e 1987 sobre o *software* de controle de uma máquina de radioterapia chamada Therac-25. Neste intervalo de tempo, devido às falhas na programação, pelo menos seis pessoas receberam altas doses da radiação, resultando em cinco mortes (LEVESON, 1995).

[22] Quando se usa os conceitos do POO para implementar aplicações distribuídas, as falhas podem ocorrem com maior freqüência, uma vez que este paradigma estabelece muitas comunicações desnecessárias entre os objetos. Estas comunicações são decorrentes das redundâncias temporais e estruturais.

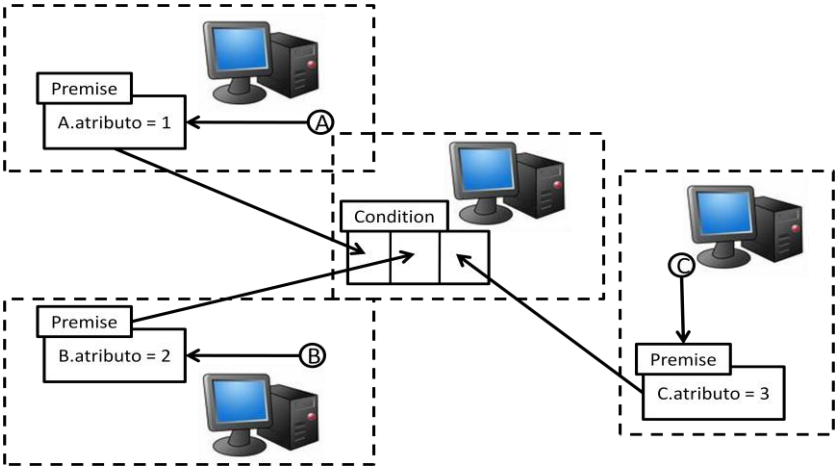


Figura 52: Comunicação distribuída com o PON

Como ilustra a figura, os objetos *Attributes* podem ser alocados juntamente com os seus respectivos objetos *Premises* a fim de minimizar as comunicações remotas. Assim, o estado do *Attribute* pode ser avaliado localmente. Neste modelo, somente quando a *Premise* muda de estado é que as comunicações remotas ocorrem. Desta forma, a solução para os problemas das redundâncias temporais e estruturais aliada com as qualidades comportamentais e de desacoplamento entre os componentes do PON provêem vantagens para o programador criar aplicações multiprocessadas.

De forma similar ao PON, algumas soluções do PD, como o algoritmo RETE de SBR, também solucionam os problemas das redundâncias. Entretanto, estas soluções também são baseadas em buscas, o que dificulta o emprego destas em ambiente multiprocessados.

Em síntese, o PON apresenta qualidades que normalmente favorecem o programador na construção de aplicações multiprocessadas. No entanto, a aplicação do PON em ambientes multiprocessados deve ser ainda melhor estudada a fim de prover, por exemplo, soluções efetivas para tratar de falhas de comunicabilidade em ambientes distribuídos e um algoritmo inteligente de balanceamento de carga.

Neste âmbito, em sua tese de doutorado (SIMÃO, 2005), Simão propôs algumas soluções incrementais ao meta-modelo, para tratamento de conflitos e garantia de determinismo, a fim deste atuar efetivamente em ambientes multiprocessados. Estas soluções devem ser consideradas a fim de formar uma solução mais efetiva à computação paralela e distribuída com o PON.

3.8 APLICABILIDADES

O PON é um paradigma que apresenta grandes expectativas. Este foi proposto para sanar as deficiências dos atuais paradigmas, apresentando-se como uma alternativa para melhorar a programação em ambientes locais e bem provavelmente também em ambientes paralelos ou distribuídos.

Devido aos conceitos herdados dos atuais paradigmas, bem como a linguagem de programação usada, o PON é uma solução aplicável à maioria dos problemas resolvíveis com estes paradigmas. No entanto, devido às particularidades do PON, este não se aplica apropriadamente a problemas de dedução lógica que exigem buscas combinatórias.

No entanto, esta incompatibilidade não é uma carência do PON, esta é uma decisão de projeto concebida principalmente para poupar processamento ao evitar estas buscas. Para isto, o PON adota a conexão de *FBEs* e *Rules* por meio de ponteiros de referência (indiretos), uma característica mais próxima ao POO[23].

Na verdade, o PON adota as regras para outro objetivo: o de exercer controle ou tomar decisões sobre os estados dos objetos. Com isto, o PON se aplica a todos os problemas resolvíveis com o Paradigma Imperativo, uma vez que a essência deste paradigma também é a manipulação dos estados de variáveis ou objetos.

Entretanto, há aplicações mais apropriadas ao estilo imperativo do que ao estilo proposto por PON. Por exemplo, o cálculo de expressões matemáticas ou mesmo problemas propriamente seqüenciais que não oferecem muitas relações causais são mais apropriados ao estilo imperativo.

Por outro lado, o PON é mais apropriado a problemas que evidenciam as dificuldades destes paradigmas. Deste modo, o PON é aplicado

- a problemas que apresentam grandes quantidades de expressões causais, o que gera processamento desnecessário por causa das redundâncias temporais e estruturais;
- a problemas que demandam maior organização da lógica da aplicação, visando facilidades de entendimento do código por uma equipe ou facilitar a manutenção;
- a problemas que necessitem maior qualidade de desacoplamento e reuso;

[23] Não obstante, o PON ainda provê um mecanismo adicional que facilita a composição de *Rules* que apresentam a mesma semântica e que somente se diferem nas instâncias dos seus objetos colaboradores. Este mecanismo permite criar *Rules* de forma genérica, ou seja, manipulando classes ao invés de objetos por meio de ponteiros de referência. Este mecanismo é chamado de *Formation Rules* e é mais bem detalhado no Apêndice C.

- quando se necessita maiores facilidades de programação e
- provavelmente a problemas a serem executados em ambientes multiprocessados.

Particularmente, os problemas supracitados são mais evidenciados em aplicações de controle, monitoramento, sistemas reativos (i.e. estilo produtor-consumidor) e qualquer outra aplicação que envolve muitas estruturas de decisão e normalmente demanda capacidades de multiprocessamento, como ocorre na implementação de games, na robótica e mais atualmente, na computação ubíqua.

3.9 CONCLUSÃO

Este capítulo apresentou a essência e as particularidades do paradigma proposto. Conforme explicitado neste capítulo, ao usar o PON, o programador estrutura o seu pensamento de maneira natural e expõe este pensamento sobre uma solução apropriada para o contexto da programação, o qual é composto por objetos com características de autonomia, independência, desacoplamento, reatividade e cooperação.

O uso destes objetos que cooperam por meio de notificações provê maiores benefícios ao programador. Com o uso destes, o programador pode construir aplicações monoprocessadas e (provavelmente) multiprocessadas mais eficientes e com menos esforços de programação. Estes benefícios são próprios ao PON, sendo que os mecanismos que provêm estes benefícios foram propostos visando, sobretudo, sanar as principais deficiências dos atuais paradigmas de programação.

Desta forma, para que o programador possa fazer uso destes objetos com respeito à nova perspectiva de programação estabelecida por PON, estes devem estar disponíveis ao programador por meio de uma linguagem de programação. Neste âmbito, um dos objetivos da presente dissertação é relatar sobre a materialização dos conceitos do PON em uma linguagem de programação, descrevendo mais sobre as facilidades de programação conquistadas.

CAPÍTULO 4

MATERIALIZAÇÃO E AVANÇOS DO PARADIGMA PROPOSTO

O objetivo deste capítulo é apresentar os assuntos pertinentes à materialização dos conceitos do Paradigma Orientado a Notificações (PON) em uma linguagem de programação. Particularmente, este capítulo aborda a materialização dos conceitos do PON na forma de uma nova versão do meta-modelo (monoprocessado) proposto inicialmente por Simão (2005).

Esta nova versão foi concebida pelo autor desta dissertação, a qual apresenta avanços na estrutura do meta-modelo e no comportamento de alguns de seus componentes. Estes avanços se fazem necessários para estabelecer o PON como um paradigma de programação efetivo e de fácil aplicação.

Também, este capítulo discute brevemente sobre algumas iniciativas de programação que materializam junções dos conceitos do Paradigma Imperativo (PI) e do Paradigma Declarativo (PD), mais especificamente do Paradigma Orientado a Objetos (POO) e dos Sistemas Baseados em Regras (SBR). Estas iniciativas são confrontadas com o paradigma proposto.

Atualmente, os conceitos do PON estão materializados na forma de um *framework* sobre uma linguagem de programação imperativa, inclusive a fim de reduzir a resistência a mudanças por parte da comunidade de programadores. Particularmente, o PON está materializado sobre a linguagem C++, uma linguagem clássica do POO. Esta materialização se deu pela junção das capacidades e vantagens da programação imperativa na linguagem C++ com as facilidades da programação declarativa incorporadas na forma de evoluções de "elementos da base de fatos" e "regras".

Atualmente, os conceitos do PON se apresentam materializados apenas sobre ambientes monoprocessados, uma vez que a versão multiprocessada ainda não se encontra finalizada. No entanto, a principal diferença entre as versões monoprocessadas e multiprocessadas está justamente na resolução de conflitos, sendo que os componentes participantes do mecanismo de notificações não se alteram de uma versão para outra. Desta forma, este capítulo descreve apenas a materialização da versão monoprocessada do PON.

Para descrever esta materialização do PON, a seção 4.1 apresenta uma aplicação hipotética a ser usada como exemplo no decorrer do capítulo, a qual se refere ao controle de um robô-motorista no trânsito, a seção 4.2 apresenta uma breve descrição sobre a adoção da linguagem imperativa C++ para a materialização, a seção 4.3 apresenta uma visão geral do

framework (incluindo a sua estrutura e seu uso na implementação da aplicação exemplo) e por sua vez, a seção 4.4 apresenta as particularidades do *framework* enfatizando alguns melhoramentos no comportamento de cada objeto participante do fluxo ativo de notificações em direção a *Rule*.

Em um foco distinto, a seção 4.5 discute brevemente sobre algumas materializações relacionadas a junções dos atuais paradigmas (POO e SBR), especialmente sobre tais materializações que ocorrem sobre a linguagem C++ confrontando-as com o *framework* do paradigma proposto. Por fim, a seção 4.5 conclui sobre o capítulo.

4.1 SISTEMA DE CONTROLE DE UM ROBÔ-MOTORISTA NO TRÂNSITO

Esta seção apresenta um exemplo sobre um cenário hipotético farto de situações causais, baseado nas percepções e ações na tomada de decisão de um robô-motorista no trânsito. Neste cenário, as situações causais são passíveis de serem representadas por expressões causais na forma de regras. Por isto, o mesmo cenário é adotado como exemplo para mais bem elucidar a materialização do PON.

Neste contexto, para que o robô-motorista possa atuar de forma efetiva em relação ao trânsito, ele abstrai o ambiente em elementos pertinentes a sua tomada de decisão. Normalmente, os elementos pertinentes à decisão se referem ao semáforo, à faixa de pedestre e ao seu próprio carro, como representado na Figura 53(a).

Estes elementos consistem nos elementos da base de fatos identificados pelo motorista, os quais são representados como *Semaphore, Crosswalk e Car* na forma de classes derivadas de *FBE* na Figura 53(b). Desta forma, o motorista realiza as tomadas de decisão no trânsito baseado na relação entre estados das instâncias destas classes. Esta tomada de decisão é naturalmente representada em forma de regra ou *Rule*, como exemplifica a Figura 53(c).

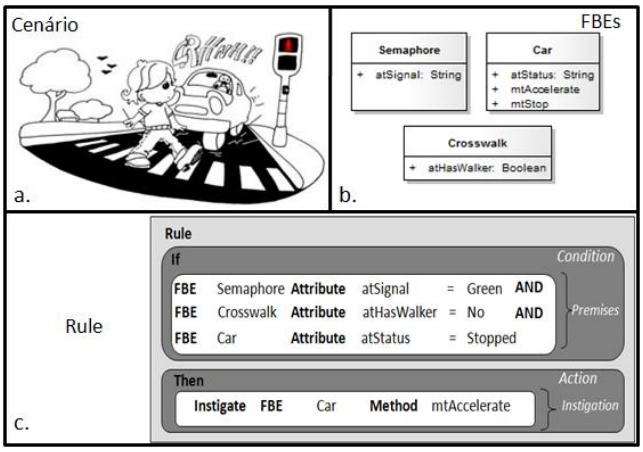


Figura 53: Cenário do trânsito

Na *Rule* exemplificada, a *Condition* é composta por uma conjunção entre três objetos *Premises*. Estas *Premises* realizam avaliações lógicas em relação aos estados dos *FBEs Semaphore*, *Crosswalk* e *Car* notificadas por eles mesmos. Assim, de acordo com as notificações conseqüentes dessas *Premises*, a *Condition* realiza o seu teste lógico para verificar a aprovação da *Rule* conectada para execução.

Com a satisfação do teste lógico, a respectiva *Rule* é aprovada para a execução do conjunto de instruções de sua *Action*. Na execução, a respectiva *Action* ativa o único *Instigation* para o qual guarda conexão. Ao ser ativado, o respectivo *Instigation* invoca o *Method mtAccelerate* do *FBE Car*, o qual é responsável por aplicar movimento ao automóvel[24].

4.2 MATERIALIZAÇÃO EM LINGUAGEM IMPERATIVA

A proposta dos conceitos do PON sobre uma linguagem POO se deve, principalmente, à popularidade destas linguagens perante a comunidade de programadores. Por isto, a linguagem C++ foi adotada para materializar os conceitos do paradigma proposto. Com esta adoção, intenciona-se minimizar a resistência à aceitação dos conceitos do paradigma pela comunidade de programadores, principalmente em relação ao que ocorreria se este fosse proposto sobre uma linguagem particular e incompatível com as atuais.

[24] Neste cenário, considera-se que um robô-motorista apresenta um conjunto particular de *Rules* e *FBEs*. Assim, quando um robô se aproxima de um cruzamento, este recebe uma mensagem com dados dos elementos reais do sistema (i.e. semáforo, faixa de pedestres) para que os estados de seus respectivos *FBEs* sejam atualizados.

Especialmente, dentre as atuais linguagens POO, escolheu-se C++ devido ao seu bom desempenho frente às demais linguagens, como o Java e C# (COWELL-SHAH, 2004). Este foi o fator determinante para a sua escolha, uma vez que uma das propostas iniciais do PON é apresentar maior eficiência às aplicações.

Outrossim, ao ser materializado em uma linguagem consolidada, o PON obtém a vantagem de usufruir de todas as capacidades e funcionalidades oferecidas nesta linguagem. Com isto, a implementação dos elementos da base de fatos e regras ocorre com maior flexibilidade e potencial do que nas linguagens ditas declarativas, como nas linguagens integradas aos *shells* dos SBR (e.g. CLIPS, OPS5 e *RuleWorks*). Normalmente, nestas linguagens, um elemento da base de fatos é representado na forma de uma simples tupla (e.g. [<object><*Attribute*><*Value*>]) e as entidades do tipo regra se restringem a executar ações sobre a Base de Fatos (i.e. inserção, alteração e remoção de elementos da base de fatos) (BROWNSTON, FARRELL, KANT *et al*, 1985).

As linguagens declarativas puras são inflexíveis e geralmente não suportam acesso direto a algumas funcionalidades comumente usadas pelos programadores das linguagens do PI (e.g. acesso ao *hardware*). Ainda, as linguagens declarativas não foram contempladas com os avanços desenvolvidos nas últimas décadas no tocante a Engenharia de *Software*. Estes avanços são relativos principalmente às linguagens imperativas orientadas a objetos, tais quais estão presentes:

- Na proposta de metodologias e ferramentas de projeto, como o processo unificado e as ferramentas CASE que a suportam;
- Nas linguagens de modelagem como a UML e mesmo a SysML (*System Modeling Language*);
- Nos padrões de *software* como os pertencentes aos catálogos comumente conhecidos como GoF (*Gang of Four*) (GAMMA, HELM, JOHNSON e VLISSIDES, 1995) e POSA (*Pattern-Oriented Software Architecture*) (SCHMIDT, STAL, ROHNERT e BUSCHMANN, 1996).

Todavia, as linguagens declarativas são caracterizadas pela facilidade de programação ao poupar o programador das particularidades das linguagens imperativas. Neste mesmo sentido, além de incorporar a flexibilidade e as capacidades do imperativo, a materialização do PON também facilita a programação ao permitir a composição dos conceitos essenciais do declarativo, regras e elementos da base de fatos, por meio de interfaces de objetos bem definidas ou então com o auxílio de uma ferramenta amigável, que torna ainda mais

transparente a instanciação dos objetos pertinentes (LUCCA, SIMÃO, BANASZEWSKI *et al*, 2008).

4.3 VISÃO GERAL DO *FRAMEWORK* EM C++

O PON, bem como sua materialização, oferece várias vantagens. Uma delas é a facilidade de programação uma vez que o conhecimento é composto em termos de regras. Para melhor elucidar estas facilidades de programação, esta seção apresenta a estrutura do *framework* sobre o exemplo do cenário do robô-motorista.

Basicamente, a estrutura do *framework* é constituída por dois principais pacotes de classes, que são o pacote *core* e o pacote *Application*:

- O pacote *core* contém as classes que modelam os objetos participantes do mecanismo de notificações, as quais já foram apresentadas no diagrama de classes da Figura 38.
- O pacote *Application* contém entre outras classes, a classe que representa o ponto de extensão do *framework*, ou seja, a classe que permite o uso do *framework*. As demais classes deste pacote correspondem àquelas que promovem a resolução de conflitos centralizada (vide seção 3.6.1).

Estes pacotes (em UML) estão representados por uma relação de importação na Figura 54. Tal representação expressa que a aplicação que deriva a classe que provê o ponto de extensão do *framework*, automaticamente está importando as classes do pacote *core*.

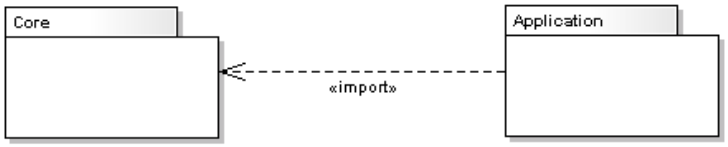


Figura 54: Estrutura do *framework*

A classe que permite a aplicação usar os serviços do *framework* (i.e. que provê o ponto de extensão) é chamada de *Application*, o mesmo nome do pacote que a incorpora. A representação da classe *Application* e as demais classes do pacote que a integra estão

ilustradas na Figura 55. Para tal representação, os atributos e métodos das classes não pertinentes a compreensão do modelo foram suprimidos para facilitar a explanação[25].

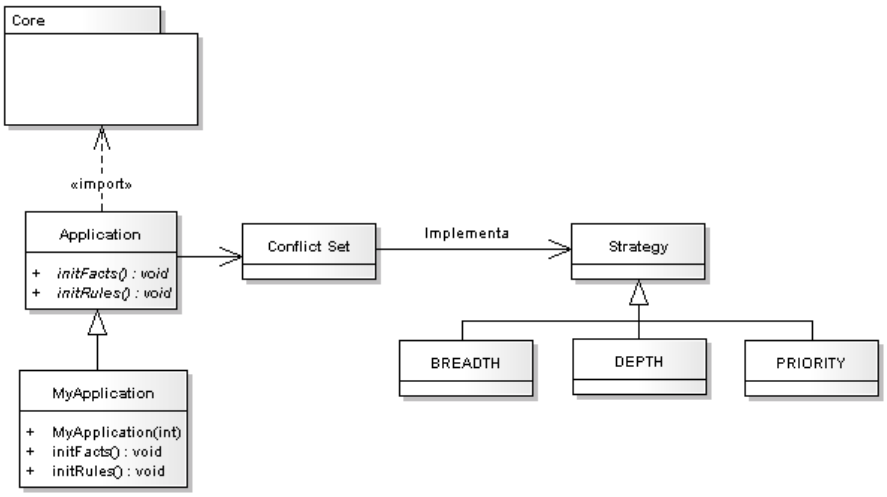


Figura 55: Instanciação do *framework*

Nesta representação, a classe *MyApplication* estende o *framework* por meio da classe *Application*. A classe *MyApplication* é definida pelo programador, a qual é usada para comportar a declaração dos *FBEs* e *Rules*.

Em sua estrutura, encontra-se a definição do método construtor com um parâmetro, sobre o qual o programador determina a estratégia de resolução de conflito desejada, caso pretenda adotar o modelo centralizado. Além do método construtor, a sua estrutura apresenta dois métodos herdados da classe *Application*. Estes se referem aos métodos *initFacts* e *initRules*, os quais são respectivamente usados para concentrar a definição dos *FBEs* e *Rules*.

Deste modo, a confecção destes métodos será realizada para definir os respectivos *FBEs* e *Rules* no contexto do exemplo apresentado. Entretanto, antes de defini-los é preciso identificar e implementar as classes dos *FBEs* para o dado cenário. Estas classes consistem nos *FBEs Semaphore*, *Crosswalk* e *Car*.

4.3.1 Estrutura da classe FBE

As classes *Crosswalk* e *Car*, por questão de padronização, podem herdar a estrutura da classe *FBE*, assim como apresenta a Figura 56. No entanto, esta não é uma prática obrigatória.

[25] Basicamente, a classe *Application* refere-se a uma implementação do padrão de *software Template Method* (GAMMA, HELM, JOHNSON e VLISSIDES, 1995), devido à estrutura pré-definida e com características de extensão.

Em PON, qualquer classe pode representar um elemento da base de fatos, bastando para isto que ela apresente referência para um *Attribute* ou um *Method*. Na mesma Figura 56, apresenta-se a estrutura estendida das classes *Attribute* e *Method*.

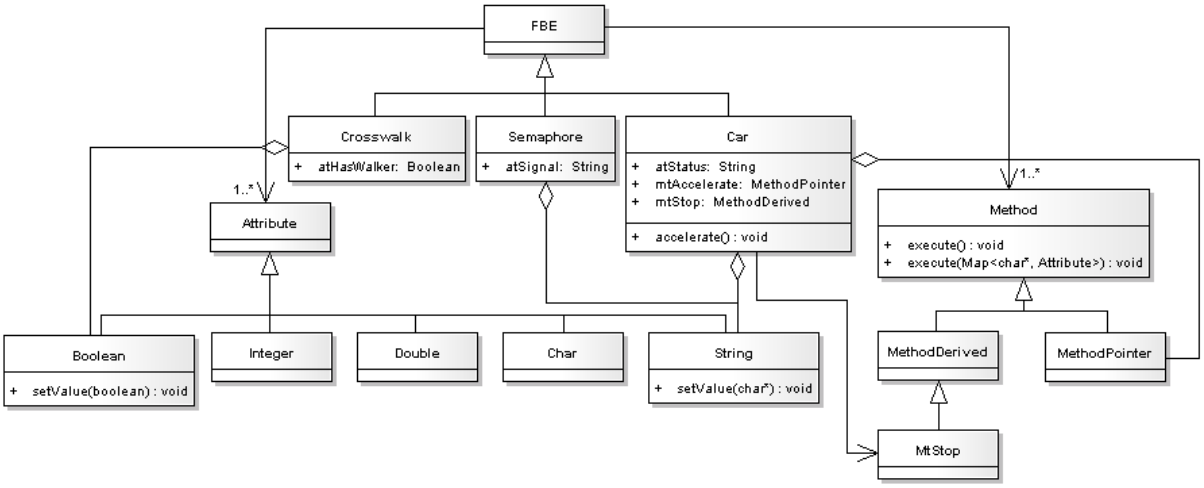


Figura 56: Estrutura estendida dos componentes colaboradores dos FBEs

Particularmente, na representação dos *FBEs Crosswalk* e *Semaphore*, há apenas a declaração de atributos, que são instâncias de duas subclasses de *Attribute* (*Boolean* e *String*). Neste âmbito, como representado na figura, a classe *Attribute* é estendida por várias subclasses, as quais encapsulam os tipos primitivos comuns das linguagens imperativas (e.g. Booleano, inteiros, ponto-flutuante, caractere e cadeia de caracteres).

Ainda, em cada uma destas subclasses do *Attribute*, há a implementação de uma versão particular de um método chamado *setValue(TipoDeDados valor)*, onde *TipoDeDados* se refere ao tipo de dados que esta subclasse do *Attribute* encapsula e *valor* se refere ao parâmetro que modifica o valor interno de uma dada instância desta subclasse. Basicamente, a responsabilidade deste método é receber um valor por parâmetro e atribuí-lo ao dado encapsulado na instância em questão, podendo gerar notificações aos objetos *Premises* conectados, caso este valor seja diferente do anterior.

Na representação do *FBE Car*, por sua vez, há a declaração de certos métodos. Estes métodos consistem em instâncias de subclasses de *Method*, esquematizadas mais à direita na mesma figura. Estas subclasses consistem na classe *MethodPointer* e *MtStop*, a qual é definida explicitamente pelo programador ao estender a classe *MethodDerived*.

Ambas as subclasses apresentam a mesma interface para serem invocadas por um objeto *Instigation*. Esta interface consiste em duas versões do método *execute*, uma com e outra sem parâmetros. Particularmente, o método execute com parâmetros (i.e. *execute(Map<char* Attribute>)*) recebe uma coleção de *Attributes*, os quais correspondem às variáveis *References* ou *Values* conectadas às *Premises* da respectiva *Rule* invocadora.

Essencialmente, uma classe *MethodPointer* modela os objetos que, cada qual, encapsula a chamada de um método definido e declarado na estrutura de um *FBE*, sendo o endereço deste método atribuído ao objeto no momento de sua criação. No exemplo, o objeto *mtAccelerate* é uma instância de *MethodPointer*, a qual encapsula a chamada do método *accelerate()* definida na classe *Car*.

A classe *MtStop*, diferentemente de *MethodPointer*, foi definida explicitamente pelo programador. Nesta, o programador estende a estrutura de *MethodDerived* e implementa o respectivo comportamento (i.e. conjunto de instruções) para fazer o carro parar. A instância *mtStop*, quando invocada pelo respectivo *Instigation*, executará o método implementado em seu próprio escopo. Esta prática atribui maior capacidade de distribuição aos objetos, uma vez que a estrutura dos *FBEs* se encontra mais desacoplada de suas implementações.

Com a identificação das classes *FBEs* e supondo que estas já foram implementadas, os objetos *FBEs* já podem ser definidos no escopo do método *initFacts()* da classe *MyApplication*. Esta definição ocorre de acordo com o trecho de código apresentado na Figura 57. Neste trecho, nas linhas 7, 9 e 11 ocorrem as instanciações dos objetos *FBE* e, nas linhas 8, 10 e 12 ocorrem as atribuições de valores para que a *Rule rlMyRule* implementada no escopo do método *initRules()*, seja aprovada.

```
 1 ...
 2 Semaphore* semaphore;
 3 Crosswalk* crosswalk;
 4 Car* car;
 5 ...
 6 void MyApplication::initFacts(){
 7   semaphore = new Semaphore();
 8   semaphore->atSignal->setValue("GREEN");
 9   crosswalk = new Crosswalk();
10   crosswalk->atHasWalker->setValue(false);
11   car = new Car();
12   car->atStatus->setValue("STOPPED");
13 }
14
15 void MyApplication::initRules(){
16   Rule* rlMyRule = new Rule(Condition::CONJUNCTION);
17   rlMyRule->addPremise(semaphore->atSignal,    "GREEN",   Premise::EQUAL);
18   rlMyRule->addPremise(crosswalk->atHasWalker,  false,    Premise::EQUAL);
19   rlMyRule->addPremise(car->atStatus,          "STOPPED", Premise::EQUAL);
20   rlMyRule->addInstigation(car->mtAccelerate);
21   rlMyRule->addInstigation(semaphore->atSignal, "RED");
22   rlMyRue->end;
23 }
24 ...
```

Figura 57: Inicialização dos *FBEs* e *Rules* em C++

4.3.2 Estrutura da classe *Rule*

A estrutura da *rlMyRule* é representada pela instanciação de um objeto *Rule* e o seu conhecimento é inserido nesta estrutura por meio da passagem de parâmetros aos métodos intuitivos da *Rule*. Em tempo de execução, de acordo com os parâmetros, o objeto *Rule* cria os objetos colaboradores e os conecta a fim de que possam constituir o mecanismo de notificações. Este processo ocorre de forma transparente ao programador.

Como se pode constatar, mesmo com a definição desta *Rule* em código de uma linguagem imperativa (C++), a classe *Rule* apresenta interfaces bem definidas e intuitivas que tornam simples a composição e a compreensão do que se deseja expressar. Por meio desta estrutura, a *Rule* se assimila em termos de capacidade de compreensão à regra representada na Figura 53.

Desta forma, por meio de uma simples análise da representação da *Rule* em C++, consegue-se facilmente compreender a sua estrutura e conseqüentemente, a sua semântica. Basicamente, a respectiva *Rule* apresenta a sua *Condition* composta por três objetos *Premises* e sua *Action* composta por dois objetos *Instigations*.

As três *Premises* atestam se o sinal do semáforo está verde, se não há pedestres atravessando a faixa e se o próprio automóvel do robô-motorista está parado. Estas *Premises* são criadas pela execução do método *addPremise*, o qual recebe três parâmetros necessários para a invocação do respectivo método construtor (i.e. *Reference*, *Value* e *Operator*). Por exemplo, na criação da primeira *Premise* da *rlMyRule*, o *Reference* corresponde ao *Attribute Semaphore->atSignal* e o *Value* corresponde à constante "*GREEN*", sendo que ambos são comparados pelo operador (*Operator*) de igualdade (i.e. *Premise::EQUAL*)[26].

Quando um novo objeto *Premise* é criado, este se auto-conecta ao(s) respectivo(s) objeto(s) *Attribute*(s) (i.e. *Reference* e/ou *Value*), armazenando sua referência em uma estrutura de dados implementada nestes elementos (i.e. lista encadeada ou tabela *hash*). Com

---

[26] No entanto, devido às funcionalidades próprias do *framework*, o objeto *Premise* somente é criado quando ele ainda não estiver sido definido em outra *Rule*. Esta funcionalidade evita avaliações redundantes. Assim, o objeto *Premise* referente a vários objetos *Rules* é compartilhado automaticamente entre as *Conditions* pertinentes.

Ainda, este mesmo processo pode ser realizado manualmente pelo programador. Isto é feito instanciando um objeto *Premise* comum a duas ou mais *Rules* e compartilhando este objeto com essas pela passagem do endereço deste objeto ao método *addPremise* de cada uma das *Rules*. Na estrutura da classe *Rule*, o método *addPremise* apresenta uma assinatura que aceita um objeto *Premise* como parâmetro, conectando-o a respectiva *Condition*.

isto, a qualquer alteração de valores neste(s) *Attribute*(s), os mesmos terão como notificar esta *Premise* e as demais que venham a ser conectadas posteriormente.

As *Premises* são relacionadas pela *Condition* exemplificada por meio de uma conjunção. Como observado na estrutura da *Rule* exemplificada, o operador lógico de uma *Condition* é definido por meio da passagem de uma constante identificadora ao construtor da *Rule*. Estas constantes podem denotar uma conjunção (pelo identificador CONJUNCTION), uma disjunção (pelo identificador DISJUNCTION) ou nenhum operador (pelo identificador SINGLE). O uso do identificador SINGLE é apropriado quando a *Condition* apresenta apenas uma *Premise* conectada, dispensando avaliações por meio de operadores lógicos.

Na mesma *Rule*, a *Action* corresponde a dois objetos *Instigations* conectados. A execução destes *Instigations* depende da forma pela qual foram definidos. Na definição da respectiva *Rule*, estes *Instigations* foram conectados a *Action* por meio da execução do método *addInstigation*, o qual apresenta duas variações em relação aos seus parâmetros. Na linha 20, o *Instigation* conectado recebe uma referência para um objeto *Method* a fim de invocar o respectivo método encapsulado. Na linha 21, o *Instigation* conectado recebe uma referência para um *Attribute* e um valor constante a ser atribuído a este *Attribute*.

Nesta última representação, o objeto *Instigation* desvia o fluxo de notificações estabelecido inicialmente (Figura 38), referente à notificação estrita de um *Method*. O *Instigation* notifica diretamente um *Attribute*, reduzindo a quantidade de notificações para atingir o mesmo objetivo. De fato, esta representação facilita a compreensão da semântica da *Rule*, pois as particularidades da alteração de um *Attribute* (i.e. valor recebido) se apresenta no mesmo escopo da *Rule* ao invés de se encontrar implementado em um método.

Esta relação direta da classe *Instigation* com a classe *Attribute* é ilustrada à direita na Figura 58. Na mesma figura, à esquerda, apresenta-se a relação entre a *Condition* e os operadores lógicos suportados[27].

[27] Esta última relação foi implementada segundo a estrutura do padrão de *software Strategy* (GAMMA, HELM, JOHNSON e VLISSIDES, 1995).

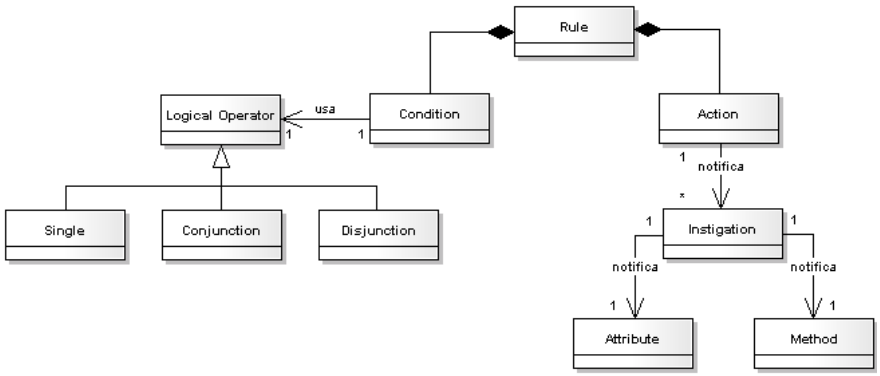


Figura 58: Estrutura estendida dos componentes colaboradores das *Rules*

4.3.3 Reflexões

Esta seção demonstrou a estrutura do *framework* proposto. Esta estrutura apresenta objetos com interfaces bem definidas que facilitam a composição de programas de forma similar às linguagens declarativas, porém com maior potencial devido ao uso das capacidades das linguagens imperativas. Da mesma forma que nas linguagens declarativas, o programador somente precisa definir os dados de entrada sobre a estrutura de uma regra, uma vez que a implementação do *framework* se responsabiliza pelas particularidades da implementação (e.g. criação dos objetos e conexões entre estes).

4.4 PARTICULARIDADES DO *FRAMEWORK* EM C++

A seção anterior apresentou uma visão geral do *framework* proposto. No entanto, algumas particularidades referentes à materialização dos componentes do PON não foram elucidados naquela seção devido à incompatibilidade com o exemplo abordado e para evitar que a explicação se tornasse mais complexa e extensa.

Desta forma, esta seção explana algumas particularidades de cada um dos componentes participantes da aprovação de uma *Rule* (i.e. *Attribute*, *Premise* e *Condition*) e a execução do próprio conteúdo da *Rule*. Estas particularidades são de uso opcional no *framework* e podem ser usadas, se assim entendido, para melhorar a eficiência dos programas e a representatividade das regras. Os objetos das classes *Action*, *Instigation* e *Method* não serão abordados nesta seção, principalmente porque uma das soluções propostas os eliminam.

4.4.1 Implementação do *Attribute*

4.4.1.1 Notificações otimizadas

A essência de execução de um *Attribute* é detectar mudanças em seu estado e notificar um conjunto de *Premises* interessadas neste evento. A priori, as referências para estas *Premises* são reunidas em uma lista encadeada. Desta forma, o *Attribute* percorre seqüencialmente cada uma destas *Premises* a fim de notificar a mudança de estado.

No entanto, esta prática se torna ineficiente quando o número de *Premises* a serem notificadas é muito alto. Esta ineficiência se deve porque grande parcela das *Premises* não é afetada por um determinado estado de um *Attribute* e mesmo assim são notificadas. Este fato é constatado no esquema à esquerda na Figura 59.

Nesta figura, o *Attribute atSignal* teve seu estado alterado de vermelho (*RED*) para verde (*GREEN*), devendo notificar apenas as *Premises* interessadas nestes dois estados para que atualizem os seus valores lógicos. Entretanto, devido ao fluxo de notificação guiado pela lista encadeada, a *Premise* que verifica se *atSignal* é amarelo (*YELLOW*) também é notificada, gerando uma avaliação lógica desnecessária. Neste simples exemplo, os efeitos na performance é imperceptível, mas em exemplos que apresentam maior variedade de estados, como em avaliações numéricas, esta prática pode ser inaceitável.

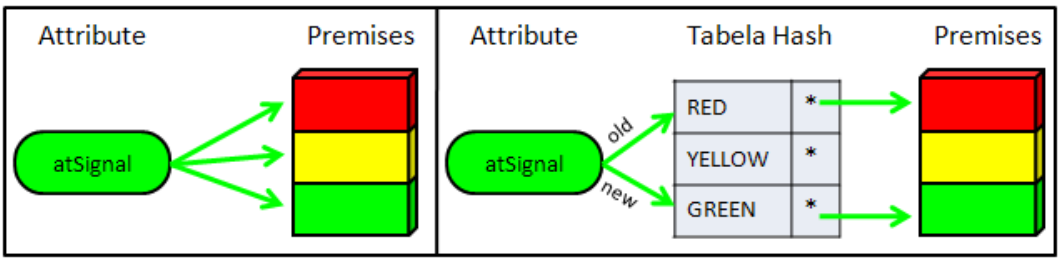


Figura 59: Notificações baseadas em lista encadeada e tabela *hash*

Para evitar estas avaliações desnecessárias, o programador pode optar por substituir a lista encadeada por uma tabela *hash*, que relaciona dois elementos: a chave e o valor. Esta opção é informada na criação de um objeto *Attribute*. Na implementação com a tabela *hash*, a chave corresponde ao estado ao qual a *Premise* compara um *Attribute* e o valor corresponde ao próprio endereço da *Premise*[28]. Com esta alteração estrutural, o *Attribute* notifica somente

[28] A chave da tabela *Hash* também pode representar intervalos entre valores para casos em que as *Premises* notificadas adotam operadores relacionais diferentes do igual, ou seja, maior do que, menor do que, maior ou igual, menor ou igual. Esta chave pode ser calculada por uma função de acordo com o valor do *Attribute*.

as *Premises* interessadas na mudança de seu estado, como ilustra o esquema à direita na Figura 59.

Neste esquema, o *Attribute atSignal* notifica somente as duas *Premises* pertinentes, as quais se referem ao estado anterior e ao atual do respectivo *Attribute*. Com esta solução, a *Premise* que atesta o sinal amarelo não é notificada, poupando recursos de processamento.

Portanto, a tabela *hash* pode ser empregada como alternativa para prover maior eficiência ao modelo proposto. A aplicação desta estrutura de dados é indicada a praticamente todos os tipos de dados suportados pelo *framework*, com exceção ao tipo Booleano. Devido a sua paridade de valores (i.e. falso ou verdadeiro), o tipo Booleano deve necessariamente notificar todas as *Premises* conectadas a cada mudança de estado.

4.4.1.2 Re-notificações

De fato, as otimizações nos *Attributes* se apresentam imprescindíveis para poupar tempo de processamento, uma vez que é nestes que se originam os fluxos de notificações. Assim, ao otimizar o comportamento destas entidades pode-se evitar que as notificações ocorram de forma desnecessária.

Com o objetivo de evitar notificações desnecessárias, a implementação atual de um *Attribute* somente permite que as *Premises* conectadas sejam notificadas quando o seu estado for alterado. Com isto, evita-se que as *Premises* re-avaliem o mesmo estado. No entanto, apesar desta solução se apresentar conveniente para a maioria das aplicações, a mesma pode não se apresentar tão favorável em outras. Há aplicações que demandam que uma regra seja reavaliada e re-executada mesmo quando os estados dos atributos referenciados continuam invariáveis.

Para que uma *Rule* seja re-aprovada para execução é necessário a ocorrência de dois fluxos de notificações. No primeiro, qualquer *Attribute* referenciado na *Condition* da *Rule* deve ter seu estado alterado para um valor que venha a afetar o estado lógico de sua *Premise* conectada, que consecutivamente irá desaprovar a respectiva *Condition*. No segundo, o estado alterado (do *Attribute*) deve ser reconstituído, assim o fluxo de notificações se reinicia em direção à re-aprovação da *Rule*.

De fato, este processo é bastante custoso em termos de processamento, uma vez que o fluxo para a desaprovação da *Rule* ocorre de forma desnecessária. Para otimizar este processo se faz necessário que um *Attribute* apresente a opção de iniciar um fluxo de notificações

mesmo quando a atribuição de um valor não corresponde a mudança de seu estado. Neste processo, o *Attribute* apresenta a qualidade de re-notificar o seu estado aos objetos colaboradores relacionados visando à re-aprovação de uma *Rule*, uma vez que esta ainda pode apresentar estado lógico verdadeiro.

No entanto, para que um *Attribute* possa re-notificar o seu estado é preciso fazer uso de uma nova versão do método *setValue(TipoDeDados valor)*, cuja assinatura é *setValue(TipoDeDados valor, bool flag),* onde *valor* se refere ao novo estado atribuído ao *Attribute* e *flag* se refere ao uso de re-notificações, quando for necessário. Se a re-notificação for desejada, o programador usa a *flag Attribute::RENOTIFY*, caso contrário basta usar a assinatura padrão do método.

O processo de re-notificação se difere do processo tradicional de notificação, uma vez que este ocorre com apenas um fluxo de notificações. O processo de re-notificação funciona da seguinte maneira: quando um *Attribute* recebe uma atribuição de valor que não varia o seu estado, ele re-notifica prontamente uma ou mais *Premises* conectadas. Cada *Premise* re-notificada não realiza o cálculo lógico, exceto quando o *Reference* e *Value* se referem à *Attributes* diferentes. Na seqüência, cada *Premise* re-notifica uma ou mais *Conditions* conectadas. Por fim, esta(s) realiza(m) o(s) seu(s) cálculo(s) lógico(s) para confirmar a continuação da veracidade de seus estados, uma vez que as demais *Premises* não participantes deste fluxo poderiam ter mudado de estados neste intervalo de tempo.

No entanto, mesmo que a *flag* seja usada, o processo de re-notificações ocorre somente quando for necessário. Em casos nos quais o *Attribute* variar de estado, as notificações ocorrem normalmente pelo processo tradicional. Portanto, o emprego das re-notificações se fazem vantajosas ao exigir apenas um fluxo de notificações e, mesmo assim, otimizado (i.e. não há a avaliação de *Premises*). Ainda, para tornar mais vantajosa, as re-notificações podem se tornar extremamente pontuais com o emprego das tabelas *hash* para guardar referência das *Premises* conectadas.

4.4.2 Implementação do *Premise*

Da mesma forma que as notificações e re-notificações dos *Attributes* podem poupar processamento, o mesmo pode ocorrer com as notificações originadas nas *Premises*. Ao perceber mudanças em seus respectivos estados lógicos, as *Premises* notificam cada uma das *Condition*s conectadas, para que estas atualizem os seus respectivos estados. Normalmente,

todas estas notificações são necessárias porque o PON segue o princípio da avaliação a priori das regras, onde uma *Condition* é avaliada em partes, no instante em que é necessário e, guarda esta informação para atualização de seu estado quando receber notificações de suas outras partes. Este processo se difere da avaliação de uma expressão causal no imperativo, onde esta é avaliada de uma única vez.

No mecanismo padrão adotado no modelo, as *Premises* notificam todas as suas *Condition*s, as quais são referenciadas em uma lista encadeada. No entanto, há situações particulares em que uma *Premise* não precisa notificar todas as *Condition*s de sua lista, basta que as notificações ocorram até que uma das *Condition*s seja satisfeita. Nestes casos, somente há a possibilidade ou necessidade de execução de uma única *Rule* em relação ao estado de uma respectiva *Premise*. No imperativo, estes casos são implementados com o uso de expressões causais aninhadas ou então por meio de blocos de escolha (blocos *switch*).

Da mesma forma, um mecanismo semelhante aos blocos de seleção do imperativo foi adotado em PON. Com isto, em certos casos, reduz-se a quantidade de *Condition*s notificadas, evitando processamentos desnecessários. Este mecanismo foi implementado com a adição de um comportamento opcional a um objeto *Premise*, a qual passa a ser chamada de *Premise Exclusive* (premissa exclusiva)[29].

Para que este comportamento seja ativado, necessita-se que o programador defina explicitamente esta opção por meio da atribuição da *flag Premise::EXCLUSIVE* ao construtor de uma *Premise*. O emprego desta *flag* à instanciação de objetos *Premises* pode ser constatada nas linhas 3, 10 ou 17 nas três *Rules* apresentadas na Figura 60. Esta *Premise* confirma o direcionamento do veículo em um cruzamento de acordo com o trajeto programado ao robô-motorista, ou seja, faz com que ele siga em frente ou então, vire à esquerda ou à direita.

[29] Basicamente, o conceito de *Premise Exclusive* em questão se refere a uma versão para ambientes monoprocessados do conceito de *Premise Exclusive* presente no contexto do modelo de resolução de conflito para ambientes multiprocessados proposto por Simão (Simão, 2005).

```
1 ...
2 Rule* rlCurveToLeft = new Rule(Condition::CONJUNCTION);
3 rlCurveToLeft->addPremise(semaphore->atSignal, "GREEN", Premise::EQUAL, Premise::EXCLUSIVE);
4 rlCurveToLeft->addPremise(crosswalk->atHasWalker, false, Premise::EQUAL);
5 rlCurveToLeft->addPremise(car->atCurve, "LEFT", Premise::EQUAL);
6 rlCurveToLeft->addInstigation(car->mtCurveToLeft);
7 rlCurveToLeft->end();
8
9 Rule* rlNoCurve = new Rule(Condition::CONJUNCTION);
10 rlNoCurve->addPremise(semaphore->atSignal, "GREEN", Premise::EQUAL, Premise::EXCLUSIVE);
11 rlNoCurve->addPremise(crosswalk->atHasWalker, false, Premise::EQUAL);
12 rlNoCurve->addPremise(car->atCurve, "NO", Premise::EQUAL);
13 rlNoCurve->addInstigation(car->mtNoCurve);
14 rlNoCurve->end();
15
16 Rule* rlCurveToRight = new Rule(Condition::CONJUNCTION);
17 rlCurveToRight->addPremise(semaphore->atSignal, "GREEN", Premise::EQUAL, Premise::EXCLUSIVE);
18 rlCurveToRight->addPremise(crosswalk->atHasWalker, false, Premise::EQUAL);
19 rlCurveToRight->addPremise(car->atCurve, "RIGHT", Premise::EQUAL);
20 rlCurveToRight->addInstigation(car->mtCurveToRight);
21 rlCurveToRight->end();
22 ...
```

Figura 60: Definição de *Premise Exclusive* em código C++

Neste exemplo, ao se aproximar de um cruzamento, o robô-motorista recalcula os seus direcionamentos e define a sua próxima manobra. Neste caso, o direcionamento determinado foi continuar seguindo em frente. Neste contexto, ao detectar o sinal verde do semáforo, uma das três *Rules* apresentadas na Figura 60 deve ser executada, uma vez que o motorista pode seguir apenas em uma direção em um certo tempo

Pela implementação padrão da *Premise*, as três *Conditions* destas *Rules* seriam avaliadas consecutivamente. Porém, com o comportamento de exclusividade, a *Premise* afetada pelo sinal do semáforo (linhas 3, 10 ou 17) notifica as *Conditions* até que a primeira delas seja satisfeita.

Este mecanismo de decisão é esquematizado na Figura 61, onde a *Premise Exclusive* percorre a sua lista de três *Condition*s de forma seqüencial, da esquerda para a direita. Na primeira notificação, a respectiva *Condition* não é satisfeita, confirmando este estado para a *Premise Exclusive*. Na seqüência, a segunda *Condition* é notificada, a qual atualiza o seu estado lógico e responde com uma confirmação de aprovação à *Premise Exclusive*. Com isto, a *Premise Exclusive* encerra o seu ciclo de notificações, evitando que as próximas *Condition*s sejam notificadas, pois estas certamente não serão satisfeitas pelo seu estado.

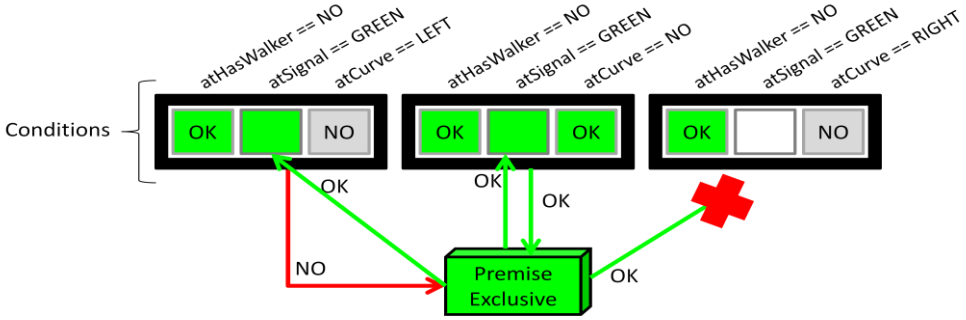


Figura 61: Esquematização da *Premise* Exclusiva

4.4.3 Implementação da *Condition*

Esta seção apresenta uma funcionalidade opcional sobre a representatividade das *Conditions*. Esta permite maior flexibilidade nas *Condition*s, ao permitir que elas organizem a relação entre suas *Premises* por meio de diferentes combinações de operadores lógicos. Além de proporcionar maiores qualidades de representação, uma variação desta funcionalidade pode prover melhor desempenho ao modelo proposto.

Como visto na seção 4.3, uma *Condition* pode relacionar as suas *Premises* por meio de dois operadores lógicos, a conjunção e a disjunção. Porém, até então, uma *Condition* podia adotar, por vez, apenas um único operador lógico para relacionar as suas *Premises* uma vez que a estrutura da *Condition* não suporta intercalações entre operadores lógicos.

A composição de premissas intercaladas por diferentes operadores forma uma expressão causal complexa. Estes tipos de expressões são comumente implementadas no PI. Por exemplo, a expressão causal apresentada na Figura 62 representa um operador de conjunção que interliga uma premissa solitária e um conjunto de três premissas relacionadas por operadores de disjunção. Nesta representação, a ordem de avaliação destas premissas é definida por meio das delimitações por parênteses, como ilustra a mesma figura.

```
1 ...
2 if( (semaphore->atSignal == "RED") &&
3     (car->atCurve == "LEFT" || car->atCurve == "RIGHT" || car->atCurve == "NO"))
4 {
5      car->stop();
6 }
7 ...
```

Figura 62: Exemplo de uma expressão causal complexa em C++

Na implementação padrão da *Condition*, esta mesma expressão causal (Figura 62) seria apresentada por três diferentes *Rules*, como ilustra a Figura 63. Nestas *Rules*, cada *Condition* representa, por meio de um operador de conjunção, a relação particular entre a *Premise* solitária (i.e. que atesta o sinal do semáforo) e uma das *Premises* do conjunto de disjunção (i.e. as que atestam a direção do percurso do motorista).

No entanto, esta replicação pode causar problemas de desempenho devido ao aumento do número de *Conditions* a serem notificadas. Por exemplo, considerando que a *Premise* replicada nas linhas 3, 9 e 15 da Figura 63 tem o seu estado alterado, esta deve notificar as três *Condition*s conectadas, enquanto que, se a *Condition* suportasse a representação de expressões complexas, a respectiva *Premise* precisaria notificar apenas uma vez.

```
1 ...
2 Rule* rlCurveToLeftStop = new Rule(Condition::CONJUNCTION);
3 rlCurveToLeftStop->addPremise(semaphore->atSignal, "RED", Premise::EQUAL);
4 rlCurveToLeftStop->addPremise(car->atCurve, "LEFT", Premise::EQUAL);
5 rlCurveToLeftStop->addInstigation(car->mtStop);
6 rlCurveToLeftStop->end();
7
8 Rule* rlNoCurveStop = new Rule(Condition::CONJUNCTION);
9 rlNoCurveStop->addPremise(semaphore->atSignal, "RED", Premise::EQUAL);
10 rlNoCurveStop->addPremise(car->atCurve, "NO", Premise::EQUAL);
11 rlNoCurveStop->addInstigation(car->mtStop);
12 rlNoCurveStop->end();
13
14 Rule* rlCurveToRightStop = new Rule(Condition::CONJUNCTION);
15 rlCurveToRightStop->addPremise(semaphore->atSignal, "RED", Premise::EQUAL);
16 rlCurveToRightStop->addPremise(car->atCurve, "RIGHT", Premise::EQUAL);
17 rlCurveToRightStop->addInstigation(car->mtStop);
18 rlCurveToRightStop->end();
19 ...
```

Figura 63: Representação padrão de uma expressão causal complexa em PON

Mesmo que esta *Premise* seja declarada como exclusiva e que a primeira *Condition* notificada seja satisfeita, ainda haverá um problema. Este se deve a replicação da mesma *Action* pelas três *Rules*[30]. A solução para estas limitações é permitir que as *Conditions* representem expressões complexas. Com isso, as mesmas três *Rules* poderiam ser definidas em apenas uma *Rule*.

Esta solução surge com a proposta de uma especialização da classe *Condition*, a *SubCondition*. Esta representa um agrupamento das *Premises* de uma *Condition* que estão relacionadas por um mesmo tipo de operador lógico. Por exemplo, a Figura 64 apresenta em forma hierárquica a representação da expressão causal imperativa da Figura 62 sobre os agrupamentos em *SubConditions*, onde cada agrupamento se refere às *Premises* delimitadas por parênteses naquela expressão. Como se pode observar, a classe *SubCondition* suporta os mesmos operadores lógicos da *Condition* (i.e. conjunção e disjunção), porém oferece maior organização a estes e permite as suas coexistências em uma mesma *Condition*.

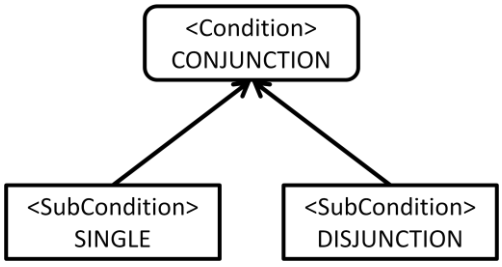


Figura 64: Representação esquemática de uma regra complexa com sub-condições

Por meio da materialização deste esquema, as três *Rules* apresentadas na Figura 63 são reapresentadas na Figura 65 em apenas uma *Rule*. As *SubConditions* agregam maior

[30] Em outros casos, quando as *Premises Exclusives* não forem empregadas, esta replicação pode resultar na execução da mesma *Action* redundantemente quando as respectivas *Rules* fores aprovadas para execução.

expressividade à *Rule* ao permitir o agrupamento de várias relações no corpo de uma única *Rule*. Isto facilita a leitura e entendimento de uma *Rule*, uma vez que o programador consegue ver todas as relações de uma única vez, ao invés de ter que examinar cada *Rule* separadamente.

```
 1 ...
 2 Rule* rlStop = new Rule(Condition::CONJUNCTION);
 3
 4 rlStop->addSubCondition(Condition::SINGLE)
 5 rlStop->addPremiseToSubCondition(semaphore->atSignal, "RED", Premise::EQUAL);
 6
 7 rlStop->addSubCondition(Condition::DISJUNCTION)
 8 rlStop->addPremiseToSubCondition(car->atCurve, "LEFT", Premise::EQUAL);
 9 rlStop->addPremiseToSubCondition(car->atCurve, "RIGHT", Premise::EQUAL);
10 rlStop->addPremiseToSubCondition(car->atCurve, "NO", Premise::EQUAL);
11
12 rlStop->addInstigation(car->mtStop);
13 rlStop->end();
14 ...
```

Figura 65: Representação de uma expressão causal complexa com sub-condições

Como ilustrado na representação da *Rule rlStop* (Figura 65), o uso de *SubConditions* é completamente intuitivo. Primeiramente, define-se o operador lógico das *SubConditions* (linhas 4 e 7) e na seqüência, conecta as respectivas *Premises* à estas *SubConditions*. Com isto, as *Premises* notificarão a *Condition* indiretamente, por meio das suas *SubConditions*. Neste contexto, as *SubConditions* farão o papel das *Premises*, notificando as respectivas *Condition*s no momento em que tiverem seus estados lógicos alterados. Para uma *Condition*, é indiferente receber notificações de *SubConditions* ou *Premises*, uma vez que os seus estados lógicos são atualizados da mesma maneira.

4.4.4 Implementação da *Rule*

Como apresentado nas seções anteriores, o PON oferece algumas opções de melhorias para tornar o seu modelo ainda mais eficiente e representativo. Essencialmente, estas melhorias se referem aos objetos participantes do chamado fluxo ativo de notificações (i.e. *Attribute*, *Premise*, *Condition*), o qual visa à aprovação de *Rules* para a execução.

O fluxo ativo de notificações representa a essência do mecanismo de notificações, pois permite estabelecer as aprovações das *Rules* ao mesmo tempo em que resolve os seus conflitos. Por sua vez, o fluxo passivo de notificações (i.e. *Action*, *Instigation*, *Method*) não carrega tantas responsabilidades, uma vez que ele se restringe às invocações dos métodos nos *FBEs*. Particularmente, os elementos deste fluxo se fazem efetivamente necessários na sua

totalidade na implementação de sistemas multiprocessados a fim de representar maior desacoplamento e independência entre as partes.

Neste sentido, na programação monoprocessada, o uso destes elementos pode ser omitido para melhorar o desempenho das aplicações. Assim, esta seção apresenta uma solução a ser usada, opcionalmente, a fim de omitir este fluxo de notificações. Esta solução permite invocar os métodos ou modificar os atributos nos *FBEs* de forma direta, sem intervenção dos objetos colaboradores.

A solução proposta refere-se à implementação do conteúdo da *Rule* diretamente em seu corpo, por meio do método invocado em sua execução. Este método é o *execute()*, o qual é invocado pela *Condition* no modelo descentralizado ou pelo objeto da classe *ConflictSet* no modelo de resolução de conflitos centralizado, assim como ilustrado na Figura 66. A implementação ocorre por meio da derivação da classe *Rule* para a subscrição do respectivo método.

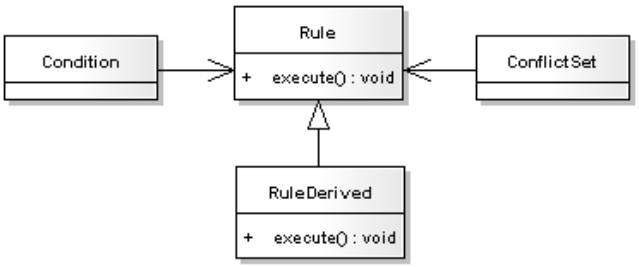


Figura 66: Estrutura da classe *Rule*

Esta implementação é ilustrada no trecho de código da Figura 67. Nesta, a ação da *Rule* é implementada com toda a flexibilidade do PI, permitindo atribuições de valores e chamadas de métodos implementados de forma imperativa. Nesta representação, o trecho de código altera a cor do sinal do semáforo para vermelho e logo imprime uma mensagem informativa na tela. Para alterar o sinal, a referência para o *Attribute atSignal* é extraído de uma tabela *hash* interna a *Rule*. Esta tabela guarda as referências para os *Attributes* conectados às *Premises* da mesma *Rule*.

```
 1 ...
 2 RuleDerived* rlCPlusPlus = new RuleDerived(Condition::CONJUNCTION);
 3 rlCPlusPlus->addPremise(semaphore->atSignal, "GREEN", Premise::EQUAL);
 4 rlCPlusPlus->addPremise(crosswalk->atHasWalker, false, Premise::EQUAL);
 5 rlCPlusPlus->end();
 6 ...
 7
 8 RuleDerived::execute(){
 9   attributes["atSignal"]->setValue("RED");
10   cout << " PARE!!! " << endl;
11 }
12 ...
```

Figura 67: Código exemplo da derivação de uma *Rule* em C++

Mesmo que esta prática venha favorecer o desempenho ao reduzir a quantidade de notificações, a necessidade de derivação da classe *Rule* para implementação se torna tediosa, diferentemente da solução padrão, onde um objeto *Rule* é apenas instanciado e tem os parâmetros de seus métodos preenchidos com os respectivos valores. Neste ponto de vista, a busca por melhorias no desempenho pode afetar a praticidade da definição de uma *Rule*.

No entanto, este problema pode ser contornado com as funcionalidades de uma ferramenta *wizard* ou automaticamente, com o emprego de outra linguagem de programação, tal como a Java. Na linguagem Java, não há a necessidade de derivação da classe para subscrever o seu método. Estes podem ser subscritos no instante em que são invocados. Neste sentido, a Figura 68 apresenta a implementação em Java da mesma *Rule* da Figura 67.

```
1 Rule rlJava = new Rule(Condition.CONJUNCTION);
2 rlJava.addPremise(semaphore.atSignal, "GREEN", Premise.EQUAL);
3 rlJava.addPremise(crosswalk.atHasWalker, false, Premise.EQUAL);
4 rlJava.execute()
5 {
6   attributes["atSignal"].setValue("RED");
7   System.out.println(" PARE!!! ");
8 }
```

Figura 68: Código exemplo da derivação de uma *Rule* em Java

Em suma, a omissão do fluxo passivo de notificações, mesmo gerando maior acoplamento entre as *Rules* e os *FBEs*, é uma solução que minimiza o número de notificações, tornando o modelo mais eficiente. Com isto, o PON pode oferecer todo o potencial do mecanismo de notificações para realizar a fase de *matching* e seleção das *Rules* e usar todo a capacidade da programação imperativa para implementar o conteúdo de execução das *Rules*.

4.4.5 Reflexões

De fato, as particularidades apresentadas agregam ainda maiores capacidades aos objetos que compõem o mecanismo de notificações, tornando-os ainda mais efetivos na resolução das deficiências dos atuais paradigmas, principalmente no tocante à eficiência de execução.

Nas seções precedentes, as soluções para estas deficiências foram explanadas com referência íntima a questões de implementação, permitindo a percepção de que os conceitos de *FBEs* e *Rules* e mesmo a forma pela qual estes interagem consiste em muito mais do que uma mera junção de conceitos do POO e SBR em uma linguagem de programação imperativa. Na verdade, estes se referem a evoluções sofisticadas dos conceitos dos atuais

paradigmas. Desta forma, mesmo que algumas iniciativas de programação proponham unir os conceitos destes paradigmas, estas não atingem os mesmos benefícios oferecidos em PON, como é constato na seção seguinte.

4.5 MATERIALIZAÇÃO DE JUNÇÕES DOS ATUAIS PARADIGMAS

Nas últimas décadas, algumas iniciativas têm sido propostas para unir os conceitos dos SBR com os conceitos do POO. O principal objetivo desta integração é oferecer maior flexibilidade e capacidade na composição das regras e elementos da base de fatos, representando os fatos por meio de estados de atributos de objetos e permitindo que os atributos e métodos destes sejam acessados diretamente pelas regras.

Estas iniciativas se assemelham parcialmente ao objetivo do PON, uma vez que estas também procuram se beneficiar das vantagens do PI e PD. No entanto, estas iniciativas apenas integram de alguma forma os conceitos já existentes a fim de aumentar a capacidade de programação dos *shells* de SBR enquanto o PON une evoluções dos conceitos atuais sobre uma nova perspectiva de programação. Desta forma, mesmo que estas iniciativas aumentem a capacidade de programação dos SBR, elas continuam apresentando as mesmas deficiências dos paradigmas que as compõem.

Algumas destas iniciativas se referem aos *shells* de SBR chamados CLIPS++, RETE++ e ILOG Rules e à linguagem de programação declarativa que estende C++, chamada R++. De uma forma geral, estas iniciativas unem os princípios do POO materializados sobre a linguagem imperativa C++ com os conceitos declarativos próprios dos SBR, inclusive a linguagem particular destes *shells* ou a sintaxe particular dos comandos adicionais impostos pela linguagem R++. Para fim de preciosismo de explicação, estas iniciativas são detalhadas no Apêndice B.

Em suma, estas iniciativas apresentam algumas limitações. Estas não integram os conceitos dos atuais paradigmas de forma simplória. Para fazer uso destas iniciativas de programação se faz necessário conhecer as sintaxes de ambas as linguagens envolvidas. Desta forma, estas iniciativas incentivam que os programas sejam construídos pela combinação das estruturas e também das sintaxes de ambos os paradigmas.

Ainda, a maioria destas iniciativas prioriza o uso das linguagens “atípicas” dos SBR ao invés das linguagens do POO, sendo que estas são as mais populares perante a comunidade de programadores. Na maioria das iniciativas, a linguagem POO é considerada apenas como uma

linguagem de extensão às linguagens dos *shells*. Por meio das linguagens declarativas dos *shells*, o programador deve compor as regras e definir o acesso aos atributos e aos métodos dos objetos/elementos da base de fatos implementados em linguagem imperativa.

Para manter compatibilidade entre as linguagens envolvidas, o código da linguagem declarativa é traduzido automaticamente para o código da linguagem imperativa através de um pré-compilador particular. No entanto, na maioria das vezes, estas traduções geram um código praticamente ilegível devido às características genéricas do pré-compilador.

Entretanto, algumas iniciativas podem oferecer uma ferramenta *Wizard* (e.g. ILOG Rules) para tornar a manipulação de ambas as linguagens menos tediosa. No entanto, o uso destas ferramentas impede o programador de fazer uso de toda a flexibilidade das linguagens imperativas.

Em um foco comparativo destas iniciativas apresentadas com o *framework* do PON, a principal diferença está no propósito e nas capacidades destas soluções. Enquanto o *framework* permite programar sobre uma nova perspectiva ao contar com a evolução dos conceitos do PI e PD, as iniciativas apresentadas apenas mesclam os conceitos já existentes e ainda o fazem de forma conturbada ao misturar as sintaxes de linguagens de ambos os paradigmas. Assim, estas iniciativas continuam apresentando as mesmas deficiências dos paradigmas mesclados enquanto que o PON oferece a solução para estas.

A atual materialização do PON provê o estilo declarativo e imperativo de programação naturalmente aos programadores habituados com as linguagens imperativas. Assim, sobre o *framework* do PON, o programador pode criar o código-fonte de um programa sobre qualquer editor de código compatível com a linguagem imperativa adotada e ainda, este código pode ser compilado por qualquer compilador que apresente a mesma compatibilidade com a linguagem empregada.

Em termos de facilidades de programação, mesmo o *framework* apresentando *interfaces* bem definidas que facilitam a manipulação dos componentes do PON em linguagens imperativas, o programador ainda pode fazer uso da ferramenta *Wizard* do PON a fim de tornar mais fácil a manipulação dos componentes e consecutivamente, melhorar a sua produtividade. Deste modo, o programador pode escolher entre manipular os componentes do PON em estilo imperativo (i.e. usando linguagem imperativa) ou no estilo declarativo (i.e. usando a ferramenta *Wizard*), sendo que ambos os estilos são de fácil aplicação em PON.

4.6 CONCLUSÃO

Este capítulo apresentou os assuntos pertinentes à materialização de uma nova versão do meta-modelo proposto inicialmente por Simão (2005). Esta nova versão se apresenta na forma de um *framework* e envolve alguns avanços estruturais e comportamentais nos elementos do PON. Ainda, uma vez que a essência do PON é fundamentada e evolui dos conceitos do POO e dos SBR, este capítulo discutiu sobre algumas iniciativas de programação que mesclam estes mesmos conceitos, mas na forma pura (i.e. sem evoluções sobre os conceitos). Sendo assim, estas não solucionam as deficiências dos atuais paradigmas.

Diferentemente destas iniciativas, o PON se fundamenta nos conceitos dos atuais paradigmas a fim de sanar as deficiências destes em ambientes monoprocessados e mesmo multiprocessados. Essencialmente, estas deficiências estão relacionadas à eficiência de execução das aplicações, ao acoplamento entre as partes das aplicações e às facilidades de programação.

Por meio do *framework* apresentado, por exemplo, o programador pode se beneficiar de maiores facilidades na programação. Por meio deste, o programador faz uso de dois estilos distintos de programação: sobre uma linguagem imperativa intuitiva baseada em interações de objetos ou então sobre uma ferramenta *Wizard* remetendo ao estilo declarativo propriamente dito. Em ambas o programador se preocupa apenas com a composição das *Rules* e a definição dos *FBEs*, uma vez que as conexões entre estes ocorrem automaticamente em *background*.

Um fato interessante é que mesmo o PON tornando a programação mais fácil ao recorrer aos conceitos do paradigma declarativo, das quais aplicações comumente se apresentam menos eficientes do que as baseadas no paradigma imperativo, as aplicações baseadas no PON podem apresentar melhor desempenho do que às baseadas nestes paradigmas, principalmente por causa das evoluções realizadas sobre estes.

Neste âmbito, um dos objetivos da presente dissertação é apresentar estudos de casos a fim de comparar a solução materializada no *framework* segundo os princípios do paradigma proposto em relação às soluções dos atuais paradigmas de programação. Especialmente, estes estudos se referem àqueles paradigmas que inspiraram diretamente a proposta do PON.

CAPÍTULO 5

ESTUDOS COMPARATIVOS

O objetivo deste capítulo é apresentar alguns estudos comparativos práticos e teóricos, em termos de eficiência de execução do paradigma proposto em relação ao Paradigma Imperativo (PI) e o Paradigma Declarativo (PD). Mais precisamente, o PON será comparado com os paradigmas que inspiraram diretamente a sua proposta, ou seja, o Paradigma Orientado a Objetos (POO) e os Sistemas Baseados em Regras (SBR) do Paradigma Lógico (PL), uma vez que se busca representar de forma prática como os conceitos evoluídos podem se apresentar mais eficientes do que os conceitos puros.

Neste âmbito, a seção 5.1 introduz os estudos comparativos, os quais ocorrerão sobre duas plataformas distintas de *hardware*: a plataforma típica dos computadores pessoais e uma plataforma de sistemas embarcados[31].

Assim, a seção 5.2 apresenta os estudos comparativos relativos à plataforma dos computadores pessoais. Esta seção é composta por cinco subseções:

- A subseção 5.2.1 apresenta a aplicação que será usada como base para a realização dos estudos comparativos práticos. Esta consiste na implementação do tradicional jogo de mira ao alvo.
- A subseção 5.2.2 apresenta algumas considerações sobre os estudos comparativos práticos, principalmente sobre o ambiente computacional e configurações de *hardware* sobre os quais os estudos foram realizados.
- A subseção 5.2.3 apresenta um estudo comparativo prático realizado entre as duas versões do *framework* do PON, ou seja, a versão proposta inicialmente por Simão e a versão evoluída desta, a qual foi concebida durante este trabalho de mestrado. Este estudo compara as versões em termos de desempenho a fim de definir a mais apropriada (mais eficiente) para ser usada na execução dos estudos comparativos entre os paradigmas em questão.
- A subseção 5.2.4 apresenta os estudos comparativos práticos realizados entre o PON e o POO do PI.

[31] Basicamente, um sistema embarcado consiste na combinação de *hardware*, *software* e certas vezes de alguns adicionais mecânicos, sendo projetado para executar uma função específica em um *hardware* com características específicas (BARR, 1999).

- A subseção 5.2.5 apresenta os estudos comparativos práticos e teóricos realizados entre o PON e os SBR do PD. Os estudos comparativos práticos ocorreram em relação a dois *shells* de SBR, um compilado e outro interpretado, ambos com a máquina de inferência constituída pelo algoritmo RETE. Os estudos comparativos teóricos foram realizados sobre os algoritmos de inferência apresentados na seção 2.8.3, inclusive o RETE, por meio do cálculo de suas complexidades algorítmicas obtidas pela análise assintótica.

Em outro foco comparativo, a seção 5.3 apresenta os estudos comparativos práticos relativos à plataforma dos sistemas embarcados. Para isto, a subseção 5.3.1 apresenta a aplicação sobre a qual foram realizados os estudos comparativos. Basicamente, a aplicação se refere a um programa que "simula" um controle para sistemas de condicionamento de ar. Neste estudo, o PON é comparado apenas com o PI, uma vez que o PD é inadequado a este tipo de plataforma. Normalmente, o PD demanda maiores capacidades de processamento do que o PI devido ao seu maior nível de abstração.

Por fim, após a apresentação dos estudos comparativos sobre ambas as plataformas, a seção 5.4 conclui sobre os resultados obtidos.

5.1 CONTEXTUALIZAÇÃO

Este trabalho de mestrado propõe o PON como uma solução para as principais deficiências dos atuais paradigmas de programação. Porém, até o presente capítulo, as vantagens do PON foram elucidadas e comparadas apenas descritivamente para com as soluções do PI e do PD. Basicamente, estas comparações teórico-descritivas se deram em torno das deficiências destes paradigmas.

Diferentemente dos capítulos anteriores, o presente capítulo apresenta estudos comparativos em relação aos paradigmas em questão com maior fundamentação experimental do que descritiva. Em sua maioria, este capítulo apresenta estudos práticos que confirmam a eficiência do PON perante os atuais paradigmas para ambientes monoprocessados, uma vez que a versão multiprocessada do PON ainda não se apresenta materializada. Entretanto, além das comparações práticas por medição de tempo de processamento, este ainda apresenta algumas comparações assintóticas entre o mecanismo de notificações e as soluções de inferência apresentadas na seção 2.8.3.

Para a concepção dos estudos práticos, uma alternativa seria realizar os experimentos sobre o sistema de simulação de células de manufatura em ANALYTICE II, uma vez que Simão já aplicou os conceitos do PON sobre estes sistemas (SIMÃO, 2005). Porém, cada um destes sistemas se apresenta demasiadamente complexo para este fim comparativo. Nestes sistemas, as situações a serem comparadas poderiam ser afetadas por interferências de outras partes do código relacionadas ao simulador. Ainda, a complexidade do código poderia impedir que os resultados de desempenho se referissem estritamente a uma situação específica, por exemplo, às redundâncias temporais ou estruturais.

Além disto, haveria a dificuldade em se integrar as linguagens declarativas dos *shells* de SBR ao ANALYTICE II devido à incompatibilidade de sintaxes das linguagens empregadas, uma vez que o ANALYTICE II foi implementado em C++. Desta forma, os estudos comparativos foram executados sobre sistemas de menor complexidade, sendo que (ademais) sistemas menores viabilizam a realização dos estudos sobre a plataforma embarcada.

Isto dito, os estudos comparativos entre os paradigmas em questão sobre as duas plataformas citadas são apresentadas nas seções 5.2 e 5.3, sendo que a seção 5.4 conclui sobre os resultados obtidos.

5.2 ESTUDOS COMPARATIVOS SOBRE A PLATAFORMA DOS COMPUTADORES PESSOAIS

5.2.1 Aplicação Mira ao Alvo

A aplicação intitulada Mira ao Alvo consiste na implementação do tradicional jogo de mira ao alvo (que possui algumas variações). Basicamente, o jogo consiste em um ambiente onde as entidades do tipo mira interagem ativamente com as entidades do tipo alvo. Neste ambiente, ambas as entidades são posicionadas a uma dada distância, sendo que a mira tenta atingir o alvo com o arremesso de um projétil.

A presente aplicação apresenta algumas variações que tornam mais complexa a interação entre as miras e alvos do que no ambiente tradicional. Estas variações são relativas à quantidade de entidades mira e alvo, a definição de novos estados para estas entidades e mesmo a inserção de novas entidades ao ambiente. Estas variações se fazem necessárias para experimentar os paradigmas em diferentes situações. No âmbito desta dissertação, cada

variação que altera a forma pela qual as miras e alvos interagem é chamada de cenário e cada entidade do cenário (i.e. miras e alvos) é chamada de personagem deste cenário.

Particularmente, esta seção expõe com maiores detalhes o cenário que menos varia do tradicional jogo de mira ao alvo, o qual serviu de base para a concepção de todos os cenários sobre os quais foram realizados os estudos comparativos para a plataforma em questão. Estes cenários serão descritos na medida em que forem aplicados na concepção dos estudos comparativos.

De um modo geral, as entidades miras e as entidades alvos são representadas respectivamente por arqueiros e maçãs, sendo que há uma maçã para cada arqueiro. Em termos de implementação, cada arqueiro e maçã é representado por um objeto (i.e. um elemento da base de fatos), os quais interagem de acordo com a validação de expressões causais pertinentes. Estas expressões causais fazem menção aos atributos/estados destes objetos.

Neste contexto, cada arqueiro e cada maçã recebem um identificador numérico, sendo que um arqueiro somente pode flechar uma maçã que apresente o identificador numérico correspondente ao seu. Ainda, os arqueiros também apresentam um atributo que denota os seus estados de pronto (i.e. *status*) para agir sobre o cenário (i.e. flechar a respectiva maçã).

Em relação às maçãs, estas também apresentam um atributo que denota os seus estados de pronto (i.e. *status*). Ainda, estas apresentam um atributo que explicita se a mesma já foi perfurada por uma flecha ou ainda não (i.e. *isCrossed*). Também, as maçãs apresentam um atributo que se refere a sua coloração (i.e. *color*), uma vez que as maçãs podem se apresentar em duas diferentes cores: vermelha ou verde. Com esta fartura de atributos, as maçãs representam um papel importante na concepção de cenários devido à possibilidade de variar os seus estados na composição de diferentes expressões causais.

Com a intenção de melhor elucidar esta aplicação, a Figura 69 ilustra a interação entre os personagens arqueiros e maçãs. Nesta, os arqueiros e maçãs estão organizados em fila de acordo com os seus números identificadores, o que assegura que um arqueiro está posicionado absolutamente à frente da sua respectiva maçã.

Neste cenário, cada arqueiro somente pode interagir com a sua respectiva maçã após a constatação de três premissas: (a) se a cor da maçã que está posicionada diretamente em sua frente é vermelha, (b) se a maçã que está posicionada diretamente em sua frente está pronta para ser atingida e ainda, (c) se ela é identificada pelo seu número correspondente. Se as três condições forem satisfeitas, o arqueiro está liberado para atingir a respectiva maçã com a

projeção de sua flecha. Desta forma, percebe-se que para cada par de arqueiros e maçãs deve haver uma expressão causal para comparar os seus estados.

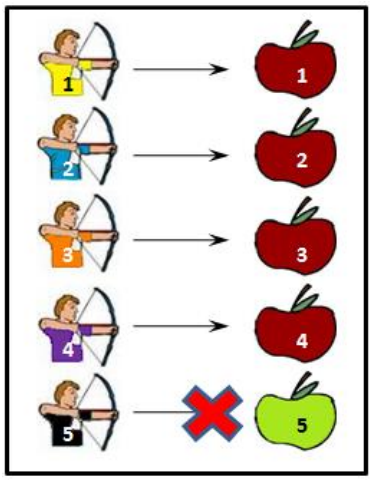


Figura 69: Cenário similar ao tradicional jogo do mira ao alvo

Conforme ilustra a Figura 69, os arqueiros e as maças identificados pelos números 1, 2, 3 e 4 confirmam a veracidade das três premissas e esses flecham as respectivas maçãs. Porém, o mesmo não ocorreu com o arqueiro identificado pelo número 5, uma vez que a sua respectiva maçã é verde, o que não satisfaz uma das premissas, i.e. premissa (a).

No entanto, este mesmo arqueiro terá outras oportunidades para interagir com a sua respectiva maçã, uma vez que os cenários em geral são compostos por várias iterações. Uma iteração consiste nas avaliações das premissas pertinentes para cada arqueiro do cenário, sendo estas falsas ou verdadeiras. Durante uma iteração, quando as condições são satisfeitas, as maçãs são sempre perfuradas, não admitindo falhas nas ações dos arqueiros. Desta forma, ao final de uma iteração, as maçãs perfuradas são substituídas por novas maçãs e todo o processo de interação entre os personagens é reinicializado.

De um modo geral, os experimentos realizados sobre os cenários podem variar os estados dos personagens para que apenas uma porcentagem pré-definida de expressões causais seja satisfeita a fim de alterar o impacto de redundâncias nas instâncias imperativas e aumentar a quantidade de notificações nas instâncias do PON. Na maioria dos cenários, esta porcentagem cresce na escala de 10% até atingir a totalidade das expressões causais. Assim, um experimento pode ser seccionado em várias escalas, chamadas de fases de um experimento. Em cada fase de um experimento, o cenário é executado pela mesma quantidade de iterações para verificar as variações de comportamento entre as instâncias comparadas.

Em suma, esta aplicação oferece um escopo ideal para realizar os estudos comparativos entre os paradigmas em questão. Este permite demonstrar sucintamente como os paradigmas se comportam, permitindo discutir sobre as suas qualidades e deficiências.

5.2.2 Considerações sobre os estudos comparativos

Para que os estudos comparativos práticos apresentassem resultados confiáveis, as aplicações derivadas do jogo Mira ao Alvo usadas na comparação da eficiência entre os paradigmas em questão foram executadas sobre o ambiente MS-DOS (*Microsoft - Disk Operating System*), uma vez que este consiste em um ambiente livre de preempções. As aplicações foram executadas sobre a versão 7.0 deste ambiente, a qual está inclusa juntamente com o sistema operacional Windows 98.

No entanto, a execução de aplicações sobre o ambiente MS-DOS puro causa limitações de acesso aos recursos computacionais devido à execução natural deste ambiente no Modo-Real dos processadores. Neste modo operacional, o processador apenas reconhece 1 MB de memória RAM sendo que apenas 640 Kbytes destes podem ser usados pelas aplicações, os quais ainda devem ser compartilhados com o próprio ambiente MS-DOS e os *drivers* do sistema. Os 384 Kbytes restantes são usados para armazenar uma cópia do BIOS (*Basic Input/Output System*) do sistema.

Desta forma, para que as aplicações a serem comparadas pudessem executar apropriadamente neste ambiente, se fez necessário que as aplicações apresentassem acesso "ilimitado" aos recursos computacionais oferecidos no Modo-Protegido, principalmente aos recursos de memória. Para isto, os códigos-fonte das aplicações comparadas foram compilados para executarem em Modo-Protegido com o compilador DJGPP (*DJ's GNU Programming Platform*[32]), que é uma versão do compilador GCC (*GNU Compiler Collection*) para ambientes MS-DOS. Com isto, as aplicações executam no MS-DOS e acessam os recursos do Modo-Protegido por meio de uma interface chamada DPMI (*DOS Protected Mode Interface*). Esta interface permite que o processador acesse toda a memória disponível no computador (DELORIE, 2003).

Por fim, as aplicações comparadas foram executadas sobre um computador com processador Pentium 4 de 1,8 GHz e 512 MB de memória e o tempo de execução de cada aplicação foi medido em milissegundos.

---

[32] A título de curiosidade, a sigla DJ refere ao criador do compilador DJGPP DJ Delorie e GNU se refere ao um projeto GNU de *software* livre, cuja definição recursiva é *GNU Is Not Unix.*

5.2.3 Estudo comparativo entre as versões do *framework* do PON

Antes de realizar os estudos comparativos em termos de desempenho entre o PON e os atuais paradigmas, se faz necessário comparar o desempenho entre as duas versões do *framework* do PON. Estas se referem à versão proposta inicialmente por Simão (versão antiga) e a evolução desta que constitui na versão descrita no capítulo 4 desta dissertação de mestrado (nova versão). Deste modo, a versão que apresentar melhor desempenho será usada na concepção dos estudos comparativos a fim de representar os conceitos do PON.

Para realizar o estudo comparativo entre as duas versões do *framework*, adotou-se o mesmo cenário apresentado na seção 5.2.1. Este cenário é usado na implementação de dois experimentos entre as versões, os quais se diferem na quantidade de iterações em que os arqueiros e maçãs se inter-relacionam. Nestes experimentos, cem arqueiros interagem com a mesma quantidade de maçãs por dez mil e cem mil iterações, os quais executam sobre o ambiente livre de preempções.

Nestes experimentos, os arqueiros são representados pela classe *Archer* e as maçãs pela classe *Apple*, ambas derivadas da classe *FBE*. Os cem arqueiros e maçãs são agrupados em estruturas de dados do tipo vetor respectivamente chamadas de *ArcherList* e *AppleList*. Estas estruturas são percorridas em um laço de repetição para a criação das cem expressões causais pertinentes, as quais são representadas por regras lógicas (*Rules*[33]) no contexto dos *frameworks*. Deste modo, a Figura 70 apresenta a implementação das *Rules* na versão antiga do *framework* (linha 1) e na versão nova do *framework* (linha 45), as quais se diferem em suas representatividades.

Nestes experimentos, dentre as três *Premises* de cada *Rule*, duas sempre apresentam estado verdadeiro e uma varia de estado em cada iteração. Esta *Premise* se refere à avaliação da cor da *Apple* (i.e. linhas 9 e linha 48), a qual se torna falsa quando a cor da *Apple* referenciada é verde e se torna verdadeira quando a cor da *Apple* referenciada é vermelha. Esta se torna verdadeira no início de cada iteração e se torna falsa pela própria execução da *Rule* que a contem, a qual altera a coloração da *Apple* referenciada para verde. As outras duas *Premises* se referem aos estados de pronto (i.e. *status*) dos respectivos *Archers* (linhas 13 e 55) e *Apples* (linhas 11 e 53).

[33] Uma *Rule* é chamada de AgenteRegra na versão antiga do *framework*.

```
/***********************************************
***************   VERSÃO ANTIGA   ***************
***********************************************/

...
for ( int i = 0; i < 100; i++)
{
        //Premises
        prAppleColorRed = new AgentePremissa(appleList->at(i)->atAppleColor, True);
        prAppleColorRed->conectaPredBoleano(&comparaBoleanos);
        prAppleStatusTrue = new AgentePremissa(appleList->at(i)->atAppleStatus, True);
        prAppleStatusTrue->conectaPredBoleano(&comparaBoleanos);
        prArcherStatusTrue = new AgentePremissa(archerList->at(i)->atArcherStatus, True);
        prArcherStatusTrue->conectaPredBoleano(&comparaBoleanos);

        //Condition
        AgenteCondicao *cdFireApple;
        cdFireApple = new AgenteCondicao();
        cdFireApple->conectaAgentePremissa(prAppleColorRed);
        cdFireApple->conectaAgentePremissa(prAppleStatusTrue);
        cdFireApple->conectaAgentePremissa(prArcherStatusTrue);

        //Action
        AgenteAcao *acFireApple;
        acFireApple = new AgenteAcao();
        acFireApple->conectaAgenteOrdem(appleList->at(i)->mtChangeToGreen);

        //Method
        mtChangeToGreen = new AgenteMetodoGen<Apple>(&Apple::ChangeToGreen);

        //Instigation
        itChangeToGreen = new AgenteOrdem(mtChangeToGreen);

        //Rule
        AgenteRegra *rlFireApple;
        rlFireApple = new AgenteRegra(cdFireApple, acFireApple);
    }
...

/***********************************************
***************   VERSÃO NOVA   ***************
***********************************************/

...
for ( int i = 0; i < 100; i++)
{
  Rule* rlFireApple = new Rule(Condition::CONJUNCTION);
  rlFireApple->addPremise(appleList->at(i)->atAppleColor,    Boolean::TRUE, Premise::EQUAL);
  rlFireApple->addPremise(appleList->at(i)->atAppleStatus,   Boolean::TRUE, Premise::EQUAL);
  rlFireApple->addPremise(archerList->at(i)->atArcherStatus, Boolean::TRUE, Premise::EQUAL);
  rlFireApple->addInstigation(appleList->at(i)->atAppleColor, false);
  rlFireApple->end();
}
...
```

Figura 70: Representação das *Rules* na versão antiga e nova do *framework* do PON

Desta forma, para verificar o comportamento das duas versões em várias situações, a quantidade de *Rules* aprovadas varia em cada fase do experimento na escala crescente de 10% pela variação da quantidade de *Apples* com coloração vermelha. Na medida em que a quantidade de *Rules* aprovadas aumenta durante as fases do experimento, pode-se verificar o comportamento das versões em relação ao aumento das notificações, uma vez que as mudanças de estados se tornarão mais freqüentes.

No primeiro experimento, ambas as versões foram comparadas sobre as mesmas condições por 10.000 iterações. Os resultados deste experimento estão ilustrados na Figura 71.

Exp. Verdad.	Antigo (ms)	Novo (ms)
10%	297	234
20%	578	485
30%	843	734
40%	1125	969
50%	1422	1234
60%	1720	1468
70%	2031	1719
80%	2297	1935
90%	2594	2235
100%	2906	2406

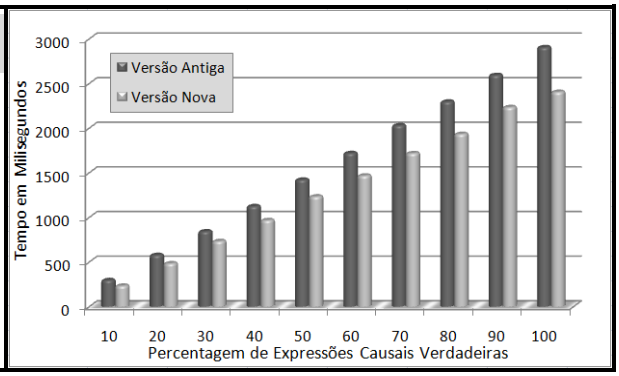


Figura 71: Resultados referentes ao primeiro experimento entre as versões do *framework* do PON

Conforme explicita o gráfico de resultados, neste experimento, ambas as versões quase que se equivalem em termos de desempenho. Todavia, mesmo com pouca diferença, a nova versão de *framework* ainda se apresenta mais eficiente, mesmo apresentando maior complexidade pela inclusão de novas funcionalidades em relação à versão antiga. Porém, as melhorias realizadas para melhorar a eficiência da nova versão do *framework* são suficientes para suportar as novas funcionalidades e manter o bom desempenho do *framework*.

No entanto, a nova versão pode se apresentar ainda mais eficiente do que a versão antiga quando a quantidade de iterações aumenta. Para confirmar este fato, outro experimento foi realizado com o intuito de aumentar a quantidade de *Rules* aprovadas seguindo os mesmos critérios do primeiro experimento. Neste novo experimento, os *Archers* interagem com as *Apples* por 100.000 iterações. Os resultados deste experimento estão expressos na Figura 72.

Exp. Verdad.	Antigo (ms)	Novo (ms)
10%	2828	2434
20%	5531	4906
30%	8281	7344
40%	12297	9782
50%	16375	12234
60%	22047	14718
70%	30781	17188
80%	37860	19578
90%	50937	22078
100%	64109	24125

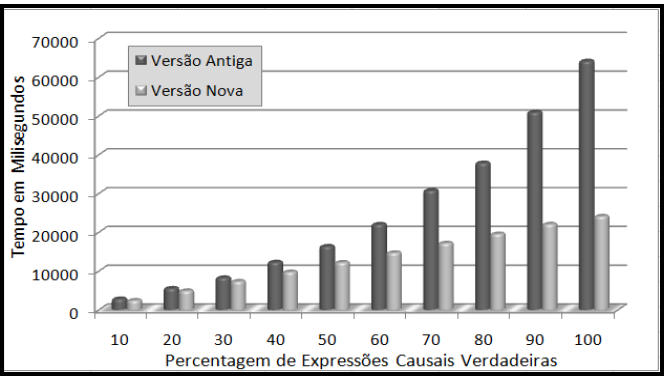


Figura 72: Resultados referentes ao segundo experimento entre as versões do *framework* do PON

Conforme os resultados explicitados no gráfico, a diferença de desempenho entre as duas versões se apresenta maior neste experimento, mesmo se tratando da execução das mesmas *Rules*. No gráfico referente ao primeiro experimento, as diferenças de desempenho entre as duas versões eram praticamente constantes. Neste gráfico, as diferenças até se

mantêm constantes quando 30% das regras são aprovadas, mas quando a porcentagem destas aprovações aumenta, o tempo de execução da versão antiga se eleva em maior escala do que o tempo da nova versão, o qual ainda se mantém em crescimento constante.

Este aumento no tempo de execução da versão antiga se deve a problemas relacionados à alocação dinâmica de memória, o que causa a extrapolação da memória principal para que dados sejam salvos temporariamente em memória virtual. Tal problema foi solucionado na nova versão do *framework* ao substituir a estrutura *union* usada para guardar os estados dos *Attributes*, a qual era instanciada a cada atribuição de estados, por tipos de dados representados por subclasses de *Attribute* (i.e. *Boolean*, *Integer*, *Double*, *Char* e *String*). Estes apresentam variáveis que guardam os valores sem a necessidade de alocação dinâmica de memória.

Como apresenta o gráfico corrente, quando as cem regras apresentam estado lógico verdadeiro, o desempenho da versão antiga corresponde aproximadamente a três vezes o desempenho da nova versão. Obviamente, se a quantidade de iterações ainda fosse aumentada, o que demandaria mais memória, esta diferença seria ainda maior. Desta forma, a nova versão potencialmente se mostra mais eficiente do que a versão anterior, sendo por isto adotada para a comparação com os demais paradigmas.

5.2.4 Paradigma Orientado a Notificação versus Paradigma Imperativo

Esta seção compara o PON com o PI, mais precisamente com o POO, em termos de processamento com o intuito de demonstrar quanto o PON é mais eficiente ao solucionar os problemas das redundâncias temporais e estruturais. As comparações se dão sobre dois cenários do jogo do Mira ao Alvo, sendo que o primeiro corresponde ao cenário já descrito na seção 5.2.1 e o segundo apresenta uma pequena variação do mesmo cenário apresentado.

Os cenários são implementados na linguagem C++ para ambos os paradigmas e executados por uma quantidade pré-estabelecida de iterações sobre o ambiente MS-DOS. Desta forma, o primeiro cenário é usado para descrever sobre os experimentos relacionados às redundâncias temporais enquanto que o segundo cenário é usado para descrever sobre os experimentos realizados sobre as redundâncias estruturais.

5.2.4.1 Primeiro cenário

No PI, a redundância temporal ocorre quando ao menos um atributo avaliado por uma premissa apresenta estado invariável e continua neste estado por mais de uma avaliação. Isto resulta em uma ou mais premissas ou uma ou mais expressões causais sendo reavaliadas sem alteração em seu estado lógico em relação à última avaliação.

Com o intuito de minimizar os efeitos das redundâncias temporais, as linguagens imperativas modernas apresentam alguns melhoramentos. Em C++ e Java, nem todas as premissas de uma expressão causal são avaliadas e reavaliadas. Nestas linguagens, se uma expressão causal apresentar conjunções entre duas ou mais premissas, quando a primeira premissa é avaliada e resulta em um estado lógico falso, subentende-se que é desnecessário avaliar as demais premissas, pois certamente o estado lógico da expressão causal será falso. Da mesma forma ocorre com as disjunções, se a primeira premissa apresenta estado lógico verdadeiro, as demais premissas não são avaliadas, subentendendo-se que a respectiva expressão causal apresenta estado lógico verdadeiro.

No entanto, para realizar estas constatações é necessário que a linguagem faça uso, implicitamente, de uma expressão causal adicional, a qual é avaliada pelo mecanismo imperativo a cada avaliação de uma premissa a fim de verificar a necessidade de prosseguir nas avaliações. Mesmo assim, esta técnica poupa significativamente os recursos de processamento principalmente em expressões causais compostas por muitas premissas. Os experimentos realizados foram implementados com respeito a esta qualidade das linguagens imperativas modernas, declarando por primeiro as premissas que mais variam de estados a fim de evitar as avaliações das premissas invariantes, quando não necessário.

Os experimentos realizados sobre o presente cenário refletem as mesmas condições do segundo experimento realizado sobre as duas versões do *framework* do PON, o qual foi executado por 100.000 iterações. Desta forma, as expressões causais apresentam praticamente a mesma semântica, como pode ser constatado na Figura 73, com exceção ao tipo de dados do *Attribute atColor*[34]. Nesta, o trecho de código entre as linhas 2 e 10 representa as expressões causais imperativas enquanto que o trecho de código entre as linhas 14 e 22 representa as

[34] Na versão da regra *rlFireApple* apresentada na Figura 73, o *Attribute atColor* se refere ao tipo de dados String enquanto que na versão apresentada na Figura 70, este se refere ao tipo de dados *Boolean* a fim de manter compatibilidade com a versão antiga do PON, uma vez que esta versão antiga não apresentava suporte aos demais tipos de dados.

Rules em PON[35]. Neste experimento, as 100 expressões causais imperativas são avaliadas pelo percorrer de um laço de repetição, onde os objetos que comportam os estados avaliados estão alocados em estruturas de dados do tipo vetor.

```
...
for(int j=0; j < 100; j++)
{
  if( (appleList->at(j)->color   == "RED") &&
      (appleList->at(j)->status  == true)  &&
      (archerList->at(j)->status == true))
  {
    appleList->at(j)->color = "GREEN";
  }
}
...


...
for (int j=0; j < 100; j++)
{
  Rule* rlFireApple = new Rule(Condition::CONJUNCTION);
  rlFireApple->addPremise(appleList->at(j)->atAppleColor,    "RED",         Premise::EQUAL);
  rlFireApple->addPremise(appleList->at(j)->atAppleStatus,   Boolean::TRUE, Premise::EQUAL);
  rlFireApple->addPremise(archerList->at(j)->atArcherStatus, Boolean::TRUE, Premise::EQUAL);
  rlFireApple->addInstigation(appleList->at(j)->atAppleColor, "GREEN");
  rlFireApple->end();
}
...
```

Figura 73: Expressões causais referentes ao primeiro cenário do estudo comparativo com o PI

Particularmente, este cenário permite identificar facilmente as redundâncias temporais no PI. Por exemplo, ao considerar que nenhuma expressão causal é satisfeita em cada iteração, a natureza do imperativo faz com que estas expressões sejam avaliadas e reavaliadas indeterminadamente até que um critério de parada seja alcançado. Este processo de avaliação consome recursos de processamento e não contribui para a execução de instruções relevantes.

Diferentemente do PI, um programa implementado com o PON não consome recursos computacionais enquanto os dados não apresentam alteração em seus estados[36]. Assim, enquanto o PI executa desnecessariamente avaliando e reavaliando expressões causais que não apresentam variação de seus estados lógicos, o mecanismo de notificações reconhece a invariabilidade dos estados e impede que as expressões relacionadas sejam avaliadas.

Para confirmar este fato, realizou-se um experimento prático sobre o corrente cenário. Neste experimento, nenhuma das 100 expressões causais apresenta estado lógico verdadeiro devido a não satisfação da primeira premissa das expressões apresentadas na Figura 73,

[35] Os trechos principais dos programas implementados com o PI e com o PON são apresentados na subseção D.1.1.1 do Apêndice D.

[36] No entanto, de forma opcional, o PON oferece o conceito de re-notificação que permite executar uma regra mais de uma vez mesmo que os estados referenciados não apresentem variação (vide 4.4.1.2).

fazendo com que todas as maçãs apresentem coloração verde. O experimento foi executado por 50.000 e 100.000 iterações e os resultados estão apresentados pelo gráfico na Figura 74.

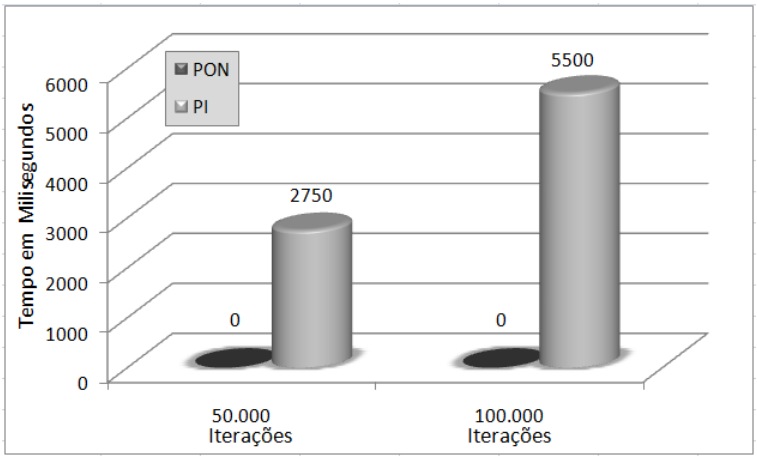


Figura 74: Resultados do experimento referente ao primeiro cenário do estudo comparativo com o PI com nenhuma expressão causal verdadeira

Conforme os resultados obtidos, este experimento confirmou a ineficiência do mecanismo imperativo e às vantagens do mecanismo de notificações. Também, este experimento permitiu perceber que a natureza de execução do imperativo é realizar buscas seqüenciais a fim de encontrar qualquer expressão causal que seja satisfeita pelos estados correntes da aplicação.

Deste modo, considerando uma centena de expressões causais dispostas seqüencialmente (sem o uso de expressões causais aninhadas) e apenas uma destas satisfeita pelos estados da aplicação, o mecanismo de execução do imperativo percorrerá e avaliará toda esta centena de expressões causais a fim de executar os comandos do escopo da expressão causal pertinente.

Em uma aplicação real, as expressões causais que precedem ou sucedem a expressão causal pertinente normalmente se referem a estados diferentes daqueles referenciados pela expressão causal em questão. Mesmo assim, estas precisam ser avaliadas por estarem inseridas no fluxo de execução seqüencial do imperativo, o que retarda o momento da avaliação da expressão causal pertinente. Este retardo no imperativo pode ser comparado analogamente a um sistema de encanamento, conforme ilustra o esquema à esquerda na Figura 75.

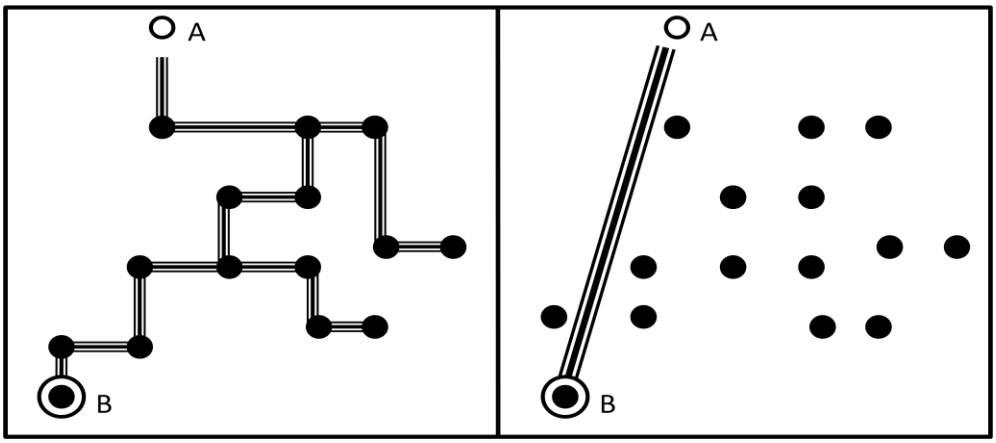


Figura 75: Analogia dos mecanismos de execução do imperativo e de notificação a um sistema de encanamento

Neste esquema, a bola branca é inserida na entrada do encanamento (i.e. ponto A) e percorre as bifurcações (pontos pretos) em direção da saída, representada pelo ponto preto circundado por um anel branco (ponto B). Analogamente, a bola branca representa a mudança de um estado de um atributo e cada ponto preto posicionado no vértice de uma bifurcação do encanamento representa uma expressão causal, sendo que a expressão causal que realmente referencia o estado modificado é representada pelo ponto preto envolto do anel branco.

Segundo o esquema apresentado, considerando a mudança de um estado (ponto A) que afeta apenas o estado lógico de uma expressão causal (ponto B), o mecanismo de execução do imperativo deve percorrer todas as expressões causais não afetadas por esta mudança (pontos pretos) para enfim avaliar a expressão causal pertinente.

No lado direito da mesma figura, está a representação do mecanismo de notificação para este mesmo caso. Nesta, com a mudança de um estado (ponto A) a expressão causal pertinente (ponto B) é notificada pontualmente, sem a interferência das demais expressões (pontos pretos).

Outrossim, para avaliar em termos de desempenho estas duas soluções foi realizado mais um experimento sobre o corrente cenário do mira ao alvo. Neste experimento, considera-se apenas uma expressão causal verdadeira em cada iteração entre as 100 expressões causais do cenário. O experimento foi executado por 50.000 e 100.000 iterações e os resultados estão expressos no gráfico apresentado na Figura 76.

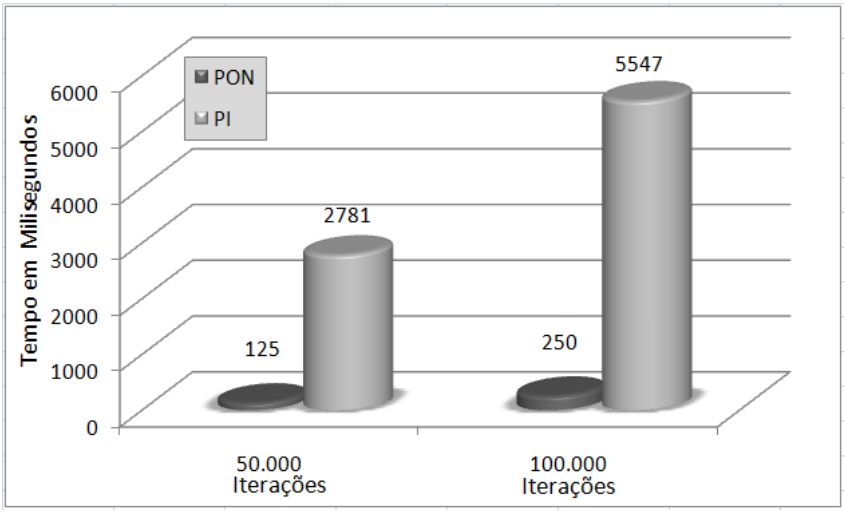


Figura 76: Resultados do experimento referente ao primeiro cenário do estudo comparativo com o PI com uma expressão causal verdadeira

Conforme ilustra o gráfico dos resultados, o mecanismo de notificações do PON poupa expressivamente os recursos de processamento em relação ao PI. Este evita buscas por expressões causais ao notificar pontualmente a expressão causal realmente afetada pela mudança de estado e imediatamente após esta mudança ter ocorrido.

Este comportamento reativo dos dados em relação às expressões causais é adequada para melhorar a eficiência de execução dos sistemas computacionais atuais e mesmo futuros. Segundo Friedman-Hill (2003), Forgy (1984) e Giarratano e Riley (1993), em um sistema computacional imperativo, a maioria das expressões causais é avaliada desnecessariamente uma vez que somente poucas delas são afetadas pela mudança de estados dos dados durante uma iteração do mecanismo imperativo. Na verdade, eles afirmam que poucos estados são alterados em cada iteração.

Segundo Friedman-Hill (2003), de acordo com a sua experiência, no máximo 20% das expressões causais podem ser afetadas em uma iteração. Segundo Forgy (1984), esta porcentagem se apresenta bem menor de acordo com cálculos e medidas realizadas sobre algumas aplicações computacionais, o que permitiu afirmar que menos de 1% das expressões causais têm seus estados alterados em cada iteração.

Estes fatos confirmam a desnecessidade de avaliar seqüencialmente e constantemente todas as expressões causais, sendo que somente as que são afetadas pelas mudanças dos dados devem ser avaliadas. No entanto, estes fatos não se aplicam absolutamente a todos os sistemas computacionais. Mesmo que raramente, pode haver sistemas em que, nos piores dos casos, a maioria ou todas as expressões causais podem ser afetadas pela mudança de um estado ou mesmo por vários estados, evitando ao máximo qualquer tipo de redundância temporal desnecessária.

Desta forma, mais um experimento foi realizado sobre o presente cenário a fim de comparar ambos os paradigmas em relação a diferentes graus de redundância temporal. Neste experimento as expressões causais afetadas pelas mudanças de estados aumentam em quantidade a cada fase do experimento a fim de reduzir os efeitos das redundâncias temporais. Os resultados deste experimento estão expressos na Figura 77.

Exp. Verdad.	PON (ms)	PI (ms)
10%	2434	5953
20%	4906	6375
25%	6141	6578
27%	6609	6687
30%	7344	6781
50%	12234	7672
75%	18437	8703
100%	24125	9672

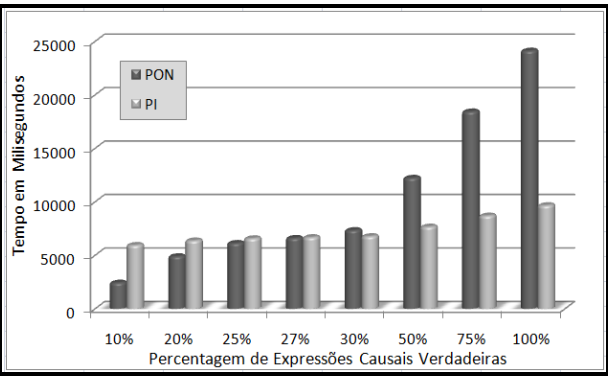


Figura 77: Resultados do experimento referente ao primeiro cenário do estudo comparativo com o PI com variações porcentuais de expressões causais verdadeiras

Como explicitado pelo gráfico, para este cenário, a adoção do mecanismo de notificações é preferível em aplicações em que 27% das expressões causais são afetadas por mudanças de dados. Entretanto, se esta porcentagem for confrontada com as afirmações de Friedman-Hill (2003) e Forgy (1984), o mecanismo de notificações do PON se apresenta mais eficiente do que o mecanismo de avaliações do PI, principalmente quando há altas quantidades de expressões causais.

No entanto, em relação às porcentagens maiores, o mecanismo eficaz e preciso de notificações se mostra menos eficiente do que o mecanismo imperativo. Estes resultados se devem à simplicidade pela qual as avaliações das expressões causais ocorrem no imperativo, auxiliados pela redução constante das redundâncias temporais em cada fase do experimento.

Esta simplicidade pode ser claramente notada pela maneira que as expressões causais são representadas. No PI, as simples estruturas das expressões causais manipulam somente dois objetos (das classes *Archer* e *Apple*) enquanto que a estrutura de uma expressão causal em PON (i.e. *Rules*) manipula os mesmos dois objetos e ainda os objetos colaboradores do mecanismo de notificação.

Na atual materialização do PON, em termos de instruções em linguagem Assembly, as expressões causais deste são certamente mais complexas e compostas por um número maior de instruções a serem processadas do que as expressões causais *if-then* do PI. Ademais, o PON está implementado como uma camada de abstração sobre a linguagem C++, o que afeta o seu desempenho. Assim sendo, estes resultados poderiam ser melhorados por uma série de

otimizações que passam certamente pela construção de um compilador conforme será discutido brevemente.

5.2.4.2 Segundo cenário

O segundo cenário apresenta os problemas das redundâncias temporais e estruturais, mas as redundâncias estruturais são evidenciadas. A redundância estrutural ocorre quando a atualização de um único estado pode afetar duas ou mais expressões causais. Desta forma, estas expressões causais apresentam uma premissa em comum, a qual é avaliada repetidamente em cada uma destas. Este tipo de redundância é resolvido em PON pelo compartilhamento de um objeto *Premise* pelas *Conditions* pertinentes.

Para avaliar a eficiência desta solução foi realizado um experimento sobre mais uma variação do jogo do Mira ao Alvo. Esta variação consiste no segundo cenário, o qual apresenta similaridades ao cenário anterior. A diferença entre estes está no acréscimo de mais um personagem ao presente cenário, uma arma de fogo, a qual é representada por um objeto da classe *Gun*. A arma de fogo tem a função de sinalizar com o seu disparo o início de uma iteração, permitindo que os arqueiros interajam com as respectivas maçãs. Esta interação é apresentada na Figura 78.

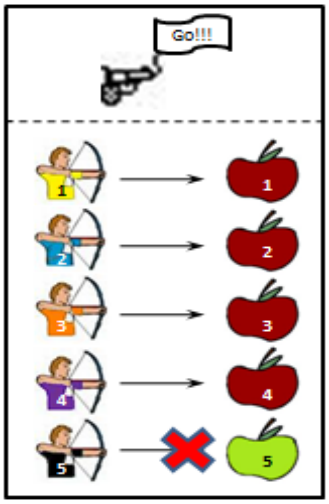


Figura 78: Representação do segundo cenário do estudo comparativo com o Paradigma Imperativo

Neste cenário, a arma constitui um elemento comum a todos os arqueiros, pois estes devem "ouvir" a sinalização emitida pela arma para poderem agir sobre as maçãs. Assim, em cada iteração, o estado da arma deve ser avaliado para cada arqueiro, este fato evidencia a

necessidade de mais uma premissa às expressões causais apresentadas no primeiro cenário. As expressões causais que incorporam premissa adicional estão apresentadas na Figura 79[37].

```
 1 ...
 2 for(int j = 0; j < 100; j++)
 3 {
 4   if( (appleList->at(j)->color   == "RED") &&
 5       (appleList->at(j)->status  == true)  &&
 6       (archerList->at(j)->status == true)  &&
 7       (gun->status == true))
 8   {
 9     appleList->at(j)->isCrossed = true;
10   }
11 }
12 ...
13
14 ...
15 Premise* prGunIsTrue = new Premise(gun->atGunStatus, Boolean::TRUE, Premise::EQUAL);
16
17 for ( int j = 0; j < 100; j++)
18 {
19   Rule* rlFireApple = new Rule(Condition::CONJUNCTION);
20   rlFireApple->addPremise(appleList->at(j)->atAppleColor,    "RED",         Premise::EQUAL);
21   rlFireApple->addPremise(appleList->at(j)->atAppleStatus,   Boolean::TRUE, Premise::EQUAL);
22   rlFireApple->addPremise(archerList->at(j)->atArcherStatus, Boolean::TRUE, Premise::EQUAL);
23   rlFireApple->addPremise(prGunIsTrue);
24   rlFireApple->addInstigation(appleList->at(j)->atAppleIsCrossed, true);
25   rlFireApple->end();
26 }
27 ...
```

Figura 79: Expressões causais referentes ao segundo cenário do estudo comparativo com o PI

Considerando os trechos de código da figura, observa-se que o PI reavalia o estado da arma (*Gun*) em cada expressão causal (linha 7) enquanto que o PON compartilha entre as expressões causais a mesma *Premise* por meio do objeto *prGunIsTrue* (linha 23). No caso deste exemplo, o código imperativo certamente poderia ser implementado de maneira mais otimizada. Por exemplo, tendo em vista a seqüencialidade e agrupamento dos *ifs*, poderia haver um *if* superior que avaliaria o estado da *Gun* e aninharia os demais *ifs*. Entretanto, o experimento é representativo e visa avaliar uma realidade prática que é o comportamento dos *ifs* não tão agrupados onde tal otimização seria inviável ou pelo menos custosa em termos de esforço de programação.

Entre os atributos que afetam a avaliação destas expressões, considera-se que apenas o objeto *Gun* tem o estado de seu atributo atualizado em cada iteração, uma vez que a variação do estado do atributo *isCrossed* (no imperativo) ou *atAppleIsCrossed* (no PON) pela execução de uma expressão causal não afeta o estado lógico da mesma. O objeto *Gun* tem o seu estado atualizado por meio de uma re-atribuição de estado, uma vez que a arma é disparada em cada iteração. Em PON, esta re-atribuição ocorre com a aplicação da funcionalidade de re-notificação ao *Attribute atGunStatus* no início de cada iteração (vide subseção 4.4.1.2).

[37] Os trechos principais dos programas implementados com o PI e com o PON são apresentados na subseção D.1.1.2 do Apêndice D.

Desta forma, as expressões causais do presente cenário se diferem das expressões causais do cenário anterior, no qual a execução de uma expressão causal modificava o estado do atributo *color* (no imperativo) ou *atAppleColor* (no PON) afetando o seu próprio estado lógico. Em PON, a execução de uma *Rule* gerava um fluxo de notificações que alterava o estado lógico de sua própria *Condition*.

Deste modo, a quantidade de notificações no presente cenário é reduzida em relação ao cenário anterior, uma vez que se deseja estritamente avaliar a solução para a redundância estrutural pelo compartilhamento da respectiva *Premise* e mesmo da redundância temporal ao guardar informações sobre estados já avaliados.

Ainda, dentre as fases do experimento, as maçãs mudam sua coloração para vermelha em escala crescente de 10% a fim de aumentar a quantidade de regras aprovadas por iteração e assim aumentar a quantidades de premissas avaliadas pelo mecanismo imperativo. Consecutivamente, com o aumento das premissas avaliadas, aumentam os efeitos da redundância temporal, mesmo que as avaliações sejam necessárias.

Desta forma, a redundância temporal ocorre quando as demais premissas são reavaliadas sem haver mudança de estados e a redundância estrutural ocorre quando o estado da arma é reavaliado pelas premissas, mas somente quando a expressão causal é aprovada porque a premissa em comum foi declarada por último na estrutura causal.

Para realizar este experimento foi considerado um grupo de 100 arqueiros e 100 maçãs que se orientam pelo disparo da arma de fogo por 100.000 iterações. Os resultados deste experimento estão apresentados na Figura 80.

Exp. Verdad.	PON (ms)	PI (ms)
10%	2297	2641
20%	2609	3250
30%	2937	3906
40%	3250	4500
50%	3578	5079
60%	3907	5719
70%	4203	6344
80%	4500	6953
90%	4828	7563
100%	5094	8141

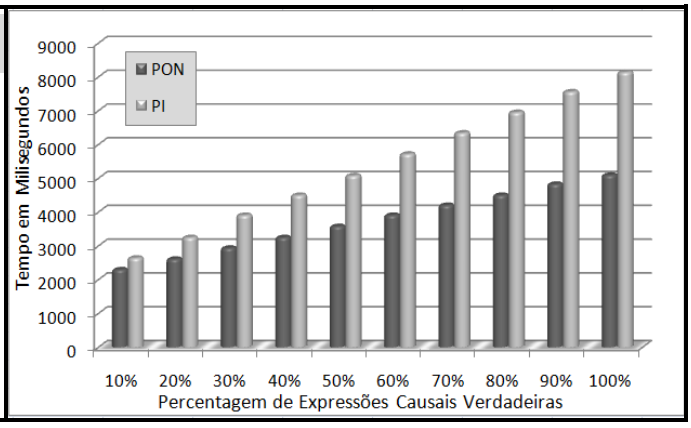


Figura 80: Resultados do experimento referente ao segundo cenário do estudo comparativo com o Paradigma Imperativo com variações porcentuais de expressões causais verdadeiras

Conforme explicita o gráfico em relação a este cenário, o PON apresenta melhor desempenho quando soluciona ambas as redundâncias. Como se pode constatar, a aplicação do PON sobre este cenário se apresentou mais favorável do que sobre o anterior devido à

capacidade do mecanismo de notificações guardar lembranças dos estados lógicos já avaliados e pelo compartilhamento do estado lógico da respectiva *Premise* por todas as *Conditions* pertinentes. Estas capacidades evitam os problemas de ambas as redundâncias.

Diferentemente, o PI não guarda lembrança dos estados lógicos já avaliados e por isto deve reavaliá-los em cada iteração. Esta situação piora quando a maioria das premissas é satisfeita, uma vez que estas devem ser necessariamente reavaliadas a fim de aprovar a expressão causal, particularmente a premissa em comum a todas as expressões causais (declarada por último). Isto pode ser constatado pelo aumento das diferenças de desempenho entre os paradigmas na medida em que cresce a quantidade de expressões causais verdadeiras.

5.2.4.3 Reflexões sobre as implementações

Nos experimentos realizados, os conceitos do PON foram comparados com uma implementação não tão otimizada do PI, na qual as expressões causais acessam os atributos dos objetos a serem avaliados de forma indireta, ou seja, por meio do percorrer de uma estrutura de dados. Esta estrutura é o *vector* (vetor) da biblioteca STL (*Standard Template Library*) do C++ (WILSON, 2007). O uso de estruturas de dados para comportar os objetos a serem avaliados prejudica excessivamente o desempenho das avaliações causais devido à necessidade de computar o endereço do respectivo objeto de acordo com o endereço do vetor.

Desta forma, para que os experimentos viessem a ocorrer sobre uma implementação otimizada do imperativo, as expressões causais dos respectivos cenários deveriam acessar diretamente cada atributo a ser avaliado. Para isto, seria preciso declarar 100 referências para a classe *Apple* e 100 referências para a classe *Archer* e instanciar manualmente cada referência. Da mesma forma, seria preciso definir manualmente 100 expressões causais (todas com a mesma estrutura, variando apenas nas instâncias das classes referenciadas), cada uma referenciando diretamente os atributos de um par de objetos das classes *Apple* e *Archer*.

Esta forma de programação é muito mais árdua do que simplesmente definir uma estrutura de dados para armazenar os objetos de cada tipo de classe, criar estes objetos e definir as respectivas expressões causais por meio de um laço de repetição que percorre as referências destes objetos nas estruturas de dados.

Porém, o simples fato de evitar o acesso aos atributos dos objetos por meio de estruturas de dados diminui extremamente a quantidade de instruções a serem executadas pelo

processador e consecutivamente reduz os ciclos de processamento necessários para recuperar as respectivas instruções em um nível mais elevado na hierarquia de memória.

Por exemplo, a Figura 81 apresenta um trecho de código em Assembly (linha 13) que representa uma típica atribuição de valor (linha 4) e uma simples expressão causal (linha 5) em linguagem imperativa. Em Assembly, cada linha de código equivale a uma instrução a ser executada pelo processador.

```
1 ...
2
3 //Código Imperativo
4 ax = 0x200;
5 if(ax == 0x1000)
6    ax = 0x3000;
7 else
8    ax = 0x2000;
9
10 ...
11
12 //Código Assembly
13 mov ax, 0x200  // (mov - move - mover)
14                // move o valor 0x200 para o registrador ax
15 cmp ax, 0x1000 // (cmp - compare - comparar)
16                //compara o valor de ax com 0x1000 e move o resultado para o registrador EFLAGS
17 je _Label_1    // (je - Jump If Equal - salta se igual)
18                // Executa as instruções do _Label_1 quando a comparação é satisfeita
19 jne _Label_2   // (jne - Jump If Not Equal - salta se não for igual)
20                // Executa as instruções do _Label_2 quando a comparação não é satisfeita
21
22 _Label_1:
23 mov ax, 0x3000 // move o valor 0x3000 para o registrador ax
24 jmp _Label_Exit// Executa as instruções do _Label_Exit para finalizar
25
26 _Label_2:
27 mov ax, 0x2000 // move o valor 0x2000 para o registrador ax
28
29 _Label_Exit:
30
31 ...
```

Figura 81: Representação de uma expressão causal imperativa em linguagem Assembly

Conforme expressa a figura, uma atribuição e mesmo uma expressão causal consistem em poucas instruções de processamento. Basicamente, uma atribuição de valor a uma variável consiste na instrução MOV (i.e. move os valores entre os registradores) e a avaliação de uma expressão causal se refere à instrução comparativa CMP (i.e. instrução que compara o valor de dois registradores) e às instruções de salto JE e JNE (i.e. instruções que guardam referência para conjuntos de instruções quando o resultado da comparação lógica é respectivamente verdadeiro ou falso), as quais guiam o fluxo do programa em relação ao resultado da respectiva avaliação causal.

Por sua vez, uma simples expressão causal que referencia os atributos por meio de uma estrutura de dados apresenta uma quantidade extremamente maior de instruções a serem processadas. A Figura 82 apresenta o trecho de código em linguagem Assembly (a partir da linha 7) para este tipo de expressão causal que referencia a estrutura de dados *vector*. Esta expressão causal é apresentada na mesma figura, mais especificamente na linha 2 do trecho de código em linguagem imperativa.

```
1 //Código Imperativo
2  if(axList->at(i)->attribute == x)){} // axList é uma instância da classe Vector da STL
3
4 ...
5
6 //Código Assembly Gerado para o Processador ARM
7   020C22E0  E1A01007  MOV         R1, R7
8   020C22E4  E59F0810  LDR         R0, [PC, #+2064]          ; [0x20C2AFC] =xList (0x2062BE4)
9   020C22E8  E5900000  LDR         R0, [R0, #+0]
10  020C22EC  EB0008DC  BL          at                        ; 0x20C4664
11  020C22F0  E1A08000  MOV         R8, R0
12
13 at:
14  020C4664  E92D4070  STMDB       SP!, {R4,R5,R6,LR}
15  020C4668  E1A04000  MOV         R4, R0
16  020C466C  E1A05001  MOV         R5, R1
17  020C4670  E1A00004  MOV         R0, R4
18  020C4674  EBFFFFD4  BL          size                      ; 0x20C45CC
19  020C4678  E1550000  CMP         R5, R0
20  020C467C  3A000001  BCC         0x20C4688
21  020C4680  E1A00004  MOV         R0, R4
22  020C4684  EBFFFC19  BL          _Xran                     ; 0x20C36F0
23  020C4688  E3A06004  MOV         R6, #0x4
24  020C468C  E1A00004  MOV         R0, R4
25  020C4690  EBFFFFC4  BL          begin                     ; 0x20C45A8
26  020C4694  E0300596  MLAS        R0, R6, R5, R0
27  020C4698  E8BD8070  LDMIA       SP!, {R4,R5,R6,PC}
28
29 size:
30  020C45CC  E92D4010  STMDB       SP!, {R4,LR}
31  020C45D0  E1A04000  MOV         R4, R0
32  020C45D4  E1A00004  MOV         R0, R4
33  020C45D8  EBFFFC3D  BL          _Size                     ; 0x20C36D4
34  020C45DC  E8BD8010  LDMIA       SP!, {R4,PC}
35
36 _Size:
37  020C36D4  E92D4010  STMDB       SP!, {R4,LR}
38  020C36D8  E1A04000  MOV         R4, R0
39  020C36DC  E1A00004  MOV         R0, R4
40  020C36E0  EBFFFFE3  BL          _Bsize                    ; 0x20C3674
41  020C36E4  E1B00120  MOVS        R0, R0, LSR #2
42  020C36E8  E8BD8010  LDMIA       SP!, {R4,PC}
43
44 _Bsize:
45  020C3674  E92D4030  STMDB       SP!, {R4,R5,LR}
46  020C3678  E1A04000  MOV         R4, R0
47  020C367C  E1A00004  MOV         R0, R4
48  020C3680  EBFFFFF7  BL          _Blast                  ; 0x20C3664
49  020C3684  E1A05000  MOV         R5, R0
50  020C3688  E1A00004  MOV         R0, R4
51  020C368C  EBFFFFF2  BL          _Bfirst                 ; 0x20C365C
52  020C3690  E0550000  SUBS        R0, R5, R0
53  020C3694  E8BD8030  LDMIA       SP!, {R4,R5,PC}
54
55 _Blast:
56  020C3664  E5900004  LDR         R0, [R0, #+4]
57  020C3668  E1A0F00E  MOV         PC, LR
58
59 _Bfirst:
60  020C365C  E5900000  LDR         R0, [R0, #+0]
61  020C3660  E1A0F00E  MOV         PC, LR
62
63 begin:
64  020C45A8  E5900000  LDR         R0, [R0, #+0]
65  020C45AC  E1A0F00E  MOV         PC, LR
66
67 ...
```

Figura 82: Representação de uma expressão causal imperativa referenciando estrutura de dados vetores em linguagem Assembly

Deste modo, mesmo que as expressões causais definidas na Figura 81 e Figura 82 apresentem a mesma complexidade causal (i.e. uma única premissa), estas se diferem descomedidamente em relação à quantidade de instruções a serem executadas. Obviamente, esta diferença se reflete no desempenho das avaliações destas duas versões de expressões causais.

Este fato pode ser constatado pelos resultados obtidos com a execução das implementações do primeiro e segundo cenário do jogo mira ao alvo com expressões causais de acesso direto aos atributos dos objetos (tal qual descrito no 2º parágrafo da presente subseção). Estas implementações foram executadas por 100.000 iterações e os resultados são apresentados graficamente na Figura 83 juntamente com os resultados obtidos com a versão das expressões causais com acesso indireto e também com o PON.

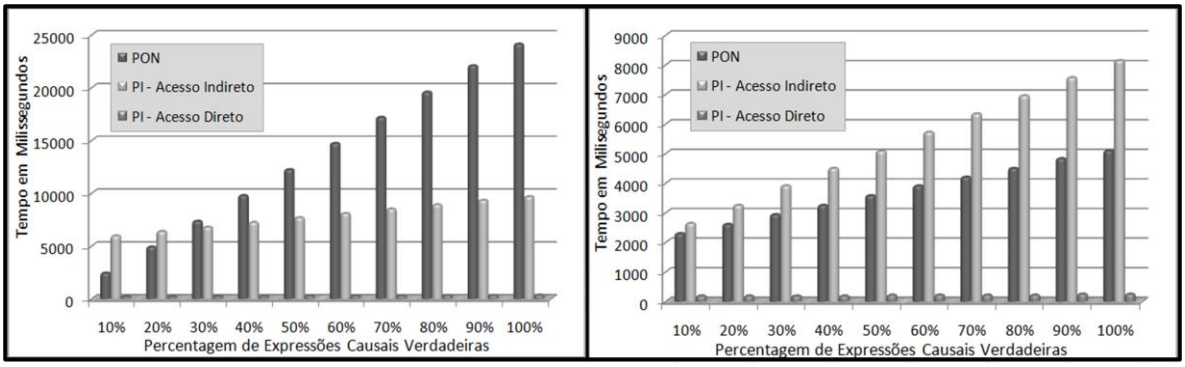


Figura 83: Experimento com as expressões causais com acesso direto aos estados dos atributos

Conforme os resultados obtidos, o acesso direto aos atributos favorece bastante o desempenho das avaliações lógicas em relação às expressões causais com acesso indireto e até mesmo com relação ao PON. Estes resultados se devem principalmente porque as instâncias que se apresentaram menos eficientes nestes cenários estão vinculadas ao percorrer de uma estrutura de dados para avaliar as expressões causais.

Como visto no capítulo 4, a materialização atual do PON também adota o uso de estruturas de dados, a qual se refere ao *vector* da STL. Em PON, como os objetos *Attributes* são ativos, as estruturas de dados são usadas para concentrar os objetos *Premises* e objetos *Conditions* a serem avaliadas pelo fluxo iniciado pelo *Attribute*. Deste modo, as interferências no desempenho causadas pelo uso de estruturas de dados é ainda maior no PON do que no imperativo com acesso indireto aos dados, uma vez que há ao uso de duas estruturas de dados para a avaliação de uma única regra (i.e. expressão causal).

Deste modo, mesmo que o PON seja comparado com o imperativo de acesso direto sobre cenários que usem todas as melhorias inclusas no *framework* (e.g. re-notificação, tabela *hash*, premissa compartilhada), ainda o PON poderia não apresentar o melhor desempenho do

que as avaliações deste tipo de expressões causais, ao menos que as redundâncias se apresentassem em grande escala na implementação imperativa. Este fato não significa que o modelo de programação do PON seja incapaz de melhorar a eficiência dos programas, mas significa que a atual forma pela qual os conceitos do PON se encontram materializados não contribuem para que se obtenham melhores resultados em termos de eficiência.

Na atual versão do PON, os conceitos são materializados sobre os recursos disponíveis na linguagem imperativa C++, inclusive as estruturas de dados, e os programas são compilados sobre compiladores que não otimizam o suficiente as interações entre os objetos colaboradores. No entanto, por se apresentar como um novo paradigma, os conceitos do PON podem ser materializados de forma independente dos recursos oferecidos pelas linguagens imperativas e também de seus compiladores.

Isto seria possível com a implementação de um compilador próprio para o PON. Este poderia consistir a uma extensão de algum compilador de código aberto para uma linguagem imperativa, como a C++. Esta extensão se encarregaria de compilar apenas as relações entre os objetos do mecanismo de notificações. Deste modo, os programadores continuariam fazendo uso da linguagem imperativa pertinente e de todas as capacidades do compilador estendido.

Deste modo, este compilador permitiria compilar o código do PON de maneira eficiente e eliminando por completo o uso das estruturas de dados para armazenar referências aos objetos a serem notificados, uma vez que os objetos notificantes e notificados seriam conectados em tempo de compilação de maneira intrínseca. A Figura 84 apresenta a forma pela qual um objeto *Attribute* seria conectado a dois objetos *Premises* em código *Assembly* supostamente gerado pelo compilador em questão.

```
1 ...
2
3 mov ax, 0x200 //instrução de atribuição a um Attribute qualquer
4 cmp ax, _estadoAnteriorAttribute //verifica se o estado deste Attribute foi alterado pela atribuição
5 je _Label_Premises //se o estado mudou, notifica as Premises pertinentes
6
7 _Label_Premises:
8   //Premise 1
9   cmp ax, _x1 // realiza o cálculo lógico para a Premise 1
10  cmp ax, _estadoAnteriorPremise1 // verifica se o estado lógico da Premise foi alterado
11  je _Label_Conditions1 //se o estado mudou, notifica as Conditions pertinentes
12
13  //Premise 2
14  cmp ax, _x2 // realiza o cálculo lógico para a Premise 2
15  cmp ax, _estadoAnteriorPremise2 // verifica se o estado lógico da Premise foi alterado
16  je _Label_Conditions2
17
18 _Label_Conditions1:
19  //se o estado lógico da Premise notificante for verdadeiro
20     //incrementa o contador de Premises verdadeiras
21     //realiza o cálculo lógico
22     //se aprovada invoca o metodo da respectiva Rule
23  //se o estado lógico da Premise notificante for falso
24     //decrementa o contador de Premises verdadeiras
25     //realiza o cálculo lógico
26
27 _Label_Conditions2:
28  //se o estado lógico da Premise notificante for verdadeiro
29     //incrementa o contador de Premises verdadeiras
30     //realiza o cálculo lógico
31     //se aprovada invoca o metodo da respectiva Rule
32  //se o estado lógico da Premise notificante for falso
33     //decrementa o contador de Premises verdadeiras
34     //realiza o cálculo lógico
35 ...
```

Figura 84: Representação hipotética da relação entre os objetos colaboradores por meio do código gerado por um compilador

Com o uso de um compilador particular, os tipos de dados do PON representados na forma de objetos poderiam ser substituídos pelos tipos de dados primitivos evoluídos com capacidade de reação. Desta forma, uma simples atribuição de valores acarretaria no início de um fluxo de notificações assim como apresentado na linha 3 da Figura 84.

Esta atribuição seria seguida por uma instrução CMP que avaliaria se o estado do *Attribute* mudou com esta atribuição (linha 4). Na seqüência, se o *Attribute* realmente mudou de estado, a instrução de salto JNE levaria ao conjunto de instruções CMP que representariam as *Premises* interessadas nesta mudança de estado (linha 7).

Da mesma forma, este processo ocorreria nas notificações das *Premises* para as suas respectivas *Conditions*, as quais também seriam representadas por simples instruções CMP além de instruções INC ou DEC para incrementar ou decrementar o seu contador de *Premises* que apresentam estado lógico verdadeiro.

Como pôde se observar, todas estas interações poderiam ocorrer sem o uso de estruturas de dados, apenas com o uso das instruções MOV, CMP, JNE, INC e DEC. A

organização destas instruções em tempo de compilação contribui bastante para reduzir a quantidade de instruções do código PON, principalmente ao eliminar as estruturas de dados.

Deste modo, com todas estas melhorias possíveis de serem implementadas sobre o modelo de programação do PON, se torna absolutamente injusto comparar a implementação atual do mecanismo de notificações com as implementações imperativas que avaliam os atributos sem fazer uso de estruturas de dados. Esta comparação apenas será justa quando o PON apresentar um compilador que permita o mesmo nível de otimizações existentes nos compiladores imperativos.

Deste modo, para demonstrar as capacidades do PON para solucionar as redundâncias temporais e estruturais no imperativo para o momento, nada mais justo do que comparar a atual versão do mecanismo de notificações com o mecanismo de avaliação que referencia indiretamente os objetos por meio de estruturas de dados.

5.2.4.4 Reflexões sobre os resultados

Considerando os argumentos expostos, os resultados dos estudos comparativos sobre o primeiro e segundo cenário são considerados verídicos, os quais demonstram o quanto o PI ainda é ineficiente devido às redundâncias temporais e estruturais.

No primeiro cenário, ao solucionar o problema das redundâncias temporais, o PON se mostrou mais eficiente do que o PI. Neste cenário, mesmo no experimento em que a quantidade de notificações foi mais significativa, o PON ainda se apresentou suficientemente mais eficiente do que o PI (i.e. na taxa de 27% das expressões causais verdadeiras) ao considerar que normalmente poucos estados variam entre as iterações. No segundo cenário, ao solucionar ambos os tipos de redundâncias sobre um experimento com a quantidade de notificações reduzida, o PON se mostrou absolutamente mais eficiente do que o PI.

Com isto, pode-se concluir que dependendo das características das aplicações (i.e. das quantidades e tipos das redundâncias a serem sanadas e a freqüência pela qual os dados têm seus estados alterados), as diferenças de desempenho entre estes dois paradigmas podem variar. Por exemplo, em cenários em que os dados têm seus estados alterados com maior freqüência em cada iteração, a simplicidade de atribuição de um valor e mesmo a passividade da busca seqüencial faz o PI prevalecer sobre o PON. Nestes cenários, se a maioria das expressões causais for afetada por estas alterações, o mecanismo seqüencial de busca pode ser

o mais adequado, uma vez que a maior parte das redundâncias temporais que vêm a existir são necessárias para aprovar cada expressão causal afetada.

Por outro lado, em cenários em que os estados variam com baixa ou mediana freqüência em cada iteração, a reatividade e a pontualidade das notificações faz o PON prevalecer sobre o PI. Nestes cenários, dependendo da quantidade de expressões causais, as redundâncias podem afetar significativamente o desempenho do PI. Assim, as notificações precisas e a qualidade de guardar lembrança dos estados já avaliados asseguram bom desempenho ao PON, tornando-o uma alternativa eficiente para aplicações ricas em expressões causais (e.g. aplicações de controle e monitoramento).

Nestes tipos de aplicações, normalmente os estados dos dados variam com pouca freqüência e a quantidade de expressões causais pode ser consideravelmente superior à quantidade correspondente aos cenários até então apresentados. Estas características formam um ambiente impróprio para que os conceitos do PI sejam aplicados.

Estas mesmas características estão presentes nos controles de células de manufatura simuladas no ANALYTICE II, as quais motivaram Simão a propor o PON, sendo esta mais adequada para aqueles propósitos do que o PI (SIMÃO, 2005). Estas características também estão presentes no sistema XCON (*e**X**pert **CON**figurer*) usado pela empresa DEC (*Digital Electronic Computer*) para configurar sistemas de computadores conforme os requerimentos dos compradores. Este apresenta aproximadamente 7000 expressões causais, o que enfatiza os problemas das redundâncias.

Inicialmente, estas expressões causais foram implementadas na linguagem imperativa FORTRAN, mas devido à ineficiência causada pelo mecanismo seqüencial do PI, as mesmas expressões causais foram reimplementadas no *shell* SBR OPS5, que adota o algoritmo RETE, uma vez que o RETE também apresenta uma solução para os problemas das redundâncias (GIARRATANO e RILEY, 1993).

Em suma, as aplicações que envolvem muitas expressões causais e nas quais os estados não mudam com tanta freqüência são mais favoráveis para a aplicação do PON. Entretanto, mesmo em aplicações que não apresentam tantos problemas de redundâncias, o PON também pode ser aplicado, mas para isto é necessário que o mecanismo de notificações seja otimizado. Entretanto, como se pode constatar pelos experimentos, mesmo o mecanismo de notificações sendo implementado na forma de uma camada abstrativa sobre a linguagem imperativa C++, o PON se mostrou suficientemente mais eficiente do que o PI, sem mencionar os aspectos qualitativos como a facilidade de programação em PON.

5.2.5 Paradigma Orientado a Notificação versus Paradigma Declarativo

Esta seção compara o PON com o PD em termos de desempenho a fim de demonstrar quanto o PON é mais eficiente ao solucionar algumas deficiências deste paradigma. Mais precisamente, os estudos comparativos ocorrem sobre dois *shells* de SBR que adotam o RETE como mecanismo de inferência, o *shell* CLIPS e *RuleWorks*, os quais respectivamente implementam a versão interpretada e compilada do RETE.

Particularmente, pelo fato da proposta do PON ser inspirada nos SBR, ambos apresentam muitas similaridades. Além destes paradigmas adotarem os elementos da base de fatos e regras como seus conceitos elementares, os mesmos também apresentam em comum as fases de um ciclo de inferência. Basicamente, um ciclo de inferência consiste nas fases de *matching*, seleção e execução, sendo que as fases de execução e seleção podem ocorrer absolutamente da mesma forma em ambos os paradigmas.

Para isto, o modelo centralizado de resolução de conflito deve ser utilizado em PON, o qual deve adotar uma estratégia de resolução de conflito em comum com o *shell* de SBR comparado. Desta forma, a principal diferença entre ambos os paradigmas consistirá na fase de *matching*, onde geralmente (em termos industriais) os SBR são implementados com o RETE e o PON apresenta o mecanismo de notificações.

Ao bem da verdade, o mecanismo de notificações guarda algumas similaridades com o RETE, uma vez que ambos solucionam os problemas das redundâncias temporais e estruturais, mesmo que de formas diferentes. Em ambos, o emprego da reatividade aos dados, o uso de um artifício particular para compartilhar ou guardar lembranças sobre estados já avaliados formam a solução para estas redundâncias.

Entretanto, a solução para estes problemas é bem mais elaborada no PON, pois neste, os próprios objetos apresentam características de reatividade ao invés desta estar condicionada a existência de uma base de fatos e os estados já avaliados são compartilhados e lembrados sem a necessidade de despender muita memória para isto. Em uma visão conceitual, o mecanismo de notificações se apresenta mais eficiente do que o RETE, pois este aparentemente demanda menos memória para solucionar as redundâncias temporais e estruturais, além de apresentar outras qualidades que não se encontram em RETE, principalmente o evitar das buscas.

Desta forma, o presente estudo comparativo não compara os paradigmas em todas as suas amplitudes, mas apenas os seus mecanismos de inferência (i.e. o mecanismo de notificações com o algoritmo de inferência RETE). Na verdade, o presente estudo

comparativo tem como objetivo definir, de modo experimental, qual dos mecanismos de inferência é o mais eficiente. Particularmente, os estudos comparativos se dão sobre dois novos cenários do jogo do mira ao alvo.

Para comparar o mecanismo de notificações com o RETE, os cenários são implementados respectivamente na linguagem C++ e nas linguagens próprias dos dois *shells* adotados. Estes cenários foram executados por uma quantidade pré-estabelecida de iterações sobre o ambiente MS-DOS. Os experimentos sobre as linguagens declarativas em ambiente MS-DOS foram possíveis devido à disponibilidade de uma versão do interpretador CLIPS para este ambiente e também pela possibilidade de compilar o código C gerado pelo *RuleWorks* para qualquer plataforma.

Entretanto, mesmo o RETE se apresentando como o algoritmo de maior impacto industrial, este não é considerado o mais eficiente. Como visto na seção 2.8.3, os algoritmos TREAT, LEAPS e HAL apresentam melhor desempenho do que o RETE. No entanto, esta dissertação não apresenta experimentos práticos do PON em relação a estas soluções de inferência, uma vez que os experimentos práticos são influenciados por diversos fatores, como pelos compiladores usados (principalmente da estratégia de otimização adotada para gerar o código de máquina), a arquitetura dos processadores, o espaço disponível na memória ou até mesmo o tamanho da memória *cache* (AHO, LABORATORIES e HILL, 2001). Por causa destes fatores, afirmar que um algoritmo é mais eficiente do que o outro pode gerar dúvidas se estes não forem comparados sobre diferentes arquiteturas, diferentes configurações de máquina e compilados sobre um mesmo compilador.

Mesmo assim, os estudos comparativos sobre estes mecanismos de inferência poderiam ser realizados sobre uma arquitetura e configuração particular, da mesma forma como esta dissertação se propõe a fazer com o RETE. Porém, a realização destes estudos foi impossibilitada devido à dificuldade em se adquirir as implementações dos algoritmos pertinentes[38]. Assim sendo, esta dissertação apresenta os estudos comparativos destas soluções teoricamente por meio da análise assintótica, a qual fornece os resultados independentemente de qualquer influência do *hardware*.

[38] Em uma tentativa de adquirir o código do TREAT, Daniel Miranker, o seu criador, declarou que já completavam quase 20 anos de quando ele propôs este algoritmo e por isso, ele não possuía de pronto uma versão do código do mesmo, muito menos uma versão que ele soubesse que funcionasse. Mesmo assim, ele se prontificou em providenciar uma versão do algoritmo, a qual poderia estar com Bernie J. Lofaso, também criador, mas afirmou que seria necessário trabalhar no código para arrumá-lo.

Na verdade, a análise assintótica não provê um resultado exato sobre o desempenho dos algoritmos em tempo de execução, mesmo porque o tempo exato de execução destes depende de vários fatores como os supracitados. Entretanto, a análise assintótica permite comparar os algoritmos de acordo com as suas complexidades algorítmicas. Desta forma, a análise assintótica se apresenta mais adequada para comparar as soluções algorítmicas do que na forma experimental, na qual a experiência de programação de um programador ou a qualidade do código gerado pode influenciar o tempo de execução.

Portanto, esta seção por meio de suas subseções apresenta os estudos comparativos prático-experimentais entre o mecanismo de notificações e o algoritmo RETE e também estudos comparativos teóricos entre o mecanismo de notificações e os algoritmos de inferência apresentados na seção 2.8.3, inclusive o RETE para confirmar a veracidade dos resultados conseguidos com os estudos práticos.

Deste modo, a seção 5.2.5.1 apresenta os estudos comparativos prático-experimentais. Nesta, a subseção 5.2.5.1.1 apresenta algumas considerações inicias sobre estes estudos e a subseção 5.2.5.1.2 apresenta os estudos comparativos de fato. Em outro foco comparativo, a seção 5.2.5.2 apresenta o estudo comparativo teórico. Nesta, a subseção 5.2.5.2.1 apresenta algumas considerações inicias sobre o estudo comparativo teórico e a seção 5.2.5.2.2 apresenta os estudos comparativos de fato.

5.2.5.1 Estudos comparativos práticos

5.2.5.1.1 Considerações sobre o estudo comparativo prático

Neste estudo, as similaridades relacionadas aos ciclos de inferência dos paradigmas permitem implementar os cenários de forma a não favorecer nenhuma solução em particular. Assim, estes estudos comparativos visam atingir resultados justos, que expressem realmente o desempenho de cada solução.

De acordo com as três fases de um ciclo de inferência, a fase de *matching* consiste na mais onerosa em termos de processamento, merecendo maior atenção por parte dos pesquisadores (MIRANKER e LOFASO, 1991). Diferentemente, as fases de seleção e execução não consomem tempo de processamento significante. Por isto, estas duas fases são implementadas de igual maneira para ambos os paradigmas a fim de enfatizar as comparações sobre a fase de *matching*. Particularmente, as duas fases foram implementadas usando a

mesma estratégia de resolução de conflitos sobre um modelo centralizado (fase de seleção) e o mesmo modo de executar regras (fase de execução) para ambos os paradigmas.

Atualmente, há varias estratégias de resolução de conflitos a serem usadas na ordenação das regras aprovadas, como a DEPTH, a BREADTH e a PRIORITY. Estas são suportadas pelo modelo de resolução de conflitos centralizado do PON (vide 3.6.1). Entretanto, a estratégia adotada neste estudo é a tradicional estratégia LEX (*LEXicographic Sort*), típica dos SBR que inferem sobre o RETE (FRIEDMAN-HILL, 2003).

Basicamente, a estratégia LEX prioriza a execução das regras que foram aprovadas por fatos de elementos inseridos/modificados mais recentemente na base de fatos (i.e. os elementos da base de fatos que apresentam valores de *Time Tag* mais altos). Nesta estratégia, se mais de uma regra vir a ser aprovada pelo mesmo fato, a regra mais específica é priorizada, ou seja, a regra que é composta pelo maior número de premissas (FRIEDMAN-HILL, 2003).

Na verdade, mesmo a estratégia LEX não sendo suportada pelo mecanismo centralizado de resolução de conflitos do PON, esta foi adotada para o presente estudo devido a presença comum desta entre os *shells RuleWorks* e CLIPS. Basicamente, o *shell RuleWorks* limita o uso de estratégias de conflito para LEX e MEA (*Means-Ends-Analysis*), enquanto que o CLIPS oferece estas duas estratégias e mais outras alternativas. Para tornar a fase de seleção compatível em ambos os paradigmas, a estratégia LEX foi integrada ao modelo de resolução de conflitos centralizado do PON.

No entanto, esta integração não ocorreu de forma simplória, uma vez que o suporte desta estratégia demandou a alteração na estrutura de alguns objetos participantes do mecanismo de notificações. A intervenção nestes objetos se fez necessário porque a estratégia em questão demanda informações dos elementos da base de fatos (i.e. valores de *Time Tag*), as quais são inexistentes e desnecessárias na implementação original do mecanismo de notificações. Também, a estratégia LEX demanda que as regras aprovadas guardem referência para os elementos da base de fatos que influenciaram diretamente as suas aprovações, sendo que esta se trata de uma funcionalidade opcional do PON.

De maneira particular, estas exigências foram implementadas em PON com a definição da variável *Time Tag* em cada classe *Attribute* e com a definição da estrutura de dados vetor na classe *Condition* para armazenar as referências para as instâncias dos *Attributes* pertinentes (conjunto de fatos). Com isto, as fases de seleção e de execução de ambos os paradigmas se equivalem, permitindo que os estudos sejam realizados apenas sobre a fase de *matching*, ou mais especificamente, entre o mecanismo de notificações e o algoritmo RETE.

5.2.5.1.2 Concretização do estudo comparativo empírico

Esta seção apresenta e discute sobre os estudos comparativos realizados sobre dois diferentes cenários do jogo “mira ao alvo” a fim de comparar os mecanismos de notificações com o algoritmo RETE.

O primeiro cenário representa um simples caso onde os mecanismos de inferência devem manipular somente uma instância de elemento da base de fatos (i.e. objeto) para cada tipo de elemento (i.e. classe). Este é um cenário favorável para o RETE executar principalmente porque a quantidade de testes de consistência realizados por Nós de Junção é extremamente reduzida, uma vez que em cada *α-memory* há somente um *α-token*. Por sua vez, o segundo cenário representa outro caso no qual os mecanismos de inferência devem manipular múltiplas instâncias de elementos da base de fatos para cada tipo de elemento.

De acordo com (TAMBE, 1991), uma grande quantidade de testes de consistência é um fator que degrada o desempenho de aplicações baseadas em RETE, principalmente quando as memórias são representadas por Listas Encadeadas. Com o uso desta representação, o tempo de processamento gasto na realização dos testes de consistência cabíveis pode aumentar proporcionalmente ao aumento da quantidade de Nós Filtros e ao tamanho da Base de Fatos (TAMBE, 1991).

De qualquer forma, por meio dos testes de consistência o RETE pode atuar na resolução de uma gama maior de problemas computacionais, como aqueles que demandam características combinatórias. O RETE soluciona problemas combinatórios por meio dos co-relacionamentos (i.e. testes de consistência) realizados pelos Nós de Junção em relação às *α-memories* pertinentes, similarmente como ocorre em operações de junção em tabelas de banco de dados (TAMBE, 1991). Além desta aplicabilidade, os co-relacionamentos permitem aos programadores compor regras em forma genérica, ou seja, referindo a classes (i.e. tipo dos elementos da base de fatos) ao invés de instâncias das mesmas. Esta é uma maneira confortável de compor regras.

Apesar das vantagens providas pelo uso dos co-relacionamentos, o PON não suporta esta funcionalidade, uma vez que um dos princípios do PON é oferecer maior eficiência às aplicações. Se o PON viesse a adotar os co-relacionamentos, isto afetaria o cumprimento deste princípio, principalmente porque geralmente estes são baseados em buscas. Assim, ao invés de aplicar buscas para relacionar os elementos da base fatos pertinentes em tempo de execução, o PON conecta-os à priori, no momento em que estes são instanciados.

Porém, o evitar dos co-relacionamentos gera algumas limitações de aplicabilidade do PON em relação às soluções de inferência mais genéricas, como o RETE. Um exemplo destas limitações se refere a não aplicabilidade do PON na solução de problemas combinatórios. Outro exemplo se refere à baixa produtividade na criação de regras quando estas referenciam instâncias ao invés de classes, ao menos que estas sejam compostas por meio das *Formation Rules* (vide Apêndice C).

Em suma, uma diferença evidente entre o mecanismo de notificações e o RETE se refere ao grau de especificidade das regras. Enquanto o PON é orientado a instâncias, o RETE é orientado a classe, o que faz este adotar buscas para correlacionar os fatos. Neste âmbito, os experimentos apresentados sobre o primeiro e o segundo cenário avaliam a eficiência destas soluções em relação a esta diferença de especificidade.

5.2.5.1.2.1 Primeiro cenário

O primeiro cenário refere-se à interação entre quatro elementos. Além dos típicos personagens arqueiro e maçã, este cenário também considera os estados de uma arma (referenciado como *Gun*) e a presença de um pássaro (referenciado como *Bird*). Estes dois últimos elementos foram adotados com a intenção de permitir a concepção de novas premissas lógicas a fim de aumentar a complexidade das regras.

Neste cenário, o estudo comparativo considera somente um exemplar (instância de classe) de cada tipo destes quatro elementos (classes), os quais interagem, sobre algumas condições, para que o arqueiro atinja a respectiva maçã. Para isto, o arqueiro deve verificar se o pássaro não se apresenta próximo da maçã, quando constatado que não, este deve aguardar a arma de fogo ser disparada para poder lançar a sua flecha. Se estas restrições forem satisfeitas, a maçã é então perfurada. Após a maçã ser atingida, esta é substituída por uma nova e a interação reinicia. Este processo se repete por uma quantidade pré-estabelecida de iterações. A interação entre estes elementos está ilustrada na Figura 85.

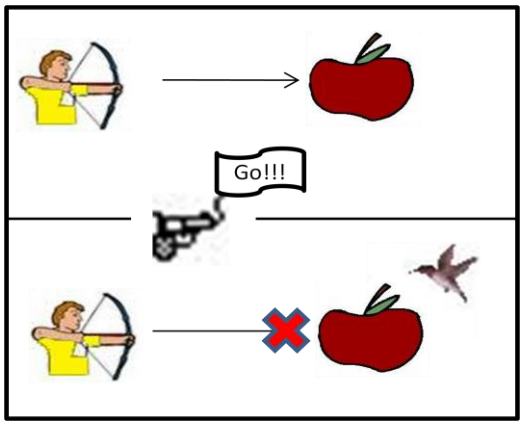


Figura 85: Primeiro cenário do estudo comparativo com o Paradigma Declarativo

Por meio da interação apresentada, este cenário proporciona uma situação favorável à execução do RETE, uma vez que as buscas desnecessárias referidas aos co-relacionamentos entre os fatos são eliminadas.

Este cenário foi implementado sobre os princípios de ambos os paradigmas, cujas implementações parciais estão apresentadas na Figura 86. Esta representa as implementações na forma estrutural (i.e. em forma de grafo) e também em termos de regras (em código-fonte) para ambos os paradigmas. Em relação às regras apresentadas, as que se referem aos SBR exibem a sintaxe da linguagem particular do *shell* CLIPS enquanto que a sintaxe da linguagem C++ é usada para representar as *Rules* no PON.

Basicamente, esta figura expõe somente as duas regras principais à interação entre os personagens do cenário, sendo que as demais se referem a questões secundárias, como controle de iterações e medição de tempo, sendo por isto, omitidas nesta representação. Estas estão apresentadas na subseção D.1.2.1 do Apêndice D. Particularmente, a regra *Rule-a* verifica as condições necessárias para o arqueiro atingir a sua respectiva maçã, enquanto que a regra *Rule-b* controla a substituição da maçã perfurada por uma nova maçã[39].

[39] A *Rule-b* é apresentada em forma simplificada para fins de representação gráfica. A sua representação completa é apresentada na seção D.1.2.1 do Apêndice D.

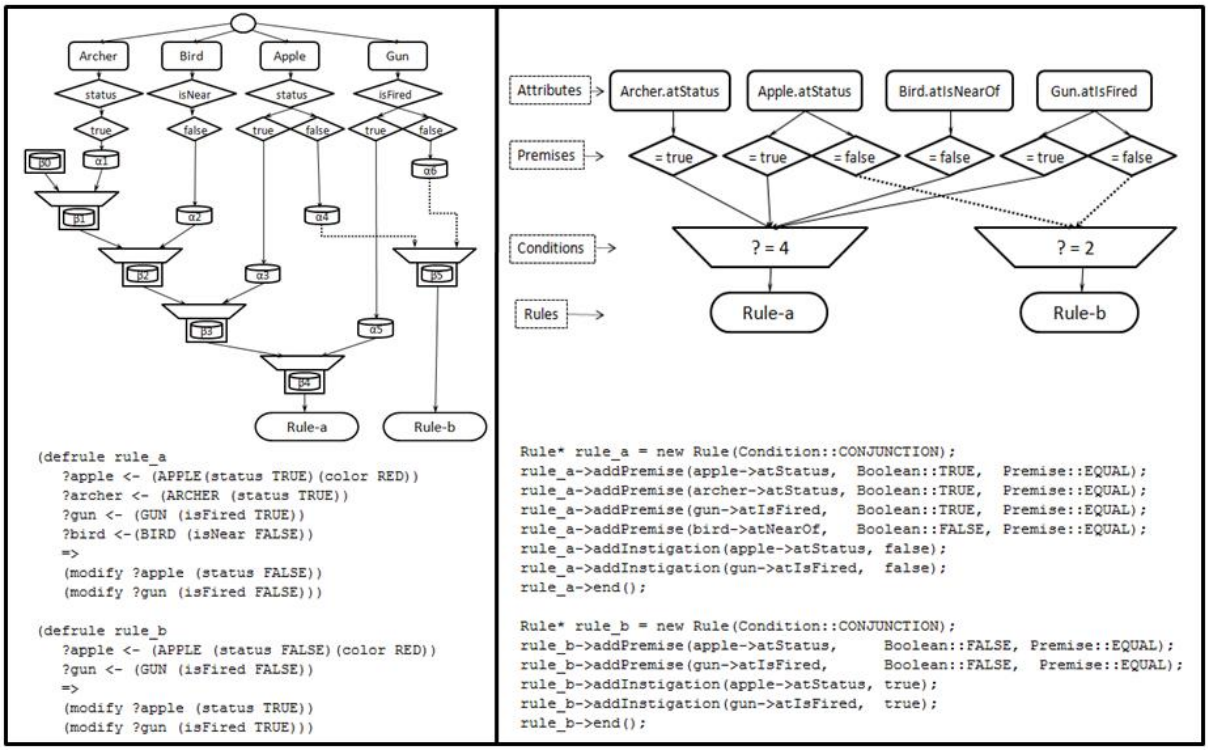


Figura 86: Representação das regras para o primeiro cenário do estudo comparativo com o Paradigma Declarativo

Neste estudo comparativo, foram realizados três experimentos sobre diferentes quantidades de iterações. Os experimentos foram executados por 10.000, 100.000 e 1.000.000 de iterações a fim de majorar as diferenças entre os resultados. Estes resultados estão ilustrados no gráfico da Figura 87.

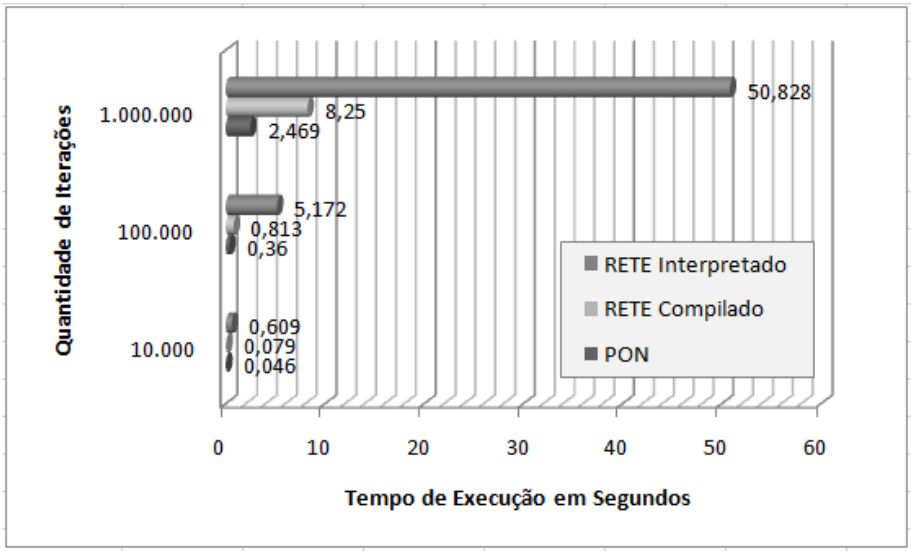


Figura 87: Resultados do experimento em relação ao primeiro cenário do estudo comparativo com o Paradigma Declarativo

De acordo com o gráfico, o mecanismo de notificações do PON se apresenta mais eficiente tanto em relação à versão interpretada (sobre o CLIPS) quanto em relação à versão compilada (sobre o *RuleWorks*) devido às suas qualidades ímpares, com especial menção às

notificações pontuais e precisas. Particularmente, no experimento de maior quantidade de iterações (i.e. um milhão de iterações), o mecanismo de notificação se apresentou quatro vezes mais rápido do que a versão compilada e vinte vezes mais rápido do que a versão interpretada.

Mesmo este cenário se apresentando favorável ao RETE ao reduzir a quantidade de testes de consistências, o RETE não apresentou um bom desempenho. Isto se deve a presença de outras deficiências além das buscas por co-relacionamento de dados. Alguns exemplos destas deficiências se referem à alocação dinâmica de memória, à ineficiência para modificar valores e às redundâncias de armazenamento de *tokens* nas memórias α e β. Estas deficiências serão discutidas com maiores detalhes na seção 5.2.5.1.2.3.

5.2.5.1.2.2 Segundo cenário

No primeiro cenário, o mecanismo de notificações e o RETE atuaram sobre a mesma quantidade de regras, ou seja, somente uma instância de regra para representar a interação entre os quatro personagens do cenário. No entanto, o que ocorre normalmente é uma diferença em relação à quantidade de regras declaradas em PON (orientadas a instâncias) e em um *shell* de SBR (orientada a classes). Normalmente, os programas implementados em PON apresentam maior quantidade de regras do que os programas implementados nos *shells*.

Desta forma, o experimento realizado sobre o segundo cenário se comporta diferentemente do experimento anterior, pois a quantidade de instâncias dos elementos da base de fatos é aumentada para conseqüentemente variar a quantidade de regras para ambos os paradigmas. Em suma, o corrente cenário compara os paradigmas em relação ao grau de generalidade/especialidade das regras.

Neste cenário, os personagens pássaro e arma de fogo foram removidos. Com isso, o cenário apresenta apenas dois tipos de personagens, porém, em maior quantidade. Na verdade, este cenário consiste em 100 arqueiros e 100 maçãs, onde cada arqueiro visa atingir a sua respectiva maçã, a qual é identificada com o seu número identificador. Conforme o cenário genérico apresentado na seção 5.2, os arqueiros e maçãs são enfileirados e de acordo com os seus números identificadores para interagirem de forma seqüencial por uma quantidade de iterações pré-estabelecida.

Neste cenário, uma iteração é separada em duas etapas. Na primeira etapa, há somente maçãs vermelhas disponíveis para serem perfuradas. Quando estas são perfuradas, as mesmas

são imediatamente substituídas por maçãs verdes. Na segunda etapa, ocorre o processo inverso. Agora, as maçãs verdes são perfuradas e substituídas por maçãs vermelhas. Este processo é esquematizado na Figura 88.

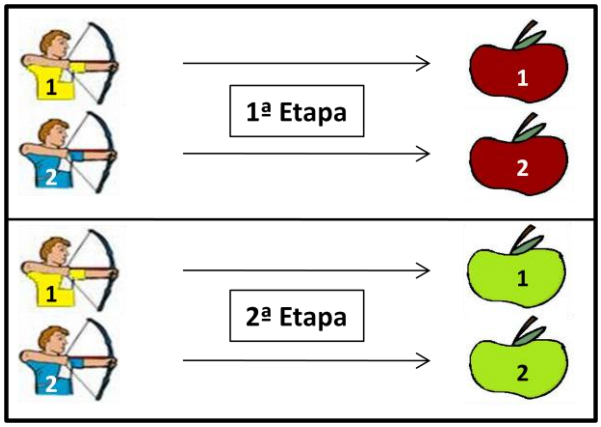


Figura 88: Segundo cenário do estudo comparativo com o Paradigma Declarativo

As duas principais regras deste cenário estão representadas na Figura 89 (em forma simplificada), as representações completas e as outras regras podem ser consultadas na subseção D.1.2.2 do Apêndice D.

À esquerda na figura estão representadas as duas regras genéricas na sintaxe do *shell* CLIPS. Estas controlam as interações entre todas as 100 instâncias dos tipos de elementos da base de fatos *Archer* e *Apple*. À direita na mesma figura estão representadas as duas mesmas regras na forma de *Rules* em PON (em forma mais específica). Porém, estas referenciam instâncias das classes *Archer* e *Apple* ao invés das próprias classes. No exemplo, as duas regras apenas controlam as interações entre as instâncias *archer1* e *apple1*.

```
(defrule Rule-1
        (archer(status TRUE)(id ?identifier))
   ?a <- (apple (status TRUE)(color RED)(id ?identifier))
   =>
   (modify ?a (color GREEN)))

(defrule Rule-2
        (archer(status TRUE)(id ?identifier))
   ?a <- (apple (status TRUE)(color GREEN)(id ?identifier))
   =>
   (modify ?a (color RED)))
```

```
1  Rule* rlRule1 = new Rule(Condition.CONJUNCTION);
2  rlRule1->addPremise(archer1->atStatus, true, Premise::EQUAL);
3  rlRule1->addPremise(apple1->atStatus, true, Premise::EQUAL);
4  rlRule1->addPremise(apple1->atColor, "RED", Premise::EQUAL);
5  rlRule1->addInstigation(apple1->atColor, "GREEN");
6  rlRule1->end();
7
8  Rule* rlRule2 = new Rule(Condition.CONJUNCTION);
9  rlRule2->addPremise(archer1->atStatus, true, Premise::EQUAL);
10 rlRule2->addPremise(apple1->atStatus, true, Premise::EQUAL);
11 rlRule2->addPremise(apple1->atColor, "GREEN", Premise::EQUAL);
12 rlRule2->addInstigation(apple1->atColor, "RED");
13 rlRule2->end();
```

Figura 89: Regras do segundo cenário do estudo comparativo com o Paradigma Declarativo

Assim, os experimentos foram realizados de acordo com as representações destas regras. Estes foram executados por 10.000, 50.000 e 100.000 iterações a fim de majorar as diferenças entre os resultados de desempenho, os quais foram contabilizados em segundos. Estes resultados são ilustrados no gráfico da Figura 90.

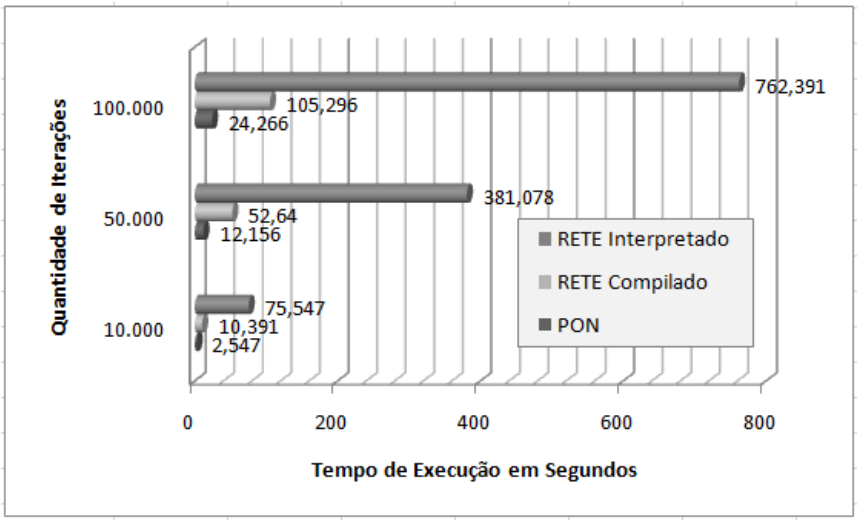


Figura 90: Resultados dos experimentos para o segundo cenário do estudo comparativo com o Paradigma Declarativo

De acordo com o gráfico, a abordagem específica do PON é mais eficiente do que a abordagem genérica dos SBR/RETE. Por meio destes resultados, pode-se concluir que é mais econômico em termos de processamento executar programas que são formados por regras específicas do que genéricas. As regras específicas participam de ciclos de inferência por meio de notificações pontuais e precisas, evitando buscas pelos fatos. Neste cenário, as buscas pelos fatos foram a causa principal do fraco desempenho do RETE.

5.2.5.1.2.3 Reflexões

Conforme demonstraram os resultados obtidos com os estudos comparativos, o mecanismo de notificações se mostrou novamente mais eficiente em relação aos mecanismos dos atuais paradigmas mais impactantes industrialmente. Desta vez, o mecanismo de notificações se mostrou mais eficiente ao evitar as deficiências existentes no algoritmo RETE, tal como as buscas pelos fatos (i.e. no percorrer de um *token* pela rede para correlacionar ou remover estados nas memórias). Além das buscas, as deficiências que de alguma forma afetaram o desempenho deste algoritmo estão descritas brevemente nos itens a seguir:

- **Complexidade Estrutural:** a representação estrutural em forma de grafo do algoritmo RETE é muito mais complexa do que a representação estrutural do mecanismo de notificações. Nesta comparação, a representação do RETE apresenta maior quantidade de elementos intermediários do que a estrutura do mecanismo de notificações. Assim sendo, enquanto o mecanismo de notificações

apresenta apenas quatro tipos de nós (i.e. *Attribute*, *Premise*, *Condition* e *Rule*) para realizar a aprovação de uma regra, a estrutura do RETE apresenta sete tipos diferentes.

- **Complexidade Estrutural Proporcional à Quantidade de Premissas em uma Regra:** o caminho do Nó Raiz até o Nó Regra na estrutura do RETE é proporcional à quantidade de Nós Filtros de uma regra. Quanto maior for a quantidade de Nós Filtros, maior será a quantidade de elementos intermediários (Nós de Junção e memórias α e β), tornando a estrutura cada vez mais complexa e o ato de percorrê-la se tornando cada vez mais lento. Diferentemente, a estrutura do mecanismo de notificações não é afetada pelo aumento do número de *Premises* conectadas a uma *Rule*, uma vez que os estados temporários dos *Attributes* em relação a estas *Premises* não são salvos em estruturas de dados similares às memórias em RETE.
- **Armazenamento Redundante de *α-tokens*:** os elementos intermediários da estrutura do RETE armazenam *tokens* redundantemente. Por exemplo, as *β-memories* armazenam os mesmos valores contidos nas *α-memórias* (*α-tokens*), uma vez que esse é representando pela concatenação destes *α-tokens*.
- **Armazenamento Redundante de *β-tokens*:** as *β-memories* também armazenam *tokens* redundantemente entre elas mesmas, uma vez que um *β-token* de uma *β–memory* reaproveita parte do conteúdo do *β-token* armazenado na *β-memory* antecessora em relação à organização estrutural da rede. Assim, em RETE, pode ocorrer de um único *α-token* ser replicado muitas vezes em diferentes *β-memories*.
- **Execuções Redundantes:** A redundância de armazenamento de *tokens* degrada o desempenho do RETE, uma vez que os *tokens* devem ser inseridos ou removidos redundantemente em diferentes nós de memória. Para modificar um *token*, o problema duplica, pois este deve percorrer duas vezes o conjunto de nós redundantes (i.e uma vez para remover o valor antigo e a outra para inserir o novo valor).
- **Ordem de Definição das Premissas:** em RETE a ordem que as premissas são organizadas para formar uma condição de uma regra afeta o seu desempenho. Em uma condição, a primeira premissa declarada (Nó Filtro) participa de mais testes

de consistência do que as premissas declaradas na seqüência[40]. Assim, os programadores de aplicações baseadas em RETE devem se preocupar com as características das premissas quando compõem uma regra. Por exemplo, o programador deve declarar as premissas que referenciam os elementos da base de fatos que mudam de estado com mais freqüência ou as premissas que referenciam uma grande quantidade de elementos da base de fatos por últimas para evitar a ocorrência de muitos testes de consistência. Mesmo que estas recomendações sejam difíceis de presumir ou mesmo tediosas para serem seguidas em cada implementação, estas são importantes para se atingir melhor desempenho com o RETE. Ciente desta deficiência, estas recomendações foram adotadas na implementação dos cenários supra-apresentados.

De fato, mesmo o algoritmo RETE sendo considerado o algoritmo de inferência mais popular, o mesmo apresenta deficiências que afetam o seu desempenho. No entanto, muitas destas deficiências são evitadas pelos algoritmos de inferência que lhe procederam (i.e. TREAT, LEAPS e HAL), permitindo que estes se apresentem mais eficientes do que o RETE.

5.2.5.2 Estudo comparativo teórico

5.2.5.2.1 Considerações sobre o estudo comparativo teórico

Mesmo que o estudo comparativo empírico entre o mecanismo de notificações e o algoritmo RETE apresente resultados confiáveis devido à significante diferença de desempenho, este ainda deixa espaço para a argüição de que os mecanismos não foram experimentados sobre todas as situações ou cenários possíveis.

Na verdade, mesmo que estes fossem comparados sobre uma maior diversidade de cenários, sobre outras aplicações, sobre diferentes compiladores e até mesmo sobre outras arquiteturas de *hardware* com diferentes configurações físicas (i.e. velocidade do processador, tamanho de memória principal e *cache*), ainda estes experimentos seriam

[40] Por exemplo, se o último Nó Filtro (premissa) de uma regra hipotética é satisfeito por um *token*, este *token* é co-relacionado por apenas um Nó de Junção. Porém, se o primeiro Nó Filtro da mesma regra é satisfeito por um *token*, este *token* deve atravessar toda a rede, sendo co-relacionado em cada Nó de Junção que se apresenta em seu trajeto até o Nó Regra.

insuficientes, uma vez que estes elementos podem ser combinados de diferentes maneiras para formar diferentes situações para serem avaliadas.

De fato, realizar experimentos empíricos sobre toda esta variedade de situações é totalmente impraticável. Geralmente, estes estudos comparativos empíricos se limitam a uma variedade pequena de situações, deixando pairar a dúvida sobre o desempenho dos mesmos em diferentes situações.

Apesar dos estudos empíricos serem realizados sobre iguais condições para as soluções comparadas, quando estas são comparadas sobre situações diferentes os resultados podem variar. Como exemplo, está a comparação de desempenho entre os algoritmos RETE e TREAT. Em (NAYAK, GUPTA e ROSENBLOOM, 1988), os experimentos realizados com os algoritmos foram executados na arquitetura SOAR[41] sobre a plataforma Unix, na qual se constatou que o algoritmo RETE apresenta melhor desempenho do que o TREAT. Em (MIRANKER e LOFASO, 1991), os experimentos sobre os mesmos algoritmos foram executados sobre a plataforma HR9000/370 (i.e. um computador com o processador Motorola 68030), na qual se constatou que o TREAT é mais eficiente do que o RETE.

Em (WANG e HANSON, 1992), os mesmos algoritmos foram adaptados e comparados no domínio de banco de dados, cujo domínio permite que os algoritmos manipulem grande quantidade de dados persistentes ao invés de uma razoável quantidade de dados em memória. Neste domínio, o TREAT se apresentou mais eficiente do que o RETE. Segundo Wang e Hanson (1992), estes resultados confirmam a supremacia do TREAT em termos de desempenho para o contexto das aplicações com dados voláteis.

Segundo Wang e Hanson (1992), os experimentos realizados sobre a arquitetura SOAR ocorreram sobre situações favoráveis ao RETE, uma vez que esta não suporta a remoção dos *tokens* de maneira tradicional (i.e. pelo percorrer de um *token* pela rede), sendo que o TREAT é mais eficiente do que o RETE nesta tarefa. No entanto, em outros estudos comparativos envolvendo os mesmos algoritmos realizados em (MIRANKER, BRANT, LOFASO e GADBOIS, 1990; LEE e CHENG, 2002), foi confirmado que realmente o TREAT supera o RETE em termos de desempenho para ambientes com dados voláteis.

Particularmente, segundo os estudos comparativos apresentados em (MIRANKER, BRANT, LOFASO e GADBOIS, 1990), dependendo do cenário em que os algoritmos são comparados, o TREAT pode se apresentar de 1/3 a 20 vezes mais rápido do que o RETE. Esta discrepância entre os resultados ilustra o grau de influência dos cenários.

---

[41] Arquitetura Cognitiva Simbólica que permite criar modelos cognitivos (DARPA, 2009)

Estes cenários utilizados são considerados como *benchmarks* padrão para comparações entre algoritmos de inferência[42]. Além destes terem sido utilizados na realização dos estudos comparativos entre o RETE e TREAT, os mesmos ainda foram empregados na concepção dos estudos que provam a eficiência dos algoritmos LEAPS (MIRANKER, BRANT, LOFASO e GADBOIS, 1990) e HAL (LEE e CHENG, 2002). Desta forma, os mesmos poderiam ter sido utilizados nos estudos comparativos entre o mecanismo de notificações e estas soluções de inferência, se não fosse pelo fato destes *benchmarks* se referirem a problemas tipicamente combinatórios.

Deste modo, para comparar o mecanismo de notificações perante os outros algoritmos e mesmo para confirmar o bom desempenho do mecanismo de notificações sobre o algoritmo RETE, os mesmos foram comparados teoricamente por meio da análise assintótica[43].

5.2.5.2.2 Considerações sobre a análise assintótica

A análise assintótica de um algoritmo consiste em estabelecer uma função que expressa em uma unidade de medida hipotética quanto de tempo de processamento e espaço em memória são necessários para executar de acordo com o volume de dados de entrada (EDMONDS, 2008). Atualmente, a análise assintótica é usada mais comumente na medição do tempo de processamento do que do espaço consumido em memória por um algoritmo, uma vez que, diferentemente de décadas atrás, o custo da memória está bem mais inferior do que o custo de processamento (AHO, LABORATORIES e HILL, 2001). Desta forma, o presente estudo apenas fará menção ao tempo de processamento dos algoritmos.

Assim, a análise assintótica é utilizada para medir teoricamente o tempo de processamento de um algoritmo à medida que o volume de dados de entrada aumenta indefinidamente. O resultado conseguido com esta análise é chamado de eficiência assintótica temporal ou complexidade temporal do algoritmo (EDMONDS, 2008).

A eficiência assintótica temporal de um algoritmo pode ser expressa para três graus diferentes de complexidade de entrada de dados: para o melhor caso, o pior caso e o caso

---

[42] Alguns benchmarks utilizados: *Manners* (um programa que organiza pessoas em uma mesa de jantar de acordo com os seus hobbies), *Tourney* (i.e. um programa que organiza os jogadores para um jogo de cartas), *Jig25* (i.e. um programa que resolve quebra cabeças) e *Rubik* (i.e. um programa que resolve o jogo do cubo)

[43] Esta forma comparativa evita a existência de qualquer interferência da plataforma de *hardware* ou mesmo da experiência do programador ou ainda do grau de otimização do compilador empregado nos *shells*.

médio. No melhor caso, o algoritmo executa sobre as configurações que o permitem executar no melhor tempo possível (i.e. quando o volume de dados de entrada é baixo) (CORMEN, LEISERSON, RIVEST e STEIN, 2002). Normalmente, o melhor caso não é considerado, pois quase sempre apresenta um volume de entrada de dados muito baixo para medir significativamente o desempenho dos algoritmos (EDMONDS, 2008).

No pior caso, o algoritmo executa sobre as condições que o fazem levar o maior tempo possível (i.e. o volume de entrada de dados é bem alto) (CORMEN, LEISERSON, RIVEST e STEIN, 2002). No caso médio, o algoritmo executa sobre um tempo médio, o qual é calculado teoricamente sobre todos os volumes de entrada de dados entre o melhor e o pior caso (EDMONDS, 2008). No caso médio, assume-se que todas as instâncias de entrada têm chances iguais de ocorrer (EDMONDS, 2008).

Na prática, o caso médio é muito mais difícil de determinar do que o tempo do pior caso, uma vez que a análise se torna matematicamente intratável devido à diversidade de entradas e também porque a noção de média das entradas nem sempre apresenta um significado óbvio. Esta pode corresponder a uma média entre o melhor e o pior caso ou ainda uma média estatística de todas as entradas possíveis (AHO, LABORATORIES e HILL, 2001).

Desta forma, a presente dissertação somente compara os mecanismos de inferência sobre o pior caso, uma vez que este provê a melhor definição matemática, é o mais fácil de analisar e o mais utilizado, pois pode mostrar a inviabilidade do algoritmo (EDMONDS, 2008). Em geral, um algoritmo que é mais eficiente no pior caso será o melhor para todos os volumes de dados de entradas, com algumas exceções para as pequenas entradas (CORMEN, LEISERSON, RIVEST e STEIN, 2002). O pior caso é representado pela notação O(f(n)) (i.e. *Big-O*), onde f(n) denota a função da complexidade do algoritmo.

5.2.5.2.3 Concretização do estudo comparativo teórico

Esta seção apresenta os comportamentos assintóticos dos algoritmos de inferência apresentados na seção 2.8.3 e os compara com a eficiência assintótica do mecanismo de notificações.

Primeiramente, segundo Miranker (1990), os algoritmos RETE e TREAT apresentam a mesma complexidade assintótica temporal e espacial para o pior caso, cuja unidade de medida se refere às comparações lógicas realizadas por estes algoritmos. No pior caso,

considera-se que os elementos da base de fatos são avaliados e aprovados por todas as premissas e todos esses têm os seus estados modificados em cada iteração. Nestas condições, o tamanho de cada *α-memory* corresponde ao mesmo tamanho da base de fatos, uma vez que nestes algoritmos cada premissa apresenta uma *α-memory* particular.

Segundo Miranker (1990), a complexidade assintótica destes algoritmos é $O(FactBaseSize^{nPremises})$, onde *FactBaseSize* corresponde ao tamanho máximo da base de fatos e *nPremises* corresponde ao número máximo de premissas em uma regra. Considerando que *nPremises* também corresponde a quantidade de *α-memories* que armazenam os elementos da base de fatos ou *tokens*, quanto maior for o valor da variável *nPremises* maior será a quantidade de comparações lógicas resultantes dos co-relacionamentos entre os elementos das *α-memories*. Desta forma, a propriedade exponencial desta função assintótica se deve às características combinatórias destes algoritmos, que faz aumentar exponencialmente a quantidade de comparações a cada inserção de uma premissa em uma regra, sendo que neste pior caso o tamanho de cada *α-memory* de uma premissa corresponde estritamente ao tamanho da base de fatos.

Na verdade, esta função assintótica se apresenta ainda mais complexa do que foi definida em (MIRANKER, BRANT, LOFASO e GADBOIS, 1990), uma vez que a análise para chegar a esta função apenas considerou as correlações entre os *tokens* armazenados nas *α-memories*. Esta função assintótica omite o tempo despendido nas comparações sobre as premissas que aprovaram os *tokens* nas *α-memories*. Desta forma, a função assintótica corrigida se apresenta da seguinte forma: $O((FactBaseSize * nPremises) * FactBaseSize^{nPremises})$, onde $(FactBaseSize * nPremises)$ representa a avaliação de todos os *tokens* representativos dos elementos da base de fatos em cada premissa[44].

Segundo Miranker (1990), o algoritmo LEAPS é assintoticamente melhor espacialmente do que o RETE e TREAT ao evitar o armazenamento de regras no conjunto de conflitos e consecutivamente evitar o armazenamento desnecessário de *tokens* nas respectivas *α-memories*, impedindo que estes sejam usados para aprovar regras que não serão executadas ou para desaprovar regras não executadas. A complexidade espacial do LEAPS consiste em $O(max(ts) * nPremises)$, onde $max(ts)$ corresponde ao maior número de *Timestamp* ou ao valor do *Timestamp* Dominante, ou seja, consiste ao limite superior de atualizações na base de fatos.

[44] Mesmo a correção não afetando a complexidade exponencial desta função assintótica, essa se faz útil para explicitar que estes algoritmos apresentam muito mais avaliações do que o especificado.

Geralmente, o algoritmo que apresenta melhor complexidade espacial também se apresenta mais eficiente em experimentos práticos, uma vez que ainda ocorrem discrepâncias no desempenho entre os níveis de memória (i.e. registradores, memória cache, memória principal e memória virtual) (AHO, LABORATORIES e HILL, 2001). Este fato foi confirmado nos experimentos empíricos realizados em (MIRANKER, BRANT, LOFASO e GADBOIS, 1990) na comparação entre o algoritmo LEAPS com o algoritmo TREAT. O LEAPS não foi comparado diretamente com o RETE, pois foi subentendido que o TREAT é mais eficiente do que o RETE. Nestes experimentos, o algoritmo LEAPS se apresentou de 2 a 3 vezes mais eficiente do que o TREAT.

Porém, a complexidade assintótica temporal do LEAPS não foi divulgada em (MIRANKER, BRANT, LOFASO e GADBOIS, 1990). No entanto, o LEAPS se apresenta como uma evolução do RETE e do TREAT[45], herdando destes a forma de correlacionar dados. Da mesma forma que ocorre no RETE e TREAT, a quantidade de correlações em LEAPS aumenta exponencialmente à medida que aumenta o número de premissas em uma regra. Deste modo, conclui-se que o LEAPS compartilha com o RETE e o TREAT a mesma complexidade assintótica temporal exponencial.

Em condições normais, a complexidade assintótica temporal destes algoritmos pode ser representado por $O((FactBaseSize * nPremises) * AvgAlphaMemorySize^{nPremise})$, onde *AvgAlphaMemorySize* corresponde ao volume médio de *tokens* nas *α-memories*, uma vez que é totalmente incomum uma *α-memory* armazenar todos os *tokens* representativos de todos elementos da base de fatos. Desta forma, constata-se que em condições normais os algoritmos RETE, TREAT e LEAPS podem apresentar um desempenho adequado se as regras não forem compostas por muitas premissas e também se as *α-memories* apresentarem volume baixo de *tokens*.

Normalmente, uma regra contém um número baixo de premissas, mas em aplicações mais complexas este mesmo número pode ser extremamente alto. Entretanto, muitas premissas têm os seus estados compartilhados pelas regras e as mesmas apresentam freqüência variável de avaliação. Geralmente, como expresso em (FORGY, 1984; FRIEDMAN-HILL, 2003; GIARRATANO e RILEY, 1993), poucos dados são afetados em

[45] O LEAPS é mais similar ao TREAT do que ao RETE. Essencialmente, o LEAPS se difere do TREAT ao permitir que os co-relacionamentos realizados por meio de buscas sejam interrompidos para a execução de uma regra. Se esta regra não alterar nenhum fato da base de fatos, a busca por co-relacionamentos continua exaustivamente da onde ela parou até que o fato com *timestamp* dominante seja comparado com todos os fatos das demais *α-memories*. Tal processo comparativo exaustivo ocorre da mesma forma em TREAT.

uma iteração e normalmente os mesmos eventos sobre esta parcela de dados se repetem durante as iterações. Mesmo assim, dependendo da entrada de dados (i.e. tamanho da base de fatos) e principalmente da complexidade das regras (i.e. número de premissas por regra), estes algoritmos podem se tornar impraticáveis para certos problemas mais complexos.

Diferentemente destes algoritmos, o HAL não compartilha a mesma complexidade temporal, uma vez que este permite evitar totalmente os co-relacionamentos ou ao menos realizá-los de uma forma mais sofisticada do que a adotada por estes algoritmos. Devido a esta qualidade flexível do algoritmo HAL, esta dissertação apenas considera o comportamento que evita totalmente os co-relacionamentos entre os dados a fim de realizar comparações justas com o mecanismo de notificações.

Deste modo, a complexidade temporal do algoritmo HAL é: *O(FactBaseSize * nPremises * nRules)*, onde *FactBaseSize* corresponde ao número de elementos da base de fatos gerenciados pelos nós Classes, *nPremises* corresponde à quantidade máxima de premissas registradas em uma classe e *nRules* corresponde à quantidade máxima de regras conectadas diretamente às classes. Para simplificar esta função assintótica, pode-se substituir cada variável da função por um valor indefinido *n*, o que resulta na seguinte função assintótica polinomial: $O(n^3)$.

Conforme expressa a função temporal polinomial, o HAL se apresenta mais eficiente do que os algoritmos supracitados ao evitar os co-relacionamentos entre os estados dos elementos da base de fatos. Empiricamente, esta supremacia temporal foi confirmada em (LEE e CHENG, 2002) sobre experimentos comparativos do HAL com o RETE e TREAT, quando o HAL se apresentou de 2 a 9 vezes mais eficiente do que o TREAT.

Nestes experimentos, o HAL não foi comparado com o LEAPS, mas assintoticamente o HAL se apresenta bem mais eficiente do que o mesmo. Entretanto, a supremacia empírica do HAL sobre o LEAPS pode ser suposta a partir das comparações já realizadas destes em relação ao TREAT, ou seja, o LEAPS se apresentou no máximo três vezes mais eficiente do que o TREAT enquanto que o HAL triplicou esta diferença.

O HAL apresenta uma solução bastante similar ao mecanismo de notificações principalmente ao permitir a supressão das correlações e permitir que as instâncias dos elementos da base de fatos e regras sejam conectadas, mesmo que indiretamente.

Desta forma, o mecanismo de notificações apresenta a mesma complexidade assintótica polinomial do algoritmo HAL, a qual é reapresentada por $O(n^3)$ ou *O(FactBaseSize * nPremises * nRules)*, onde *FactBaseSize* corresponde ao tamanho máximo de objetos *Attributes*, *nPremises* corresponde ao tamanho máximo de objetos *Premises*

notificados por estes *Attributes* e *nRules* corresponde ao tamanho máximo de objetos *Conditions* notificados por estas *Premises*[46].

Esta função assintótica representa a quantidade de notificações entre os objetos colaboradores que também corresponde à quantidade de avaliações lógicas. A constatação desta função assintótica pode ser realizada pela análise da Figura 91, a qual demonstra as relações por notificações entre os objetos colaboradores. Nesta os *Attributes*, *Premises*, *Conditions* e *Rules* correspondem respectivamente aos símbolos com abreviações *Att*, *Pr*, *Cd* e *Rl*.

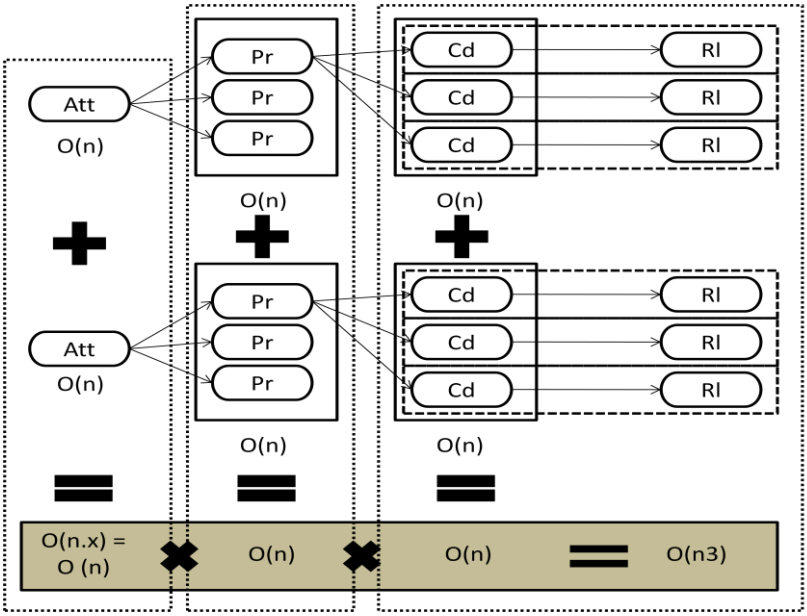


Figura 91: Representação esquemática da função assintótica do mecanismo de notificações

Nesta função, o produto de *FactBaseSize* * *nPremises* corresponde às avaliações das premissas em relação a base de fatos (i.e. conjunto de *Attributes*). Desta forma, se as avaliações das premissas fossem desconsideradas como ocorreu em (MIRANKER, BRANT, LOFASO e GADBOIS, 1990), a função assintótica do mecanismo de notificações seria apenas *O(nRules)* ou *O(n)*, uma função linear devido às comunicações pontuais.

[46] Este cálculo assintótico foi realizado por Simão em sua tese. Este cálculo se encontra na seção 3 do capítulo 5. Esta seção apresenta com maiores detalhes sobre a complexidade assintótica do PON trazendo, por exemplo, reflexões sobre o caso médio.

5.2.5.2.4 Reflexões

De acordo com os estudos comparativos teóricos o mecanismo de notificações se apresenta mais eficiente do que os algoritmos que enfatizam as buscas por co-relacionamento de dados e se equivale em termos de desempenho ao algoritmo HAL.

Em relação aos algoritmos baseados em buscas, o mecanismo de notificações se difere principalmente pelas conexões a priori e pontuais entre os objetos colaboradores e pela especificidade dos relacionamentos entre estes. Por exemplo, os objetos *Premises* referenciam pontualmente no máximo dois objetos *Attributes*, enquanto que as *α-memories* relacionadas às premissas dos algoritmos em questão podem referenciar inúmeros elementos da base de fatos. Também, os objetos *Premises* não têm os seus elementos correlacionados, o que não torna o desempenho do mecanismo influenciado pelo aumento da quantidade de *Premises* conectadas às *Conditions*.

Em relação ao HAL, apesar das grandes semelhanças, o mecanismo de notificações se diferencia na comunicação mais pontual entre os objetos. Enquanto que no HAL somente os componentes genéricos se comunicam, no mecanismo de notificações as próprias instâncias podem ser comunicar e de forma direta, evitando que entidades genéricas despendam buscas para relacionar as instâncias comunicantes.

Além disto, o HAL trata um elemento da base de fatos como uma entidade modular composta por vários atributos, sendo que a cada modificação de um estado de um elemento da base de fatos e a posterior atribuição deste à respectiva classe, cada um de seus atributos são avaliados em relação às premissas, mesmo que muitos deles não apresentem alteração de estado. No mecanismo de notificações, apenas os atributos (*Attributes*) de um elemento da base de fatos (*FBE*) que alteram de estado são avaliados e não todos os atributos deste elemento.

Ainda, as instâncias do mecanismo de notificações apresentam a capacidade de comunicação autônoma e independente, ou seja, sem a intervenção de um terceiro elemento, como a máquina de inferência. Esta comunicação mais pontual, autônoma e independente favorece a aplicabilidade do mecanismo de notificações em ambientes multiprocessados. Em PON, cada instância (e.g. *Attribute*, *Premise*) apresenta capacidades evidentes de distribuição devido as suas independências. No HAL, as instâncias se apresentam acopladas às suas entidades representativas (i.e. nó Classe ou nó Regra) e dependentes destas.

Mesmo o mecanismo de notificações se equivalendo assintoticamente ao algoritmo HAL, o PON apresenta qualidades que se sobressaem ao HAL em condições normais,

principalmente porque as comunicações entre os componentes ocorrem de forma mais direta no PON do que no HAL. Neste sentido, as qualidades do PON podem colaborar para que este exiba melhor desempenho em um oportuno estudo comparativo empírico entre os algoritmos. Assim, da mesma forma que os estudos empíricos demonstraram que o TREAT supera o RETE, uma vez que a análise assintótica é incapaz de definir o tempo real de execução de cada algoritmo, o mecanismo de notificações também pode superar o algoritmo HAL. Em suma, ambos apresentam soluções sofisticadas para realizar inferência, mas a sofisticação do mecanismo de notificações se mostra mais evidente ao eliminar totalmente as buscas por elementos passivos.

5.3 ESTUDO COMPARATIVO SOBRE PLATAFORMA EMBARCADA

Uma vez que os estudos comparativos práticos entre algoritmos podem ter os seus resultados influenciados por diversos fatores, tais como compiladores, plataformas e configurações de *hardware*, torna-se necessário que os mesmos sejam comparados sobre uma maior variedade de cenários. Neste sentido, esta seção apresenta um estudo comparativo prático entre uma aplicação implementada com o PON e outra implementada com o POO em uma plataforma embarcada. Este estudo tem como objetivo verificar se os resultados conseguidos por PON na plataforma dos computadores pessoais, onde os recursos computacionais são mais abundantes, se repetem em uma plataforma na qual os recursos são mais limitados.

Mais precisamente, o estudo comparativo ocorre sobre a plataforma embarcada relativa à placa de *hardware eSysTech* eAT55. Geralmente, esta placa é adotada na execução de experimentos, avaliações e treinamentos no desenvolvimento de sistemas embarcados. A placa eAT55 é ilustrada na Figura 92 junto com um visor de cristal líquido (i.e. LCD – *Liquid Crystal Display*) de 4 linhas e 20 colunas e um simples teclado com 5 teclas[47].

[47] Alguns cenários da aplicação Mira ao Alvo foram executados nesta plataforma e apresentaram resultados similares aos obtidos na plataforma dos computadores pessoais.

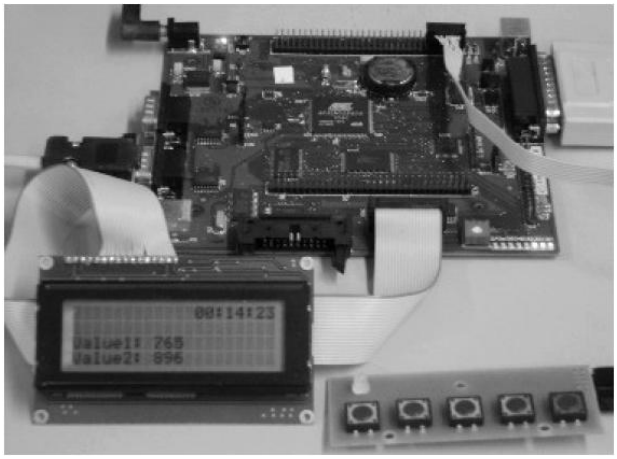


Figura 92: Placa EAT55 da eSysTech

A placa eAT55 inclui o processador AT91M5580 com núcleo ARM7TDMI de 32 bits, sendo tal processador manufaturado pela *Atmel Corporation*. Atualmente, o núcleo ARM é o mais adotado por fabricantes de processadores na construção de *hardware* para sistemas embarcados, tal como a *Intel*, *Motorola*, *Samsung* e *Sharp*. Essencialmente, além do processador AT91M5580, a placa eAT55 também inclui uma memória estática de acesso aleatório (SRAM – *Static Random Access Memory*), uma memória *flash*, um conversor analógico para digital, um conversor digital para analógico, um relógio de Tempo-Real (RTC – *Real-Time Clock*) e interfaces serial, paralela, USB (*Universal Serial Bus*) e PS/2 (*Personal System/2*) (ESYSTECH, 2009).

Normalmente, os *softwares* embarcados executam sobre a plataforma eAT55 sobre o *kernel* de Tempo Real denominado X. O *kernel* X corresponde a uma camada de *software* que executa funcionalidades típicas de sistemas operacionais para plataformas embarcadas, suportando inclusive execuções multitarefa. Basicamente, o X gerencia tarefas e provê alguns serviços, como os serviços de sincronização, temporização e trocas de mensagens, simplificando o trabalho de desenvolvimento de *software* embarcado (ESYSTECH, 2008).

Em (KÜNZLE, 2006), o *kernel* X foi empregado na construção de um *software* embarcado implementado em linguagem C++ que simula um monitor de temperatura de uma máquina industrial sobre a placa eAT55. Nesta simulação, a máquina industrial é representada por um computador conectado à placa via porta serial. Basicamente, este *software* recebe as informações de temperatura enviadas randomicamente do computador e as exibe na tela LCD juntamente com outras informações, tal como as relativas ao tempo corrente e muitas vezes mensagens de erro enviadas pelo computador.

Pelo fato de que este *software* já se encontra implementado sobre os conceitos do POO e de forma otimizada devido ao seu propósito de ser empregado em um estudo de caso para a

validação de uma ferramenta computacional que analisa Redes de Petri Temporais (KÜNZLE, 2006), o mesmo poderia ter sido utilizado na composição do estudo comparativo em questão. No entanto, sobre uma análise mais aprofundada sobre o contexto para o qual este *software* se aplica, foi concluído que este se apresenta inadequado para a realização do respectivo estudo, principalmente porque apresenta uma baixa quantidade de relações causais.

Desta forma, se fez necessário evoluir este contexto a fim de permitir a realização do estudo comparativo sobre um novo contexto mais oportuno à aplicação do PON, ou seja, em um contexto em que as redundâncias temporais e estruturais se apresentem mais significativas.

Neste sentido, a mesma estrutura do *software* monitor de temperatura foi mantida (i.e. interação entre o computador e a placa por meio de mensagens), mas o contexto do *software* foi evoluído para a simulação de um Sistema de Condicionamento de Ar[48]. Neste âmbito, a subseção 5.3.1 apresenta o contexto deste sistema e a subseção 5.3.2 apresenta os resultados dos estudos comparativos práticos realizados sobre as implementações deste sistema sobre os conceitos do PON e do POO.

5.3.1 Sistema de Condicionamento de Ar

Esta seção aborda a descrição do contexto relativo ao Sistema de Condicionamento de Ar sobre duas perspectivas. Primeiramente, a subseção 5.3.1.1 descreve os conceitos gerais do sistema como a estrutura, os componentes que fazem parte desta estrutura e mesmo os estados assumidos por estes componentes. Na seqüência, a subseção 5.3.1.2 descreve as particularidades relativas à implementação deste sistema na forma de um *software* simulado sobre a plataforma embarcada.

5.3.1.1 Conceitos gerais

O Sistema de Condicionamento de Ar considerado nesta dissertação é hipoteticamente integrado a um edifício de 16 andares. Cada andar possui uma bomba de calor, uma entrada

[48] A concepção deste sistema foi baseada no Sistema de Condicionamento de Ar exemplificado em (FRIEDMAN-HILL, 2003).

de ar ajustável e um sensor de temperatura (i.e. termômetro), os quais têm os seus estados controlados pelo componente centralizador chamado de Aplicação de Controle, assim como esquematizado na Figura 93.

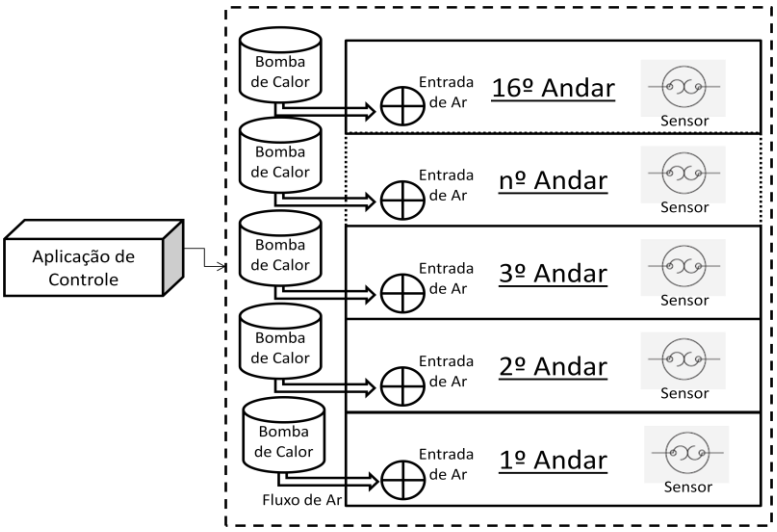


Figura 93: Sistema de ar condicionado

Neste ambiente, uma bomba de ar tem a função de aquecer ou resfriar o ar em um determinado andar. Desta forma, em um dado momento uma bomba de calor pode se encontrar no estado de HEATING (aquecendo), COOLING (resfriando) ou OFF (desligado). O ar enviado por uma bomba de ar entra no respectivo andar pela entrada de ar ajustável, a qual pode assumir apenas dois estados: OPEN (aberta) ou CLOSE (fechada).

Por sua vez, um sensor de temperatura pode identificar os seguintes estados que representam a temperatura de um andar:

- **TARGET:** a temperatura pré-configurada para o andar ou temperatura ideal.
- **UPPER e LOWER GUARD:** correspondem às temperaturas na faixa aceitável que variam respectivamente 2 ºC para mais ou para menos da temperatura ideal.
- **HOT e COLD:** correspondem respectivamente às temperaturas que variam 6ºC para mais ou para menos da temperatura ideal.
- **TOO HOT e TOO COLD:** correspondem respectivamente às temperaturas acima do estado HOT e abaixo do estado COLD.

Neste ambiente, a Aplicação de Controle controla os estados das bombas de calor e das entradas de ar a fim de atingir a temperatura ideal para cada andar de acordo como o envio periódico dos estados de temperatura pelos respectivos sensores. Deste modo, para cada estado de temperatura recebido, a Aplicação de Controle analisa um conjunto de relações causais e decide sobre as alterações de estados pertinentes.

Estas relações causais são passíveis de serem representadas em forma de tabelas relacionais. Por exemplo, a Tabela 5 apresenta a relação entre os estados de um sensor de temperatura e os estados de uma bomba de calor.

Nesta tabela, considerando que o estado atual de uma bomba de calor é COOLING (resfriamento), se o estado de temperatura recebida for igual à temperatura ideal (i.e. TARGET) ou acima desta (i.e. UPPER GUARD, HOT ou TOO HOT), a Aplicação de Controle mantém o estado corrente da bomba de calor. No entanto, se o estado de temperatura recebida for abaixo da temperatura desejada (i.e. LOWER GUARD, COLD, TOO COLD), a Aplicação de Controle emite um sinal para desligar a bomba de calor, ou seja, esta altera o estado da bomba de calor para OFF.

Tabela 5: Estados da bomba de calor

	TOO COLD	**COLD**	**LOWER GUARD**	**TARGET**	**UPPER GUARD**	**HOT**	**TOO HOT**
OFF	Inicie o Aquecimento	Inicie o Aquecimento				Inicie o Resfriamento	Inicie o Resfriamento
HEATING					Desliga	Desliga	Desliga
COOLING	Desliga	Desliga	Desliga				

Da mesma forma que a Aplicação de Controle atua sobre uma bomba de calor por relacionar o seu estado com a temperatura recebida, o mesmo ocorre em relação às entradas de ar.

Os dados presentes na Tabela 6 permitem controlar o estado de uma entrada de ar de um determinado andar de acordo com a temperatura enviada pelo respectivo sensor quando o estado da bomba de calor é COOLING. Por exemplo, considerando que a bomba de calor deste andar assume o estado COOLING, quando a Aplicação de Controle recebe a informação de que o estado de temperatura do respectivo sensor se apresenta abaixo da temperatura ideal (i.e. LOWER GUARD, COLD, TOO COLD), esta emite um sinal para fechar a respectiva entrada de ar, ou seja, altera o estado desta para CLOSE. Neste caso, a entrada de ar somente será aberta novamente quando o estado de temperatura recebida se apresentar superior a temperatura ideal (UPPER GUARD, HOT, TOO HOT).

Tabela 6: Estados da entrada de ar para a bomba de calor no estado COOLING

	TOO COLD	**COLD**	**LOWER GUARD**	**TARGET**	**UPPER GUARD**	**HOT**	**TOO HOT**
OPEN	Fechar Entrada	Fechar Entrada	Fechar Entrada				
CLOSE					Abrir Entrada	Abrir Entrada	Abrir Entrada

5.3.1.2 Conceitos específicos

Como pôde se observar, o contexto dos Sistemas de Condicionamento de Ar se apresenta abundante em relações causais, o que evidencia as ocorrências das redundâncias temporais e estruturais em relação ao contexto do *software* monitor de temperatura proposto por (KÜNZLE, 2006). Entretanto, estes contextos se apresentam similares no que se refere às instigações das avaliações causais, as quais ocorrem a partir do recebimento de informações relativas à temperatura.

Desta forma, a estrutura do *software* monitor de temperatura foi adaptada e também evoluída para a implementação do contexto dos Sistemas de Condicionamento de Ar. Como parte desta adaptação, o computador conectado à placa passou a representar os sensores de temperatura do edifício e a placa passou a comportar o código referente à Aplicação de Controle (i.e. as expressões causais). No entanto, esta implementação não fez uso do visor LCD adotado para exibir as temperaturas monitoradas.

Como parte desta evolução, cada componente do Sistema de Condicionamento de Ar (i.e. bombas de calor, entradas de ar e sensores) foi representado como um objeto. Estes objetos são instâncias das classes *HeatPump*, *Vent* e *Sensor*, as quais são representadas em notação UML na Figura 94 respectivamente sobre os princípios do POO e PON. Ambas as representações de classes apresentam os seus atributos/*Attributes* com os valores padrão ("*default*") de inicialização. Os valores a serem assumidos por estes atributos se referem aos estados dos componentes conforme foi apresentado na subseção anterior.

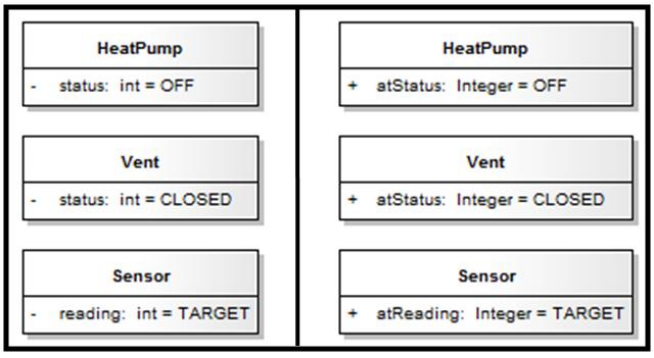


Figura 94: Classes do sistema de condicionamento de ar

Dentre estas classes, a classe *Sensor* é usada para representar computacionalmente os sensores "físicos" simulados pelo computador conectado à placa. Diferentemente, as outras duas classes não apresentam um correspondente "físico", estas apenas são representadas computacionalmente por não apresentarem características autônomas para mudar de estado.

A relação entre os objetos da classe *Sensor* e os respectivos sensores “físicos” simulados (i.e. computador conectado à placa) ocorre por meio da Aplicação de Controle. Esta recebe o estado de temperatura enviado por um sensor “físico” de um determinado andar do edifício e atribui este estado ao atributo do objeto que representa computacionalmente este sensor. Este atributo consiste no atributo *reading* da classe *Sensor* sobre a representação imperativa e no *Attribute atReading* do *FBE Sensor* sobre a representação no PON.

Todos os sensores do edifício apresentam a capacidade de informar a Aplicação de Controle sobre os estados de temperatura lidos. Os sensores informam estes estados por meio do envio de uma mensagem para a Aplicação de Controle, sendo que esta apenas pode tratar uma única mensagem por vez.

As mensagens se apresentam em dois formatos ou tipos diferentes: a mensagem simples e a mensagem composta. Uma mensagem simples contém o estado da temperatura relativo a apenas um sensor e a mensagem composta pode conter um ou mais estados de temperaturas relativos a um ou mais sensores.

Por exemplo, se 10 sensores precisarem informar as suas temperaturas à Aplicação de Controle em qualquer momento, estes o fazem por meio de mensagens simples. Assim, a Aplicação de Controle receberá 10 mensagens e tratará destas 10 mensagens seqüencialmente. No entanto, se os mesmos 10 sensores precisarem informar as suas temperaturas à Aplicação de Controle em um momento combinado, estes o fazem por meio de uma mensagem composta. Assim, a Aplicação de Controle receberá apenas 1 mensagem e tratará desta mensagem de uma única vez.

O tratamento de uma mensagem ocorre da seguinte forma: com o recebimento de uma mensagem, a Aplicação de Controle extrai o estado de temperatura (no caso da mensagem simples) ou os estados de temperatura (no caso de uma mensagem composta) e atribui este(s) estado(s) aos atributos pertinentes. Na seqüência, estes estados são avaliados em relação às respectivas expressões causais de forma passiva no mecanismo imperativo ou de forma ativa no mecanismo de notificações.

A Aplicação de Controle comporta um total de 256 expressões causais. Estas são representadas na forma de *if-then* quando a aplicação se apresenta implementada com o POO e na forma de *Rules* quando esta se apresenta implementada com o PON. Na verdade, estas expressões causais são particionadas por andar, sendo que os estados dos componentes de cada andar são controlados de forma particular por 16 expressões causais. A Figura 95 apresenta uma destas expressões causais, a qual é representada na forma de *if-then* com acesso indireto aos atributos (linha 3) e também na forma de *Rule* (linha 12).

Basicamente, esta expressão causal verifica se (a) a bomba de ar de um andar *i* está desligada, (b) se o estado da temperatura recebida de um sensor do andar *i* é UPPER_GUARD e (c) se a entrada de ar do andar *i* está fechada. Caso estes estados sejam confirmados, a respectiva entrada de ar é aberta. O conjunto completo das expressões causais imperativas e *Rules* que se diferem em estrutura semântica são apresentadas na seção D.2 do Apêndice D, inclusive a apresentada na Figura 95.

```
1 ...
2
3 if((heatPumpList->at(i)->state == HeatPump::OFF)       &&
4    (sensorList->at(i)->reading == Sensor::UPPER_GUARD) &&
5    (ventList->at(i)->state == Vent::CLOSE))
6 {
7     ventList->at(i)->state = Vent::OPEN;
8 }
9
10 ...
11
12 Rule* rl_4 = new Rule(Condition::CONJUNCTION);
13 rl_4->addPremise(heatPumpList->at(i)->atState, HeatPump::OFF, Premise::EQUAL);
14 rl_4->addPremise(sensorList->at(i)->atReading, Sensor::UPPER_GUARD, Premise::EQUAL);
15 rl_4->addPremise(ventList->at(i)->atState, Vent::CLOSE, Premise::EQUAL);
16 rl_4->addInstigation(ventList->at(i)->atState, Vent::OPEN);
17 rl_4->end();
18
19 ...
```

Figura 95: Representação da única expressão causal aprovada no estudo comparativo sobre a plataforma embarcada

Em relação às expressões causais imperativas, a quantidade de vezes que estas são avaliadas pela Aplicação de Controle depende do tipo de mensagem recebida. Se a aplicação receber uma mensagem simples, todas as 256 expressões causais são percorridas para avaliar o único estado contido na mensagem. No entanto, se a aplicação receber uma mensagem composta, as mesmas 256 expressões causais são percorridas para avaliar todos os estados contidos na mensagem. Diferentemente do PI, o mecanismo de notificações atua da mesma forma sobre os dois tipos de mensagens, uma vez que as avaliações são pontuais e imediatas à atribuição de um estado.

5.3.2 Estudos comparativos práticos

Esta seção apresenta os estudos comparativos práticos relativos à implementação do Sistema de Condicionamento de Ar com os conceitos do PON e do POO em linguagem

C++[49]. Estes estudos práticos foram seccionados em dois cenários, uma vez que as avaliações das expressões causais imperativas são avaliadas de forma distinta de acordo com o tipo de mensagem recebida.

Estes cenários são referenciados respectivamente como Primeiro Cenário e Segundo Cenário. Basicamente, o Primeiro Cenário se refere ao recebimento de mensagens simples pela Aplicação de Controle enquanto que o Segundo Cenário se refere ao recebimento de mensagens compostas.

Desta forma, a subseção 5.3.2.1 apresenta algumas considerações sobre a realização dos estudos práticos, a subseção 5.3.2.2 apresenta os resultados dos estudos práticos para o Primeiro Cenário e a subseção 5.3.2.3 apresenta os resultados dos estudos práticos para o Segundo Cenário. Por fim, a seção 5.3.2.4 permite uma reflexão sobre os resultados obtidos com estes novos estudos práticos.

5.3.2.1 Considerações sobre os estudos práticos

Em ambos os cenários, cada experimento é dividido em 16 fases. Cada fase consiste em uma quantidade diferente e crescente de sensores que informam suas temperaturas à Aplicação de Controle por meio de mensagens simples no Primeiro Cenário e de mensagens compostas no Segundo Cenário.

Por exemplo, na primeira fase de cada experimento apenas um sensor informa a sua temperatura, o que ocorre por meio de uma mensagem simples no Primeiro Cenário e por meio de uma mensagem composta no Segundo Cenário[50]. Na última fase de cada experimento os 16 sensores informam as suas temperaturas, o que ocorre por meio de várias mensagens simples no Primeiro Cenário ou por uma única mensagem composta no Segundo Cenário.

Com respeito a esta interação, o tempo de execução de uma fase do experimento consiste no tempo gasto pela Aplicação de Controle para tratar da(s) mensagem(s) recebida(s) na respectiva fase. Este tempo é computado considerando que um sensor apenas pode

---

[49] A ferramenta de desenvolvimento utilizada foi o *IAR Embedded Workbench* para processadores ARM (SYSTEMS, 2008). Esta ferramenta oferece um editor de código C/C++, o compilador *ARM IAR C/C++* e ainda permite que a aplicação desenvolvida seja facilmente embarcada na placa eAT55 por meio de uma conexão por porta paralela.

[50] Deste modo, uma mensagem composta que contém apenas um estado de temperatura é equivalente a uma mensagem simples, pois é tratado da mesma forma pela Aplicação de Controle imperativa.

informar a sua temperatura uma única vez por fase, sendo que este estado é reinicializado para o valor padrão ("*default*") em cada nova fase, ou seja, para o estado TARGET.

Devido às limitações do *hardware* da plataforma em questão, esta única interação por fase se faz suficiente para evidenciar a solução do PON para os problemas das redundâncias temporais e estruturais no PI. Na verdade, as redundâncias temporais e estruturais se apresentam em maior grau no Primeiro Cenário, uma vez que todo o conjunto de expressões causais imperativas é avaliado especialmente para cada mensagem. No segundo cenário, estas redundâncias se apresentam menos significativas, uma vez que o mesmo conjunto de expressões causais é avaliado apenas uma vez para tratar de todos os estados incluídos na mensagem composta.

Os tempos de execução para cada fase dos experimentos foram computados sobre um caso que se apresenta mais caro em termos de processamento para o PON, no qual se exige maior quantidade de notificações. Este caso ocorre quando um componente sensor informa o estado de temperatura UPPER_GUARD, uma vez que este estado é o mais referenciado pelas expressões causais. Entre as 16 expressões causais por andar 4 delas avaliam este estado. Desta forma, considera-se que cada sensor sempre informará o estado da temperatura como UPPER_GUARD em ambos os cenários para ambas as implementações dos paradigmas.

Em relação aos estados dos outros componentes (i.e. bomba de calor e entrada de ar), estes são mantidos invariáveis de acordo com os estados de inicialização apresentados na Figura 94 (i.e. o estado padrão de cada bomba de calor é OFF e de cada entrada de ar é CLOSE). Ainda, o conjunto das instruções executáveis das expressões causais *if-then* e o conjunto de *Instigations* no PON foram omitidos (pelo uso do símbolo de comentário "//") a fim de evitar que estes estados fossem alterados. Esta decisão foi tomada para impedir que as alterações sobre os *Attributes* do PON iniciassem fluxos de notificações que viessem resultar em avaliações ou mesmo aprovações de outras expressões causais (i.e. *Rules*). Esta ocorrência inviabilizaria a realização dos estudos comparativos da forma planejada.

Sobre estas considerações, quando um *Attribute atReading* receber o estado de temperatura UPPER_GUARD este notifica a única *Premise* interessada após recuperar o endereço desta em sua estrutura de dados Tabela *Hash* interna[51]. Esta *Premise* tem o seu estado lógico compartilhado por quatro *Conditions* devido à solução para as redundâncias estruturais.

[51] O uso da Tabela *Hash* evita que *Premises* não interessadas no estado UPPER_GUARD sejam notificadas.

Deste modo, com a alteração do estado lógico desta *Premise*[52], esta gera quatro fluxos de notificações em direção às respectivas *Conditions*. Dentre estes fluxos, apenas um resulta na aprovação de uma *Rule*. A *Rule* aprovada se refere à *Rule.4*, a mesma apresentada na Figura 95, sendo que as demais *Premises* são satisfeitas pelos estados de inicialização padrão dos respectivos objetos da classe *HeatPump* e *Vent*. O processo de aprovação desta *Rule* pelo fluxo inicializado em *atReading* é esquematizado na Figura 96.

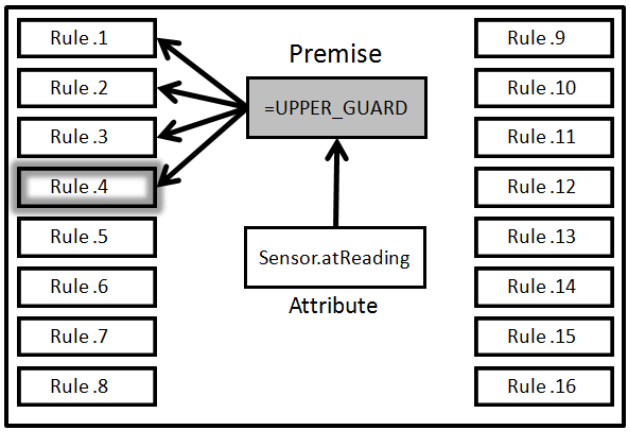


Figura 96: Comportamento do PON em ambos os cenário do estudo comparativo com o PON e o POO na plataforma embarcada

Assim, uma vez que os *Attributes* são reativos e adotam a Tabela *Hash* para notificarem estritamente as *Premises* interessadas em seus estados e com o compartilhamento destas pelas *Rules* pertinentes, o PON soluciona as redundâncias temporais e estruturais para os cenários em questão. Desta forma, os ganhos em desempenho obtidos ao solucionar estas redundâncias são apresentados nas duas seguintes seções.

5.3.2.2 Primeiro cenário

No Primeiro Cenário, um número incremental de sensores envia apenas mensagens simples em cada fase do experimento. Em relação à implementação com o PI, a Aplicação de Controle avalia as 256 expressões causais para cada mensagem recebida. Assim, se os 16 sensores enviarem as suas mensagens para a Aplicação de Controle, esta deverá avaliar um total de 4096 expressões causais, sendo que apenas 16 expressões causais serão satisfeitas[53].

---

[52] Ocorre a mudança de estado porque o *Attribute atReading* é reinicializado para o estado TARGET em cada nova fase do experimento.

[53] Certamente, a implementação com o PI poderia ser otimizada ao definir as expressões causais em métodos/procedimentos relativos a cada andar. Assim, de acordo com a mensagem recebida de um determinado andar, o estado de temperatura que esta inclui seria direcionado para ser avaliado em relação às expressões

Em relação à implementação com o PON, a Aplicação de Controle trata da mensagem recebida com apenas 4 fluxos de notificações. Assim, se os 16 sensores enviarem as suas mensagens para a Aplicação de Controle, haverá um total de 64 fluxos de notificações, sendo que 16 destes fluxos resultam na aprovação das *Rules* pertinentes.

Estas implementações foram executadas e os resultados obtidos em tempo de execução são apresentados na Figura 97.

Andares	PON	PI
1	367	8.542
2	851	17.073
3	1.218	25.569
4	1.709	34.098
5	2.071	42.537
6	2.556	50.976
7	2.924	59.402
8	3.410	67.781
9	3.779	76.172
10	4.265	84.535
11	4.634	92.860
12	5.120	101.171
13	5.489	109.504
14	5.975	117.747
15	6.344	126.043
16	6.830	134.182

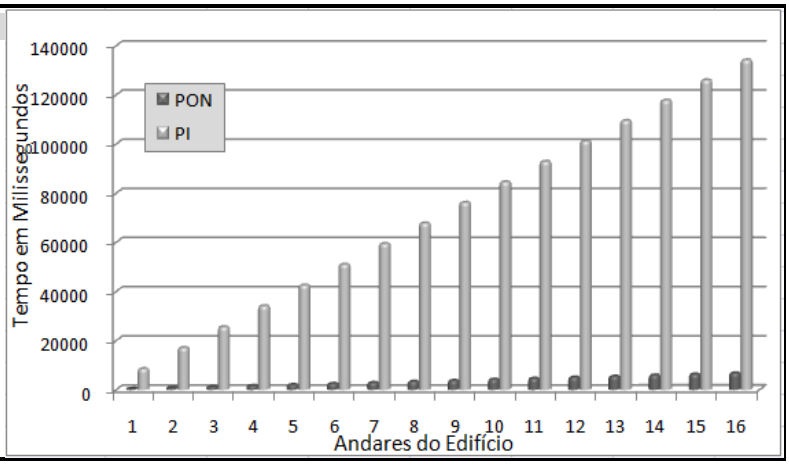


Figura 97: Resultados do experimento sobre a plataforma embarcada sobre o primeiro cenário

Conforme explicita o gráfico, o PON apresenta excelentes resultados em relação ao PI ao solucionar as redundâncias temporais e estruturais. Desta forma, mesmo que as avaliações causais ocorram de forma otimizada no PI (i.e. evitando as avaliações de todas as premissas de uma conjunção quando se verifica que uma destas apresenta estado lógico falso), estas otimizações são insuficientes para assegurar um bom desempenho. Ainda, pode-se perceber pela análise destes resultados a inviabilidade de aplicação do PI para atuar em sistemas que apresentam grande quantidade de expressões causais.

5.3.2.3 Segundo cenário

No Segundo Cenário, um número incremental de sensores envia os respectivos estados de temperatura na forma de uma mensagem composta em cada fase do experimento. Em relação à implementação com o PI, a Aplicação de Controle avalia as 256 expressões causais

---

causais definidas no método/procedimento pertinente. No entanto, esta otimização não foi adotada porque se preferiu considerar o mesmo nível de dificuldade para compor as expressões causais. Estas otimizações acarretariam na adição de códigos para direcionamento (um bloco *switch* ou um conjunto de *ifs* aninhados) que afetariam as comparações avaliativas. Ainda, sem estas otimizações consegue-se simular uma aplicação mais complexa que apresenta grande quantidade de redundâncias temporais e estruturais.

para cada mensagem composta recebida, a qual envolve no mínimo uma e no máximo 16 informações de temperatura. Neste caso, estas 256 expressões causais são avaliadas uma única vez por fase, podendo satisfazer os estados lógicos de no mínimo uma e no máximo 16 expressões causais.

Desta forma, as avaliações desnecessárias das expressões causais são reduzidas em cada fase do experimento, mas ainda se apresentam significativas. Mesmo que uma mensagem comporte 16 estados de temperatura para a satisfação das 16 expressões causais, ainda 240 expressões causais são avaliadas desnecessariamente.

Em relação à implementação com o PON, a Aplicação de Controle apresenta o mesmo comportamento do cenário anterior. Desta forma, estas implementações foram executadas e os resultados obtidos em tempo de execução são apresentados na Figura 98.

Andares	PON	PI
1	367	8.540
2	851	8.545
3	1.218	8.594
4	1.709	8.603
5	2.071	8.653
6	2.556	8.663
7	2.924	8.713
8	3.410	8.723
9	3.779	8.774
10	4.265	8.784
11	4.634	8.841
12	5.120	8.852
13	5.489	8.903
14	5.975	8.914
15	6.344	8.965
16	6.830	8.975

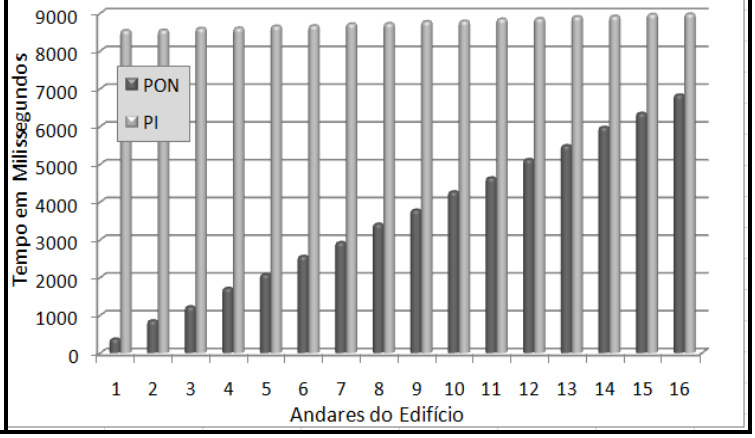


Figura 98: Resultados do experimento sobre a plataforma embarcada sobre o segundo cenário

Conforme explicita o gráfico, mesmo neste cenário em que as redundâncias se apresentam em menor grau, o PON ainda apresenta melhor desempenho. O PON se apresenta mais eficiente em 100% das fases ao gerar no máximo 64 fluxos de notificações, os quais afetam apenas 25% do total de expressões causais (64 de 256 expressões causais) sendo que 6,25% do total de expressões causais (16 de 256 expressões causais) são realmente aprovadas. Diferentemente, as avaliações passivas do PI afetam 100% das expressões causais para que os mesmos 6,25% do total de expressões causais sejam satisfeitas (16 de 256 expressões causais).

Entretanto, o tempo de execução da Aplicação de Controle imperativa se apresenta praticamente constante entre as fases do experimento ao avaliar o mesmo conjunto de expressões causais a cada fase. Por sua vez, o tempo de execução da Aplicação de Controle criada com o PON se eleva em uma proporção constante a cada fase do experimento devido ao aumento dos fluxos de notificações.

5.3.2.4 Reflexões

Estes experimentos serviram para confirmar a superioridade do PON em relação ao PI em um dado contexto, no qual as expressões causais acessam os dados de forma indireta por meio de um vetor. Conforme os resultados apresentados, o PON se apresenta mais eficiente mesmo quando executado sobre uma plataforma com recursos computacionais limitados.

Esta superioridade do PON sobre o PI nestes cenários ocorreu devido à quantidade significativa de redundâncias temporais e estruturais. Esta larga diferença em termos de desempenho entre os paradigmas permitiu que o PON fosse avaliado sobre estes mesmos cenários em relação ao PI cujas expressões causais acessam diretamente os dados. Estes estudos foram executados e os resultados estão apresentados na Figura 99.

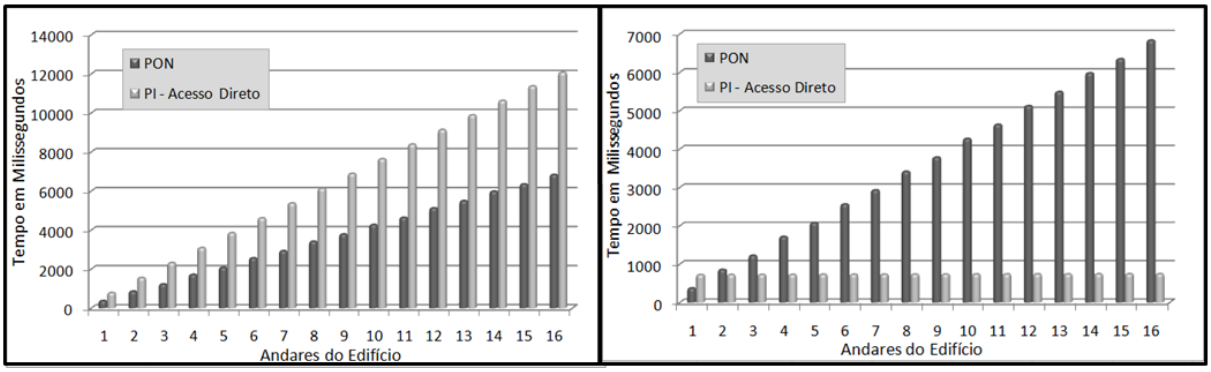


Figura 99: Experimento com as expressões causais com acesso direto aos estados dos atributos sobre o primeiro e segundo cenário

Conforme explicitam os gráficos, as redundâncias ocorrem de forma tão significativa nestes cenários que permitem o PON apresentar melhor desempenho em algumas situações, mesmo não apresentando todas as otimizações necessárias como discutido na subseção 5.2.4.3.

Em relação ao Primeiro Cenário (gráfico à esquerda), o PON se mostrou mais eficiente em 100% das fases dos experimentos devido à grande quantidade de redundâncias evitadas. Em relação ao Segundo Cenário (gráfico à direita), onde as redundâncias se apresentam menos significativas, o PON ainda se apresentou mais eficiente em relação a informação de temperatura de um único andar, o que corresponde a 6,25% da quantidade de dados alterados para este cenário. Esta percentagem se apresenta bem maior do que a taxa calculada por Forgy (1984), o qual constatou que menos de 1% dos estados são alterados em uma iteração.

Em suma, conclui-se por meio dos resultados obtidos que em situações nas quais as redundâncias se apresentam em demasia, o desempenho das aplicações imperativas é fortemente afetado. Também, os resultados obtidos permitem a conclusão de que o modelo do

PON apresenta grandes qualidades ao solucionar as redundâncias de maneira tal que podem superar até os problemas relacionados à sua atual materialização. Outrossim, ao ser constatado que a atual materialização do PON apresenta melhores resultados do que as duas versões do PI quando as redundâncias são significativas, então a materialização otimizada do PON (i.e. com a supressão das estruturas de dados vetor) poderá oferecer resultados ainda mais gratificantes.

5.4 CONCLUSÃO

Este capítulo apresentou os estudos comparativos práticos e teóricos do PON em relação aos atuais paradigmas, principalmente sobre aqueles que inspiraram a sua concepção. Conforme os resultados obtidos com estes estudos, o PON se mostrou suficientemente mais eficiente em alguns casos (considerando inclusive as constatações de Forgy (1984) em relação à taxa da variação de estados em uma iteração) e absolutamente mais eficiente em outros.

Mais precisamente, nos estudos comparativos em relação ao POO, o PON se mostrou suficientemente mais eficiente quando os problemas das redundâncias temporais e estruturais não eram tão significativos nas aplicações imperativas quando comparadas a situações justas no tocante a sua materialização (i.e. expressões causais imperativas acessando dados em vetores). No entanto, o PON se mostrou absolutamente mais eficiente quando as redundâncias se apresentaram em quantidade plausível para contornar os problemas relacionados à sua atual materialização. Ainda, os estudos permitiram concluir que o PON mantém o mesmo bom desempenho de suas aplicações quando estas executam sobre uma plataforma com recursos computacionais limitados.

Em relação aos SBR, o PON se apresentou experimentalmente mais eficiente do que as aplicações de SBR com inferência em RETE, o algoritmo de maior impacto industrial. Ainda, o PON, ou mais precisamente, o mecanismo de notificações do PON se apresentou teoricamente mais eficiente do que o RETE e seus derivados (i.e. TREAT e LEAPS) por meio da análise assintótica. Neste contexto, apesar do mecanismo de notificações e o HAL apresentarem a mesma complexidade assintótica para o pior caso, o mecanismo de notificações é supostamente melhor do que o HAL em condições normais de execução devido à eliminação total de buscas.

De um modo geral, este capítulo demonstrou que o PON resolve as deficiências relacionadas à ineficiência de execução dos atuais paradigmas, principalmente daqueles que

lhe deram inspiração, se apresentando como uma solução eficiente, de fácil aplicação e intuitiva à cognição humana que, entretanto, ainda necessita de avanços em sua materialização para apresentar melhor desempenho de maneira absoluta.

CAPÍTULO 6

CONCLUSÃO E TRABALHOS FUTUROS

Este capítulo apresenta a conclusão final sobre o trabalho e abre perspectivas para trabalhos futuros. Desta forma, a seção 6.1 apresenta a conclusão do trabalho situando o PON como um paradigma de fato e a seção 6.2 apresenta os trabalhos futuros que contribuirão para aumentar as capacidades e benefícios na programação com o PON.

6.1 CONCLUSÃO

O PON, mesmo se inspirando em qualidades do POO e de SBR, evolui os seus componentes elementares para apresentar uma nova forma de estruturar o pensamento sobre os problemas computacionais. Com o PON, o desenvolvedor percebe os problemas computacionais naturalmente por meio de objetos/elementos da base de fatos e regras, os quais entretanto atuam sobre um ambiente computacional ideal.

Neste ambiente ideal, os objetos/elementos da base de fatos deixam de ser passivos e se comportam de forma autônoma e reativa a cada mudança de estado, notificando pontualmente as regras pertinentes responsáveis pelo tratamento do respectivo estado, sempre quando estritamente necessário. Os objetos/elementos da base de fatos (i.e. os *FBEs* com seus *Attributes* e *Methods*) se encontram relacionados às regras (i.e. *Rules* e seus objetos colaboradores), as quais são independentes e executam de maneira otimizada graças a estes relacionamentos por notificações pontuais e pertinentes, conforme apresentado e discutido nesta dissertação.

Desta forma, o PON altera a forma pela qual os programas são computados por serem baseados no ambiente ideal e permite que o programador conceba programas pensando sobre as particularidades deste ambiente, tal como nas capacidades ativas dos objetos e regras. Entretanto, devido às facilidades de programação do PON por meio da ferramenta Wizard, o programador pode até se abstrair (da consciência) desta forma de programar, mesmo assim podendo obter bons resultados. Porém, esta consciência se faz necessária para que o programador consiga decidir sobre os problemas que o PON se aplica adequadamente e principalmente para melhor se beneficiar das vantagens do PON.

De fato, o PON se apresenta como um novo paradigma de programação. A essência do PON se enquadra na definição do termo paradigma elaborada por Kunh (1970) relativa à

ciência física, cuja interpretação para a ciência da computação consente o entendimento de que um paradigma oferece uma forma particular de "enxergar" os problemas do mundo real a serem tratados computacionalmente. Da mesma forma, a essência do PON se enquadra na definição do termo paradigma elaborada por Roy e Haridi (2004) particular à ciência da computação, na qual um paradigma oferece uma nova forma de estruturar o pensamento do desenvolvedor na concepção dos programas.

Assim sendo, o PON oferece uma nova forma de computar e programar, a qual permite "curar" as deficiências dos atuais paradigmas de programação. Estas deficiências são relativas à ineficiência de execução, dificuldades de programação e forte acoplamento entre as partes dos programas.

Para solucionar a deficiência relativa à ineficiência de execução, o PON evita a realização de avaliações causais sobre elementos passivos e as redundâncias decorrentes pelo uso de inferência por notificações como já discutido. Isto faz com que o PON seja absolutamente mais eficiente do que as soluções atuais de inferência do PD conforme demonstraram os estudos experimental-práticos e os estudos teóricos realizados neste trabalho. Isto também faz com que o PON seja aceitável em relação a certas aplicações baseadas no PI e absolutamente mais eficiente que outras, as quais apresentam as redundâncias em quantidade plausível para contornar os problemas relacionados à sua atual materialização conforme demonstraram os experimentos. Provavelmente, com melhorias relativas à materialização do PON, como o advento de um compilador particular, estes resultados poderão se apresentar ainda mais gratificantes.

Para solucionar a deficiência relativa às dificuldades de programação, o PON permite que o desenvolvedor programe por meio de elementos intuitivos à cognição humana, os quais são facilmente editados por meio de interfaces bem definidas em linguagem imperativa ou pelo uso da ferramenta *Wizard*. Desta forma, o PON permite que o desenvolvedor programe em estilo declarativo por meio da ferramenta *Wizard* e, quando maior flexibilidade se fizer necessária, o PON permite que o desenvolvedor adote as facilidades de linguagem imperativa presentes no *framework*.

Para solucionar a deficiência relativa ao forte acoplamento, o PON permite que o desenvolvedor construa aplicações que apresentem acoplamento mínimo devido à independência dos *FBEs*, *Rules* e seus colaboradores (graças à abordagem por notificações), o que contribui para aumentar o nível de reuso dos *FBEs*. Ainda, o fato do PON reduzir o nível de acoplamento entre as partes das aplicações e otimizar suas interações pode provavelmente

permitir que estas sejam mais facilmente distribuíveis e melhor obtenham benefícios da computação multiprocessada.

Em suma, o PON apresenta soluções para deficiências dos atuais paradigmas, constituindo-se como uma solução útil para o futuro da ciência da computação. Segundo Alan Kay, um dos idealizadores do POO e criador da linguagem Smalltalk, "a melhor maneira de prever o futuro é inventá-lo" (GASCH, 2005)[54]. Neste sentido, o PON foi inventado por Simão e teve as suas qualidades e vantagens apresentadas nesta dissertação. De acordo com o que foi expresso sobre o PON, permite prever que ele apresenta potencial para se posicionar como uma solução de programação de bom aceite perante a comunidade de programadores. Este fato propiciaria o fácil desenvolvimento de aplicações para que estas executem de maneira mais eficiente em ambientes monoprocessados e provavelmente em ambientes multiprocessados.

6.2 TRABALHOS FUTUROS

O presente trabalho é o pioneiro na apresentação dos conceitos do PON como um paradigma de programação ao público. Assim sendo, este abre perspectivas de pesquisa para gerar melhoramentos sobre o PON e sua materialização de forma a contribuir para tornar a programação ainda mais fácil, os programas ainda mais eficientes e aumentar as suas aplicabilidades. Ainda, novas pesquisas podem ser iniciadas para estudar a aplicação do PON em ambientes multiprocessados, bem como para propor avanços sobre as capacidades e comportamentos dos objetos notificantes a fim de expandir as suas aplicabilidades nestes ambientes.

6.2.1 Melhoramentos na facilidade de programação

Para tornar a programação ainda mais fácil com o PON, se faz necessário uma metodologia de programação que permita conceber as relações entre os objetos participantes do mecanismo de notificações de forma ainda mais intuitiva à cognição humana (i.e. ainda mais natural que objetos e regras). Esta metodologia e o respectivo ferramental devem

[54] Traduzido do inglês "*The best way to predict the future is to invent it*".

direcionar o programador para desenvolver aplicações com o PON de maneira gráfico-natural ao seu modo de pensar. Esta metodologia poderia ser materializada em forma de uma linguagem gráfica de modelagem própria para o PON (da "mesma forma" que a UML é particular ao POO).

Esta linguagem de modelagem poderia ser implementada sobre uma nova versão da ferramenta *Wizard* do PON, permitindo ao programador compor os *FBEs* e *Rules* de forma mais visual do que textual, ou seja, de forma a simular as suas percepções e deduções gráficas realizadas sobre as entidades e situações no mundo real. O programador poderia representar os *FBEs*, juntamente com os seus estados e comportamentos, por meio de símbolos geométricos (i.e. círculos, retângulos, etc.) ao invés de se preocupar com as suas implementações em forma de código. Com isto, o programador visualizaria as representações dos *FBEs* e os manipularia com o auxilio do *mouse* (i.e. arrastando os símbolos), relacionando-os a fim de formar graficamente um conjunto de *Rules*.

6.2.2 Melhoramentos no tocante à Engenharia de Software

Na verdade, a metodologia e o respectivo ferramental integrados à ferramenta *Wizard* se apresentariam como um princípio de uma metodologia de Engenharia de *Software* para o PON, uma vez que as suas qualidades visuais naturalmente permitem que as aplicações sejam projetadas antes destas serem definitivamente implementadas[55].

No entanto, para que os projetos das aplicações detenham maior qualidade, faz-se necessário que as representações dos *FBEs* e *Rules* concebidas na ferramenta *Wizard* sejam transformadas automaticamente em notações mais formais (e vice-versa) com maiores capacidades representativas do ponto de vista da engenharia.

Neste sentido, com particular menção aos *FBEs*, as representações gráficas destes (na forma de classes e objetos) na ferramenta Wizard poderiam ser representadas mais formalmente por meio das notações de classes e objetos na linguagem de modelagem UML. Tal representação seria adequada devido à popularidade da UML perante os desenvolvedores habituados com as linguagens do POO.

---

[55] Nesta ferramenta, primeiro a estrutura dos *FBEs* é definida por meio dos símbolos gráficos para que depois os seus métodos sejam devidamente implementados.

Com especial menção às *Rules*, as suas representações gráficas poderiam ser convertidas, por exemplo, em Redes de Petri a fim de permitir alguma análise formal. As Redes de Petri permitem representar graficamente e analisar formalmente sistemas computacionais que envolvem características de paralelismo, concorrência, sincronização, exclusão mútua e mesmo que apresentem regras causais (SIMÃO, 2005) (KORDIC, 2008). Na verdade, algumas pesquisas precedentes já realizaram estudos para demonstrar a adequabilidade do mapeamento entre regras e Redes de Petri (BAKO e VALETTE, 1990) (HE *et al*, 1999) (EHRIG *et al*, 2008).

Ademais, Simão (2005) salienta que as *Rules* e suas decomposições (mecanismo de notificações) apresentam ainda maior proximidade ao modelo teórico de Redes de Petri do que as regras típicas de SBR. Assim sendo, as Redes de Petri podem ser utilizadas para representar as *Rules* e seus elementos colaboradores de uma forma simples e intuitiva com a intenção de melhor visualizar as relações entre os objetos notificantes e notificados e mesmo para visualizar os direcionamentos dos fluxos de notificação iniciados nos *Attributes*.

Assim sendo, um trabalho futuro pertinente seria o desenvolvimento de técnicas de Engenharia de *Software* para o PON envolvendo conceitos da UML e as capacidades representativas das Redes de Petri.

6.2.3 Melhoramentos na eficiência de execução

Normalmente, o uso da ferramenta *Wizard* tornaria muito mais fácil o ato de programar. Mesmo assim, o programador poderia preferir usufruir de toda a flexibilidade das linguagens imperativas do que das facilidades providas por esta ferramenta. Deste modo, se faz necessário que novas versões do *framework* (também utilizado pelo *Wizard*) sejam criadas para materializar o PON sobre uma maior variedade de linguagens de programação. Estas versões são necessárias para permitir que os benefícios do PON estejam presentes também em linguagens diferentes da C++ devido a resistências dos programadores em mudar de linguagens de programação. Estas versões permitiriam maior popularidade do PON perante a comunidade de programadores.

Para tornar os programas ainda mais eficientes quando desenvolvidos com o PON, se faz necessário a construção de um compilador particular que otimize as relações entre os objetos participantes do mecanismo de notificações. Estas otimizações incluem a eliminação de estruturas de dados vetores para armazenar os objetos notificados (i.e. *Premises* e

Conditions), uma vez que os objetos notificantes e notificados seriam conectados em tempo de compilação como foi apresentado na subseção 5.2.4.3.

Este compilador poderia consistir em uma extensão de compiladores de código aberto disponível para as linguagens de programação que viessem dar suporte ao PON. Esta extensão se encarregaria de compilar apenas as relações entre os objetos do mecanismo de notificações. Deste modo, os programadores continuariam fazendo uso da linguagem de programação pertinente e de todas as capacidades do compilador estendido.

Este compilador poderia ser integrado à ferramenta *Wizard* do PON e ser usado de maneira transparente ao programador. Com esta integração, o compilador geraria o código de máquina ou Assembly pertinente sem o programador precisar instigar os serviços deste compilador ou muito menos tomar conhecimento sobre o compilador que está sendo usado.

6.2.4 Estudo da aplicabilidade em ambientes multiprocessados

Para que o PON seja adotado efetivamente na construção de aplicações multiprocessadas (paralelas e distribuídas) ainda se fazem necessárias algumas melhorias. Nestas melhorias há diferenças para ambientes que compartilham memória entre os nós de processamento (i.e. ambientes paralelos) e ambientes nos quais cada nó apresenta a sua memória particular (i.e. ambientes distribuídos).

Em ambientes distribuídos, faz-se necessário conceber uma plataforma que automatize e torne transparente ao programador todas as questões de comunicação entre os nós de processamento remotos e um mecanismo de balanceamento de carga.

A plataforma a ser concebida poderia adotar uma camada de *software*/*middleware* pré-definida como o CORBA (*Common Object Request Broker Architecture*), DCOM (*Distributed Component Object Model*) ou o RMI (*Remote Method Invocation*), estas auxiliariam na implementação das abstrações relativas à comunicabilidade entre os nós de processamento.

O mecanismo de balanceamento de carga poderia potencialmente ser concebido baseando-se, por exemplo, em soluções fundamentadas em algoritmos evolucionários. Assim, os objetos participantes do mecanismo de notificações poderiam ser distribuídos de forma balanceada e cooperar (por notificações) conforme os endereços definidos na plataforma.

Em ambientes paralelos, por sua vez, onde as particularidades de comunicação pela rede são desnecessárias, apenas a implementação do modelo de escalonamento de *Rules* descrito no Apêndice A é suficiente para as aplicações executarem efetivamente.

No entanto, em ambos os ambientes se faz necessário um modelo eficaz para a resolução de conflitos entre as *Rules*. Para isto, o modelo de resolução de conflitos descentralizado descrito no Apêndice A ou o modelo de resolução de conflitos desenvolvido por Simão (2005) devem ser usados.

6.2.5 Estudo da aplicabilidade na Inteligência Artificial

Atualmente, o PON se apresenta como uma solução de programação inspirada na forma pela qual a mente humana deduz sobre os problemas do mundo por meio da avaliação de um conjunto pré-definido de "regras" em relação à percepção dos "fatos".

No entanto, a máquina cognitiva humana apresenta qualidades muito mais sofisticadas do que apenas a capacidade de tomada de decisão sobre as percepções de estados. Uma destas qualidades consiste na capacidade de aprendizado.

A capacidade de aprender de um ser humano permite que o seu conhecimento seja desenvolvido conforme as suas experiências vivenciadas, sendo que o conhecimento é passível de ser representado em forma de regras. Esta capacidade de aprendizado permite que diferentes pessoas apresentem bases de conhecimento diferentes (i.e. regras diferentes) e consecutivamente funções diferentes na sociedade ao aprenderem ou entenderem de assuntos diferentes.

Da mesma forma, pessoas com experiências de vida diferentes (i.e. aprendizado de regras diferentes) pensam muitas vezes de forma adversa conforme os seus "conjuntos de regras". Certamente, esta diversidade de conhecimento ou diferentes pontos de vista não seria possível se cada pessoa nascesse com o mesmo "conjunto de regras" pré-definido e sem a capacidade de auto-evoluir este "conjunto".

Desta forma, uma possível evolução sobre o PON seria a concepção de um mecanismo que permita que novas regras sejam concebidas autonomamente de acordo com as particularidades do problema computacional a ser resolvido. Esta capacidade (de aprendizado) do ser humano se faz necessária na programação para que aplicações computacionais aprendam autonomamente a melhor solução para um dado tipo de problema.

Esta capacidade de aprendizado se faz útil para conceber soluções efetivas para problemas computacionais que não apresentam uma solução algorítmica exata (e.g. problema de gerenciamento de cadeia de suplementos) ou mesmo quando a solução algorítmica se apresenta tão complexa que torna impraticável a sua implementação, como em soluções que apresentam muitas relações causais a serem definidas. A definição manual destas relações por meio de regras pode se tornar tediosa ou mesmo incompleta devido à possibilidade do desenvolvedor esquecer-se de definir alguma relação pertinente. Para estes problemas, o aprendizado dinâmico de regras seria a solução.

Na verdade, esta capacidade de aprendizado relativa às aplicações computacionais já é há algum tempo foco de estudo da Inteligência Artificial (IA), resultando na concepção de várias técnicas de aprendizado. O conjunto destas técnicas é comumente referenciado pelo termo aprendizagem de máquina. Sucintamente, o objetivo da aprendizagem de máquina é permitir que aplicações computacionais tomem decisões cada vez mais acertadas baseando-se em experiências anteriores.

Com a utilização de aprendizado de máquina sobre o PON, para compor *Rules*, a sua viabilidade poderia ser testada em uma gama maior de problemas (reais). Uma linha de pesquisa em voga que apresenta estes requisitos se refere à resolução de problemas de gerenciamento de cadeias de suprimentos[56].

De fato, muitas pesquisas vêm aplicando de maneira efetiva os conceitos de aprendizado de máquina para resolver este tipo de problema. Algumas destas pesquisas aplicam estes conceitos para que SBR ou Sistemas Especialistas adquiram a capacidade de aprender novas regras a fim de solucionar este tipo de problema (PIRAMUTHU, 2005; CARBONNEAUA, LAFRAMBOISEA e VAHIDOV, 2008), outras aplicam estes conceitos sobre a colaboração entre agentes de *software* (i.e. sistemas multiagentes) com o mesmo objetivo de solucionar este tipo de problema (FOX, BARBUCEANU e TEIGEN, 2000; KIMBROUGH, WU e ZHONG, 2002; STRADER, LIN e SHAW, 1998; BRITO, TACLA e SIMÃO, 2008).

De acordo com estas pesquisas, pode-se presumir que o problema de gerenciamento de cadeias de suprimentos é factível por meio do aprendizado de regras e também pelo uso das colaborações e outras qualidades dos agentes de *software* (i.e. autonomia, reatividade,

[56] Segundo SIMCHI-LEVI, KAMINSKY e SIMCHI-LEVI (2000), Gerenciamento de Cadeia de Suprimentos é um conjunto de abordagens para integrar eficientemente fornecedores, fábricas, depósitos e armazéns, sendo que as mercadorias são distribuídas na quantidade certa, no local certo e na hora certa, minimizando os custos globais do sistema e satisfazendo os requisitos impostos.

representatividade), uma vez que os participantes de uma cadeia de suprimentos (fábricas, consumidores, fornecedores, etc.) podem ser naturalmente representados por agentes (BRITO, TACLA e SIMÃO, 2008). Na verdade, os agentes de *software* são similares aos *FBEs*, uma vez que as mesmas qualidades relativas à autonomia, reatividade, representatividade e colaboração dos agentes se encontram presentes nos *FBEs*.

Em suma, o PON apresenta um modelo que pode ser evoluído para agregar maior inteligência às aplicações que contribui para que este seja aplicado em um número maior de problemas computacionais. Um trabalho futuro seria conceber este modelo e aplicá-lo em um problema computacional que obtenha benefícios destas qualidades de aprendizado. Este problema seria o gerenciamento de cadeias de suprimento, uma vez que os *FBEs* (como os agentes de *software*) se apresentam teoricamente adequados para representar os participantes de uma cadeia de suprimentos e o aprendizado por regras ou *Rules* também se apresenta como uma solução bastante adequada a este problema.

6.2.6 Estudo da aplicabilidade na Computação Ubíqua

Atualmente, a Computação Ubíqua se apresenta como uma linha de pesquisa em evidência, sendo esta aplicada na concepção de ambientes ditos inteligentes. Um ambiente inteligente é um pequeno “mundo” onde aparelhos inteligentes trabalham continuamente para tornar mais confortável a vida das pessoas que habitam neste ambiente (COOK e DAS, 2004).

Um exemplo de ambiente inteligente consiste em uma casa inteligente, onde os aparelhos (e.g. aparelho de som, telefone, secadora ou máquina de lavar roupas) se comunicam entre si e com os moradores da casa. Assim, uma máquina de lavar roupa pode “questionar” ao aquecedor de água se a temperatura da água está adequada ou o aparelho de som pode autonomamente abaixar o seu volume quando o telefone recebe uma chamada ou ainda a secadora de roupas pode avisar por meio do sistema de comunicação interno da casa quando as roupas estiverem secas (LOKE, 2006).

Atualmente, as casas inteligentes podem ser classificadas em duas principais categorias: casas programáveis e casas adaptativas. As casas programáveis têm os seus comportamentos definidos por meio de ações pré-configuradas que devem ser executadas de acordo com os eventos detectados pelos sensores dos aparelhos. Por sua vez, as casas adaptáveis são aquelas que têm capacidades para aprender sobre os estados e ações a serem

executadas no ambiente. Esta se adapta aos hábitos dos moradores da casa, dispensando a programação explícita (i.e. linhas de código) para configurar algum serviço (LOKE, 2006).

Quanto às casas programáveis, estas podem demandar um esforço significante para configurar as funcionalidades dos aparelhos ou colaborações entre eles de acordo com as mudanças nos hábitos de uma família. Esta situação se agrava quando a configuração deve ser feita especialmente em cada casa e não há uma ferramenta amigável (i.e. um *wizard*) disponível. Normalmente, os moradores de uma casa inteligente invocam os serviços de um suporte técnico para realizar as configurações cabíveis, uma vez que lidar diretamente com as tarefas de programação está longe das capacidades e interesses dos típicos moradores destas casas. Ainda, a requisição freqüente dos serviços do suporte técnico pode se tornar caro e ao mesmo tempo tedioso aos moradores.

Desta forma, a programação relativa às configurações destes aparelhos deve ser facilitada, permitindo que um morador típico realize as configurações necessárias sem recorrer para o suporte técnico. Neste sentido, um trabalho futuro poderia ser a aplicação dos conceitos do PON para atuar no controle dos comportamentos e relações entre estes aparelhos inteligentes. Isto seria feito por meio de uma versão da ferramenta *Wizard* do PON que facilitaria a atualização/remoção ou mesmo a adição de novos comportamentos e relações entre os aparelhos. Com isto, um morador comum poderia ajustar as regras para atingir a melhor configuração para o ambiente de acordo com as necessidades de sua família. Assim, o uso dos conceitos do PON por meio da ferramenta *Wizard* tornaria mais fácil a configuração do ambiente inteligente, uma vez que as regras são naturais à cognição humana.

Em uma casa inteligente, as *Rules* poderiam executar em um servidor que se comunicaria com os sensores distribuídos por toda a casa. Neste cenário, cada sensor notificaria o servidor quando necessário sobre as suas percepções a fim de ativar a execução das *Rules* pertinentes. Em outro cenário, as *Rules* poderiam ser integradas aos próprios aparelhos inteligentes permitindo que as colaborações pelo mecanismo de notificações ocorram de forma distribuída. Este cenário se apresentaria como um ambiente apropriado para experimentar o comportamento do mecanismo de notificações distribuído quando este estiver definitivamente materializado. Por fim, as casas adaptáveis seriam um ambiente ideal para aplicar as capacidades de aprendizado do PON quando estas forem realmente concebidas.

APÊNDICE A

MODELO DE RESOLUÇÃO DE CONFLITOS DESCENTRALIZADO PARA AMBIENTES MULTIPROCESSADOS

Este apêndice apresenta um modelo de resolução de conflitos descentralizado para ambientes multiprocessados que se mostra como uma evolução do modelo de resolução de conflitos descentralizado para ambientes monoprocessados descrito na seção 3.6.2. Apesar daquele modelo monoprocessado se apresentar como uma solução eficiente, o mesmo não se adéqua em sua forma pura a ambientes com múltiplos nós de processamento.

Em ambientes multiprocessados, as mudanças de estados nos *Attributes* podem ocorrer em paralelo, gerando fluxos paralelos de notificação. Por meio destes fluxos, várias regras podem ser aprovadas para executarem em paralelo. Deste modo, a inadequação do modelo monoprocessado está na falta de um mecanismo para evitar que as regras acessem concorrentemente um mesmo *Attribute* exclusivo (*Attribute-Exclusive*). Oportunamente, um *Attribute* é dito exclusivo quando duas ou mais regras concorrem para alterar o seu estado de maneira exclusiva.

Neste sentido, o autor desta dissertação concebeu um mecanismo faltante que permite adaptar o modelo descentralizado monoprocessado para atuar em ambientes multiprocessados. A seção A.1 apresenta este modelo com maiores detalhes. Neste mesmo contexto, Simão também propôs um modelo descentralizado de resolução de conflitos para ambientes multiprocessados, o qual também pode ser aplicado a ambientes monoprocessados (SIMÃO, 2005). Este modelo é apresentado em maiores detalhes em (SIMÃO, 2005)[57].

Ainda, para que estes modelos possam atuar efetivamente em ambientes multiprocessados, principalmente em ambientes com memória compartilhada, se faz necessário um mecanismo que escalone as execuções das regras a fim de aumentar o aproveitamento dos recursos de processamento. Neste sentido, a seção A.2 apresenta um mecanismo de escalonamento que pode ser integrado em ambos os modelos de resolução de conflito para ambientes multiprocessados. Por fim, a seção A.3 apresenta uma breve reflexão sobre estas soluções.

---

[57] Mais precisamente, o modelo proposto por Simão se encontra no Capítulo 5, na subseção 5.4.2 de sua tese. A sua tese, por sua vez, se encontra no endereço http://www2.cpgei.ct.utfpr.edu.br/teses-e-dissertacoes/teses/teses-2005.

A.1 MODELO DESCENTRALIZADO PARA AMBIENTES MULTIPROCESSADOS

Basicamente, este modelo de resolução de conflitos considera os *Attributes* como fonte geradora dos conflitos, considerando este como um *Attribute-Exclusive*. Sendo assim, para que uma *Condition* seja aprovada, além de ela apresentar todas as suas *Premises* com os estados lógicos verdadeiros, esta também precisa adquirir exclusividade de acesso aos respectivos *Attributes-Exclusives* conectados a estas *Premises*.

Neste modelo, a responsabilidade de assegurar a exclusividade de acesso aos *Attributes-Exclusives* é delegada às próprias *Premises* conectadas, as quais adotam um mecanismo simples para manter a sincronização. O mecanismo utilizado se refere ao uso de uma variável Booleana de controle (*flag*), a qual é empregada a cada objeto *Attribute-Exclusive* a fim de identificar se o mesmo está ou não disponível para acesso. Se esta *flag* apresentar o estado lógico verdadeiro, significa que alguma *Premise* já adquiriu acesso a este *Attribute*, caso contrário, o *Attribute* se apresenta livre para uso.

Assim, quando todas as *Premises* de uma *Condition* tiverem obtido exclusividade de acesso aos seus respectivos *Attributes-Exlusives*, a *Condition* é considerada aprovada. Caso contrário, as *Premises* que asseguram a exclusividade de acesso (as quais atribuíram o valor da *flag* para verdadeiro) devem alterar o valor da *flag* para falso, a fim de permitir que outras *Premises* possam acessar o conteúdo dos respectivos *Attributes-Exclusives*.

Para melhor elucidar esta solução, considerar-se-á o simples exemplo explicitado na Figura 100. Este exemplo visa apresentar a resolução de conflitos entre a *Condition.1* e a *Condition.2*, sendo que a *Condition.1* apresenta maior prioridade de execução em relação à *Condition.2*. As prioridades de execução destas *Conditions* são respectivamente 4 e 2, conforme representa a figura.

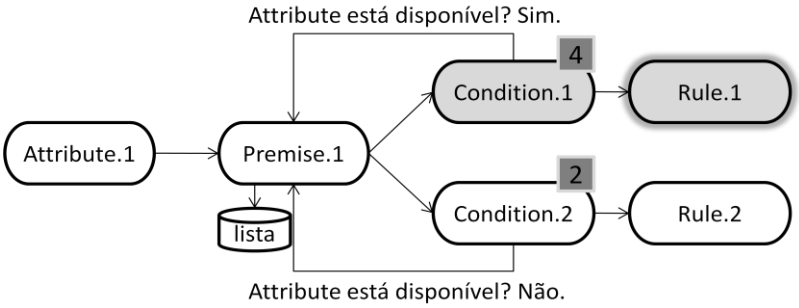


Figura 100: Representação de conflito em ambiente multiprocessado

Estas *Conditions* estão em conflito quando a *Premise.1* é satisfeita e acaba notificando-as sobre o seu estado atual. Como estas *Conditions* estão conectadas apenas a esta *Premise* (a qual referencia o *Attribute.1*), os seus estados também são satisfeitos.

Entretanto, neste modelo, mesmo que as *Conditions* estejam satisfeitas, estas ainda não podem ser aprovadas para execução. Para que sejam aprovadas, estas devem assegurar exclusividade de acesso ao(s) *Attribute*(s) conectado(s) a(s) sua(s) *Premise(s)*. Assim, as *Conditions* contra notificam a(s) respectiva(s) *Premise*(s) para que estas adquiram a exclusividade de acesso e solucionem os conflitos, se houver.

No exemplo em questão, ambas as *Conditions* contra notificam a *Premise.1* e esperam pela confirmação para serem definitivamente aprovadas. Por sua vez, a *Premise.1* adquire exclusividade de acesso ao *Attribute.1* e confirma a aprovação de apenas uma das duas *Conditions* entre as quais está conectada, ou seja, a qual apresenta maior prioridade[58]. Conforme esta política de seleção, a *Condition.1* é aprovada, fazendo com que a *Rule.1* seja imediatamente executada.

Em relação à *Condition.2*, esta é armazenada em uma lista de espera na *Premise.1*. Esta lista é própria a qualquer *Premise*. Desta forma, quando a *Rule.1* for executada e esta execução não gerar um fluxo de notificação que altere o estado lógico da *Premise.1*, a *Condition.2* será extraída desta lista pela *Premise.1* e será notificada para que realize o seu cálculo lógico a fim de confirmar a sua aprovação. Neste caso, o estado lógico da *Condition.2* não foi afetado porque esta é composta apenas pela *Premise.1*. Entretanto, se a *Condition.2* fosse composta por mais de uma *Premise*, o estado lógico desta poderia ter sido afetado, uma vez que as outras *Premises* poderiam ter seus estados lógicos alterados neste intervalo de tempo.

Sucintamente, se uma *Premise* contiver elementos na lista de espera e não for afetada pelo fluxo de notificação gerado por uma regra que a contém, ela extrai a *Condition* de maior prioridade armazenada na lista e notifica-a para que esta realize o seu teste lógico. Se o teste lógico for satisfeito, esta é aprovada, senão, outra *Condition* é extraída da lista e a mesma é notificada, se houver.

A.2 MECANISMO DE ESCALONAMENTO DE REGRAS

Apesar do modelo de resolução de conflitos apresentado neste apêndice e o modelo de resolução de conflitos proposto por Simão solucionarem os conflitos em ambientes

[58] Caso elas apresentassem a mesma prioridade, uma delas poderia ser escolhida aleatoriamente para a aprovação.

multiprocessados, estes ainda não apresentam um mecanismo que permita escalonar apropriadamente a execução das regras para tirar maior proveito da capacidade de processamento dos nós de execução. Neste sentido, esta seção apresenta um mecanismo que realiza de forma efetiva esta tarefa. Este mecanismo será explicado considerando apenas o modelo de resolução de conflito apresentado neste apêndice.

Para que o mecanismo de escalonamento seja integrado ao modelo de resolução de conflito apresentado, se faz necessário alterar a forma pela qual as regras são executadas neste modelo. Assim, ao invés de uma *Condition* aprovada (e com conflitos resolvidos) instigar a execução imediata de uma *Rule*, essa deve repassar o controle da execução da respectiva *Rule* para o componente escalonador de regras, chamado de *Scheduler Rules*.

Desta forma, quando uma *Condition* é aprovada por este modelo descentralizado, a regra conectada não é executada imediatamente, ela é inserida em uma fila de escalonamento de execução no componente *Scheduler Rules* por meio de uma interface bem definida, conforme ilustra a Figura 101. Esta fila é usada para ordenar a execução das regras em ambientes multiprocessados ao atribuir estas regras aos processos ou nós de processamento disponíveis. Estas atribuições ocorrem com respeito à prioridade de execução de cada regra.

Como observado na Figura 101, a fila apresenta uma quantidade pré-estabelecida (e configurável) de posições, as quais condizem aos níveis de prioridades das regras. Nesta fila, cada uma das posições armazena uma referência para outra fila, formando uma espécie de matriz bidimensional.

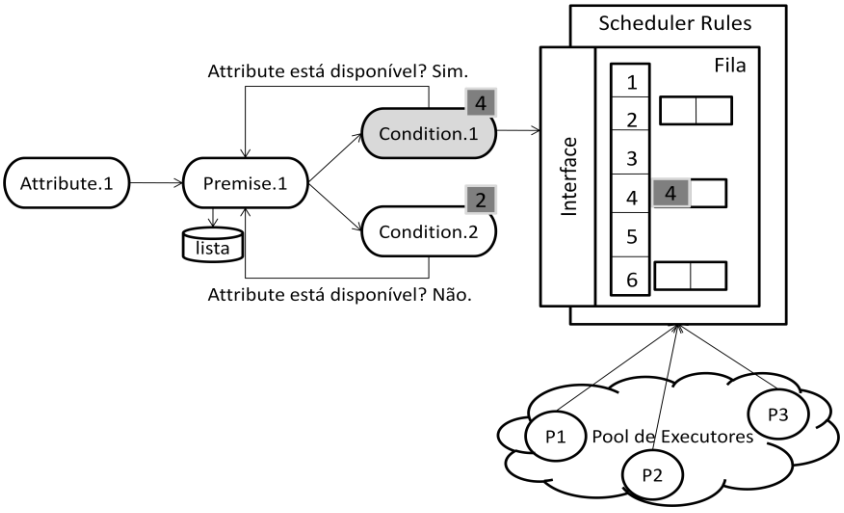


Figura 101: Modelo de escalonamento

De fato, o uso da fila incentiva a execução das regras de maior prioridade. Neste modelo, as regras são executadas pelos nós de processamento que formam o *Pool de Executores*, o qual também apresenta tamanho configurável. O *Pool* é gerenciado pelo

Scheduler Rules (representado por um processo ou *thread* de execução), o qual tem a responsabilidade de atribuir cada regra inserida na fila para execução em algum nó de processamento.

Esta relação entre o *Scheduler Rules* e os elementos do *Pool* segue o modelo de delegação (i.e. mestre-escravo), onde o *Scheduler Rules* é o mestre e os nós de processamento são os escravos (HUGHES e HUGHES, 2003). A sincronização entre estes elementos ocorre de forma colaborativa, no instante em que um escravo termina a sua tarefa (execução de uma regra), ele notifica o *Scheduler Rules* sobre o término, obtendo outra regra para execução (a de mais alta prioridade), se houver.

Para mais bem elucidar o funcionamento do mecanismo de escalonamento de regras, será adotado o mesmo exemplo relativo à aprovação da *Rule.1* apresentado na seção A.1. Para isto, será adotado o esquema apresentado na Figura 101 que exibe a relação entre as respectivas *Conditions* em conflito (i.e. *Condition.1* e *Condition.2*) e o componente *Scheduler Rules*.

Deste modo, após a aprovação da *Condition.1, a Rule.1* é inserida na fila de escalonamento de acordo com a sua prioridade (i.e. prioridade 4). Certamente, esta regra será executada após a execução de outras que apresentam maior prioridade (quando houver), pois o *Scheduler Rules* percorre a fila em ordem decrescente em relação à prioridade.

Ao percorrer a fila, o *Scheduler Rules* delega a execução de uma regra a qualquer nó de processamento ocioso, sendo que, se a mesma regra for aprovada mais de uma vez durante os ciclos de execução, certamente esta não executará todas estas vezes sobre um mesmo nó de processamento.

Neste sentido, considerando a execução da regra conectada a *Condition.1*, esta pode ser delegada a qualquer nó de processamento pelo *Scheduler Rules*. Considerando que a respectiva regra foi delegada para execução sobre o nó de processamento T1, este executará a *Action* desta regra, podendo vir a alterar, por exemplo, o estado do *Attribute.1*. Este evento gera um fluxo de notificações, podendo acarretar na aprovação de outra regra. Se uma nova regra for aprovada por este fluxo, quando ela for executada, ela poderá ser executada por um nó de processamento diferente do nó de processamento T1. Com isto, conclui-se que o nó de processamento que aprova uma regra pode não ser o mesmo que a executa, o que evidencia a independência que este modelo oferece em relação ao contexto de execução das regras.

A.3 REFLEXÕES

De fato, o modelo descentralizado e o próprio mecanismo de escalonamento de regras se mostram (teoricamente) apropriados para o que se propõem. Ainda, o fato de o modelo poder ser integrado ao escalonador de regras, o qual se apresenta como um componente centralizador das regras aprovadas, não o desclassifica como um modelo descentralizado. Na verdade, o modelo resolve os conflitos de forma descentralizada, pois as regras já são aprovadas (inseridas na fila de escalonamento) com os seus respectivos conflitos resolvidos.

APÊNDICE B

INICIATIVAS DE JUNÇÃO DOS CONCEITOS DOS ATUAIS PARADIGMAS

Este apêndice apresenta algumas soluções que unem os conceitos do Paradigma Orientado a Objetos (POO) e dos Sistemas Baseados em Regras (SBR) a fim de confrontá-las com o paradigma proposto.

Portanto, a fim de apresentar a materialização destas junções inter-paradigmas, as quais mesclam a linguagem imperativa C++ com linguagens particulares de seus *shells* de SBR, o presente apêndice está assim organizado: a seção B.1 apresenta o *shell* CLIPS++; a seção B.2 apresenta o *shell* RETE++; a seção B.3 apresenta o *shell* ILOG Rules; e por fim, a seção B.4 apresenta a linguagem R++.

B.1 CLIPS++

O *shell* CLIPS++ (OBERMEYER e MIRANKER, 1994) é uma iniciativa que usa os conceitos do C++ para melhorar as capacidades de programação de SBR. Particularmente, ele permite integrar objetos implementados na linguagem C++ ao *shell* (de SBR) CLIPS, o qual adota o algoritmo de inferência RETE (vide 2.8.3.6).

Atualmente, o CLIPS apresenta uma linguagem parcialmente compatível com os conceitos do POO, uma vez que este *shell* permite que o programador expresse os elementos da base de fatos na forma de objetos por meio da linguagem de extensão COOL. Entretanto, esta linguagem não assegura as mesmas capacidades disponíveis nas linguagens típicas do POO, tal como C++ e Java. Um exemplo desta precariedade da linguagem COOL está na inexistência de suporte à declaração e implementação de métodos em suas classes. Deste modo, os objetos COOL são usados em CLIPS com o mesmo sentido de que as estruturas são usadas em C/C++, isto é, apenas para armazenar estados em um componente modular.

Neste âmbito, para prover maiores capacidades aos programadores dos SBR, os criadores do CLIPS++ evoluíram a linguagem do CLIPS substituindo por completo a linguagem COOL pela linguagem C++ e também, o algoritmo RETE pelo algoritmo LEASP, o qual se apresenta mais eficiente do que o RETE (MIRANKER, BRANT, LOFASO e GADBOIS, 1990).

Desta forma, os programadores que adotam o CLIPS++ passam a implementar os elementos da base de fatos como objetos por meio da linguagem de extensão C++. Assim, em

CLIPS++, os elementos da base de fatos podem guardar estados em atributos para serem referenciados diretamente pelas condições das regras e ainda, estes elementos podem conter métodos para serem acessados pelas ações das regras. Desta maneira, o emprego de objetos típicos do POO pode oferecer alguns serviços antes inacessíveis ou limitados na programação de SBR, tal como o acesso a recursos de baixo-nível por meio de chamadas de sistemas, o gerenciamento de rede e o acesso e manipulação de banco de dados.

Porém, o uso dos objetos implementados em C++ em CLIPS++ não ocorre de forma natural aos programadores C++. Para que estes sejam usados se faz necessário manipular indiretamente estes objetos por meio da linguagem declarativa do CLIPS++. Esta linguagem emprega comandos e sintaxes não familiares aos programadores C++, necessitando assim, que estes dediquem algum tempo para o aprendizado destes novos conceitos. Além disto, o emprego da linguagem declarativa demanda o uso de um pré-compilador para traduzir o código declarativo para código C++, o que enfatiza a perda de naturalidade na programação pelos programadores C++.

B.2 RETE++

Basicamente, o *shell* RETE++[59] (HALEY, 2001) é definido como uma extensão do *shell* Eclipse. O Eclipse é implementado na linguagem ANSI C e seu mecanismo de inferência é baseado em RETE. A linguagem adotada em Eclipse é similar àquela do CLIPS (i.e. com sintaxe similar a linguagem LISP), uma vez que o Eclipse consiste em uma evolução do CLIPS e mesmo de outros *shells*, tais como o OPS5 e o ART. Por sua vez, o RETE++ também evolui o Eclipse ao adotar a sua linguagem declarativa e permitir que o seu código (implementado em C) seja encapsulado dentro de uma taxonomia de classes em C++.

Com o RETE++, os serviços do Eclipse podem estar disponíveis às classes de qualquer programa em C++, uma vez que os elementos da base de fatos são representados por meio de classes em C++. Para isto, as classes C++ que representam os elementos da base de fatos devem ser definidas na linguagem declarativa do *shell*, sendo a versão imperativa propriamente dita é gerada automaticamente por meio de um pré-compilador.

As classes representativas dos elementos da base de fatos em C++ apresentam uma estrutura particular que se faz necessária para que o código C++ mantenha referência ao

[59] RETE++ é um produto registrado pela empresa *The Haley Enterprise*.

algoritmo RETE. Em geral, para representar um elemento da base de fatos se faz necessário estender por herança uma classe particular chamada *ReteInstance* e representar cada tipo de dados em seus atributos por meio de uma representação genérica correspondente à classe *ReteFieldValue*.

De maneira particular, estas duas classes provêm algumas características adicionais ao RETE++. Elas permitem que as classes C++ representativas dos elementos da base de fatos (definidas em linguagem declarativa) sejam manipuladas diretamente por classes C++ definidas manualmente pelo usuário em qualquer outro editor de código. Também, essas classes podem ser manipuladas diretamente dentro do escopo de regras definidas na linguagem declarativa, as quais podem atribuir valores aos atributos de suas instâncias a fim de instigar a execução do algoritmo RETE.

Entretanto, as regras não apresentam toda a flexibilidade desejada para manipular os objetos a partir do seu escopo. Isto se deve porque as regras são implementadas inteiramente em linguagem declarativa, estando indisponível nesta toda flexibilidade existente na linguagem C++.

Assim, o programador que adota o RETE++ somente pode manipular os objetos (elementos da base de fatos) C++ por meio das funções típicas de qualquer SBR, ou seja, a inserção, a remoção e modificação de fatos à base de fatos. Em RETE++, estas funções são mapeadas para instruções naturais do C++, tal como o invocar de um método construtor (inserção), o invocar de um método destrutor (remoção) e atribuição de valores a um atributo de um objeto (modificação).

B.3 ILOG RULES

ILOG RULES[60] (ALBERT, 1994) é um produto comercial que adota as capacidades da linguagem C++ na composição de regras e elementos da base de fatos. Com isto, o ILOG Rules permite ao programador representar os elementos da base de fatos por meio de classes C++ e definir regras para que estas referenciem os objetos destas classes em seus escopos.

Entretanto, assim como ocorre nas outras iniciativas de junção apresentadas, a concepção das regras não é realizada diretamente na linguagem C++, mas em linguagem declarativa. Estas representações declarativas demandam de um pré-compilador para serem

[60] ILOG Rules é um produto registrado pela empresa *ILOG S.A.*

traduzidas para C++, a fim de que manipulem os elementos da base de fatos representados por objetos C++.

Em ILOG Rules, qualquer mudança nos estados dos objetos/elementos da base de fatos afeta o processo de inferência que é baseado no algoritmo RETE. Este processo é afetado mesmo se as mudanças ocorrem por meio da manipulação explícita por um objeto qualquer da aplicação, mesmo este não se tratando de uma regra ou de outro objeto representativo de um elemento da base de fatos.

Entretanto, a modificação de estado de um objeto/elemento da base de fatos não ocorre de forma habitual aos programadores C++, demandando alguns esforços adicionais. Além de implementar a atribuição de valores ao atributo de um objeto/elemento da base de fatos, o programador deve notificar explicitamente o processo de inferência sobre esta atribuição. Deste modo, para programar com C++ em ILOG Rules é necessário despender muita atenção, uma vez que o esquecimento da invocação do método que realiza a notificação ao processo de inferência pode gerar erros de execução.

Geralmente, os programadores que adotam o *shell* ILOG Rules não fazem uso direto da linguagem C++, uma vez que este *shell* permite a composição das regras e elementos da base de fatos por meio de uma ferramenta *wizard*. De fato, esta ferramenta facilita a concepção de regras e a própria manipulação dos elementos da base de fatos, o que colabora para melhorar a produtividade do programador.

No entanto, esta abordagem tem as suas desvantagens. Primeiramente, a ferramenta se trata de um produto comercial com propriedade da ILOG, quando uma ferramenta pública seria a melhor opção. Além disso, pelo fato de que a ferramenta é comercial e suas interfaces e sintaxes da linguagem são proprietárias, isto complica a integração ou portabilidade de uma aplicação construída nesta para outra ferramenta. Ainda, a ferramenta impede que o programador tenha acesso a toda a flexibilidade das linguagens imperativas.

B.4 R++

R++ é um resultado de colaboração entre pesquisadores e desenvolvedores da empresa AT&T (LITMAN, PATEL-SCHNEIDER, MISHRA *et al*, 2002). Particularmente, R++ consiste em uma nova linguagem de programação que estende a linguagem C++, incluindo a esta o conceito de regras dos SBR.

No R++, uma regra é considerada como um novo membro de uma classe, como também são os atributos e métodos. Assim, as regras são definidas de forma similar aos métodos, apresentando as mesmas capacidades que estes no acesso aos estados dos atributos. Entretanto, as regras em R++ diferem dos métodos em termos de sintaxe e na maneira em que seus serviços são invocados.

Para criar uma regra em R++ é necessário adotar uma sintaxe especial, a qual apresenta similaridades à sintaxe pura do C++. No entanto, esta sintaxe apresenta algumas palavras-chaves não compatíveis. Sendo assim, para manter a compatibilidade entre as linguagens R++ e C++ se faz necessário traduzir o código do formato de regra em R++ para código em C++. Esta tradução é executada por um pré-compilador particular à linguagem R++.

Outrossim, a sintaxe da linguagem R++ também se difere da linguagem C++ na declaração dos atributos de um elemento da base de fatos. Para que uma ou várias regras pertinentes sejam avaliadas após a mudança de estado de um atributo relacionado, se faz necessário que este atributo apresente a palavra-chave *monitor* antecedendo a sua declaração (e.g. *monitor* <*tipo de dados*> <*nome do atributo*>).

Por fim, pelo fato de que as regras são avaliadas estritamente de forma reativa a eventos ocorridos nos atributos relacionados, a linguagem R++ não apresenta as capacidades suficientes para ser adotada como uma solução efetiva de SBR. Na verdade, o R++ não adota um algoritmo de inferência inteligente, este apenas reage a eventos sobre os atributos dos elementos da base de fatos, levando a problemas de redundâncias temporais e estruturais. Sendo assim, estas mesmas regras podem ser representadas por expressões causais na forma de *se-então* dentro do escopo de um método.

APÊNDICE C

REGRAS DE FORMAÇÃO OU FORMATION RULES

O conceito de *Formation Rule* (*FR*) ou Regra de Formação foi proposto por Simão (SIMÃO, 2001; SIMÃO, STADZISZ e KÜNZLE, 2003) para permitir a composição de *Rules* específicas (i.e. as quais referenciam instâncias de classes) a partir da representação genérica de uma *Rule* (i.e. que referenciam classes).

O uso das *FRs* é bastante útil quando o conhecimento causal de uma *Rule* é comum para diferentes conjuntos de instâncias de *FBEs*, ou seja, quando várias regras ditas específicas representam a mesma estrutura e apenas se diferenciam nas instâncias referenciadas.

Esta praticidade é evidenciada no cenário do robô-motorista, o qual é composto por vários semáforos e faixas de pedestres, sendo que para cada par destes elementos é necessário compor uma ou mais *Rules* específicas. Este trabalho é tedioso devido à repetição da mesma tarefa, uma vez que o programador deve relacionar manualmente as instâncias de *FBEs* pertinentes a cada *Rule*, sendo que estas apresentam a mesma estrutura semântica.

Neste contexto, as *FR* facilitam este processo de composição das *Rules*. A maneira usada pelas *FR* para criar as *Rules* específicas é explicado a seguir conforme está expresso em (SIMÃO, STADZISZ e KÜNZLE, 2003).

Uma *FR* é mais genérica do que uma *Rule* tradicional. As suas premissas somente analisam se um *FBE* é de uma determinada classe sem considerar, em princípio, os valores dos atributos. Uma *FR* filtra e cria combinações de *FBEs*, onde cada combinação resulta na criação de uma *Rule*. Cada *FR* é composta por uma entidade *Condition-Form* (i.e. Forma ou modelo para criar *Conditions*) e uma entidade *Action-Form* (i.e. Forma ou modelo para criar *Actions*).

Uma *Condition-Form* é conectada a um conjunto de *Premise-Form*, sendo que a função principal de uma *Premise-Form* é filtrar ou encontrar instâncias de *FBE* pertencentes a uma dada classe formando *Premises*. Cada *Premise-Form* especifica uma entidade *Reference* (para um dado *Attribute* de uma classe), uma entidade *Value* e uma entidade *Operator* que permitem a filtragem, mas sem a realização de cálculo lógico sobre estas entidades, uma vez que esta apenas é usada como um modelo para a criação das *Premises* tradicionais.

O papel de uma *Condition-Form* é encontrar combinações entre os elementos filtrados pelas respectivas *Premises-Forms*. Por sua vez, um *Action-Form* consiste em uma seqüência de *Instigations-Forms*, os quais servem como modelo para a criação dos *Instigations*

tradicionais. As *Conditions-Forms* e *Actions-Forms* podem compartilhar *Premises-Forms* e *Instigations-Forms*, evitando redundâncias no processo de filtragem.

Assim, por meio da associação entre as entidades *Forms* criam-se as *Rules* pertinentes. Desta forma, após a criação de uma *Condition*, as *Premises* relacionadas são criadas e conectadas. Na criação de uma *Premise*, esta recebe as entidades *Reference*, *Value* e *Operator* de acordo com as definições na respectiva *Premise-Form*. Com isto, a *Premise* executa o seu cálculo lógico que resulta no valor correspondente ao seu estado lógico inicial.

Na criação de uma *Condition*, após esta apresentar as suas *Premises* conectadas, esta também realiza o seu cálculo lógico a fim de definir seu estado lógico inicial. Na medida em que mais *Conditions* são criadas e necessitam das *Premises* já existentes, mais conexões entre elas são feitas, evitando redundâncias. Uma *Action*, uma vez criada, é conectada a uma seqüência de *Instigations* de acordo com as especificações da respectiva *Action-Form*. Os *Instigation-Forms* somente são usados para criar as *Actions*.

Em suma, uma *FR* aplica conceitos genéricos a fim de criar *Rules* específicas, mantendo as mesmas capacidades de notificação entre os objetos colaboradores. A função principal da *FR* é gerar automaticamente a partir de um modelo várias instâncias de *Rules* que compartilham a semântica deste modelo. A Figura 102 ilustra uma representação hipotética de uma *FR* que pode ser adotada para gerar as instâncias das *Rules* referentes ao cenário do robô-motorista.

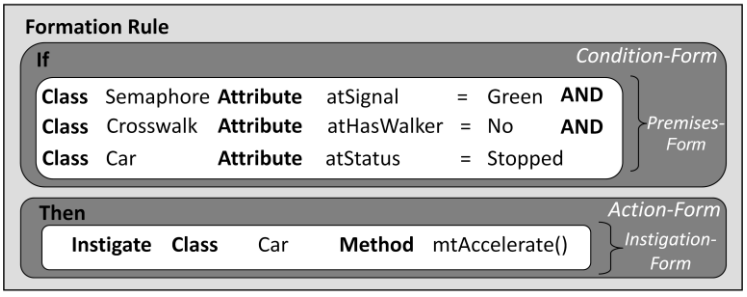


Figura 102: Representação de uma *Formation Rule*

APÊNDICE D

CÓDIGOS-FONTE RELATIVOS AOS ESTUDOS COMPARATIVOS

Este apêndice apresenta os códigos-fonte parciais dos experimentos comparativos realizados sobre a plataforma de computadores pessoais e a plataforma embarcada com o Paradigma Orientado a Notificações (PON) em relação ao Paradigma Imperativo (PI) e o Paradigma Declarativo (PD).

A seção D.1 apresenta os códigos-fonte relativos à plataforma dos computadores pessoais por meio de suas subseções. Deste modo, a subseção D.1.1 apresenta os códigos-fonte referentes aos experimentos envolvendo o PI para os respectivos Primeiro e Segundo Cenário. Por sua vez, a subseção D.1.2 apresenta os códigos-fonte relativos aos experimentos com o PD, mais precisamente com os Sistemas Baseados em Regras (SBR) que inferem sobre o algoritmo RETE, para os respectivos Primeiro e Segundo Cenário. Nestas representações, os códigos-fonte imperativos são apresentados sobre a linguagem C++ e os códigos-fonte declarativos são apresentados sobre a linguagem do *shell* CLIPS.

Por fim, a seção D.2 apresenta os códigos-fonte relativos à plataforma embarcada por meio de suas subseções. Deste modo, a subseção D.2.1 apresenta os códigos-fonte que consiste essencialmente nas expressões causais imperativas e nas *Rules* do PON para os respectivos Primeiro e Segundo Cenário, uma vez que as expressões causais são as mesmas para ambos os cenários.

D.1 PLATAFORMA DOS COMPUTADORES PESSOAIS

D.1.1 Paradigma Orientado a Notificações versus Paradigma Imperativo

D.1.1.1 Primeiro cenário

D.1.1.1.1 Paradigma Imperativo

```
...
void SimpleApplication::initFacts(){

  archerList = new std::vector<Archer*>();
  appleList = new std::vector<Apple*>();

  for( int i = 0; i < 100; i++ ){
    Archer *archerTmp;
    archerTmp = new Archer();
    archerTmp->status = true;
    archerList->push_back(archerTmp);

    Apple *appleTmp;
    appleTmp = new Apple();
    appleTmp->status = true;
    appleTmp->color = "GREEN";
    appleList->push_back(appleTmp);
  }
}

void SimpleApplication::codeApplication(){
  long iterations  = 0;
  clock_t start, finish;
  double duration;

  cout << "Quantidade de Iterações: ";
  cin >> iterations;
  cout << "Porcentagem de Maças Vermelhas: ";
  cin >> percentage;

  start = clock();

  imperativeExperiment();

  finish = clock();
  duration = finish - start;

  cout << "\n Tempo transcorrido: \n";
  cout << (durationNop / CLOCKS_PER_SEC) * 1000;
  cout << " ms" << "\n";
}

void SimpleApplication::imperativeExperiment(){
  for(int i = 0; i < iterations; i++){

    for(int j = 0; j < percentage; j++){
        appleList->at(j)->color = "RED";
    }

    for(int k = 0; k < 100; k++){
        if((appleList->at(k)->color   == "RED") &&
           (appleList->at(k)->status  == true)  &&
           (archerList->at(k)->status == true))
        {
            appleList->at(k)->color = "GREEN";
        }
    }
  }
}
...
```

D.1.1.1.2 Paradigma Orientado a Notificações

```
...
void SimpleApplication::initFacts(){
  archerList = new std::vector<Archer*>();
  appleList = new std::vector<Apple*>();

  for( int i = 0; i < 100; i++ ){
    Archer *archerTmp;
    archerTmp = new Archer();
    archerTmp->atArcherStatus->setValue(true);
    archerList->push_back(archerTmp);

    Apple *appleTmp;
    appleTmp = new Apple();
    appleTmp->atAppleStatus->setValue(true);
    appleTmp->atAppleColor->setValue("GREEN");
    appleList->push_back(appleTmp);
  }
}

void SimpleApplication::initRules(){
 for ( int j = 0; j < 100; j++ ){
   Rule* rlFireApple = new Rule(Condition::CONJUNCTION);
   rlFireApple->addPremise(appleList->at(j)->atAppleColor,    "RED",         Premise::
    EQUAL);
   rlFireApple->addPremise(appleList->at(j)->atAppleStatus,   Boolean::TRUE, Premise::
    EQUAL);
   rlFireApple->addPremise(archerList->at(j)->atArcherStatus, Boolean::TRUE, Premise::
    EQUAL);
   rlFireApple->addInstigation(appleList->at(j)->atAppleColor, "GREEN");
   rlFireApple->end();
 }
}

void SimpleApplication::codeApplication(){
  long iterations  = 0;
  clock_t start, finish;
  double duration;

  cout << "Quantidade de Iterações: ";
  cin >> iterations;
  cout << "Porcentagem de Maças Vermelhas: ";
  cin >> percentage;

  start = clock();

  notificationExperiment();

  finish = clock();
  duration = finish - start;

  cout << "\n Tempo transcorrido: \n";
  cout << (durationNop / CLOCKS_PER_SEC) * 1000;
  cout << " ms" << "\n";
}

void SimpleApplication::notificationExperiment(){
  for(int i = 0; i < iterations; i++){
    for(int j = 0; j < percentage; j++){
        appleList->at(j)->atAppleColor->setValue("RED");
    }
  }
}
...
```

D.1.1.2 Segundo cenário

D.1.1.2.1 Paradigma Imperativo

```
...
void SimpleApplication::initFacts(){

  archerList = new std::vector<Archer*>();
  appleList = new std::vector<Apple*>();

  gun = new Gun();
  gun->status = false;

  for( int i = 0; i < 100; i++ ){
    Archer *archerTmp;
    archerTmp = new Archer();
    archerTmp->status = true;
    archerList->push_back(archerTmp);

    Apple *appleTmp;
    appleTmp = new Apple();
    appleTmp->status = true;
    appleTmp->color = "GREEN";
    appleList->push_back(appleTmp);
  }
}

void SimpleApplication::codeApplication(){
  long iterations  = 0;
  int percentage;
  clock_t start, finish;
  double duration;

  cout << "Quantidade de Iterações: ";
  cin >> iterations;
  cout << "Porcentagem de Maças Vermelhas: ";
  cin >> percentage;

  start = clock();

  imperativeExperiment();

  finish = clock();
  duration = finish - start;

  cout << "\n Tempo transcorrido: \n";
  cout << (durationNop / CLOCKS_PER_SEC) * 1000;
  cout << " ms" << "\n";
}

void SimpleApplication::imperativeExperiment(){

  for(int j = 0; j < percentage; j++){
      appleList->at(j)->color = "RED";
  }

  for(int i = 0; i < iterations; i++){
    gun->status = true;

    for(int j=0; j < 100; j++){
      if( (appleList->at(j)->color   == "RED") &&
          (appleList->at(j)->status  == true)  &&
          (archerList->at(j)->status == true)  &&
          (gun->status == true))
      {
         appleList->at(j)->isCrossed = true;
      }
    }
  }
}
...
```

D.1.1.2.2 Paradigma Orientado a Notificações

```
...
void SimpleApplication::initFacts(){
  archerList = new std::vector<Archer*>();
  appleList = new std::vector<Apple*>();

  gun = new Gun();
  gun->atGunStatus->setValue(false);

  for( int i = 0; i < 100; i++ )
  {
    Archer *archerTmp;
    archerTmp = new Archer();
    archerTmp->atArcherStatus->setValue(true);
    archerList->push_back(archerTmp);

    Apple *appleTmp;
    appleTmp = new Apple();
    appleTmp->atAppleStatus->setValue(true);
    appleTmp->atAppleColor->setValue("GREEN");
    appleList->push_back(appleTmp);
  }
}

void SimpleApplication::initRules(){
 Premise* prGunIsTrue = new Premise(gun->atGunStatus, Boolean::TRUE, Premise::EQUAL);
 for ( int j = 0; j < 100; j++ ){
   Rule* rlFireApple = new Rule(Condition::CONJUNCTION);
   rlFireApple->addPremise(appleList->at(j)->atAppleColor,    "RED",         Premise::
    EQUAL);
   rlFireApple->addPremise(appleList->at(j)->atAppleStatus,   Boolean::TRUE, Premise::
    EQUAL);
   rlFireApple->addPremise(archerList->at(j)->atArcherStatus, Boolean::TRUE, Premise::
    EQUAL);
   rlFireApple->addPremise(prGunIsTrue);
   rlFireApple->addInstigation(appleList->at(j)->atAppleIsCrossed, true);
   rlFireApple->end();
 }
}

void SimpleApplication::codeApplication(){
  long iterations  = 0;
  clock_t start, finish;
  double duration;

  cout << "Quantidade de Iterações: ";
  cin >> iterations;
  cout << "Porcentagem de Maças Vermelhas: ";
  cin >> percentage;

  start = clock();

  notificationExperiment();

  finish = clock();
  duration = finish - start;

  cout << "\n Tempo transcorrido: \n";
  cout << (durationNop / CLOCKS_PER_SEC) * 1000;
  cout << " ms" << "\n";

  getchar();
}

void SimpleApplication::notificationExperiment(){
  for(int j = 0; j < percentage; j++){
    appleList->at(j)->atAppleColor->setValue("RED");
  }

  for(int i = 0; i < iterations; i++){
    gun->atGunStatus->setValue(true, Attribute::RENOTIFY);
  }
}
...
```

D.1.2 Paradigma Orientado a Notificações versus Paradigma Declarativo

D.1.2.1 Primeiro cenário

D.1.2.1.1 Paradigma Declarativo

```
1  //Elementos da base de fatos
2  (deftemplate MAIN::ARCHER
3      (slot status (default TRUE)))
4
5  (deftemplate MAIN::APPLE
6      (slot status (default TRUE))
7      (slot color  (default RED)))
8
9  (deftemplate MAIN::GUN
10     (slot isFired (default TRUE))
11
12 (deftemplate MAIN::UTIL
13     (slot countLoop (type INTEGER) (default 0))
14     (slot totalLoop (type INTEGER)))
15
16 //Regras
17 (defrule MAIN::rlGetInformation
18     (declare (salience 10))
19     =>
20     (printout t "Quantidade de Iterações: ")
21     (bind ?t (read))
22     (assert (APPLE (status TRUE) (color RED)))
23     (assert (ARCHER (status TRUE)))
24     (assert (BIRD (nearOfApple FALSE)))
25     (assert (GUN (isFired TRUE)))
26     (assert (UTIL (totalLoop ?t))))
27
28 defrule MAIN::rl_1
29     ?apple <- (APPLE (status TRUE) (color RED))
30     ?archer <- (ARCHER (status TRUE))
31     ?gun <- (GUN (isFired TRUE))
32     ?bird <- (BIRD (nearOfApple FALSE))
33     =>
34     (modify ?apple (status FALSE))
35     (modify ?gun (isFired FALSE)))
36
37 defrule MAIN::rl_2
38     ?apple <- (APPLE (status FALSE) (color RED))
39     ?util <- (UTIL (countLoop ?cl) (totalLoop ?tl))
40     ?gun <- (GUN (isFired FALSE))
41     =>
42     (bind ?aux (+ ?cl 1))
43     (modify ?util (countLoop ?aux))
44     (modify ?apple (status TRUE))
45     (modify ?gun (isFired TRUE)))
46
47 defrule MAIN::rl_3
48     (declare (salience 20))
49     ?util <- (UTIL (countLoop ?x) (totalLoop ?x))
50     ?gun <- (GUN (isFired TRUE))
51     =>
52     (modify ?gun (isFired NO_MORE)))
53
```

D.1.2.1.2 Paradigma Orientado a Notificações

```
...
//Inicialização dos FBEs
Archer* archer = new Archer();
archer->atArcherStatus->setValue(true);
Apple* apple = new Apple();
apple->atAppleStatus->setValue(true);
Gun* gun = new Gun();
gun->atIsFired->setValue(true);
Bird* bird = new Bird();
bird->atNearOfApple->setValue(false);
Util* util = new Util();
util->atCountLoop->setValue(0);

...

//Inicialização das Rules
Rule* rlHitTheApple = new Rule(Condition::CONJUNCTION);
rlHitTheApple->addPremise(apple->atAppleStatus,   Boolean::TRUE,  Premise::EQUAL);
rlHitTheApple->addPremise(archer->atArcherStatus, Boolean::TRUE,  Premise::EQUAL);
rlHitTheApple->addPremise(gun->atIsFired,         Boolean::TRUE,  Premise::EQUAL);
rlHitTheApple->addPremise(bird->atNearOfApple,    Boolean::FALSE, Premise::EQUAL);
rlHitTheApple->addInstigation(apple->atAppleStatus, false);
rlHitTheApple->addInstigation(gun->atIsFired, false);
rlHitTheApple->end();

RlRestart* rlRestart = new RlRestart(Condition::CONJUNCTION);
rlRestart->addPremise(apple->atAppleStatus, Boolean::FALSE, Premise::EQUAL);
rlRestart->addPremise(gun->atIsFired,       Boolean::FALSE, Premise::EQUAL);
rlRestart->addPremise(util->atCountLoop, util->atCountLoop, Premise::EQUAL);
rlRestart->end();

RlEndApplication* rlEndApplication = new RlEndApplication(Condition::CONJUNCTION);
rlEndApplication->addPremise(util->atCountLoop, util->atTotalLoop, Premise::EQUAL);
rlEndApplication->addPremise(gun->atIsFired,    Boolean::TRUE,     Premise::EQUAL);
rlEndApplication->end();

...

//Implementação Direta das Actions para as Rules Derivadas
void RlRestart::executeRule(){
    aux = util->atCountLoop->getValue();
    aux = aux + 1;
    util->atCountLoop->setValue(aux);
    apple->atAppleStatus->setValue(true);
    gun->atIsFired->setValue(true);
}

void RlEndApplication::executeRule(){
    exit(EXIT_SUCCESS);
}
...

```

D.1.2.2 Segundo cenário

D.1.2.2.1 Paradigma Declarativo

```
1  //Elementos da Base de Fatos
2  (deftemplate Archer
3      (slot id (type integer))
4      (slot status))
5
6  (deftemplate Apple
7      (slot identifier (type integer))
8      (slot status)
9      (slot color))
10
11 (deftemplete Flag
12     (slot flag))
13
14 (deftemplate Util
15     (slot countLoop (type integer))
16     (slot totalLoop (type integer)))
17
18 //Regras
19 (defrule rl_1
20     ?apple <- (Apple (status TRUE) (color RED) (id ?ID))
21     (Archer (status TRUE) (id ?ID))
22     (Flag (flag FALSE))
23 -->
24     (modify ?apple (color GREEN)))
25
26 (defrule rl_2
27     ?apple <- (Apple (status TRUE) (color RED) (id ?lastID))
28     (Archer (status TRUE)  (id ?lastID))
29     ?myFlag <- (Flag (flag FALSE))
30 -->
31     (modify ?myFlag (flag TRUE))
32     (modify ?apple (color GREEN)))
33
34 (defrule rl_3
35     ?apple <- (Apple (status TRUE) (color GREEN) (id ?ID))
36     (Archer (status TRUE) (id ?ID))
37     (Flag (flag TRUE))
38 -->
39     (modify ?apple (color RED)))
40
41 (defrule rl_4
42     ?apple <- (Apple (status TRUE) (color GREEN) (id ?lastID))
43     (Archer (status TRUE) (id ?lastID))
44     ?myFlag <- (Flag (flag TRUE))
45     ?util <- (Util (countLoop ?x))
46 -->
47     (bind ?aux (+ ?x 1))
48     (modify ?apple (color RED))
49     (modify ?util (countLoop ?aux))
50     (modify ?myFlag (flag FALSE)))
51
52 (defrule rl_5
53     (Util (countLoop ?x) (totalLoop ?x))
54 -->
55     (halt))
56
```

D.1.2.2.2 Paradigma Orientado a Notificações

```
...
for ( iteratorArcher = archerList->begin(), iteratorApple = appleList->begin(),
      iteratorArcher != archerList->end(); ++iteratorArcher, ++iteratorApple){
    Rule* rlArcherHitTheTarget = new Rule(Condition::CONJUNCTION);
    rlArcherHitTheTarget->addPremise((*iteratorArcher)->atAppleStatus, Boolean::TRUE,
    Premise::EQUAL);
    rlArcherHitTheTarget->addPremise((*iteratorApple)->atAppleStatus,  Boolean::TRUE,
    Premise::EQUAL);
    rlArcherHitTheTarget->addPremise((*iteratorApple)->atAppleColor,   "RED",
    Premise::EQUAL);
    rlArcherHitTheTarget->addPremise(auxiliar->atFlag,                 Boolean::FALSE,
    Premise::EQUAL);
    rlArcherHitTheTarget->addInstigation((*iteratorApple)->atAppleColor, "GREEN");
    rlArcherHitTheTarget->end();
}

Rule* rlLastRedApple = new Rule(Condition::CONJUNCTION);
rlLastRedApple->addPremise(lastArcher->atAppleStatus, Boolean::TRUE,  Premise::EQUAL);
rlLastRedApple->addPremise(lastApple->atAppleStatus,  Boolean::TRUE,  Premise::EQUAL);
rlLastRedApple->addPremise(lastApple->atAppleColor,   "RED",          Premise::EQUAL);
rlLastRedApple->addPremise(auxiliar->atFlag,          Boolean::FALSE, Premise::EQUAL);
rlLastRedApple->addInstigation(auxiliar->atFlag, true);
rlLastRedApple->addInstigation(lastApple->atAppleColor, "GREEN");
rlLastRedApple->end();

for ( iteratorArcher = archerList->begin(), iteratorApple = appleList->begin();
    iteratorArcher != archerList->end(); ++iteratorArcher, ++iteratorApple){
    Rule* rlArcherEvaluateGreenApple = new Rule(Condition::CONJUNCTION);
    rlArcherEvaluateGreenApple->addPremise((*iteratorArcher)->atAppleStatus, Boolean::
    TRUE, Premise::EQUAL);
    rlArcherEvaluateGreenApple->addPremise((*iteratorApple)->atAppleStatus,  Boolean::
    TRUE, Premise::EQUAL);
    rlArcherEvaluateGreenApple->addPremise((*iteratorApple)->atAppleColor,   "GREEN",
        Premise::EQUAL);
    rlArcherEvaluateGreenApple->addPremise(auxiliar->atFlag,                 Boolean::
    TRUE, Premise::EQUAL);
    rlArcherEvaluateGreenApple->addInstigation((*iteratorApple)->atAppleColor, "RED");
    rlArcherEvaluateGreenApple->end();
}

RlLastGreenApple* rlLastGreenApple = new RlLastGreenApple(Condition::CONJUNCTION);
rlLastGreenApple->addPremise(lastArcher->atAppleStatus, Boolean::TRUE,     Premise::
    EQUAL);
rlLastGreenApple->addPremise(lastApple->atAppleStatus,  Boolean::TRUE,     Premise::
    EQUAL);
rlLastGreenApple->addPremise(lastApple->atAppleColor,   "GREEN",           Premise::
    EQUAL);
rlLastGreenApple->addPremise(auxiliar->atFlag,          Boolean::TRUE,     Premise::
    EQUAL);
rlLastGreenApple->addPremise(util->atCountLoop,         util->atTotalLoop, Premise::
    DIFFERENT);
rlLastGreenApple->end();

void RlLastGreenApple::executeRule(){
    ...
    apple->atAppleColor->setValue("RED");
    auxiliar->atFlag->setValue(false);
    aux = util->atCountLoop->getValue();
    aux = aux + 1;
    util->atCountLoop->setValue(aux);
}

RlFinish* rlFinish = new RlFinish(Condition::CONJUNCTION);
rlFinish->addPremise(util->atCountLoop, util->atTotalLoop, Premise::EQUAL);
rlFinish->end();

void RlFinish::executeRule(){
    ...
    control->finishingTime();
    std::cout << " Finish!!! " << std::endl;
    exit(EXIT_SUCCESS);
}
...
```

D.2 PLATAFORMA EMBARCADA

D.2.1 Paradigma Orientado a Notificações versus Paradigma Imperativo

D.2.1.1 Primeiro e segundo cenário

D.2.1.1.1 Paradigma Imperativo

```
for(int i=0; i < nTotalAndares; i++){
    if((heatPumpList->at(i)->state == HeatPump::OFF)      &&
       (sensorList->at(i)->reading == Sensor::TOO_COLD)){
        heatPumpList->at(i)->state = HeatPump::HEATING;
            ventList->at(i)->state = Vent::CLOSE;
    }

    if((heatPumpList->at(i)->state == HeatPump::OFF)      &&
       (sensorList->at(i)->reading == Sensor::COLD)){
            heatPumpList->at(i)->state = HeatPump::HEATING;
    }

    if((ventList->at(i)->state == Vent::OPEN)             &&
       (heatPumpList->at(i)->state == HeatPump::HEATING)  &&
       (sensorList->at(i)->reading == Sensor::COLD)){
            ventList->at(i)->state = Vent::CLOSE;
    }

    if((ventList->at(i)->state == Vent::OPEN)             &&
       (heatPumpList->at(i)->state == HeatPump::HEATING)  &&
       (sensorList->at(i)->reading == Sensor::TOO_COLD)){
            ventList->at(i)->state = Vent::CLOSE;
    }

    if((heatPumpList->at(i)->state == HeatPump::COOLING)  &&
       (sensorList->at(i)->reading == Sensor::HOT)        &&
       (ventList->at(i)->state == Vent::CLOSE)){
            ventList->at(i)->state = Vent::OPEN;
    }

    if((heatPumpList->at(i)->state == HeatPump::COOLING)  &&
       (sensorList->at(i)->reading == Sensor::TOO_HOT)    &&
       (ventList->at(i)->state == Vent::CLOSE)){
            ventList->at(i)->state = Vent::OPEN;
    }

    if((heatPumpList->at(i)->state == HeatPump::HEATING)  &&
       (sensorList->at(i)->reading == Sensor::UPPER_GUARD)){
            heatPumpList->at(i)->state = HeatPump::OFF;
    }

    if((heatPumpList->at(i)->state == HeatPump::OFF)       &&
       (sensorList->at(i)->reading == Sensor::UPPER_GUARD)&&
       (ventList->at(i)->state == Vent::CLOSE)){
            ventList->at(i)->state = Vent::OPEN;
    }

    if((heatPumpList->at(i)->state == HeatPump::OFF)       &&
       (sensorList->at(i)->reading == Sensor::LOWER_GUARD)&&
       (ventList->at(i)->state == Vent::CLOSE)){
            ventList->at(i)->state = Vent::CLOSE;
    }

```

```
54      if((heatPumpList->at(i)->state == HeatPump::OFF) &&
55         (sensorList->at(i)->reading == Sensor::TOO_HOT)){
56              heatPumpList->at(i)->state = HeatPump::COOLING;
57              ventList->at(i)->state = Vent::OPEN;
58      }
59
60      if((heatPumpList->at(i)->state == HeatPump::OFF) &&
61         (sensorList->at(i)->reading == Sensor::HOT)){
62              heatPumpList->at(i)->state = HeatPump::COOLING;
63      }
64
65      if((heatPumpList->at(i)->state == HeatPump::COOLING)  &&
66         (sensorList->at(i)->reading == Sensor::LOWER_GUARD)){
67              heatPumpList->at(i)->state = HeatPump::OFF;
68      }
69
70      if((heatPumpList->at(i)->state == HeatPump::HEATING)  &&
71         (sensorList->at(i)->reading == Sensor::LOWER_GUARD)&&
72         (ventList->at(i)->state == Vent::CLOSE)){
73              ventList->at(i)->state = Vent::OPEN;
74      }
75
76      if((ventList->at(i)->state == Vent::OPEN)             &&
77         (heatPumpList->at(i)->state == HeatPump::COOLING)  &&
78         (sensorList->at(i)->reading == Sensor::LOWER_GUARD)){
79              ventList->at(i)->state = Vent::CLOSE;
80      }
81
82      if((heatPumpList->at(i)->state == HeatPump::COOLING)  &&
83         (sensorList->at(i)->reading == Sensor::UPPER_GUARD)&&
84         (ventList->at(i)->state == Vent::CLOSE)){
85              ventList->at(i)->state = Vent::OPEN;
86      }
87
88      if((ventList->at(i)->state == Vent::OPEN)             &&
89         (heatPumpList->at(i)->state == HeatPump::HEATING)  &&
90         (sensorList->at(i)->reading == Sensor::UPPER_GUARD)){
91              ventList->at(i)->state = Vent::CLOSE;
92      }
93  }
94
```

D.2.1.1.2 Paradigma Orientado a Notificações

```
...
Premise* premiseTmp;

for(int i = 0; i < nTotalAndares; i++){
  premiseTmp =  new Premise(sensorList->at(i)->atReading, Sensor::UPPER_GUARD, Premise::EQUAL);
  prSensorUPPER_GUARD->push_back(premiseTmp);
  premiseTmp =  new Premise(sensorList->at(i)->atReading, Sensor::LOWER_GUARD, Premise::EQUAL);
  prSensorLOWER_GUARD->push_back(premiseTmp);
}

for(int i = 0; i < nTotalAndares; i++){
  rl_1 = new Rule(Condition::CONJUNCTION);
  rl_1->addPremise(heatPumpList->at(i)->atState, HeatPump::OFF,    Premise::EQUAL);
  rl_1->addPremise(sensorList->at(i)->atReading, Sensor::TOO_COLD, Premise::EQUAL);
  rl_1->addInstigation(heatPumpList->at(i)->atState, HeatPump::HEATING);
  rl_1->addInstigation(ventList->at(i)->atState, Vent::CLOSE);
  rl_1->end();

  rl_2 = new Rule(Condition::CONJUNCTION);
  rl_2->addPremise(heatPumpList->at(i)->atState, HeatPump::OFF, Premise::EQUAL);
  rl_2->addPremise(sensorList->at(i)->atReading, Sensor::COLD,  Premise::EQUAL);
  rl_2->addInstigation(heatPumpList->at(i)->atState, HeatPump::HEATING);
  rl_2->end();

  rl_3 = new Rule(Condition::CONJUNCTION);
  rl_3->addPremise(heatPumpList->at(i)->atState, HeatPump::HEATING, Premise::EQUAL);
  rl_3->addPremise(sensorList->at(i)->atReading, Sensor::COLD,      Premise::EQUAL);
  rl_3->addPremise(ventList->at(i)->atState,     Vent::OPEN,        Premise::EQUAL);
  rl_3->addInstigation(ventList->at(i)->atState, Vent::CLOSE);
  rl_3->end();

  rl_4 = new Rule(Condition::CONJUNCTION);
  rl_4->addPremise(heatPumpList->at(i)->atState, HeatPump::HEATING, Premise::EQUAL);
  rl_4->addPremise(sensorList->at(i)->atReading, Sensor::TOO_COLD,  Premise::EQUAL);
  rl_4->addPremise(ventList->at(i)->atState,     Vent::OPEN,        Premise::EQUAL);
  rl_4->addInstigation(ventList->at(i)->atState, Vent::CLOSE);
  rl_4->end();

  rl_5 = new Rule(Condition::CONJUNCTION);
  rl_5->addPremise(heatPumpList->at(i)->atState, HeatPump::COOLING, Premise::EQUAL);
  rl_5->addPremise(sensorList->at(i)->atReading, Sensor::HOT,       Premise::EQUAL);
  rl_5->addPremise(ventList->at(i)->atState,     Vent::OPEN,        Premise::EQUAL);
  rl_5->addInstigation(ventList->at(i)->atState, Vent::CLOSE);
  rl_5->end();

  rl_6 = new Rule(Condition::CONJUNCTION);
  rl_6->addPremise(heatPumpList->at(i)->atState, HeatPump::COOLING, Premise::EQUAL);
  rl_6->addPremise(sensorList->at(i)->atReading, Sensor::TOO_HOT,   Premise::EQUAL);
  rl_6->addPremise(ventList->at(i)->atState,     Vent::CLOSE,       Premise::EQUAL);
  rl_6->addInstigation(ventList->at(i)->atState, Vent::OPEN);
  rl_6->end();

  rl_7 = new Rule(Condition::CONJUNCTION);
  rl_7->addPremise(heatPumpList->at(i)->atState, HeatPump::HEATING, Premise::EQUAL);
  rl_7->addPremise(prSensorUPPER_GUARD->at(i));
  rl_7->addInstigation(heatPumpList->at(i)->atState, HeatPump::OFF);
  rl_7->end();

  rl_8 = new Rule(Condition::CONJUNCTION);
  rl_8->addPremise(heatPumpList->at(i)->atState, HeatPump::OFF,       Premise::EQUAL);
  rl_8->addPremise(sensorList->at(i)->atReading, Sensor::UPPER_GUARD, Premise::EQUAL);
  rl_8->addPremise(prSensorUPPER_GUARD->at(i));
  rl_8->addInstigation(ventList->at(i)->atState, Vent::OPEN);
  rl_8->end();

  rl_9 = new Rule(Condition::CONJUNCTION);
  rl_9->addPremise(heatPumpList->at(i)->atState, HeatPump::OFF,       Premise::EQUAL);
  rl_9->addPremise(sensorList->at(i)->atReading, Sensor::LOWER_GUARD, Premise::EQUAL);
  rl_9->addPremise(prSensorLOWER_GUARD->at(i));
  rl_9->addInstigation(ventList->at(i)->atState, Vent::CLOSE);
  rl_9->end();
```

```
72
73    rl_10 = new Rule(Condition::CONJUNCTION);
74    rl_10->addPremise(heatPumpList->at(i)->atState, HeatPump::OFF,    Premise::EQUAL);
75    rl_10->addPremise(sensorList->at(i)->atReading, Sensor::TOO_HOT, Premise::EQUAL);
76    rl_10->addInstigation(ventList->at(i)->atState, Vent::OPEN);
77    rl_10->addInstigation(heatPumpList->at(i)->atState, HeatPump::COOLING);
78    rl_10->end();
79
80    rl_11 = new Rule(Condition::CONJUNCTION);
81    rl_11->addPremise(heatPumpList->at(i)->atState, HeatPump::OFF,     Premise::EQUAL);
82    rl_11->addPremise(sensorList->at(i)->atReading, Sensor::HOT,       Premise::EQUAL);
83    rl_11->addInstigation(heatPumpList->at(i)->atState, HeatPump::COOLING);
84    rl_11->end();
85
86    rl_12 = new Rule(Condition::CONJUNCTION);
87    rl_12->addPremise(heatPumpList->at(i)->atState, HeatPump::COOLING, Premise::EQUAL);
88    rl_12->addPremise(prSensorLOWER_GUARD->at(i));
89    rl_12->addInstigation(heatPumpList->at(i)->atState, HeatPump::OFF);
90    rl_12->end();
91
92    rl_13 = new Rule(Condition::CONJUNCTION);
93    rl_13->addPremise(heatPumpList->at(i)->atState, HeatPump::HEATING, Premise::EQUAL);
94    rl_13->addPremise(ventList->at(i)->atState,     Vent::CLOSE,       Premise::EQUAL);
95    rl_13->addPremise(prSensorLOWER_GUARD->at(i));
96    rl_13->addInstigation(ventList->at(i)->atState, Vent::OPEN);
97    rl_13->end();
98
99    rl_14 = new Rule(Condition::CONJUNCTION);
100   rl_14->addPremise(heatPumpList->at(i)->atState, HeatPump::COOLING, Premise::EQUAL);
101   rl_14->addPremise(ventList->at(i)->atState, Vent::OPEN,            Premise::EQUAL);
102   rl_14->addPremise(prSensorLOWER_GUARD->at(i));
103   rl_14->addInstigation(ventList->at(i)->atState, Vent::CLOSE);
104   rl_14->end();
105
106   rl_15 = new Rule(Condition::CONJUNCTION);
107   rl_15->addPremise(heatPumpList->at(i)->atState, HeatPump::COOLING, Premise::EQUAL);
108   rl_15->addPremise(ventList->at(i)->atState, Vent::CLOSE,           Premise::EQUAL);
109   rl_15->addPremise(prSensorUPPER_GUARD->at(i));
110   rl_15->addInstigation(ventList->at(i)->atState, Vent::OPEN);
111   rl_15->end();
112
113   rl_16 = new Rule(Condition::CONJUNCTION);
114   rl_16->addPremise(heatPumpList->at(i)->atState, HeatPump::HEATING, Premise::EQUAL);
115   rl_16->addPremise(ventList->at(i)->atState,     Vent::OPEN,        Premise::EQUAL);
116   rl_16->addPremise(prSensorUPPER_GUARD->at(i));
117   rl_16->addInstigation(ventList->at(i)->atState, Vent::CLOSE);
118   rl_16->end();
119 }
120 ...
```

REFERÊNCIAS BIBLIOGRÁFICAS

ABBAS, A. **Grid Computing: A Practical Guide to Technology and Applications**. Charles River Media, 2000.

AHMED, S. **CORBA Programming Unleashed**. Sams Publishing, 1998.

AHO, A. V., LABORATORIES, B., HILL, M. **Data Structures and Algorithms**, 2001.

ALBERT, P. ILOG Rules, **Embedding Rules in C++: Results and Limits**. OOPSLA´94 - Workshop Embedded Object-Oriented Production Systems (EOOPS), 1994.

ARAUJO, J. M. **Código de Trânsito Brasileiro Anotado** (3ª Edição). Letras Jurídicas, 2007.

BAKO, B., VALETTE, R. **Towards a Descentralization of Rule-Based Systems Controlled by Petri Nets: An Application to FMS,** Fourth International Symposium on Knowledge Engineering, 1990.

BACKUS, J. W., BEEBER, R. J., GOLDBERG, R. **The FORTRAN Automatic Coding System**. Western Joint Computer Conference. Los Angeles, 1957.

BANASZEWSKI, R. F., SIMÃO, J. M., TACLA, C. A., STADZISZ, P. C. **Notification Oriented Paradigm (NOP) – A Software Development Approach based on Artificial Intelligence Concepts**. VI Congress of Logic Applied to Technology - LAPTEC 2007. Santos, 2007.

BANASZEWSKI, R. F. **Advances on Notification Oriented Paradigm**. Qualificação de Mestrado, Universidade Tecnológica Federal do Paraná - UTFPR, Curso de Pós-Graduação em Engenharia Elétrica e Informática Industrial - CPGEI, Curitiba, 2008.

BANERJEE, P., CHANDY, J. A., GUPTA, M., HODGES IV, E. W., HOLM, J. G., LAIN, A. **The Paradigm Compiler for Distributed-Memory Multicomputer**. IEEE Computer, *28* (10), pp. 37 – 47, 1995.

BARR, M. **Programming Embedded Systems in C and C++**. O'Reilly, 1999.

BELLIFEMINE, F., CAIRE, G., GREENWOOD, D. **Developing Multi-Agent Systems with JADE**. John Wiley & Sons, 2007.

BERGIN, T., GIBSON, R. **History of Programming Languages** (2a Edição). Addison Wesley, 1996.

BERSHAD, B., LAZOWSKA, E., LEVY, H. **PRESTO: A System for Object-Oriented Parallel Programming**. Software—Practice & Experience, 18 (8), pp. 713 – 732, 1988.

BLACKBURN, P., BOS, J. E STRIEGNITZ, K. **Learn Prolog Now!** College Publications, ISBN: 1904987176, 2006.

BONANNI, P. **Falcon 4.0: Prima's Official Strategy Guide**. Prima Games, 1999.

BRITO, R. C., TACLA, C. A, SIMÃO, J. M. **Um Sistema Multiagente para Auxiliar nas Decisões Logísticas de Alocação de Petróleo em Portos**. Sistemas de Informação e Gestão do Conhecimento XV SIMPEP, Bauru. Anais do XV SIMPEP, 2008.

BROOKS, R. A. **Cambrian Intelligence: The Early History of the New AI**. MIT Press, Cambridge, MA, 1999.

BROOKSHEAR, J. G. **Coputer Science: An Overview** (9 ed.). Addison Wesley, 2006.

BROWNSTON, L., FARRELL, R., KANT, E., MARTIN, N. **Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming**, Boston, MA, USA: Addison-Wesley Publishing Company, 1985.

CARBONNEAUA, R., LAFRAMBOISEA, K. e VAHIDOV, R. **Application of Machine Learning Techniques for Supply Chain Demand Forecasting**. European Journal of Operational Research, Volume 184, Issue 3, Pages 1140-1154, Elsevier, 2008.

COLMERAUNER, A., ROUSSEL, P. **The Birth of PROLOG**, 1992.

COOK, D., DAS, S. **Smart Environments: Technology, Protocols and Applications**, Wiley-Interscience, 2004.

CORMEN, T. H., LEISERSON, C., RIVEST, R., STEIN, C. **Algoritmos: Teoria e Prática**. (V. D. de Souza, Trad.) Campus, 2002.

CORNEY, J. S., KIBLER, D. F. **Parallel Interpretation of Logic Programs**. Communications of the ACM, 1981.

COWELL-SHAH, W. C. **Nine Language Performance Round-up**. 2004. Disponível em OSNews: http://www.osnews.com/story/5602. Acessado em 24/01/2009.

CRNKOVIC, I., LARSSON, M. **Building Reliable Component-Based Software Systems**. Artech House, 2002.

DAHL, O.-J., NYGAARD, K. ***SIMULA: an ALGOL-based Simulation Language***. Communications of the ACM, *9*, pp. 671 – 678, 1967.

DARPA. **SOAR**. 2009. Disponível em University of Michigan: http://sitemaker.umich.edu/soar/home. Acessado em 2/2/2009.

DAVIS, R., BUCHANAN, B. G., SHORTLIFFE, E. H. **Production Systems as a Representation for a Knowledge-based Consultation Program**. Artificial Intelligence, pp. 15-45, 1977.

DELORIE, D. **DJGPP**. 2003. Disponível em http://www.delorie.com/djgpp/. Acessado em 2/2/2009.

DÍAZ, M., GARRIDO, D., ROMERO, S., RUBIO, B., SOLER, E., TROYA, J. M. **A Component-Based Nuclear Power Plant Simulator Kernel: Research Articles**. Concurrency and Computation: Practice & Experience, 19 (5), pp. 593 – 607, 2007.

DIGITAL Equipment Corporation. **RukeWorks: Knowledge Management.** 2008. Disponível em: www.ruleworks.co.uk. Acessado em 14/10/2008.

DIJKSTRA, E. **Go To Statement Considered Harmful**. Communications of the ACM, pp. 147-148, 1968.

DOORENBOS, R. B. **Production Matching for Large Learning Systems**. Carnegie Mellon University, Computer Science Department, Pittsburgh, PA, USA, 1995.

D'SOUZA, D. F., Wills, A. C. **Objects, Components, and Frameworks with UML: The Catalysis Approach**. Boston, MA, USA: Addison-Wesley Longman Publishing Co, 1998.

EDMONDS, J. **How to Think About Algorithms**. Cambridge University, 2008.

EHRIG, H., et al. **Petri Net Transformations**. Petri Nets: Theory and Applications, por Vedran Kordic. Berlin: I-Tech Education and Publishing, 2008.

ESYSTECH. **eAT55 ARM Evaluation Board**. 2009. Disponível em Página Pessoal do Professor Douglas P. B. Renaux: http://www.lit.cpdtt.cefetpr.br/douglas/disciplinas/manual%20eAT55.pdf. Acessado em 5/2/ 2009.

ESYSTECH. **X Real Time Kernel**. 2008. Disponível em Embedded Systems Technologies: www.esystech.com.br/produtos/XKernel/XKernel.php. Acessado em 31/07/2008.

FAISON, T. **Event-Based Programming: Taking Events to the Limit**. Apress, 2006.

FEIGENBAUM, E. A. **Knowledge Engineering in the 1980**. Stanford University, Department of Computer Science, Stanford/CA, 1982.

FERBER, J. **Multi-agent Systems – An Introduction to Distributed Artificial Intelligence**, Addison-Wesley, 1995.

FERG, S. **Event-Driven Programming: Introduction, Tutorial, History**. 2006. Disponível em http://eventdrivenpgm.sourceforge.net/event_driven_programming.pdf Acessado em 17/12/2008.

FERGUSON, M. **A Conspiração Aquariana**. Rio de Janeiro: Record, 1980.

FERREIRA, A. B. **Novo Dicionário Aurélio**. (5.0). Curitiba, Paraná, Brasil: Positivo, 2006.

FORGY, C. **Initial Assessment of Architectures for Production Systems**. (M. Kaufmann, & C. Palo Alto, Eds.) *Nationall Conference of Artificial Intelligence* , pp. 116-120, 1984.

FORGY, C. **RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem**. *Artificial Intelligence* , *19* (1), pp. 17-37, 1982.

FOX, M.S., BARBUCEANU M. e TEIGEN, R. **Agent-Oriented Supply-Chain Management**. The International Journal of Flexible Manufacturing Systems 12, pp. 165–188, 2000.

FRIEDMAN-HILL, E. **Jess in Action: Rule Based System in Java**. Greenwich, CT, USA: Manning Publications Co, 2003.

GAMMA, E., HELM, R., JOHNSON, R., VLISSIDES, J. **Design Patterns: Elements of Reusable Object-Oriented Software**. Addison Wesley, 1995.

GAO, Z. J., TSAO, J. H., & WU, Y. **Testing and Quality Assurance for Component-Based Software**. Artech House, 2003.

GAS POWERED GAMES. 2007. **Supreme Commander**. Disponível em: http://www.supremecommander.com. Acessado em 26/2/2009.

GASCH, S. **Alan Kay**. 2005. Disponível em http://ei.cs.vt.edu/~history/GASCH.KAY.HTML. Acessado em 5/2/2009.

GIARRATANO, J. e G. RILEY. **Expert Systems: Principles and Practice**. Boston, MA: PWS Publishing, 1993.

GRAHAM, P. **On Lisp: Advanced Techniques for Common Lisp**, 2003.

GUPTA, A. **Parallelism in Production Systems**. PHD Thesis, Camergie Mellon University, Department of Computer Science, 1986.

HALEY, E. **Reasoning about RETE++**. 2001. Disponível em www.haley.com. Acessado em 23/6/2008.

HE, X., CHU, W.C., YANG H., YANG S. J. H. **A New Approach to Verify Rule-Based Systems Using Petri Nets.** Twenty-Third Annual International Computer Software and Applications Conference (IEEE), 1999.

HEATH, S. **Embedded Systems Design** (2ª Edição). Newnes, 2003.

HENNESSY, J. L., PATTERSON, D. A. **Computer Architecture: A Quantitative Approach** (4ª Edição). Morgan Kaufmann, 2007.

HERMENEGILDO, M. V. **An Abstract Machine for Restricted AND-Parallel Execution Logic Programming**. University of Texas, Austin, Texas, 1985.

HUGHES, C., HUGHES, T. **Parallel and Distributed Programming Using C++**. Addison Wesley, 2003.

HYDE, R. **The Art of Assembly Language**. No Starch Press, 2003.

ISHIDA, T. **Metbds and Effectiveness of Parallel Rule Firing**. IEEE Conference on Artificial Intelligence Applications, pp. 166-122, 1990.

JENNINGS, N.R. e WOOLDRIDGE M. **Agent-Oriented Software Engineering**. 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, 1999.

KAISLER, S. **Software Paradigm**. John Wiley & Sons, 2005.

KALP, D., TAMBE, M., GUPTA, A., FORGY, C., NEWELL, A., ACHARYA, A., et al.. **Parallel OPS5 User' s Manual**. Technical Report, Carnegie Mellon University, Computer Science Department, 1988.

KARP, P.D. **The Design Space of Frame Knowledge Representation Systems**, SRI International Artificial Intelligence, 1992.

KAY, A. C. **The Early History of Smalltalk**. In: T. J. Bergin, & R. G. Gibson, History of Programming Languages---*II* (pp. 511 - 598). New York, NY, USA: ACM, 1996.

KAYE, P., LAFLAMME, R., MOSCA, M. **An Introduction to Quantum Computing**. Oxford University Press, 2007.

KELLY, M. A., SEVIORA, R. E. **An Evaluation of DRete on CUPID for OPS5 Matching**. Eleventh International Joint Conference on Artificial Intelligence, pp. 84-90, 1989.

KIMBROUGH, S.O., WU, D.J. e ZHONG, F. **Computers Play the Beer Game: Can Artificial Agents Manage Supply Chains*?*** Decision Support Systems 33, pp. 323–333, 2002.

KING, K. N. **C Programming: A Modern Approach** (2ª Edição), W. W. Norton & Company, ISBN: 0393979504, 2008.

KINNERSLEY, B. **Collected Information On About 2500 Computer Languages, Past and Present**. 2008. Disponível em The Language List: http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm. Acessado em 22/10/2008.

KORDIC, V. **Petri Nets: Theory and Applications**. I-Tech Education and Publishing, 2008.

KOSCIANSKI, A., ROSINHA, L. F., STADZISZ, P. C., KÜNZLE, L. A. **FMS Design and Analysis: Developing a Simulation Environment**. XV International Conference on CAD/CAM, Robotics and Factories of the Future, *II*, p. 25, 1999.

KRUCHTEN, P. **The Rational Unified Process: An Introduction** (3ª Edição). Addison-Wesley, 2003.

KÜNZLE, E. **Implementação e Avaliação de uma Ferramenta Computacional para Análise de Redes de Petri Temporais**. Dissertação de Mestrado, Universidade Tecnológica Federal do Paraná - UTFPR, Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial - CPGEI, Curitiba, PR, 2006.

KUHN, T. S. **The Structure of Scientific Revolutions** (1ª Edição). Chicago: University of Chicago, 1962.

KUHN, T. S. **The Structure of Scientific Revolutions**. Chicago: University of Chicago, 1970.

LADDAD, R. **AspectJ in Action: Practical Aspect-Oriented Programming**. Greenwich, CT, USA: Manning, 2003.

LAFORE, R. **Object-Oriented Programming in C++** (4ª Edição). Sams, 2002.

LAU, S., YAN, J. C. **Parallel Processing and Expert Systems**. NASA Technical Memorandum, NASA, 1991.

LAW, A., MCCOMAS, M. **Simulation of Manufacturing Systems**. In: P. A. Farrington, H. B. Nembhard, D. T. Sturrock, & G. W. Evans (Ed.), 1999 Winter Simulation Conference (WSC'99), I, pp. 56-59. Atlanta, GA, USA, 1999.

LEE, P.-Y., CHENG, A. M. **HAL: A Faster Match Algorithm**. IEEE Transactions on Knowledge and Data Engineering, *14* (5), pp. 1047-1058, 2002.

LITMAN, D., PATEL-SCHNEIDER, P., MISHRA, A., CRAWFORD, J., DVORAK, D. **R++: Adding Path-Based Rules to C++**. IEEE Transactions on Knowledge and Data Engineering , *14* (3), pp. 638-658, 2002.

LOKE, S. **Context-Aware Pervasive Systems: Architectures for a New Breed of Applications**. AUERBACH, 2006.

LUCCA, J., SIMÃO, J. M, BANASZEWSKI, R. F., STADZISZ, P. C., TACLA, C. A. **Interface sobre Meta-modelo de Controle do Simulador ANALYTICE II e suas Utilizações para Comparações de Políticas de Controle de Manufatura**. Seminário de Iniciação Científica e Tecnológica da UTFPR - SICITE 2008, 2008.

MALDOVAN, D. **A Model for Parallel Processing of Production Systems**. IEEE International Conference on Systems, Man, Cybernetics, (pp. 568-573), 1986.

MASTERMAN, M. **The Nature of a Paradigm** (I. Lakatos, & A. Musgrave, Eds.) Criticism and the Growth of Knowledge, pp. 59-89, 1970.

MCCARTHY, J. **Recursive Functions of Symbolic Expressions and their Computation by Machine**. Communications of the Association for Computing Machinery, pp. 184-195, 1960.

MCDERMOTT, J., NEWELL, A., & MOORE, J. **The Efficiency of Certain Production System Implementations**. Pattern-Directed Inference Systems , pp. 155-176, 1978.

MELLE, W. V. **A Domain-Independent Production Rule System for Consultation Programs**. Joint Conference on Artificial Intelligence , pp. 923-925, 1979.

MEYER, J.-J. C. **Agent Languages and Their Relationship to Other Programming Paradigms**, Intelligent Agents V: Agents Theories, Architectures, and Languages (ATAL'98), 1555, pp. 309 – 316, 1998.

MILES, R. **AspectJ Cookbook**. O'Reilly, 2004.

MIRANKER, D. P. **TREAT: A better Match Algorithm for AI Production Systems**. Sixth National Conference on Artificial Intelligence - AAAI'87, (pp. 42-47), 1987.

MIRANKER, D. P. **TREAT: A New and Efficient Match Algorithm for AI Production Systems**, 1989.

MIRANKER, D. P., LOFASO, B. **The Organization and Performance of a TREAT-Based Production System Compiler**. IEEE Transactions on Knowledge and Data Engineering , III (1), pp. 3-10, 1991.

MIRANKER, D. P., BRANT, D. A., LOFASO, B., GADBOIS, D. **On the Performance of Lazy Matching in Production Systems**. 8th National Conference on Artificial Intelligence AAAI (pp. 685-692). AAAI Press / The MIT Press, 1990.

MITCHELL, J. C. **Concepts in Programming Languages**. Cambridge University Press, 2003.

MOORE, G. **Intel Developer Forum (IDF)**. São Francisco, EUA, 2007.

MOORE, G. **Cramming More Components nto Integrated Circuits**. Eletronics Magazine , 1965.

NAYAK, P., GUPTA, A., ROSENBLOOM, P. **Comparison of the Rete and Treat Production Matchers for Soar**. AAAI'88, 1988.

NEIMAN, D. E. **UMASS Parallel OPS5 User's Manual and Technical Report**. Technical Report, University of Massachusetts, Amherst, MA, 1992.

NEWELL, A., SIMON, H. A. **Human Problem Solving**. Englewood Cliffs, NJ, USA: Prentice-Hall, 1972.

NWANA, H.S. **Software Agents: An Overview. Knowledge Engineering Review**, Vol.11, n.3, 1-40. Cambridge University Press, 1996.

OBERMEYER, L., MIRANKER, D. P. **CLIPS++: Embedding CLIPS into C++**. Third CLIPS Conference Proceedings , I, pp. 29-33, 1994.

O'CONNELL, M. J. **Carbon Nanotubes: Properties and Applications**. CRC, 2006.

ODERSKY, M., ALTHERR, P., CREMET, V., EMIR, B., MAN, S. **An Overview of the Scala Programming Language**. Technical Report, Switzerland, 2004.

OLIVEIRA, S., STEWART, D. **Writing Scientific Software: A Guided to Good Style**. Cambridge University Press, 2006.

OXFORD English Dictionary. **Oxford English Dictionary** (2ª Edição) (J. Simpson, & E. Weiner, Eds.) Clarendon Press, 1989.

PENDER, T. **UML Bible**. John Wiley & Sons, 2003.

PIRAMUTHU, S. **Machine Learning for Dynamic Multi-Product Supply Chain Formation**. Expert Systems with Applications, Volume 29, Issue 4, Pages 985-990, Elsevier, 2005.

PIROGOV, V. **The Assembly Programming Master Book**. A-LIST, 2005.

POO, D., KIONG, D. **Object-Oriented Programming and Java**. Springer, 2008.

QUEINNEC, C. **LISP in a Small Pieces**. Cambridge: Cambridge University, 1996.

QUINN, M. J. Designing Efficient Algorithms for Parallel Computers. University of New Hampshire. McGraw-Hill, 1987.

RAO, A. GEORGEFF, M. **BDI Agents: From Theory to Practice**, Proceddings of the First International Conference on MultiAgent Systems (ICMAS'95), p.312-319, 1995.

REILLY, D., REILLY, M. **Java Network Programming and Distributed Computing**. Addison-Wesley, 2002.

RICCIOLI, G. B. **Almagestum Novum**. Bologna, 1651.

ROY, P. V. **Multiparadigm Programming in Mozart/Oz: Second International Conference** (Vol. 3389). Charleroi, Belgium: Springer, 2005.

ROY, P. V., HARIDI, S. **Concepts, Techniques, and Models of Computer Programming**. MIT Press, 2004.

RUSSELL, S. J., NORVIG, P. **Artificial Intelligence: A Modern Approach** (2ª Edição). New Jersey: Prentice Hall, 2002.

SCHILDT, H. **C Completo e Total: Revista e Atualizada** (3ª Edição). Makron Books, 1997.

SCHMIDT, D., STAL, M., ROHNERT, H., BUSCHMANN, F. **Pattern-Oriented Software Architecture: A System of Patterns** (Vol. I). John'Wiley & Sons, 1996.

SCOTT, M. L. **Programming Language Pragmatics**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc, 2000.

SIMÃO, J. M. **Proposta de uma Arquitetura de Controle para Sistemas Flexíveis de Manufatura Baseada em Regras e Agentes**. Dissertação de Mestrado, Universidade Tecnológica Federal do Paraná - UTFPR, Curso de Pós-Graduação em Engenharia Elétrica e Informática Industrial - CPGEI, Curitiba, 2001.

SIMÃO, J. M. **A Contribution to the Development of a HMS Simulation Tool and Proposition of a Meta-Model for Holonic Control**. Doctoral Thesis, UTFPR, CPGEI, Curitiba, Brazil, 2005.

SIMÃO, J.M., STADZISZ, P.C. **An Agent-Oriented Inference Engine applied for Supervisory Control of Automated Manufacturing Systems**. In: J. Abe, & J. Silva

Filho, Advances in Logic, Artificial Intelligence and Robotics (Vol. 85, pp. 234-241). Amsterdam, The Netherlands: IOS Press Books, 2002.

SIMÃO, J.M., STADZISZ, P. C. **Paradigma Orientado a Notificações (PON) – Uma Técnica de Composição e Execução de Software Orientado a Notificações**. Pedido de Patente submetida ao INPI/Brazil (Instituto Nacional de Propriedade Industrial) em 2008 e a Agência de Inovação/UTFPR em 2007. No. INPI Provisório 015080004262.Patente submetida ao INPI. Brasil, 2008.

SIMÃO, J.M., STADZISZ, P.C. **Inference Process Based on Notifications: The Kernel of a Holonic Inference Meta-Model Applied to Control Issues**. IEEE Transactions on Systems, Man and Cybernetics. Part A, Systems and Humans, Vol. 39, Issue 1, 238-250, Digital Object Identifier 10.1109/TSMCA.2008.20066371, 2009a.

SIMÃO, J.M., STADZISZ, P.C., TACLA, C.A. **Holonic Control Meta-Model**. IEEE Transactions on Systems, Man and Cybernetics. Part A, Systems and Humans, Artigo aceito, 2009b.

SIMÃO, J.M., STADZISZ, P.C., BANASZEWSKI, R. F., TACLA, C. A. **Holonic Manufacturing Execution Systems for Customised and Agile Production: Manufacturing Plant Simulation**. V Congresso Brasileiro de Engenharia de Fabricação, 2008.

SIMÃO, J.M., STADZISZ, P.C., MOREL, G. **Manufacturing Execution System for Customized Production**. Journal of Material Processing Technology , 179 (1-3), 268, 2006.

SIMÃO, J.M., FABRO, J., STADZISZ, P.C., ARRUDA, L., ISHIMATSU, S. **An Agent-Oriented Fuzzy Inference Engine**. VI Simpósio Brasileiro de Automação Inteligente. Bauru, 2003.

SIMÃO, J.M., STADZISZ, P.C., KÜNZLE, L. **Rule and Agent-oriented Architecture to Discrete Control Applied as Petri Net Player**. (G. Torres, J. Abe, M. Mucheroni, & C. P.E., Eds.) 4th Congress of Logic Applied to Technology - LAPTEC 2003 , 101, p. 217, 2003.

SIMCHI-LEVI, D., KAMINSKY, P., SIMCHI-LEVI, E. **Designing and Managing the Supply Chain**, McGraw-Hill Higher Education, 2000.

STADZISZ, P.C., RENAUX, D. **Software Embarcado**. In: XIV Escola Regional de Informática SBC (Vol. 1, pp. 107-155). Guarapuava, 2007.

STOLFO, S.J. **Five Parallel Algorithms for Production Systems Execution on DADO Machine**. Proceedings National Conference on Artificial Intelligence (AAAI-84) , pp. 300-307, 1984.

STRADER, T.J., LIN, F.R. e SHAW, M.J. **Simulation of Order Fulfillment in Divergent Assembly Supply Chains**, Journal of Artificial Societies and Social Simulation 1, 1998.

SYSTEMS, I. **IAR Embedded Workbench**. 2008. Disponível em www.iar.se. Acessado em 13/9/2008.

TAMBE, M.S. **Eliminating Combinatorics from Production Match**. Universidade de Carnegie Mellon, Departamento de Ciência da Computação, Pittsburgh, PA, USA, 1991.

TANENBAUM, A.S. **Modern Operating Systems** (2ª Edição). Prentice Hall, 2001.

TANENBAUM, A. S. e STEEN, M. V. **Distributed Systems: Principles and Paradigms** (2ª Edição). Prentice Hall, 2006.

TIOBE. TIOBE **Programming Community Index for October 2008**. 2008. Disponível em TIOBE Software - The Coding Standards Company: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html. Acessado em 23/11/2008.

TURING, A. M. **On Computable Numbers, with an Application to the Entscheidungsproblem**. London Mathmatical Society , pp. 230-265, 1936.

WANG, Y., HANSON, E. N. **A Performance Comparison of the RETE and TREAT Algorithms for Testing Database Rule Conditions**. IEEE, 1992.

WATT, D. **Programming Language Design Concepts**. J. Willey & Sons, 2004.

WEISS, G. **Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence**. MIT Press, Massachusetts, 1999.

WILSCY, M., PARAMESWARAN, N. **A Distributed Production System for Problem Solving**. IEEE, 1992.

WILSON, G. V., LU, P. **Parallel Programming Using C++**. The MIT Press, 1996.

WILSON, M. **Extended STL: Collections and Iterators**. Addison Wesley, 2007.

WOLF, W. **High-Performance Embedded Computing: Architectures, Applications, and Methodologies**. Morgan Kaufmann, 2007.

WOOLDRIDGE, M. e JENNINGS, N., **Intelligent Agents: Theory and Practice**. The Knowledge Engineering Review, 10 (2), 115-152, 1995.

RESUMO:

Os atuais paradigmas de programação de *software*, mais precisamente o Paradigma Imperativo (PI) e o Paradigma Declarativo (PD), apresentam deficiências que afetam: (a) o desempenho das aplicações; (b) a facilidade de programação no PI ou as funcionalidades na programação no PD; e (c) a obtenção de "desacoplamento" entre os módulos de *software*, o que dificulta seus reaproveitamentos bem como o uso de multiprocessamento.

Com o objetivo de prover uma solução para este conjunto de deficiências, foi desenvolvido o Paradigma Orientado a Notificações (PON) derivado de uma teoria de controle e inferência precedente. O PON se inspira nos conceitos do PI (e.g. objetos) e do PD (e.g base de fatos e regras) oferecendo maiores vantagens do que estes paradigmas. Basicamente, o PON usa objetos para tratar de fatos e regras na forma de composições de outros objetos menores. Todos estes objetos apresentam características comportamentais de certa autonomia, independência, reatividade e colaboração por meio de notificações pontuais. Estas características são voltadas à realização participativa das funcionalidades do *software* por esses objetos.

Em suma, este trabalho apresenta o PON classificando-o como um paradigma efetivo, disserta sobre as qualidades e vantagens dele e a quais contextos se aplicam e principalmente, o compara com os paradigmas vigentes. Estas comparações se dão por meio de explicações e estudos práticos e teóricos, onde a eficiência de execução é salientada. O trabalho conclui sobre as vantagens e pertinências do PON, bem como abre perspectivas de pesquisa sobre ele, onde o multiprocessamento é um exemplo.

PALAVRAS-CHAVE

Paradigmas de Programação

Paradigma Orientado a Notificações

Mecanismo de Inferência por Notificações

ÁREA/SUB-ÁREA DE CONHECIMENTO

1.03.00.00-7 Ciência da Computação

1.03.01.01-1 Computabilidade e Modelos de Computação

2009

Nº 500

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Administração
Baixar livros de Agronomia
Baixar livros de Arquitetura
Baixar livros de Artes
Baixar livros de Astronomia
Baixar livros de Biologia Geral
Baixar livros de Ciência da Computação
Baixar livros de Ciência da Informação
Baixar livros de Ciência Política
Baixar livros de Ciências da Saúde
Baixar livros de Comunicação
Baixar livros do Conselho Nacional de Educação - CNE
Baixar livros de Defesa civil
Baixar livros de Direito
Baixar livros de Direitos humanos
Baixar livros de Economia
Baixar livros de Economia Doméstica
Baixar livros de Educação
Baixar livros de Educação - Trânsito
Baixar livros de Educação Física
Baixar livros de Engenharia Aeroespacial
Baixar livros de Farmácia
Baixar livros de Filosofia
Baixar livros de Física
Baixar livros de Geociências
Baixar livros de Geografia
Baixar livros de História
Baixar livros de Línguas

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo