UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA
E INFORMÁTICA INDUSTRIAL

LEONARDO ARAUJO SANTOS

LINGUAGEM E COMPILADOR PARA O PARADIGMA ORIENTADO A NOTIFICAÇÕES: AVANÇOS PARA FACILITAR A CODIFICAÇÃO E SUA VALIDAÇÃO EM UMA APLICAÇÃO DE CONTROLE DE FUTEBOL DE ROBÔS

DISSERTAÇÃO

CURITIBA

2017

LEONARDO ARAUJO SANTOS

LINGUAGEM E COMPILADOR PARA O PARADIGMA ORIENTADO A NOTIFICAÇÕES: AVANÇOS PARA FACILITAR A CODIFICAÇÃO E SUA VALIDAÇÃO EM UMA APLICAÇÃO DE CONTROLE DE FUTEBOL DE ROBÔS

Dissertação apresentada ao Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial da Universidade Tecnológica Federal do Paraná como requisito parcial para obtenção do grau de "Mestre em Ciências" - Área de Concentração: Engenharia de Computação.

Orientador: Prof. Dr. Jean Marcelo Simão

Co-orientador: Prof. Dr. João Alberto Fabro

CURITIBA

2017

Dados Internacionais de Catalogação na Publicação

S237L
2017

Santos, Leonardo Araujo
Linguagem e compilador para o paradigma orientado a notificações: avanços para facilitar a codificação e sua validação em uma aplicação de controle de futebol de robôs / Leonardo Araujo Santos.-- 2017.
274 p. : il. ; 30 cm

Texto em português, com resumo em inglês
Disponível também via World Wide Web
Dissertação (Mestrado) - Universidade Tecnológica Federal do Paraná. Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial, Curitiba, 2017
Bibliografia: p. 128-135

1. Paradigma orientado a notificações. 2. Futebol – Simulação por computador. 3. Robótica. 4. Robôs. 5. Framework (Programa de computador). 6. C++ (Linguagem de programação de computador). 7. Engenharia elétrica – Dissertações. I. Simão, Jean Marcelo. II. Fabro, João Alberto. III. Universidade Tecnológica Federal do Paraná. Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial. IV. Título.

CDD: Ed. 22 – 621.3

Biblioteca Central da UTFPR, Câmpus Curitiba

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
Câmpus Curitiba

Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial

Título da Dissertação Nº. ____

Linguagem e Compilador para o Paradigma orientado a Notificações: Avanços para Facilitar a codificação e sua Validação em uma Aplicação de Controle de Futebol de Robôs

por

Leonardo Araujo Santos

Orientador: Prof. Dr. Jean Marcelo Simão (UTFPR)
Coorientador: Prof. Dr. João Alberto Fabro (UTFPR)

Esta dissertação foi apresentada como requisito parcial à obtenção do grau de MESTRE EM CIÊNCIAS – Área de Concentração: ENGENHARIA DE COMPUTAÇÃO do Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial – CPGEI – da Universidade Tecnológica Federal do Paraná – UTFPR, às 15:30h do dia 31 de março 2017. O trabalho foi aprovado pela Banca Examinadora, composta pelos professores doutores:

Prof. Dr. Paulo Cézar Stadzisz
(Presidente – UTFPR)

Prof. Dr. André Murbach Maidl
(PUC-PR)

Prof. Dr. André Schneider de Oliveira
(UTFPR)

Prof. Dr. Robson Ribeiro Linhares
(UTFPR)

Visto da coordenação:

Prof. Jean Carlos Cardozo da Silva, Dr.
(Coordenador do CPGEI)

AGRADECIMENTO

Agradeço, primeiramente, a Deus, por ter me concedido vida e saúde para concluir este trabalho.

Agradeço à minha esposa, Jaqueline Prudencio Santos, por todo o carinho e incentivo que tanto me ajudaram a alcançar este objetivo.

Agradeço aos meus pais Henrique Guimarães dos Santos e Ione Araujo Santos e minha irmã Ingrid Santos Andor, por todo amor ao longo de toda a minha vida.

Agradeço aos professores Jean Marcelo Simão e João Alberto Fabro pelos conselhos e orientação e co-orientação durante este período de mestrado.

Agradeço aos professores Paulo Cézar Stadzisz, André Murbach Maidl, André Schneider de Oliveira e Robson Linhares, membros da banca, pela disponibilidade em avaliar este trabalho e apresentar sugestões e correções pertinentes.

RESUMO

Araujo Santos, Leonardo. LINGUAGEM E COMPILADOR PARA O PARADIGMA ORIENTADO A NOTIFICAÇÕES: AVANÇOS PARA FACILITAR A CODIFICAÇÃO E SUA VALIDAÇÃO EM UMA APLICAÇÃO DE CONTROLE DE FUTEBOL DE ROBÔS. 293 f. Dissertação – Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial, Universidade Tecnológica Federal do Paraná. Curitiba, 2017.

As técnicas de programação baseadas no Paradigma Imperativo (PI) e Paradigma Declarativo (PD) apresentam limitações como redundâncias e acoplamentos, os quais podem prejudicar o desenvolvimento e execução de *softwares*. Visando solucionar algumas das deficiências desses paradigmas de programação surgiu o Paradigma Orientado a Notificações (PON), o qual apresenta uma nova forma de realizar avaliações lógico-causais. Isto é realizado através de entidades computacionais de pequeno porte, reativas e desacopladas que colaboram por meio de notificações pontuais. O PON foi originalmente materializado na forma de um *Framework*, implementado na linguagem de programação C++. Este foi projetado de forma a fornecer interfaces de alto nível que facilitassem o desenvolvimento de aplicações PON. Entretanto, internamente, o *Framework* usa estruturas de dados que induzem a uma sobrecarga de processamento computacional e consequente prejuízo ao desempenho das aplicações. Visando solucionar esse problema, uma linguagem de programação específica para o PON, nomeada LingPON, e respectivo compilador, foram criados recentemente. Entretanto, em um primeiro esforço, apenas algumas poucas e simples aplicações PON foram criadas utilizando a LingPON, fato este que não permite sua consolidação mais efetiva. Isso se deve, principalmente, a algumas limitações que existem na atual versão da linguagem, as quais dificultam o desenvolvimento de aplicações complexas que solucionem problemas realísticos. Neste sentido, este trabalho propõe uma nova versão da LingPON (versão 1.2), baseada em agregações de entidades, na qual é possível criar aplicações complexas de forma mais fácil e direta. Os avanços são validados por meio do desenvolvimento de um *software*, utilizando a LingPON (versão 1.0 e 1.2), para algo reconhecidamente complexo. O *software* em questão trata do controle para partidas de futebol de robôs (Robocup). Ao final, o *software* desenvolvido utilizando o LingPON é comparado quantitativamente e qualitativamente com um *software* equivalente desenvolvido o utilizando o *Framework* PON 2.0 e outro *software* equivalente desenvolvido com o Paradigma Orientado a Objetos utilizando a linguagem de programação C++ (este programado por outrem). Os resultados obtidos mostraram que ao utilizar a nova versão da LingPON, apresentada neste trabalho, é possível desenvolver aplicações PON de forma mais simples e com menor esforço. Ademais, as aplicações PON desenvolvidas apresentaram maior facilidade de manutenção quando comparadas a aplicação PI. Esses resultados obtidos, com o desenvolvimento de uma aplicação conhecidamente complexa utilizando a nova versão da LingPON, vêm ao encontro de colaborar na demonstração de propriedades previstas na própria teoria do PON.

Palavras-chave: Paradigma Orientado a Notificações, LingPON, Futebol de Robôs

ABSTRACT

Araujo Santos, Leonardo. LANGUAGE AND COMPILER FOR THE NOTIFICATION-ORIENTED PARADIGM: ADVANCES TO FACILITATE CODING AND ITS VALIDATION IN A ROBOT CONTROL SOCCER APPLICATION. 293 f. Dissertação – Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial, Universidade Tecnológica Federal do Paraná. Curitiba, 2017.

Programming techniques based on the Imperative Paradigm (PI) and Declarative Paradigm (PD) have limitations such as redundancies and couplings, which may hamper the development and execution of software. In order to solve some of the deficiencies of these programming paradigms, the Notification Oriented Paradigm (NOP) has emerged, which presents a new way of performing logical/cause revisions. This is accomplished through small, reactive and decoupled computational entities that collaborate through timely notifications. The NOP was originally materialized in the form of a Framework, implemented in the C++ programming language. It is designed to provide high-level interfaces that facilitate the development of NOP applications. However, internally the Framework has data structures that induce an overhead of computational processing and consequent damage to the performance of the applications. In order to address this problem, a specific programming language for the NOP, named LingPON, and its compiler have been created recently. However, only a few, simple, NOP applications were created using LingPON, which does not allow its more effective consolidation. This is essentially for some limitations that exist in the current language version, as well as difficulties in developing complex applications that solve real problems. In this sense, this work proposes a new version of LingPON (version 1.2), with new features such as aggregations of entities, allowing an easier and more direct development of complex applications. The advances are validated by developing a more complex software using LingPON (version 1.0 and 1.2). The software in question deals with the autonomous control for soccer playing robots (Robocup). In the end, software developed using LingPON is compared quantitatively and qualitatively with equivalent software developed using the NOP 2.0 Framework and other equivalent software developed with the Object Oriented Paradigm using a C++ programming language (this one programmed by others). The results show that the use of the new version of LingPON, presented here, allows an easier development of NOP applications. Also, that the developed application was easier to maintain when compared to the PI application. These results, with the development of a known complex application using a new version of LingPON, allows the demonstration of the properties expected in the NOP theory itself.

Keywords: Notification Oriented Paradigm, LingPON, Robots Soccer, RoboCup

LISTA DE FIGURAS

Figura 1: Exemplo de uma *Rule*. 19

Figura 2: Atividades do método de pesquisa. 29

Figura 3: Taxonomia de paradigmas de programação com o Paradigma Orientado a Notificações em destaque. Extraído de [Xavier 2014]. . . . 32

Figura 4: Evento, condição detectada e notificação [Faison 2006]. 36

Figura 5: Paradigma Orientado a Eventos adaptado de [Hansen e Fossum 2010]. 36

Figura 6: Arquitetura de um Sistema Baseado em Regras (SBR). 37

Figura 7: Inferência por notificações [Simão et al. 2014]. 41

Figura 8: Modelo Centralizado de Resolução de Conflitos [Banaszewski 2009]. 43

Figura 9: Cálculo assintótico do mecanismo de notificações [Banaszewski 2009] 46

Figura 10: Impacto nas alterações de estado de *Attributes* ativos (adaptado de [Ronszcka 2012]). 47

Figura 11: Impacto nas alterações es estado de *Attributes* impertinentes (adapatado de [Ronszcka 2012]) 48

Figura 12: Exemplo de reativação de uma entidade desativada (adapatado de [Ronszcka 2012]) 48

Figura 13: Estrutura do *Framework* PON [Linhares et al. 2011] 51

Figura 14: Diagrama de Classes do pacote *Core* [Linhares et al. 2011] 52

Figura 15: Diagrama de Classes dos subpacotes *Attributes* e *Conditions* [Linhares et al. 2011] 53

Figura 16: Diagrama de classes do procedimento inicial de uma aplicação PON [Ronszcka 2012] 54

Figura 17: Diagrama de atividades UML do compilador PON. 63

Figura 18: Exemplo de análise semântica [Ferreira 2016] 64

Figura 19: Diagrama de classes utilizada pelo compilador PON para representar entidades PON [Ferreira 2016]. 65

Figura 20: Dimensões do campo oficial da categoria SSL, em milímetros. . . . 68

Figura 21: Sistema *RoboCup* SSL [Yoon 2015] 69

Figura 22: Interface gráfica da aplicação *grSim Simulator*. 70

Figura 23: Interface gráfica da aplicação *Referee Box*. 71

Figura 24: Diagrama representativo das aplicações que compõem o ambiente simulado Robocup SSL. 72
Figura 25: Fluxo de compilação de código utilizando o pré-compilador PON. . 82
Figura 26: Diagrama de atividades simplificado de uma partida de futebol de robôs. 92
Figura 27: Diagrama de atividades para escolha do batedor de penalidade máxima. 94
Figura 28: Diagrama de classes simplificado da solução desenvolvida em PI. . 95
Figura 29: Diagrama de classes do *software* de controle para partida de futebol de robôs em PON. 102
Figura 30: Gráfico linhas de código-fonte para cada uma das soluções apresentadas. 109
Figura 31: Gráfico quantidade de *tokens* presentes no código-fonte de cada uma das soluções apresentadas. 110
Figura 32: Campo de jogo (dimensões em milímetros). 146
Figura 33: Dimensões máxima do robô (em milímetros). 147
Figura 34: Posição do marcador central (azul ou amarelo), e dos 4 marcadores laterais coloridos (magenta ou verde claro). 147
Figura 35: Marcadores coloridos utilizados para a identificação. 148
Figura 36: Esquema geral de funcionamento do Futebol de Robôs da categoria SSL. Fonte: http://wiki.robcup.org/Small_Size_League 148
Figura 37: Interface do programa SSL Referee Box, para envio de comandos do árbitro. 149
Figura 38: Elementos do diagrama de objetos PON [Kossoski et al. 2014]. . . 159
Figura 39: Diagrama de objetos PON das *Rules* rlRobotMoveX, rlRobotMoveY, rlAngleMove e rlBallFar. 161
Figura 40: Diagrama de objetos PON da *Rule* rlStartTargetToBall. 161
Figura 41: Diagrama de objetos PON das *Rules* rlStartFreePartner e rlStartFreePartnerPass. 162
Figura 42: Diagrama de objetos PON das *Rules* rlStartNoFreePartner e rlStartNoFreePartnerKick. 162
Figura 43: Diagrama de objetos PON das *Rules* rlStartEnemyPositionKick e rlStartEnemyFieldKick 163
Figura 44: Diagrama de objetos PON das *Rules* rlGoalkeeperStopCloseGoal e rlGoalKeeperStopFarGoal. 163
Figura 45: Diagrama de objetos PON das *Rules* rlGoalkeeperStartInsideAreaClosestBall e rlGoalkeeperStartInsideAreaClosestBallKick. 164

Figura 46: Diagrama de objetos PON das *Rules* rlGoalkeeperStartInsideArea e rlGoalkeeperStartOutsideArea. 164
Figura 47: Diagrama de objetos PON das *Rules* rlGoalkeeperBluePenaltyYellow e rlGoalkeeperYellowPenaltyBlue. 165
Figura 48: Diagrama de objetos PON das *Rules* rlDefenderLeftStopBallFar e rlDefenderLeftStopBallClose. 165
Figura 49: Diagrama de objetos PON das *Rules* rlDefenderLeftBlueDirectKickBlue e rlDefenderLeftYellowDirectKickYellow. 166
Figura 50: Diagrama de objetos PON das *Rules* rlDefenderLeftBlueIndirectKickBlue e rlDefenderLeftYellowIndirectKickYellow. 166
Figura 51: Diagrama de objetos PON das *Rules* rlDefenderLeftBluePenaltyYellow e rlDefenderLeftYellowPenaltyBlue. 166
Figura 52: Diagrama de objetos PON da *Rule* rlDefenderLefStartBallNotClose. 167
Figura 53: Diagrama de objetos PON das *Rules* rlDefenderRightStopBallFar e rlDefenderRightStopBallClose. 167
Figura 54: Diagrama de objetos PON das *Rules* rlDefenderRightBlueDirectKickBlue e rlDefenderRightYellowDirectKickYellow. 167
Figura 55: Diagrama de objetos PON das *Rules* rlDefenderRightBlueIndirectKickBlue e rlDefenderRightYellowIndirectKickYellow. 168
Figura 56: Diagrama de objetos PON das *Rules* rlDefenderRightBluePenaltyYellow e rlDefenderRightYellowPenaltyBlue. 168
Figura 57: Diagrama de objetos PON da *Rule* rlDefenderRightStartBallNotClose. 168
Figura 58: Diagrama de objetos PON da *Rule* rlMidfieldOnlyStop. 169
Figura 59: Diagrama de objetos PON das *Rules* rlMidfieldOnlyBlueKickoff e rlMidfieldOnlyYellowKickoff. 169
Figura 60: Diagrama de objetos PON das *Rules* rlMidfieldOnlyBlueReadyKickoffBlue e rlMidfieldOnlyYellowReadyKickoffYellow. 169
Figura 61: Diagrama de objetos PON das *Rules* rlMidfieldOnlyBlueDirectKick e rlMidfieldOnlyYellowDirectKick. 170
Figura 62: Diagrama de objetos PON das *Rules* rlMidfieldOnlyBlueIndirectKick e rlMidfieldOnlyYellowIndirectKick. 170
Figura 63: Diagrama de objetos PON das *Rules* rlMidfieldOnlyBluePenaltyBlue e rlMidfieldOnlyYellowPenaltyYellow. 170
Figura 64: Diagrama de objetos PON das *Rules* rlMidfieldOnlyBluePenaltyYellow e rlMidfieldOnlyYellowPenaltyBlue. 171
Figura 65: Diagrama de objetos PON das *Rules* rlMidfieldOnlyBlueReadyPenaltyBlue e rlMidfieldOnlyYellowReadyPenaltyYellow. 171

Figura 66: Diagrama de objetos PON das *Rules* rlStrikerLeftStopTeamLeft e rlStrikerLeftStopTeamRight. 172
Figura 67: Diagrama de objetos PON das *Rules* rlStrikerLeftBlueDirectKick e rlStrikerLeftYellowDirectKick. 172
Figura 68: Diagrama de objetos PON das *Rules* rlStrikerLeftBlueIndirectKick e rlStrikerLeftYellowIndirectKick. 173
Figura 69: Diagrama de objetos PON das *Rules* rlStrikerLeftBluePenaltyBlue e rlStrikerLeftYellowPenaltyYellow. 173
Figura 70: Diagrama de objetos PON das *Rules* rlStrikerLeftBluePenaltyYellow e rlStrikerLeftYellowPenaltyBlue. 173
Figura 71: Diagrama de objetos PON da *Rule* rlStrikerLeftStartBallNotClose. 174
Figura 72: Diagrama de objetos PON das *Rules* rlStrikerRightStopTeamLeft e rlStrikerRightStopTeamRight. 174
Figura 73: Diagrama de objetos PON das *Rules* rlStrikerRightBlueDirectKick e rlStrikerRightYellowDirectKick. 174
Figura 74: Diagrama de objetos PON das *Rules* rlStrikerRightBlueIndirectKick e rlStrikerRightYellowIndirectKick. 175
Figura 75: Diagrama de objetos PON das *Rules* rlStrikerRightBluePenaltyBlue e rlStrikerRightYellowPenaltyYellow. 175
Figura 76: Diagrama de objetos PON das *Rules* rlStrikerRightBluePenaltyYellow e rlStrikerRightYellowPenaltyBlue. 175
Figura 77: Diagrama de objetos PON das *Rules* rlStrikerRightStartBallNotClose.176
Figura 78: Representação da aplicação Mira ao Alvo [Banaszewski 2009] . . . 217
Figura 79: Exemplo de uma *Rule* presente na aplicação Mira ao Alvo 218
Figura 80: Representação do Sistema de Condicionamento de Ar [Banaszewski 2009]. 219
Figura 81: Ambiente gerado pelo simulador [Ronszcka et al. 2011]. 221
Figura 82: Figura conceitual de um exoesqueleto do projeto Hardiman I da General Electric. 223
Figura 83: Casos de uso do Sistema de Vendas. 224
Figura 84: *Rule* responsável por finalizar a venda [Ferreira et al. 2013]. 225
Figura 85: Configuração Inicial da Torre de Hanói [Krug 2016]. 229

LISTA DE TABELAS

Tabela 1: *Rules*, *Conditions* e suas *Premises* e *Actions* instigadas do software de controle PON para futebol de robôs. 103

Tabela 2: Resultados obtidos a partir dos experimentos de contabilidade de linhas de código e quantidade de *tokens* presentes no código fonte e nível de manutenibilidade. 120

Tabela 3: Definição das *Premises* utilizadas pelas *Rules* na aplicação de controle de futebol de robôs. 160

SUMÁRIO

1 INTRODUÇÃO **17**

1.1 PARADIGMA ORIENTADO A NOTIFICAÇÕES (PON) 18

1.2 MOTIVAÇÃO 21

1.3 JUSTIFICATIVA 22

1.3.1 Questão da agregação de *Rules* 23

1.3.2 Questão de agregação entre *FBEs* 24

1.3.3 Reflexão 25

1.4 OBJETIVOS 26

1.5 ATIVIDADES DO MÉTODO DE PESQUISA 27

1.5.1 Revisão do Estado da Arte 28

1.5.2 Reflexão e Proposta 28

1.5.3 Definição dos Objetivos 28

1.5.4 Desenvolvimento Investigativo 28

1.5.5 Comparações e Validações 30

1.5.6 Conclusão 30

2 REVISÃO DO ESTADO DA ARTE **31**

2.1 PARADIGMAS DE PROGRAMAÇÃO 31

2.1.1 Paradigma Imperativo 33

2.1.2 Paradigma Orientado a Eventos 36

2.1.3 Paradigma Declarativo 37

2.2 PARADIGMA ORIENTADO A NOTIFICAÇÕES 38

2.2.1 Relação entre o PON e os Paradigmas Imperativo e Declarativo . . 39

2.2.2 Mecanismo de Notificação do PON 41
2.2.3 Resolução de Conflitos no PON 42
2.2.4 Propriedades Inerentes ao PON 44
2.2.5 PON – Utilização x Compreensão 44
2.2.6 Cálculo Assintótico da Inferência do PON 45
2.2.7 *Attributes* Impertinentes 46
2.2.8 Regras de Formação 48
2.3 MATERIALIZAÇÕES DO PON 49
2.3.1 *Framework* PON 50
2.3.2 Linguagem e Compilador para o PON - LingPON 1.0 56
2.3.2.1 Linguagem de programação PON 56
2.3.2.2 Compilador para o PON 62
2.3.3 Outras Materializações em Software do PON 66
2.4 FUTEBOL DE ROBÔS - ROBOCUP 67
2.4.1 Ambiente Simulado *RoboCup* SSL 69
2.5 REFLEXÃO SOBRE A REVISÃO DO ESTADO DA ARTE 73

3 DESENVOLVIMENTO 75

3.1 CONTRIBUIÇÕES PARA A LINGPON 75
3.1.1 Agregação entre *FBEs* 76
3.1.2 Agregação de *Rules* em *FBEs* 79
3.1.3 Correção de erro: Geração de código-alvo C++ com múltiplas instâncias de *FBE* 86
3.2 ESTUDO DE CASO - SOFTWARE DE CONTROLE PARA O FUTEBOL DE ROBÔS 91
3.2.1 Especificação do *Software* de controle para uma partida de futebol de robôs 91

3.2.2 Solução desenvolvida sob o viés do Paradigma Imperativo - Programação Orientada a Objetos 94
3.2.3 Soluções desenvolvidas sob o viés do Paradigma Orientado a Notificações 101
3.2.3.1 Solução desenvolvida sob o viés do Framework PON 2.0 . 104
3.2.3.2 Solução desenvolvida sob o viés do Paradigma Orientado a Notificações - LingPON 1.0 105
3.2.3.3 Solução desenvolvida sob o viés do Paradigma Orientado a Notificações - LingPON 1.2 106
3.2.4 Comparações entre a aplicação de controle de futebol de robôs desenvolvida nos Paradigma Orientado a Objetos e no Paradigma Orientado a Notificações 108
3.2.4.1 Comparações de complexidade de código-fonte entre a aplicação de controle de futebol de robôs desenvolvida em PI/POO e PON 108
3.2.4.2 Comparações de manutenibilidade entre a aplicação de controle de futebol de robôs desenvolvida em PI/POO e PON 111
3.2.5 Reflexão sobre as comparações 118

4 CONCLUSÃO E TRABALHOS FUTUROS 121

4.1 CONCLUSÃO 121
4.2 TRABALHOS FUTUROS 124
4.2.1 Suporte a múltiplos arquivos de código-fonte 124
4.2.2 Utilização de bibliotecas externas 124
4.2.3 Simplificação da sintaxe da LingPON 125
4.2.4 Teste de unidade para o compilador 125
4.2.5 Melhorar experimento de nível de manutenibilidade 126
4.2.6 Estudo de Agentes aplicados ao PON 126

REFERÊNCIAS **128**

Apêndice A – DESCRIÇÃO DAS ALTERAÇÕES REALIZADAS NA LINGPON **136**

A.1 AGREGAÇÃO DE *FBES* 136

A.2 *FBE RULES* 141

A.3 CORREÇÃO DE ERRO: GERAÇÃO DE CÓDIGO-ALVO C++ COM MÚLTIPLAS INSTÂNCIAS DE *FBE* 142

Apêndice B – ESPECIFICAÇÃO TÉCNICA - ROBOCUP SMALL SIZE LEAGUE **145**

B.1 DESCRIÇÃO DO AMBIENTE DE JOGO 145

B.2 DESCRIÇÃO DOS ROBÔS 146

B.2.1 Uniformes 147

B.3 REQUISITOS FUNCIONAIS DA APLICAÇÃO DE CONTROLE 151

Apêndice C – CONJUNTO DE *RULES* APLICADAS À SOLUÇÃO DESENVOLVIDA SOB O VIÉS DO PON **159**

C.1 DIAGRAMA DE OBJETOS PON 159

C.2 CÓDIGO-FONTE DAS *RULES* DESENVOLVIDAS PARA O *SOFTWARE* PON 177

Apêndice D – ANALISADOR LÉXICO DESENVOLVIDO PARA A CONTAGEM DO NÚMERO DE *TOKENS* **210**

Apêndice E – CENSO DAS APLICAÇÕES PON **212**

E.1 MIRA AO ALVO 217

E.2 SISTEMA DE CONDICIONAMENTO DE AR 219

E.3 SIMULADOR DE JOGO (*PACMAN*) 220

E.4 SIMULADOR DE TRANSPORTE INDIVIDUAL 222

E.5 SISTEMA DE VENDAS 224

E.6 PORTÃO ELETRÔNICO 226
E.7 CONTROLE DA ILUMINAÇÃO EM UMA CIDADE VIRTUAL 3D . . . 226
E.8 WARSHIPATTACK GAME 227
E.9 CTA SIMULATOR 228
E.10 TORRE DE HANÓI 228
E.11 ALGORITMO TRIANGULAR MESH SLICING 229
E.12 REFLEXÃO 230

Apêndice F – PROGRAMAÇÃO ORIENTADA A AGENTES 232

Anexo A – BNF DA LINGPON 1.0 234

Anexo B – BNF DA LINGPON 1.2 237

Anexo C – RELATÓRIOS AINDA NÃO PUBLICADOS SOBRE APLICAÇÕES PON 241

Anexo D – RELATÓRIO DA DISCIPLINA LINGUAGENS/COMPILADORES - 2015 254

Anexo E – RELATÓRIO DA DISCIPLINA LINGUAGENS/COMPILADORES - 2016 271

1 INTRODUÇÃO

Os principais paradigmas de programação vigentes na indústria de desenvolvimento de *software* são o Paradigma Imperativo (PI) e o Paradigma Declarativo (PD). Aplicações desenvolvidas sob esses paradigmas de programação apresentam, em sua maioria, processamento desnecessário. Isto ocorre, principalmente, por motivos como redundâncias em avaliações causais e/ou utilização de estruturas de dados computacionalmente custosas.

Visando solucionar algumas das deficiências dos paradigmas usuais de programação em relação a avaliações causais desnecessárias e acopladas, o Paradigma Orientado a Notificações (PON) foi inicialmente proposto por J. M. Simão na forma de uma solução de controle discreto e de inferência para sistemas de manufatura inteligentes [Simão 2001], evoluindo posteriormente para a forma de um paradigma de programação [Simão e Stadzisz 2008, Ronszcka 2012, Banaszewski 2009].

O PON apresenta uma nova forma de realizar avaliações de expressões lógico-causais. No PON, tais expressões são expressas na forma de regras (*Rules*) e o fluxo de execução da aplicação é determinado por notificações pontuais e precisas entre entidades computacionais de pequeno porte. Dessa forma, é possível eliminar, ou ao menos amenizar, as limitações presentes no PI e PD.

Os conceitos do PON foram inicialmente implementados ou materializados computacionalmente na forma de um *Framework*. Este *Framework* foi construído utilizando a linguagem de programação C++. Tal *Framework* encontra-se em sua terceira versão [Valença 2013, Ronszcka et al. 2011]. Tais materializações possibilitaram a criação de aplicações PON e consequente demonstração de seus principais conceitos. Entretanto, o desempenho das aplicações desenvolvidas utilizando o *Framework* ficaram aquém do que era esperado pela teoria do PON [Ferreira et al. 2013, Simão et al. 2012].

Posteriormente, uma linguagem de programação, nomeada LingPON, e respectivo compilador, específicos para o PON, foram desenvolvidos [Ferreira 2016]. Por utilizar estruturas de dados mais enxutas quando comparada com o *Framework*, a linguagem de

programação PON apresentou resultados, em termos de desempenho, mais próximos ao esperado pela teoria do PON [Ferreira 2016].

Apesar de trazer avanços em questão de desempenho, a LingPON não representou um grande avanço no tocante à facilidade de programação. Conforme apresentado nas seções subsequentes, o desenvolvedor que deseja criar um *software* sob o viés do PON utilizando a atual versão da LingPON necessita escrever muitas linhas de código, fato este que não contribui para a consolidação do paradigma em questão.

Além disso, um número não muito expressivo de aplicações foram desenvolvidas utilizando a LingPON até o presente momento, fato este que não permite uma real compreensão das limitações da linguagem de programação. Dessa forma, vislumbra-se a possibilidade de desenvolver uma aplicação conhecidamente complexa, nomeadamente uma aplicação de controle de futebol de robôs, utilizando a LingPON para que a mesma possa ser avaliada de forma mais efetiva. Ademais, essa aplicação enriquecerá o rol de aplicações PON e poderá, até mesmo, ser de valor considerável para a consolidação do PON entre os paradigmas de programação vigentes.

Dessa forma, este trabalho pretende evoluir a especificação da LingPON e seu respectivo compilador de forma a facilitar e agilizar o desenvolvimento de aplicações PON, além de usar o PON em uma aplicação de complexidade reconhecida.

Neste capítulo introdutório, a Seção 1.1 apresenta uma contextualização sobre o Paradigma Orientado a Notificações. A Seção 1.2 detalha a motivação para este estudo. A Seção 1.3, por sua vez, apresenta a justificativa para este estudo. Por fim, a Seção 1.4 elenca os objetivos pretendidos com este trabalho [1].

1.1 PARADIGMA ORIENTADO A NOTIFICAÇÕES (PON)

O PON encontra alguma inspiração nos paradigmas usuais de programação, nomeadamente o Paradigma Declarativo (PD) e o Paradigma Imperativo (PI), aproveitando a flexibilidade algorítmica e abstração em forma de classes/objetos da Programação-Paradigma Orientado a Objetos (POO) do PI e a representação do conhecimento em regras dos Sistemas Baseados em Regras (SBR) do Paradigma Lógico (PL) do PD. Sendo assim, o PON usa inclusive parte de ambos os estilos de programação em seu modelo, evoluindo-os no que se refere ao processo de inferência ou cálculo lógico-causal [Xavier 2014, Simão e

[1] Pertinente registrar que parte destes esforços de pesquisa, aqui considerados, foram apresentados na forma de Trabalho Individual (“Qualificação de Mestrado”) junto ao CPGEI/UTFPR em 03 de Junho de 2016

Stadzisz 2008, Simão et al. 2009, Banaszewski 2009, Linhares et al. 2011, Simão et al. 2012]

Visando solucionar, ou ao menos amenizar, algumas das deficiências encontradas naqueles paradigmas, tais como a repetição de expressões lógicas e reavaliações desnecessárias delas (i.e. redundâncias estruturais e temporais) e, particularmente, o acoplamento forte de entidades quanto às avaliações ou cálculo lógico-causal, o PON apresenta outra forma de realizar tais avaliações ou inferências. Isto é realizado via entidades computacionais de pequeno porte, reativas e desacopladas que colaboram por meio de notificações pontuais, sendo tais entidades criadas a partir do 'conhecimento' de regras [Linhares et al. 2011, Simão et al. 2012, Simão et al. 2012, Simão et al. 2012].

No PON, as entidades computacionais que possuem atributos (*Attribute*) e métodos (*Methods*) são genericamente chamadas de *FBEs* (*Fact Base Elements*). Por meio de seus *Attributes* e *Methods*, as entidades de *FBE* são passíveis de correlação lógico-causal por meio de *Rules*, as quais constituem elementos fundamentais do PON [Xavier 2014, Simão e Stadzisz 2008, Linhares et al. 2011, Simão et al. 2012, Simão et al. 2012, Simão et al. 2012].

A Figura 1 apresenta um exemplo de *Rule*, justamente na forma de uma regra lógico-causal. A *Rule* é uma entidade computacional composta por outras duas entidades, *Condition* e *Action*. A *Condition* trata da decisão da *Rule*, enquanto a *Action* trata da execução das ações associadas à *Rule*. Assim sendo, a *Condition* e a *Action* tratam o conhecimento lógico e causal associado a *Rule* [Simão e Stadzisz 2008, Linhares et al. 2011, Simão et al. 2012, Simão et al. 2012, Simão et al. 2012].

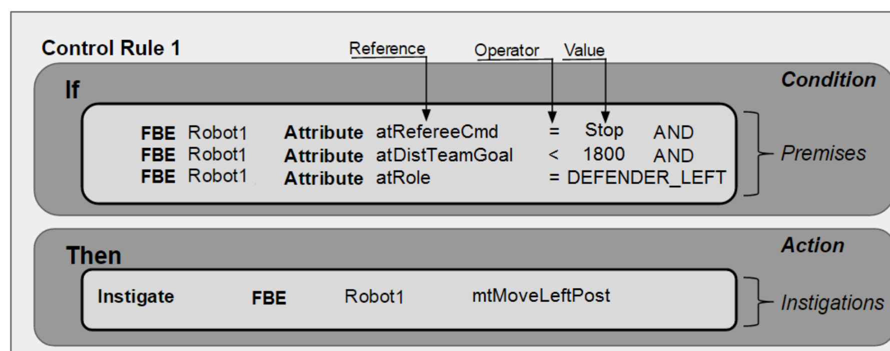

Figura 1: Exemplo de uma *Rule*.

A *Rule* apresentada na Figura 1 faz parte de um sistema de controle de robôs em uma partida de futebol de robôs. Essa *Rule* refere-se à decisão de movimento de um determinado Robô (*Robot*) em uma partida de futebol, em função do estado de seus *Attributes*.

Neste contexto, a *Condition* da *Rule* apresentada na Figura 1 é composta por três

entidades *Premises*. Estas, realizam, respectivamente as seguinte verificações: a) *o comando recebido pelo Robô (Robot1.atRefereeCmd) representa a interrupção da partida (Stop)?* b) *a distância entre a o Robô e o gol que ele está defendendo (Robot1.atDistTeamGoal) é menor que 1800 cm.?* c) *a função que o Robô está exercendo no time (Robot1.atRole) é a de lateral-esquerdo ("DEFENDER_LEFT")?.* Portanto, é possível concluir que os estados dos atributos dos *FBEs* compõem os fatos a serem avaliados pelas *Premises* [Simão e Stadzisz 2008, Simão et al. 2009, Banaszewski 2009].

Cada *Premise* avalia o estado de um ou dois *Attributes* de *FBE*. Em tempo, para cada mudança de estado de um *Attribute* de um *FBE*, ocorrem automaticamente, via notificações, avaliações (lógicas) somente nas *Premises* relacionadas com eventuais mudanças nos seus estados. Semelhantemente, a partir da mudança de estado das *Premises*, ocorrem automaticamente avaliações somente nas *Conditions* relacionadas com eventuais mudanças em seus estados [Simão e Stadzisz 2008, Simão et al. 2009, Banaszewski 2009, Simão et al. 2012].

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. Isto se dá em cada Premise pela comparação, usando um operador lógico, do valor notificado com outro valor, este uma constante ou um valor notificado por outro *Attribute*. Se o valor lógico da entidade *Premise* se alterar, essa notifica um conjunto de entidades *Conditions* conectadas para que seus estados lógicos sejam reavaliados [Banaszewski 2009].

Desse 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, a entidade *Condition* é satisfeita quando todas as entidades *Premises* que a compõem apresentam estado lógico verdadeiro, decorrendo na aprovação da sua respectiva *Rule*. Com isto, a entidade *Action* conectada a esta *Rule* é executada, podendo invocar *Methods* de *FBEs* através das entidades *Instigations* [Banaszewski 2009]. Usualmente, os *Methods* alteram os estados dos *Attributes* do *FBE*, fazendo assim com que um novo ciclo de notificações se inicie. No exemplo apresentado, a *Action* contém apenas uma *Instigation*, a qual instiga um *Method* que faz movimentar o Robô para uma posição específica (*Stop Position*).

Isto posto, nota-se que a essência da computação no PON está na forma como as responsabilidades de um programa estão distribuídas entre entidades computacionais autônomas e reativas que colaboram entre si através de notificações pontuais. Este

arranjo forma o mecanismo de notificações, o qual determina o fluxo de execução das aplicações, permitindo execução otimizada e minimamente acoplada de processamento lógico-causal, útil para o aproveitamento correto de mono-processamento, bem como para o processamento distribuído [Simão e Stadzisz 2008, Simão et al. 2010, Belmonte et al. 2012, Simão et al. 2012, Simão et al. 2012, Simão et al. 2012, Linhares et al. 2015, Linhares 2015, Ferreira 2016].

Os conceitos do PON foram primeiramente materializados sobre a programação orientada a objetos (POO), através de um *Framework* desenvolvido utilizando a linguagem de programação C++ em uma versão prototipal concebida por Simão em 2007. Subsequentemente, a chamada versão 1.0 foi desenvolvida em 2009 por Banaszewski [Banaszewski 2009]. Em 2012, uma nova versão (versão 2.0) otimizada para ambientes monoprocessados foi desenvolvida por Valença [Valença 2013] e Ronszcka [Ronszcka 2012]. Posteriormente, foi desenvolvida uma linguagem de programação específica para o PON (LingPON, versão 1.0), acompanhada do respectivo compilador [Ferreira 2016], a qual já apresenta uma nova versão prototipal [Pordeus et al. 2015]. Atualmente, o *Framework* PON (particularmente a versão 2.0) e a LingPON (particularmente a versão 1.0) representam as principais materializações em *software* do PON.

1.2 MOTIVAÇÃO

Muitos assuntos relacionados ao PON foram estudados nos últimos anos, entre eles pode-se destacar: Desenvolvimento Orientado a Notificações (DON) [Wiecheteck et al. 2011, Wiecheteck 2012, Medonça et al. 2015], padrões de projetos aplicados a softwares PON [Ronszcka 2012], linguagem nativa de programação PON (LingPON) e respectivo compilador [Ferreira 2016], teste funcional em software PON [Kossoski et al. 2014], coprocessadores para aceleração de aplicações desenvolvidas utilizando o PON (CoPON) [Peters 2012], PON em *hardware* digital (PON HD) [Kerschbaumer et al. 2015, Simão et al. 2012], processador nativo em PON (ARQPON) [Linhares 2015] e comparações entre abordagens orientada a eventos e orientada a notificações [Xavier 2014].

Apesar destes avanços, como é ainda um tanto natural academicamente no tocante à prova de conceito, um número não verdadeiramente significativo de aplicações, que sejam complexas[2] e que tenham sido comparados a versões equivalente em paradigmas

[2]Segundo [Russell e Norvig 2009], a complexidade de um sistema está relacionado à certas características de seu ambiente de execução. As características consideradas para determinar a complexidade de uma tarefa consideram se o ambiente é parcialmente ou completamente observável, determinístico ou estocástico, estático ou dinâmico e discreto ou contínuo e se envolve um único agente ou multi-agentes.

vigentes, foram desenvolvidos em PON. Isso se agrava quando se observa a materialização de linguagem e compilador para o PON, aqui nomeada apenas como LingPON. Isto se deve, inclusive, ao fato desta ser uma materialização recente, a qual ainda está em desenvolvimento para se tornar uma linguagem de programação mais completa. Como relatado em [Ferreira 2016], ainda faz-se necessário, dentre outros, evolui-la no tocante a certas facilidades de programação, visando proporcionar maior agilidade no desenvolvimento de aplicações PON.

Dessa forma, este trabalho vislumbra a possibilidade de evoluir a LingPON, de forma a facilitar o desenvolvimento de aplicações PON por meio do aprimoramento de agregação de entidades no LingPON, a luz do próprio PON, o que leva a redução de redundâncias de linhas de código fonte em aplicações feitas neste paradigma nessa sua materialização. Ademais, este trabalho também visa a demonstração da utilização da LingPON no desenvolvimento de uma aplicação pertinente e de maior complexidade, quando comparada com as aplicações previamente desenvolvidas.

A aplicação visada neste presente trabalho está no âmbito de controle de futebol de robôs, um domínio reconhecidamente com considerável complexidade, uma vez que se trata de um ambiente estocásticos, dinâmico, não determinístico, contínuo e com múltiplos agentes [Visser e Burkhard 2007, Russell e Norvig 2009]. Ainda, este domínio de aplicação permitirá tecer comparações entre a versão evoluída da LingPON aqui proposta, a versão até então vigente da LingPON (versão 1.0), o Framework PON na sua versão 2.0 (última versão em C++) e mesmo comparações para com a Programação/Paradigma Orientado a Objetos (POO) em C++. Isto se dará por meio de versões equivalentes da mesma aplicação nestes diferentes meios, sublinhando que a versão POO foi elaborada por outrem, o que permitiria comparação um tanto mais isenta.

1.3 JUSTIFICATIVA

A oferta de facilidades na programação é uma característica inerente à teoria do PON, principalmente devido ao fato de ele, neste âmbito, ter sido inspirado em conceitos do PD. De fato, um conjunto de autores consideram a programação em PD menos difícil que em PI [Kaisler 2005, Gabbrielli e Martini 2010]. Assim, a programação em PON é tida como menos difícil que programação em PI porque aquela é considerada intuitiva à forma cognitiva humana, característica esta herdada do PD [Panescu et al. 2015, Banaszewski 2009]. Neste sentido, um dos objetivos fundamentais do PON, através de suas materializações, é proporcionar o desenvolvimento de *software* com alguma facilidade

de programação [Banaszewski 2009].

Entretanto, até o presente momento, um número não grande de aplicações em PON (aproximadamente 35) foram desenvolvidas utilizando a LingPON [3]. Ainda, as aplicações PON desenvolvidas apresentavam escopo reduzido, pois visavam o estudo do desempenho do PON em cenários específicos de comparação. O fato de apenas algumas aplicações PON terem sido desenvolvidas utilizando a LingPON está relacionado inclusive a algumas limitações da linguagem, relativas a restrições de agregação de entidades PON na gramática da LingPON. Tais restrições fazem com que o desenvolvedor tenha que reescrever, por diversas vezes, linhas de código muito semelhantes. Estas limitações são apresentadas e explicadas a seguir, através da apresentação de exemplos.

1.3.1 QUESTÃO DA AGREGAÇÃO DE *RULES*

De forma análoga ao que existe em aplicações desenvolvidas segundo a POO, nas quais objetos são criados como sendo instâncias de classes, em aplicações PON é possível (e necessário) criar instâncias de *FBEs*. Em aplicações desenvolvidas utilizando a LingPON, cada uma das *Rules* relaciona-se com instâncias de *FBEs*. Neste sentido, para cada nova instância de FBE criada, novas Rules e, consequentemente, novas linhas de código devem ser adicionadas ao sistema, resultando assim em aumento no código fonte a ser escrito. Isto é assaz normal, mas há situações nas quais ocorre um aumento desnecessário de complexidade na atual versão da LingPON (versão 1.0).

De forma a exemplificar o problema em questão, pode-se imaginar uma aplicação PON que controle o funcionamento de robôs em uma partida de futebol de robôs. O comportamento do robô depende da posição (goleiro, zagueiro e etc.) designada ao mesmo na partida. Caso o robô seja um zagueiro, seu comportamento esperado é o de defender. Entretanto, caso seja um atacante, um comportamento mais ofensivo é esperado. Sendo assim, um robô poderia ser modelado como sendo um *FBE* que apresenta apenas um *Attribute* (*posição*) e dois *Methods* (*defender* e *atacar*). Desse modo, havendo apenas um robô (R1) presente na partida, o sistema de controle poderia ser criado com as *Rules*:

- *Rule* 1: Se *posição* de R1 é 'zagueiro' então defender.
- *Rule* 2: Se *posição* de R1 é 'atacante' então atacar.

Entretanto, caso seja necessário expandir tal aplicação de forma a controlar

[3]As aplicações PON desenvolvidas até o presente momento são listadas no Apêndice E deste trabalho.

dois robôs (R1 e R2), será imprescindível duplicar o número de *Rules* que regem o comportamento do sistema afim de que ambos apresentem um funcionamento correto. Nesse caso, o sistema apresentaria as seguintes *Rules*:

- *Rule* 1: Se *posição* de R1 é 'zagueiro' então defender.
- *Rule* 2: Se *posição* de R1 é 'atacante' então atacar.
- *Rule* 3: Se *posição* de R2 é 'zagueiro' então defender.
- *Rule* 4: Se *posição* de R2 é 'atacante' então atacar.

Nesse pequeno exemplo é possível evidenciar um cenário no qual haverá redundância de *Rules* no código fonte da aplicação PON desenvolvida utilizando a LingPON. Portanto, vislumbra-se a possibilidade de evoluir a LingPON de forma a evitar a redundância na declaração de *Rules* em cenários similares ao exemplificado, através da inclusão do conceito de "agregação"de *Rules* em *FBEs*.

1.3.2 QUESTÃO DE AGREGAÇÃO ENTRE *FBES*

Na LingPON, define-se que os *Attributes* de um *FBE* devem ser de tipos primitivos, isto é, *boolean*, *integer*, *float*, *char* ou *string* [Ferreira 2016]. Entretanto, existem casos nos quais há a necessidade de se criar *Attributes* que façam referência a outro *FBE* de forma a aumentar o encapsulamento de *Attributes* e *Methods*.

Para exemplificar tal cenário, pode-se imaginar uma aplicação PON na qual seja necessário criar um *FBE* que simule o comportamento de um time de futebol. O time é composto por dois jogadores, sendo que cada jogador possui um nome, um número e uma posição de jogo.

Na atual versão do LingPON, o desenvolvedor deverá construir um *FBE Time* o qual apresentará dois *Attributes* nome (*nome1* e *nome2*), dois *Attributes* número (*numero1*, *numero2*) e dois *Attributes* posição (*posicao1*, *posicao2*). Em se tratando de um time de dois jogadores, isso não é um grande problema. Entretanto, caso seja necessário expandir o número de jogadores no time, será imprescindível criar novos *Attributes* para cada novo jogador.

Dessa forma, visando aumentar o encapsulamento na LingPON, vislumbra-se a possibilidade de evoluir a LingPON de forma a permitir que *FBEs* relacionem outros *FBEs*, ou seja, possuam *Attributes* que sejam definidos por um *FBE*. No exemplo apresentado,

seria possível criar um *FBE Jogador* e um *FBE Time*, o qual possuiria quantos *Attributes* do tipo *Jogador* quanto necessário para compor um time.

1.3.3 REFLEXÃO

Esses são exemplos que demonstram a necessidade de evolução da LingPON. Outros exemplos seriam a necessidade de mecanismos de herança, declaração de vetores e alocação dinâmica de memória. Portanto, reforça-se a necessidade de validação do LingPON, inclusive com as novas características.

Neste sentido, esta dissertação de mestrado se propõe a evoluir a LingPON no tocante a facilidade de programação por meio de aprimoramento de relações de agregação entre entidades PON na LingPON. Particularmente, a evolução englobará as soluções vislumbradas na subseção 1.3.1 (i.e. agregação de Rules em FBEs) e na subseção 1.3.2 (i.e. agregação de FBEs em FBE), bem como a correção de erros ou *bugs* existente na atual versão do LingPON, tais como erros na geração de código alvo quando o código LingPON apresenta múltiplas instâncias do mesmo *FBE*.

Esta evolução da LingPON (i.e. linguagem e compilador do PON) se dará por meio de mudanças na gramática ou BNF (*Backus normal form*), no analisador léxico, no analisador sintático, no analisador semântico e no gerador de código relativos ao LingPON. Isto inclusive por meio de ferramental de desenvolvimento de linguagens e compiladores, como as ferramentas *Flex* e *Bison*.

Ainda, tanto esta evolução da LingPON, como ela na íntegra serão neste trabalho validados por experimentos em uma aplicação com nível de complexidade que supera as aplicações anteriores. Desse modo, faz-se necessário realizar um levantamento e análise, em termos de quantidade de entidades PON, das aplicações previamente desenvolvidas utilizando o PON.

Em suma, a aplicação utilizada para a validação das evoluções da LingPON será o controle de futebol de robôs que permite inclusive testes em um estudo de caso de complexidade reconhecida em vários domínios da ciência da computação e afins [Visser e Burkhard 2007] [4]. Neste contexto, a própria versão anterior do compilador será testada, bem como a última versão do Framework PON em C++ (*Framework* PON 2.0) via versões da aplicação desenvolvidas nessas materializações precedentes do PON. Portanto, isso permitirá compor comparações entre as principais materializações correntes em

[4] Os requisitos funcionais da aplicação de controle de futebol de robôs são apresentados no Apêndice deste trabalho.

software do PON. Por fim, as aplicações nessas materializações serão comparadas como um versão da aplicação desenvolvida por outrem em POO. Neste, âmbito as comparações tratarão inclusive do grau de dificuldade de lidar com nova demanda (i.e. mudança) em funcionalidades de *software* no tocante ao POO e ao PON.

1.4 OBJETIVOS

Tendo em vista o apresentado até agora em termos de motivação e justificativa, a dissertação de mestrado desenvolvida com base neste trabalho apresenta como objetivos principais [5]:

- Evoluir a tecnologia LingPON (linguagem e seu respectivo compilador), no tocante a tratar agregações de entidades PON, nomeadamente *FBEs* que agregam *Rules* e/ou *FBEs*. Isto com o intuito de evitar redundância de código e, assim, facilitar a programação ou desenvolvimento de aplicações em PON.
- Desenvolver com a tecnologia LingPON até então vigente, com a tecnologia LingPON proposta neste trabalho e com o *Framework* PON 2.0, para cada uma dessas materializações, uma aplicação de complexidade reconhecida. Nomeadamente, esta aplicação é a reconhecida aplicação de controle para partida de futebol de robôs. Tal desenvolvimento visa contribuir para demonstrar e consolidar a factibilidade do PON em um contexto mais efetivo.
- Comparar entre si, por meio das aplicações em PON para controle futebol de robôs, as três materializações do PON consideradas, bem como a mesma aplicação feita em POO/PI por outrem. Tais comparações se darão no tocante ao número de linhas de código e *tokens*[6] presentes no código-fonte e dificuldade de manutenção de *software* em função de nova demanda.

Para atingir estes objetivos principais, o presente trabalho de pesquisa tem os seguintes objetivos específicos:

- Evoluir a especificação da LingPON (linguagem e seu respectivo compilador) de forma a evitar a redundância na declaração de *Rules* quando as mesmas devem

[5] Os objetivos principais deste trabalho poderia ser apresentado de forma tradicional, em um único parágrafo, com o mesmo valor semântico dos três itens. Entretanto, os objetivos são apresentados separadamente de forma a facilitar a compreensão e avaliação dos mesmos.

[6] Um *token* é um par constituído de um nome de *token* e um valor de atributo opcional identificado a partir de um lexema (sequência de caracteres que associa um padrão a um *token*) [Aho et al. 1995].

ser aplicadas sobre todas as instâncias de um determinado *FBE*. Isto se dará pela possibilidade de *FBE* poder agregar Rules;

- Evoluir a especificação da LingPON (linguagem e seu respectivo compilador) para suportar agregação de *FBEs* através da declaração de *Attributes* de tipos não primitivos. Isto se dará pela possibilidade de FBE poder agregar FBE;
- Realizar levantamento e análise das aplicações PON desenvolvidas até o presente momento, em termos de entidades PON presentes no código-fonte e complexidade de funcionamento, de forma a evidenciar a simplicidade destas aplicações. Isto principalmente para as materializações do PON, nomeadamente *Framework* PON 2.0 e LingPON 1.0;
- Desenvolver aplicações de controle para o futebol de robôs utilizando as seguintes materializações do PON: *Framework* PON 2.0 (C++), atual versão da LingPON (versão 1.0) e a nova versão da LingPON (chamada de versão 1.2[7]), esta apresentada neste trabalho;
- Comparar a aplicação desenvolvida utilizando a nova versão da LingPON com uma solução existente, construída sob o PI, e com soluções desenvolvidas utilizando o *Framework* PON 2.0 e a atual versão da LingPON, utilizando número de linhas e número de *tokens* como critérios de complexidade de código-fonte;
- Verificar propriedades do PON de facilidade de programação, através da análise de nível de manutenibilidade por meio de alteração de requisito, utilizando as materializações pertinentes a este trabalho (i.e. *Framework* PON 2.0, atual versão da LingPON e a nova versão da LingPON) em uma situação prática a ser comparada com aplicação equivalente (mesmo funcionamento) desenvolvida em C++ PI/POO por outrem.

1.5 ATIVIDADES DO MÉTODO DE PESQUISA

A metodologia desta pesquisa envolveu tarefas teóricas e práticas, com comparações qualitativas e quantitativas. Dessa forma, foram realizadas seis tarefas principais durante a pesquisa e desenvolvimento deste trabalho, são elas: Revisão do Estado da Arte e da Técnica, Reflexão e Proposta, Definição dos Objetivos, Desenvolvimento Investigativo,

[7]A versão da LingPON desenvolvida e apresentada neste trabalhado foi definida como sendo 1.2 por suportar duas novas funcionalidades, a saber a agregação de *FBE* em *FBEs* e *Rule* em *FBE*.

Comparações e Validações e, por fim, Conclusão, conforme apresentado na Figura 2. A seguir, cada uma destas fases é apresentada.

1.5.1 REVISÃO DO ESTADO DA ARTE

Primeiramente, com o objetivo de prover uma base sólida para a argumentação e desenvolvimento deste trabalho, foram angariadas referências bibliográficas sobre os paradigmas de programação apresentados neste trabalho, nomeadamente o Paradigma Imperativo, Paradigma Declarativo e Paradigma Orientado a Notificações. Posteriormente foi realizado um estudo sobre as materializações do PON em *software*, a saber o *Framework* PON C++ 2.0 e a LingPON. Por fim, foi realizado um censo das aplicações PON desenvolvidas até o presente momento, de forma a evidenciar a baixa complexidade na maioria das aplicações PON desenvolvidas até o presente momento.

1.5.2 REFLEXÃO E PROPOSTA

Esta etapa teve como objetivo identificar as limitações da atual versão da LingPON, principalmente no tocante à agregação de entidades. Ademais, com base nos dados obtidos através do censo de aplicações PON, contatou-se a necessidade de desenvolver a apresentar um estudo de caso mais complexo e apropriados aos anteriores para validar as materializações do PON em *software*: *Framework* PON C++ 2.0 e LingPON. Como resultado desta reflexão, surgiu a proposta de desenvolver uma aplicação de controle de futebol de robôs utilizando as materializações PON *software*. Para isso, um estudo e análise de uma aplicação já existente foram realizados. Por fim, foi proposto a utilização da aplicação de controle de futebol de robôs para a validação das materializações PON em *software*.

1.5.3 DEFINIÇÃO DOS OBJETIVOS

Utilizando os resultados obtidos a partir da etapa de reflexão e propostas, foram definidos os três objetivos principais apresentados deste trabalho, os quais são apresentado na Seção 1.4.

1.5.4 DESENVOLVIMENTO INVESTIGATIVO

A etapa de desenvolvimento iniciou-se pelo desenvolvimento de uma aplicação de futebol de robôs, funcionalmente equivalente a uma solução já existente e desenvolvida em

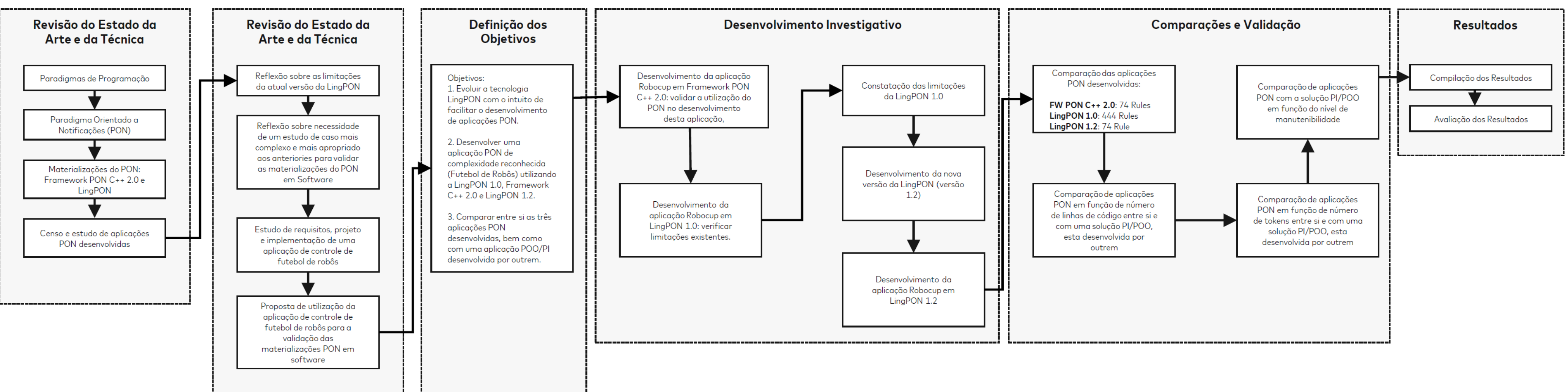

Figura 2: Atividades do método de pesquisa.

PI/POO, utilizando o *Framework* PON C++ 2.0 e a LingPON. Na aplicação desenvolvida em *Framework* PON C++ 2.0, foi possível validar a relação de agregações de entidades PON, conforme desenvolvido e apresentado em [Ronszcka 2012]. Na aplicação desenvolvida em LingPON, por sua vez, contatou-se que limitações referentes à agregação de entidades PON dificultava o desenvolvimento de aplicações PON. Sendo assim, foi desenvolvida uma nova versão da LingPON (versão 1.2) que solucionou as duas limitações observadas. Utilizando esta nova versão da LingPON, uma nova aplicação de controle de futebol de robôs foi desenvolvida.

1.5.5 COMPARAÇÕES E VALIDAÇÕES

As aplicações PON de controle de futebol de robôs desenvolvidas neste trabalho foram comparadas em relação às entidades PON presentes em seu código-fonte. Posteriormente, as aplicações PON foram comparadas, em termos de linhas de código e quantidade de *tokens* presentes no código-fonte, entre si e com a aplicação PI/POO que foi utilizada como base na etapa de desenvolvimento investigativo. Por fim, as aplicações PON e a aplicação PI/POO foram comparadas em termos de facilidade de manutenção, através de um experimento de adição de um novo requisito funcional à cada um dos sistemas.

1.5.6 CONCLUSÃO

A última etapa realizada durante o desenvolvimento deste trabalho contou com a compilação dos resultados obtidos através das comparações realizadas na etapa anterior. Posteriormente realizou-se a etapa de avaliação dos resultados obtidos, a qual permitiu avaliar os resultados obtidos durante o desenvolvimento deste trabalho.

2 REVISÃO DO ESTADO DA ARTE

Neste capítulo são apresentados os principais conceitos sobre os quais se desenvolve este trabalho. Em primeiro lugar, a Seção 2.1 apresenta uma sucinta reflexão sobre os paradigmas de programação usuais da computação. Em seguida, a Seção 2.2 apresenta o Paradigma Orientado a Notificações (PON) de forma mais aprofundada àquela apresentada na Subseção 1.1.2. A Seção 2.3 detalha as atuais materializações do PON em *software*. Ainda, a seção 2.4 apresenta o que é o futebol de robôs. Findando o capítulo, a Seção 2.5 apresenta algumas reflexões sobre o presente capítulo.

2.1 PARADIGMAS DE PROGRAMAÇÃO

Existem diversas definições para o significado da palavra paradigma no contexto da computação. 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) utilizados 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 estruturar o pensamento na concepção de soluções em *software* [Van-Roy e Haridi 2004].

De forma sucinta, paradigma de programação é o modelo utilizado para compreender um problema do mundo real, de forma que o mesmo possa ser solucionado por sistemas computacionais, tradicionalmente *software*. Usualmente, o modelo do paradigma de programação está disponível para o programador na forma de linguagem de programação própria ou em *framework* sobre outra linguagem de outro paradigma. Neste sentido, as linguagens de programação são as ferramentas capazes de tornar o paradigma aplicável e permitir que artefatos de *software* possam ser criados para solucionar problemas [Banaszewski 2009].

Van Roy (2009) apresentou uma taxonomia de como os paradigmas de programação

são relacionados e qual é o caminho das linguagens de programação até os paradigmas e conceito relacionados a eles. Esta taxonomia é apresentada na Figura 3.

Nesta taxonomia, cada paradigma de programação é definido por um conjunto de conceitos de programação e organizado em uma linguagem básica simples, nomeada *kernel language* [Roy et al. 2009]. Para classificá-los, Van Roy utilizou características pertinente a cada um deles, tais como conceitos de registros (*record*), recipientes com escopo léxico (*closure*), independência (concorrência) e estado nomeado (*named state*), além do não-determinismo observável [1].

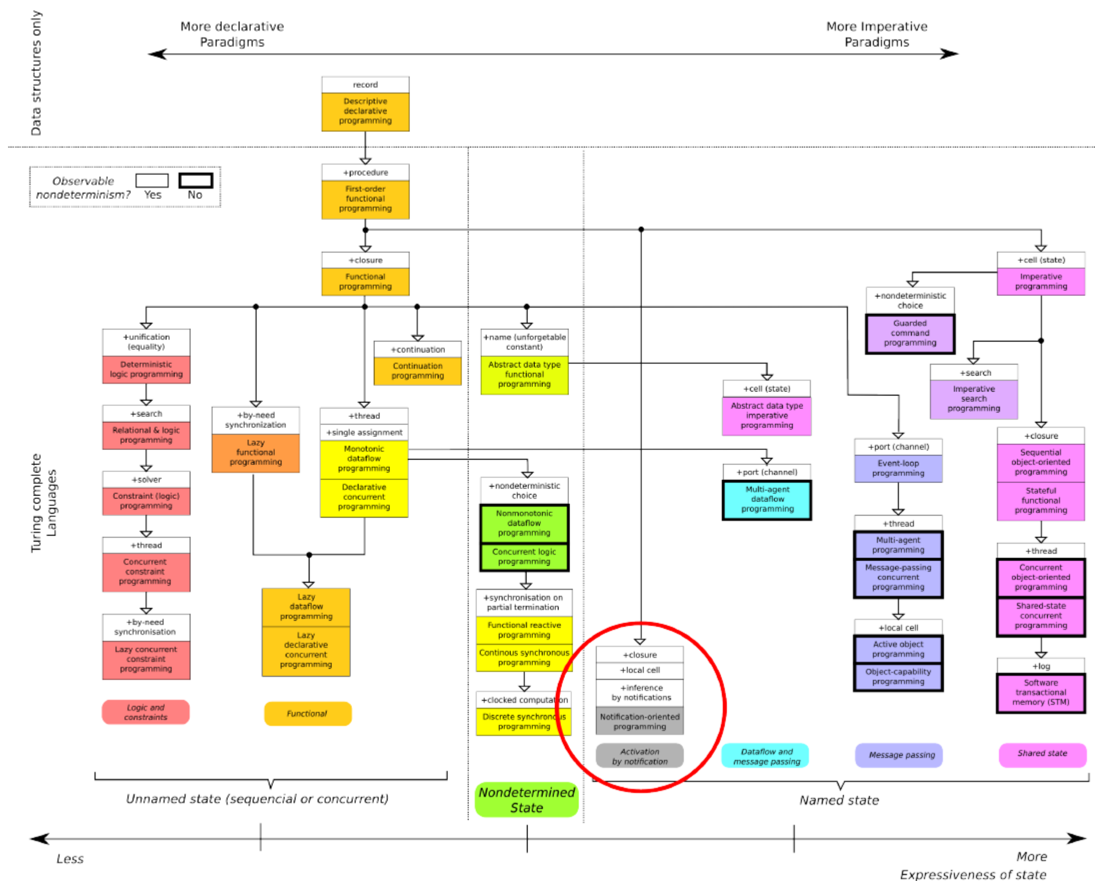

Figura 3: Taxonomia de paradigmas de programação com o Paradigma Orientado a Notificações em destaque. Extraído de [Xavier 2014].

Atualmente, há vários paradigmas de programação. Entretanto, de maneira resumida, uma forma de classificá-los seria em Paradigma Imperativo (PI) e Paradigma Declarativo (PD) [Van-Roy e Haridi 2004]. O PI ainda pode ser subdividido em Paradigma Procedimental (PP) e Paradigma Orientado a Objetos (POO). Por sua vez, o PD pode

[1] Não-determinismo é quando a execução de um programa não pode ser completamente determinada pela sua especificação, isto é, em algum momento durante a execução a especificação permite ao programa escolher o que fazer a seguir. Sendo assim, o não-determinismo observável ocorre quando um usuário pode ver resultados diferentes de execuções a partir da mesma configuração interna [Roy et al. 2009].

ser subdividido em Paradigma Funcional (PF) e Paradigma Lógico (PL). Isto considerado, conforme citado na introdução deste trabalho, o PON encontra alguma inspiração no POO e no PD. Neste sentido, de forma a compreender as características do PON e suas vantagens, faz-se necessário entender os conceitos relacionados ao POO e ao PL, particularmente do Sistemas Baseados em Regras (SBR). Por esse motivo, as subseções sequentes apresentam maiores detalhes sobre tais paradigmas de programação.

A revisão sobre paradigmas de programação apresenta neste trabalho foi baseada em trabalhos anteriores desenvolvidos pelo grupo de pesquisa da UTFPR sobre o PON, sendo as principais referências as seguintes [Banaszewski 2009, Xavier 2014].

2.1.1 PARADIGMA IMPERATIVO

O Paradigma Imperativo (PI) engloba dois dos subparadigmas mais utilizados no desenvolvimento de *software*: Paradigma Procedimental (PP) e Paradigma Orientado a Objetos (POO). A utilização de ambos os subparadigmas do PI tornou-se popular na indústria de *software* devido a questões como inércia cultural, riqueza de abstração e flexibilidades algorítmicas [Xavier 2014].

No caso do PP/PI, as variáveis e comandos são organizados em funções e procedimentos, os quais permitem alcançar um grau significativo de modularidade no código implementado [Watt 2004, Banaszewski 2009].

O POO/PI, por sua vez, apresenta um nível de abstração considerado mais rico e natural para o ser humano, quando comparado ao PP/PI. Os *softwares* baseados neste subparadigma são compostos por entidades modulares denominadas objetos, as quais representam objetos do mundo real, apresentando somente as características pertinentes para a implementação do sistema computacional [Poo et al. 2007]. As entidades objetos, em termos técnicos, agrupam atributos (similares a variáveis do PP) e métodos (similares a procedimentos ou funções do PP) relacionados de maneira a estimular coesão e desacoplamento [Pressman e Maxim 2016, Brookshear 2002, Watt 2004].

Independentemente do subparadigma utilizado, *softwares* baseados no PI são concebidos como sequências de instruções. Esse mecanismo de execução sequencial consiste em buscas 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*) usualmente executados dentro de laços de repetição (e.g *for*, *while* e *do-while*).

Devido à forma de busca presente na execução de *softwares* PI, as linhas de

código se tornam interdependentes, causando assim problemas de redundância temporal e estrutural em sua execução. Estes problemas podem, inclusive, acarretar degradação de desempenho dos programas desenvolvidos.

A redundância temporal ocorre na avaliação desnecessária e repetida de expressões causais na presença de estados previamente avaliados e inalterados. A redundância estrutural, por sua vez, ocorre quando o conhecimento sobre o valor *Booleano* de uma expressão lógica não é compartilhado entre outras expressões causais pertinentes, causando assim reavaliações desnecessárias [Banaszewski 2009].

Ambas redundâncias apresentadas são observadas no Código 1. Neste, três expressões causais verificam os estados de x e y dos objetos A e B a fim de alterar o estado y do objeto B.

Código 1: Exemplo de redundância temporal e estrutural em aplicações PI [Banaszewski 2009].

```
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 ...
```

A redundância temporal é observada na avaliação repetida da primeira expressão causal (linha 7) a cada ciclo interativo, mesmo quando a mesma não apresenta variação em relação às avaliações anteriores. A redundância estrutural é observada na avaliação da primeira condição das duas últimas expressões causais (linhas 11 e 15). Esta condição é avaliada duas vezes em um mesmo ciclo, sem alterações em seu valor lógico.

Além de problemas de redundância, o PI também apresenta problemas relacionados a composição de programas. Geralmente, nos programas criados com os conceitos do PI, o código que envolve o conhecimento lógico-causal da aplicação encontra-se 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 [Banaszewski 2009].

Os programadores normalmente encontram dificuldades na manipulação das linguagens de programação imperativas, principalmente no que diz respeito ao controle de fluxo de execução através de *loops* (e.g *for*, *while* e *do-while*) e por sintaxes pouco intuitivas, como por exemplo a atribuição de valores por meio do símbolo "=" e de igualdade por meio do símbolo "==".

Ademais, a maior dificuldade na programação imperativa se encontra na forma com a qual o código é organizado. Apesar do POO oferecer avanços em termos de abstrações de maior compreensão, o código OO normalmente é de difícil compreensão, não só pela sintaxe, mas também pela organização lógica do código-fonte [Banaszewski 2009].

Devido aos relacionamentos entre os objetos, torna-se difícil compreender uma funcionalidade do sistema analisando apenas uma classe. Normalmente, é preciso considerar as demais classes relacionadas. Essa dificuldade se deve à dispersão da lógica da aplicação em diferente métodos de diferentes objetos coligados, o que leva a necessidade de analisar o fluxo de chamada de método para entender a lógica [Banaszewski 2009].

Esse problema continua mesmo na manutenção de programas. Devido aos relacionamentos entre os objetos que compõem o *software* OO, a alteração de uma pequena parcela do conhecimento em algum ponto do código-fonte pode, muitas vezes, resultar na necessidade de alteração em outros, criando assim uma reação em cadeia em termos de alterações a serem realizadas.

Sendo assim, a inclusão de uma simples expressão causal (*se-então*) pode se tornar uma tarefa difícil e complexa com o POO. Primeiramente, deve-se encontrar a posição

correta que a expressão deve ser inserida no código-fonte. Na sequência, deve-se analisar o fluxo de execução, uma vez que a mesma não pode afetar a avaliação das demais expressões e deve apresentar oportunidade de ser avaliada [Banaszewski 2009].

2.1.2 PARADIGMA ORIENTADO A EVENTOS

Considerado por alguns como um paradigma de programação atualmente associado principalmente ao POO (ainda que haja tecnologias com PP/PI e mesmo PD) [Brookshear 2012, Ferg 2006], a programação orientada a eventos (POE) busca amenizar a redundância temporal presente nas aplicações através do controle de fluxo determinado por eventos. Um evento, por sua vez, é uma condição detectada que pode disparar uma notificação, a qual será enviada para um recebedor definido em tempo de execução [Faison 2006] (conforme Figura 4).

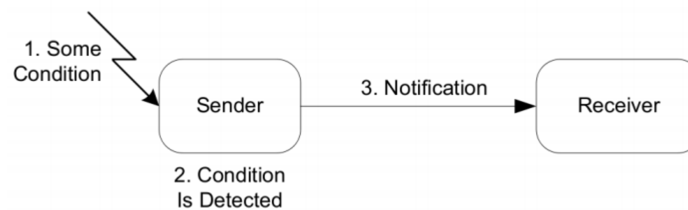

Figura 4: Evento, condição detectada e notificação [Faison 2006].

Um evento ocorre de forma imprevisível e pode ser exemplificado por um botão pressionado, uma interrupção de *hardware* ou uma mensagem recebida, oriunda de uma entidade externa à aplicação [Xavier 2014]. Esse evento instiga uma determinada ação (*i.e* método), conforme Figura 5. A ação normalmente está contida em um tipo determinado de módulo, como um bloco, objeto, observador, consumidor ou, até mesmo, agente [Eugster et al. 2003, Faison 2006, Hansen e Fossum 2010].

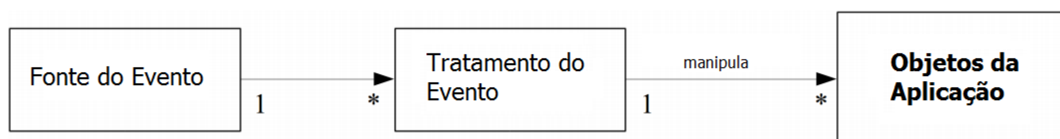

Figura 5: Paradigma Orientado a Eventos adaptado de [Hansen e Fossum 2010].

Em geral, um *software* baseado em eventos é mais simples sob o ponto de vista estrito de código [Faison 2006]. Isto ocorre porque a POE propicia componentes menores, razoavelmente coesos e desacoplados amenizando redundâncias. Entretanto, ainda que amenize questões de redundâncias, POE não as resolve pois em cada procedimento ou

similar disparado por evento tal deficiência não raro permanece. Ademais, aumenta-se a dispersão lógica e os relacionamentos do *software*, sendo necessário percorrer e compreender todo o conjunto de unidades de uma aplicação [Pordeus 2016, Faison 2006, Brookshear 2012, Xavier 2014].

2.1.3 PARADIGMA DECLARATIVO

Diferentemente do PI, o Paradigma Declarativo (PD) possibilita uma programação de mais alto nível quando comparada ao PI, permitindo ao programador focar mais na organização do conhecimento para a solução do problema computacional do que na forma de implementação do mesmo. Neste sentido, o PD exige que o programador descreva as regras que compõem o conhecimento lógico-causal do sistema, ao invés de escrever Códigos na forma de sequência de instruções que solucionem o problema, como ocorre no PI [Riley e Giarratano 1993, Krug 2016].

No PD, enfatizando Sistemas Baseados em Regras (SBR), *softwares* são compostos por elementos da Base de Fatos, os quais podem possuir atributos e métodos similares a objetos da POO. Posteriormente, define-se uma Base de Regras com relações causais relativas às entidades da Base de Fatos. Estas duas bases são processadas por meio de uma Máquina de Inferência, a qual compara as regras e fatos (e.g estados dos atributos) gerando assim novos fatos e, portanto, um novo ciclo de inferência [Xavier 2014]. Estes componentes formam a arquitetura dos SBR, como apresentado na Figura 6.

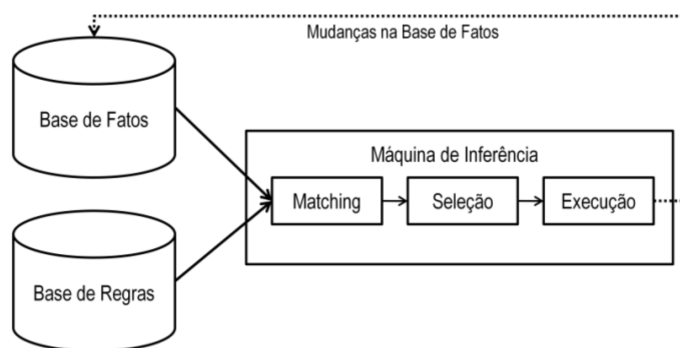

Figura 6: Arquitetura de um Sistema Baseado em Regras (SBR).

Um ciclo de inferência do SBR consiste em três fases distintas:

- Matching: essa fase compara os fatos em relação as regras. As regras aprovadas são então colocadas, de forma desordenada, em um repositório chamado conjunto de conflito (i.e *conflict set*).

- Seleção: nessa fase as regras presente no conjunto de conflito são ordenadas segundo uma estratégia de resolução de conflitos, como a estratégia baseada na prioridade das regras ou a estratégia relativa a recenticidade dos fatos que ativaram as regras, para formar o conjunto ordenado de regras chamado *Agenda*.

- Execução: nessa fase a primeira regra da *Agenda* é selecionada e a ação relacionada a mesma é executada. Nesta execução, a regra pode inserir novos elementos na Base de Fatos ou, até 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. Nos primeiros SBR, durante a fase de *matching*, cada regra era avaliada contra os estados da cada um dos elementos da base de fatos. Com isso, essa fase do ciclo de inferência correspondia a aproximadamente 90% do tempo de execução dos SBR [Miranker e Lofaso 1991]. A ineficiência destes sistemas ocorria principalmente devido à avaliação redundante entre fatos e regras, uma vez que muitos desses testes realizados durante a fase de *matching* apresentavam o mesmo resultado dos ciclos anteriores, gerando assim desperdício de processamento [Banaszewski 2009].

Para solucionar, ou ao menos amenizar, a ineficiência da fase de *matching*, algumas soluções foram propostas. Basicamente os novos Códigos de inferência buscam guardar os estados já avaliados em ciclos anteriores. Dessa forma, as regras são comparadas apenas com os estados dos elementos da base de fatos atualizados recentemente. Estas soluções são chamadas de Códigos de inferência incrementais. Exemplos deste tipo de Código são o RETE [Forgy 1982], o TREAT [Miranker 1987], o LEAPS [Miranker et al. 1990] e o HAL [Lee e Cheng 2002].

2.2 PARADIGMA ORIENTADO A NOTIFICAÇÕES

Esta seção detalha o Paradigma Orientado a Notificações (PON), o qual foi brevemente introduzido na Subseção 1.1. Primeiramente, nesta presente seção, a Subseção 2.2.1 apresenta a relação existente entre o PON e os paradigmas Imperativo e Declarativo. A Subseção 2.2.2 apresenta o mecanismo de notificação do PON. Por sua vez, a Subseção 2.2.3 aborda o mecanismo de resolução de conflitos e garantias de determinismo em aplicações PON. A Subseção 2.2.4, particularmente, reflete e contextualiza sobre as propriedades inerentes ao PON. A Subseção 2.2.5 define as características de utilização e compreensão do PON. Ainda, a Subseção 2.2.6, detalha a função assintótica do PON em relação ao processo

de resolução de cálculo lógico-causal. Subsequentemente, a subseção 2.2.7 apresenta os detalhes sobre a propriedade do PON denominada *Attributes* impertinentes. Por fim, a subseção 2.2.8 apresenta o conceito de Regras de Formação do PON.

2.2.1 RELAÇÃO ENTRE O PON E OS PARADIGMAS IMPERATIVO E DECLARATIVO

Conforme apresentado no Código 2, escrito em linguagem de programação PON (a qual será apresentado na Seção 2.3.2), o PON aproveita características de ambos os paradigmas previamente apresentados, a saber o PI e o PD. Similarmente ao que ocorre no Paradigma Imperativo, mais especificamente na POO através de seus objetos, no PON é possível declarar entidades computacionais que possuam atributos e métodos. Além disso, a forma como o conhecimento lógico-causal da aplicação é representado no PON é semelhante à base de fatos e de regras dos SBR do Paradigma Declarativo [Banaszewski 2009].

Código 2: Exemplo de código PON.

```
1  fbe Robot
2      attributes
3             boolean atGameStarted false
4      end_attributes
5      methods
6          method mtRunToBall ( ... )
7      end_methods
8  end_fbe
9
10 inst
11     Robot robot1
12 end_inst
13
14 rule rlRunToBallRobot1
15     condition
16         subcondition condition1
17             premise prRobot1Started robot1.atGameStarted == true
18        end_subcondition
19    end_condition
20    action
21        instigation inRobot1Started robot1.mtRunToBall();
22    end_action
23 end_rule
```

Embora aproveite conceitos dos dois paradigmas citados, o PON apresenta uma nova forma de estruturar e executar a lógica de programas computacionais, fato este que justificaria sua classificação como um paradigma [Linhares 2015]. O modelo do PON impõe a expressão da dinâmica de funcionamento de um *software* e da sua lógica de causa e efeito por meio de notificações potencialmente executadas em paralelo [Linhares 2015]. Essa característica o diferencia do comportamento de programas PI, nos quais a lógica do programa é totalmente dependente da sequência de execução e dos SBR, nos quais a sequência de execução é abstraída e dependente do mecanismo de inferência monolítico utilizado [Linhares 2015].

2.2.2 MECANISMO DE NOTIFICAÇÃO DO PON

Um novo conceito para a construção e execução de aplicações de *software* é introduzido pelo Paradigma Orientado a Notificações (PON). No PON, as aplicações são compostas por pequenas entidades reativas e desacopladas, as quais colaboram por meio de notificações precisas e pontuais, ditando assim o fluxo de execução de tais aplicações. Esse novo modo de estruturar e executar *software* tende a proporcionar uma melhora no desempenho das aplicações e, potencialmente, facilitar o desenvolvimento de aplicações distribuídas [Simão e Stadzisz 2008, Simão e Stadzisz 2009, Simão et al. 2012].

Graças ao orquestramento da cadeia de notificações pontuais entre as entidades PON, emergido do código PON em tempo de construção, o fluxo de execução das aplicações PON é realizado de modo transparente ao desenvolvedor. Isto diferencia-se do fluxo de execução encontrado em aplicações do PI, incluindo o subparadigma OO, no qual o desenvolvedor informa de maneira explícita o laço de iteração através de comandos como *while* e *for*. No PON, a repetição ocorre de forma natural na perspectiva de execução da aplicação a partir da mudança de estado de um *Attribute*, conforme exemplificado na Figura 7.

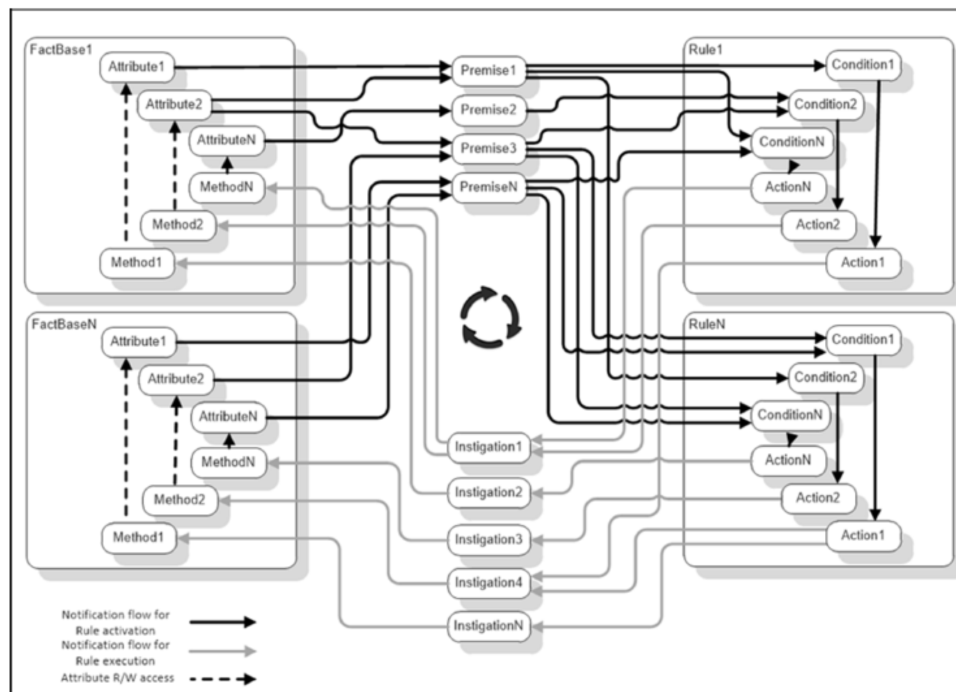

Figura 7: Inferência por notificações [Simão et al. 2014].

O fluxo de execução inicia-se a partir da mudança de estado de um *Attribute* de um determinado *FBE*. Após sua mudança de estado, o *Attribute* notifica todas as *Premises* pertinentes, a fim de que estas reavaliem seus estados lógicos. Caso o valor lógico de uma

Premise se altere, a *Premise* colabora com a avaliação lógica de uma ou de um conjunto de *Conditions* conectadas, o que ocorre por meio da notificação sobre a mudança relacionada ao seu estado lógico [Banaszewski 2009].

Consequentemente, cada *Condition* notificada avalia o seu valor lógico de acordo com as notificações recebidas das *Premises* e com um dado operador lógico, sendo normalmente tal operador de conjunção ou disjunção. Assim, no caso de uma conjunção por exemplo, quando todas as *Premises* que integram uma *Condition* são satisfeitas, a *Condition* também é satisfeita. Isto resulta na aprovação de sua respectiva *Rule* que pode então ser executada [Banaszewski 2009].

Sendo assim, quando uma dada *Rule* aprovada está pronta para ser executada (*i.e* aprovada e com conflitos resolvidos conforme discutido na próxima subseção), a sua *Action* é ativada. Uma *Action*, por sua vez, é conectada a um ou vários *Instigations*. Os *Instigations* colaboram com as atividades das *Actions*, acionando a execução de algum serviço de um objeto *FBE* por meio dos seus *Methods*. Usualmente, os *Methods* alteram os estados dos *Attributes*, recomeçando assim o ciclo de notificações. [Banaszewski 2009]

Oportunamente, as conexões entre os objetos notificantes são estabelecidas em tempo de criação. Por exemplo, toda *Premise* quando criada deve possuir pelo menos um *Attribute* associado. Uma vez que um *Attribute* é referenciado em uma *Premise*, o *Attribute* considera automaticamente esta *Premise* como sendo interessada em receber notificações sobre qualquer alteração em seu estado. Assim, o *Attribute* identifica todas as *Premises* interessadas e notifica-as quando o seu estado é alterado. Ainda, mecanismo similar ocorre em relação às *Premises* e as *Conditions*, bem como as relações entre *Conditions* e *Rules* [Banaszewski 2009].

2.2.3 RESOLUÇÃO DE CONFLITOS NO PON

Um conflito ocorre quando duas ou mais *Rules* referenciam um mesmo *FBE* e demandam exclusividade de acesso ao mesmo. Isto posto, as *Rules* concorrem para adquirir acesso exclusivo a este *FBE*, sendo que somente uma destas *Rules* deve ser executada por vez afim de garantir determinismo e consistência. Neste âmbito, visando solucionar conflitos entre as *Rules*, o fluxo de execução das mesmas é determinado segundo uma estratégia preestabelecida. Visando alcançar o fluxo de execução pretendido, o desenvolvedor pode escolher qual estratégia será utilizada para a resolução de conflitos [Banaszewski 2009].

Em um ambiente monoprocessado, a resolução de conflitos visa estabelecer uma

ordem de execução para *Rules*, de forma que apenas uma *Rule* possa ser executada por vez. Para isso, é empregado um escalonador de *Rules* formado por uma estrutura de dados do tipo linear (*e.g* fila, lista ou pilha) [Banaszewski 2009]. Tais estruturas recebem as *Rules* na ordem em que são aprovadas e as organiza de acordo com os preceitos da estratégia de resolução de conflitos adotada [Banaszewski 2009], conforme ilustrado na Figura 8.

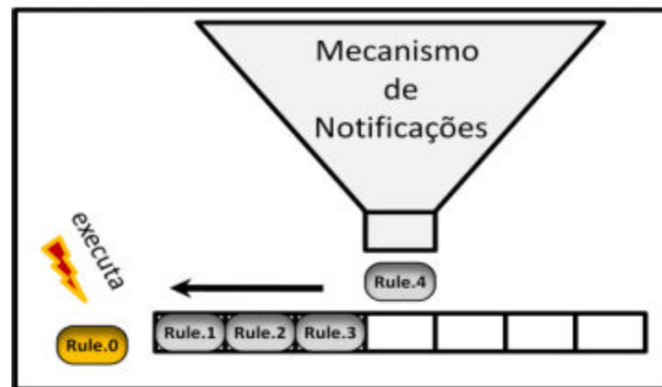

Figura 8: Modelo Centralizado de Resolução de Conflitos [Banaszewski 2009].

Os modelos de resolução de conflitos empregados para o PON em ambientes monoprocessados são:

- **BREADTH** ou Largura: baseia-se no escalonamento *First In, First Out (FIFO)*, ou seja, refere-se à execução de *Rules* seguindo a estrutura de dados do tipo fila;
- **DEPTH** ou Profundidade: baseia-se no escalonamento *Last In, First Out (LIFO)*, ou seja, refere-se à execução de *Rules* seguindo a estrutura de dados do tipo pilha; e
- **PRIORITY** ou Prioridade: organiza as *Rules* de acordo com as prioridades definidas na criação dass mesmas.

Quando nenhuma estratégia de resolução de conflito for definida em uma aplicação PON, utiliza-se a estratégia NO_ONE por padrão. Essa estratégia faz com que as entidades *Rules* não sejam enviadas ao escalonador, sendo imediatamente executadas após respectiva aprovação.

As estratégias de resolução de conflitos apresentadas são particularmente aplicáveis a aplicações PON monoprocessadas, ainda que possam ser úteis em soluções multiprocessadas e distribuídas [Banaszewski 2009]. Entretanto, a resolução de conflitos em aplicações PON concorrentes e/ou distribuídas deveriam ser realizadas a partir de soluções que lhe sejam mais apropriadas (*i.e.* não centralizadoras). Ainda que este trabalho não seja relativo à aplicações PON em sistemas concorrentes ou distribuídos, os trabalhos [Simão

2005, Banaszewski 2009, Simão et al. 2010] apresentam soluções úteis para a resolução de conflitos em ambientes distribuídos, bem como soluções correlatas para certa garantia de determinismo.

2.2.4 PROPRIEDADES INERENTES AO PON

É notável que a essência computacional no PON está organizada e distribuída entre entidades reativas que colaboram por meio de notificações pontuais. Este arranjo forma o mecanismo de notificações apresentado na subseção 2.2.2 o qual determina o fluxo de execução das aplicações PON.

Através desse mecanismo, as responsabilidades de um programa PON são divididas entre as diferentes instâncias do modelo. Neste sentido, a colaboração por meio de notificações pontuais e precisas representaria a solução para parte das deficiências dos atuais paradigmas de programação.

Neste âmbito, ao evitar buscas sobre entidades passivas, o PON implicitamente evita redundâncias estruturais e temporais que tanto afetam o desempenho de aplicações no PI e mesmo no PD [Banaszewski 2009]

Ademais, os objetos participantes da cadeia de notificação do PON apresentam-se desacoplados, devido à comunicação realizada por meio de notificações pontuais e precisas. Dessa forma, pode-se considerar que o PON seja apropriado para a execução em ambientes multiprocessados, uma vez que cada objeto notificante precisa apenas conhecer os endereços dos objetos a serem notificados para que o ciclo de notificações ocorra [Banaszewski 2009, Simão et al. 2012]. Isto tem sido demonstrado em trabalhos sobre o PON [Peters et al. 2012, Linhares et al. 2015, Belmonte et al. 2016]

2.2.5 PON – UTILIZAÇÃO X COMPREENSÃO

O PON permite uma nova maneira de estruturar, executar e pensar os artefatos de *software*. Ainda que o PON permita compor *software* em alto nível na forma de regras sem o conhecimento de sua essência, conhecê-la é importante em certas situações [Simão et al. 2012]. A compreensão dos princípios do PON é importante para aplicações complexas, onde o fluxo de notificações é intenso e precisa-se de maior formalismo e rastreabilidade, como em aplicações de tempo real e controle discreto. Na verdade, esse tipo de aplicação pode exigir apoio de ferramentas formais para elaboração do projeto [Simão et al. 2012].

Um exemplo particular de formalismo é o DON (*Desenvolvimento Orientado a*

Notificações, proposto prototipalmente no ano de 2011 em [Wiecheteck et al. 2011]. O DON é um método para projeto de *software* PON. A solução proposta é baseada em um perfil UML que expressa os conceitos PON e viabiliza a sua aplicação na etapa de modelagem de um processo de engenharia de *software* [Wiecheteck et al. 2011, Linhares 2015]. Atualmente, uma Metodologia de Projeto de *Software* Orientada a Notificações (MON) está sendo desenvolvida como uma metodologia iterativa e incremental, na qual os elementos fundamentais do PON são considerados desde os primeiros níveis de modelagem [Mendonça 2016].

2.2.6 CÁLCULO ASSINTÓTICO DA INFERÊNCIA DO PON

Em seu pior cenário, a complexidade assintótica polinomial do PON é representada 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 *nConditions* corresponde ao tamanho máximo de objetos *Conditions* notificados por estas *Premises* [Simão 2005, Banaszewski 2009]. A função assintótica apresentada para o PON, no pior cenário, demonstra uma função bastante similar, mas ainda assim mais eficiente, ao mecanismo de notificações do Código *HAL* e mais eficiente do que os Códigos de inferência *RETE, TREAT* e *LEAPS*, sendo esses dois derivados do *RETE* [Banaszewski 2009].

Essa função assintótica representa a ordem da quantidade de notificações que ocorre entre objetos colaboradores, o que também corresponde à ordem da quantidade de avaliações lógicas. A constatação desta função assintótica pode ser realizada pela análise da Figura 9, a qual demonstra as relações por notificações entre os objetos colaboradores. Nesta, os *Attributes*, *Premises*, *Conditions* e *Rules* correspondem, respectivamente, aos símbolos *Att*, *Pr*, *Cd* e *Rl* [Banaszewski 2009].

Outra forma menos ortodoxa de analisar a complexidade assintótica do PON é considerar o caso médio, ao invés do pior caso, uma vez que o pior caso é irrealista dado que seria improvável que todos os *Attributes* notificassem todas as *Premisses* e assim por diante.

Neste sentido, a análise da complexidade do caso médio é iniciada analisando-se o começo do processo de notificação do PON através da entidade *Attribute*. Assim, as principais variáveis envolvidas em uma notificação de um *Attribute* são demonstradas pela Equação 1.

$$ppgcaFB_{at} = NumPremises + NumRules \qquad (8)$$

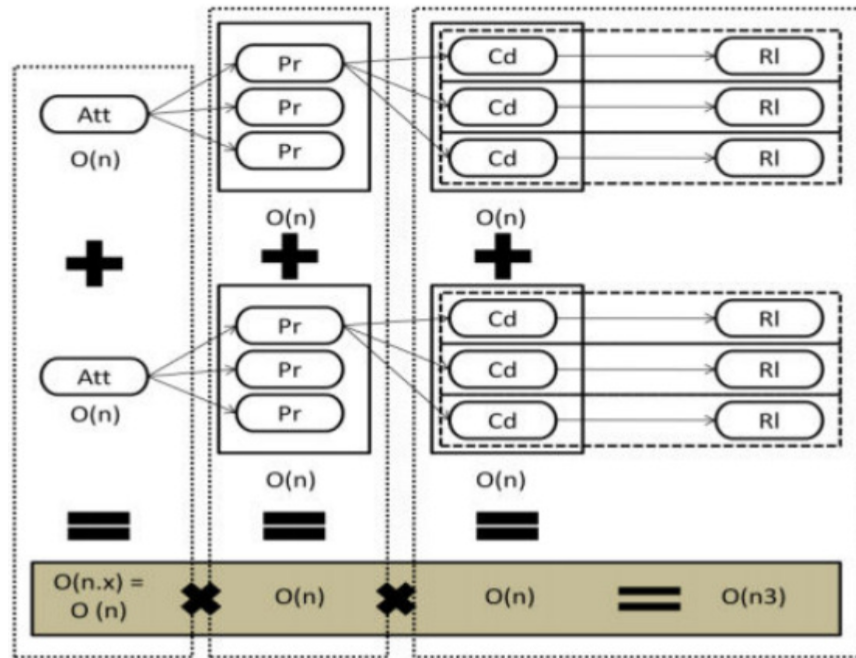

Figura 9: Cálculo assintótico do mecanismo de notificações [Banaszewski 2009]

A variável *NumPremises* representa a soma de entidades *Premises* relacionadas ao respectivo *Attribute* e a variável *NumRules* é a soma das entidades *Rules* relacionadas a cada entidade *Premise* contada em *NumPremises*. Portanto, se for considerado simplesmente cada ciclo de inferência como a instigação de um *Attribute* e w sendo o número de todos os *Attributes* existentes, uma média possível seria:

$$TMedium(x) = (FBAT.1() + ... + FBAT.w())/w \tag{9}$$

Assim, o resultado desta média seria uma ordem de (n), o que implicaria uma complexidade linear [Simão 2005].

2.2.7 *ATTRIBUTES* IMPERTINENTES

A reatividade presente nos *Attributes* proporciona, na maioria dos casos, uma execução livre de avaliações redundantes e desnecessárias, comuns aos paradigmas de programação usuais. Entretanto, existem casos em que a variação de um *Attribute* encadeia uma sequência de notificações indesejáveis [Ronszcka 2012].

Isso ocorre em situações nas quais um dado *Attribute* apresenta mudanças frequentes de estado, disparando o fluxo de notificações a cada variação, sem afetar efetivamente a aprovação da *Rule* associada. Isto poderia ser grave em casos de muitas notificações

desnecessárias, as quais impactariam negativamente o desempenho de uma aplicação PON [Ronszcka 2012].

De modo a ilustrar o problema da impertinência, considera-se o exemplo ilustrado na Figura 10. Neste exemplo, são apresentado dois *Attributes* distintos de um *FBE* que representa um robô SSL da *RoboCup*. O primeiro *Attribute* é *atWithBall* (*Boolean*), o qual indica se o robô está com a posse da bola, e o outro é *atDistanceGoal* (*Double*), o qual representa a distância do robô ao gol adversário. Ambos participam de uma *Condition/Rule* composta por duas *Premises*, sendo que a primeira *Premise* avalia se o estado de *atWithBall* é verdadeiro, enquanto a outra *Premise* avalia se o valor de *atDistanceGoal* é menor do que um dado valor preestabelecido.

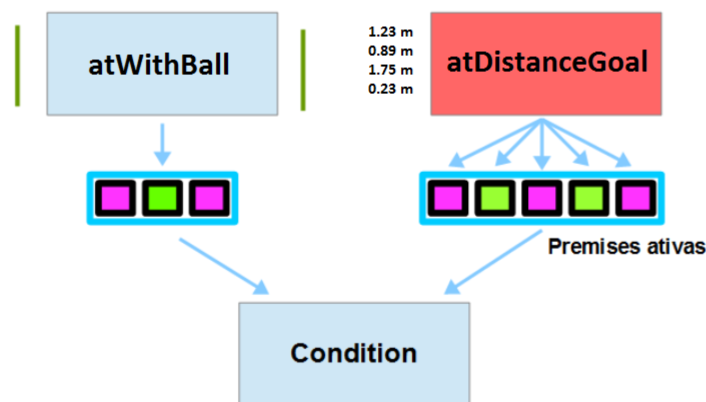

Figura 10: Impacto nas alterações de estado de *Attributes* ativos (adaptado de [Ronszcka 2012]).

O *Attribute atWithBall* apresentaria poucas variações em seu estado, permanecendo a maior parte do tempo com o estado *false*, disparando assim o fluxo de notificações esporadicamente. Por sua vez, o *Attribute atDistanceGoal* apresentaria alterações constantes em seu estado, uma vez que o robô está em constante movimentação pelo campo de jogo, gerando assim constantes notificações a *Rule* associada.

Neste sentido, um *Attribute* como *atDistanceGoal* pode ser categorizado como 'impertinente'. Neste dado contexto, o *Attribute* passa a ter suas funções reativas desabilitadas temporariamente, conforme ilustrado na Figura 11. Dessa forma, as variações de estado do *Attribute atDistanceGoal* não iniciariam o fluxo de notificações para as *Conditions-Rules* associados ao *Attribute*.

Neste âmbito, quando o conjunto dos *Attributes* aprovar suas respectivas *Premises* em uma dada *Condition-Rule*, essa deve solicitar a reativação das notificações para a *Premise* composta pelo *Attribute* impertinente. Assim, uma vez que o *Attribute atWithBall*

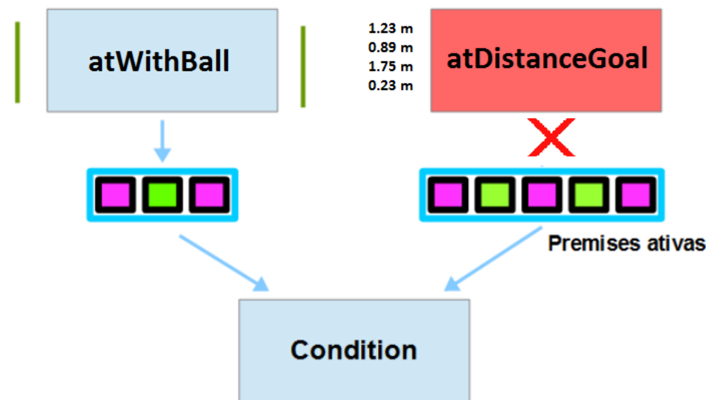

Figura 11: Impacto nas alterações es estado de *Attributes* impertinentes (adapatado de [Ronszcka 2012])

.

apresentasse estado verdadeiro, a *Conditiion* ilustrada solicitaria a reativação da *Premise* correspondente ao *Attribute atDistanceGoal*, conforme ilustrado na Figura 12

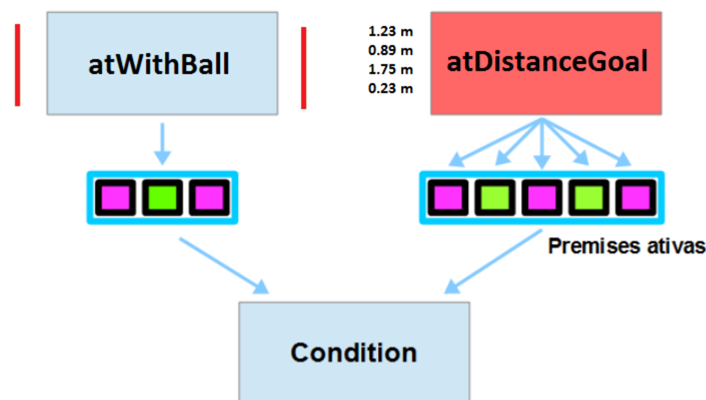

Figura 12: Exemplo de reativação de uma entidade desativada (adapatado de [Ronszcka 2012])

.

Dessa forma, o *Attribute atWithBall*, ao apresentar o estado verdadeiro, colocaria tal *Condition* em "ponto de aprovação", o que reativaria as funções reativas da *Premise* desconsiderada. Assim, a entidade *Premise* em questão permitiria ser notificada novamente pelo *Attribute atDistanceGoal*. Por fim, após a devida aprovação e execução da *Rule* em questão, o *Attribute* impertinente voltaria a ignorar tal *Premise* até que seja requisitado novamente por outra *Condition* [Ronszcka 2012].

2.2.8 REGRAS DE FORMAÇÃO

Definido por Simão em [Simão 2001], uma Regra de Formação ou *Formation Rules* é uma entidade sob a forma de regra que explica como será criada uma ou mais *Rules*.

Desse modo, uma *Formation Rule* não é uma *Rule* a ser executada, mas sim uma forma genérica de constituição de um conjunto de *Rules* [Simão 2001].

Neste sentido, a utilização de *Formation Rule* permite a criação de *Rules* específicas, a partir da representação genérica de uma *Rule*. Este conceito é bastante útil quando o conhecimento causal de uma *Rule* é comum para diferentes conjuntos de instâncias de *FBEs*, ou seja, um conjunto de *Rules* específicas se diferencia apenas nas instâncias de *FBEs* referenciadas [Pordeus 2016].

2.3 MATERIALIZAÇÕES DO PON

Os conceitos do PON foram primeiramente materializados sobre o POO, através de um arquétipo ou *Framework* desenvolvido com a linguagem de programação C++. Tal materialização possibilitou a demonstração dos principais conceitos do PON, permitindo a criação de aplicações sob o domínio desse novo paradigma. Entretanto, as aplicações desenvolvidas utilizando o *Framework* apresentaram desempenho não tão satisfatório quanto esperado a luz do cálculo assintótico do PON. Isto se dá principalmente devido ao uso de estruturas de dados computacionalmente caras, tais como listas duplamente encadeadas da biblioteca *Standard Template Library* (STL), que foram utilizadas no seu desenvolvimento. [Banaszewski 2009, Valença 2013].

Concluiu-se então que existia a necessidade de melhorias no estado da técnica para que o PON atingisse o que fora vislumbrado em seu estado da arte. Neste âmbito, foi desenvolvida uma linguagem de programação específica, acompanhada do respectivo compilador, que traduz o código PON para código-alvo mais puro, isto é, com menos depêndencia de conceitos de outros paradigmas, evitando assim o uso de estruturas de dados complexas [Ferreira 2016].

Ademais, vislumbrou-se a possibilidade de materializar o PON através de implementações em *hardware*, de modo que aplicações pudessem ser concebidas seguindo o modelo PON de maneira mais fidedigna no tocante ao paralelismo [Pordeus 2016]. Alguns trabalhos foram realizados neste âmbito, com destaque a [Witt et al. 2011, Jasinski 2012, Peters 2012, Kerschbaumer et al. 2015]. Recentemente, como outro avanço no âmbito de PON em hardware, Linhares (2015) propôs uma nova abordagem para execução de *software* PON, na forma de uma arquitetura de computador denominada *Notification Oriented Computer Paradigm* (NOCA) [Linhares et al. 2015].

Entretanto, esse trabalho visa a evolução do estado da técnica do PON no âmbito

de *software*. Dessa forma, as próximas subseções apresentam maiores detalhes apenas sobre as materializações em software do *PON* que são consideradas assaz estáveis e testadas.

Além de detalhar as materializações em *software* do PON, este trabalho apresenta também um censo das aplicações PON desenvolvidas até o presente momento. Maiores informações sobre estas aplicações são apresentadas no Apêndice E.

2.3.1 *FRAMEWORK* PON

O *Framework* PON consiste em uma estrutura de elementos orientada a objetos, construída com a utilização da linguagem de programação C++, correspondentes às entidades que compõem o modelo do PON e a luz do PON. O principal objetivo deste *Framework* é oferecer uma interface de programação para a implementação de aplicações sob o viés do PON, definindo as abstrações necessárias para compor os *FBEs* e respectivas *Rules*, tanto *FBEs* quanto *Rules* com seus respectivos constituintes [Linhares 2015, Valença 2013].

A versão 1.0 do *Framework* PON foi desenvolvida por Banaszewski [Banaszewski 2009] a partir de uma versão prototipal concebida por Simão em 2007 [Simão et al. 2012], sendo esta versão prototipal derivado de seus esforços de dissertação de mestrado e tese de doutorado no âmbito do atualmente chamado Controle Orientado a Notificações (CON) [Simão 2001, Simão 2005]. Por ser um conjunto de classes materializados sobre o PI, o *Framework* PON 1.0 tem sua implementação baseada em percorrimentos sobre estruturas de dados, fornecidas pela STL - *Standard Template Library*[2], para avaliação de relações lógico-causais e envio de notificações [Linhares 2015]. A principal vantagem de uma materialização construída sobre o POO/PI é a possibilidade de rápida prototipação de aplicações PON para teste sobre plataformas de computação convencionais, o que inicialmente permitiu a demonstração da viabilidade do PON. Entretanto, esta forma de implementação é desvantajosa à filosofia do PON em si, pois fundamenta seu funcionamento em estruturas de dados caras e percurso sequencial sobre estas estruturas, o que é natural em PI [Linhares 2015].

Dadas algumas questões de implementação do *Framework* PON em sua versão 1.0, em particular a degradação de desempenho causada por sua implementação baseada em estruturas de dados baseadas em *Standard Template Lists* (STL) do C++, Ronszcka e Valença (2012) efetuaram uma série de otimizações com o objetivo de melhorar o desempenho de execução de aplicações construídas utilizando o *Framework*. Um dos

[2]Biblioteca padrão/standard de gabaritos/templates do C++.

resultados deste trabalho é uma nova versão do *Framework* PON (versão 2.0) baseada em uma variedade de estruturas de dados mais otimizadas do que as equivalentes fornecidas pela STL. Tais novas estruturas contemplam vetores (PONVECTOR), listas (PONLIST) e tabelas *hash* (PONHASH) [Ronszcka 2012, Valença 2013] . Esta versão do *Framework* apresentou ganhos significativos de desempenho em diversas aplicações quando comparado à sua versão 1.0 [Valença 2013].

Conforme ilustrado na Figura 13, o *Framework* PON 2.0 é subdividido em três pacotes principais. O pacote *Application* possui apenas a classe *Application*, a qual relaciona uma aplicação PON às demais classes que compõem o *Framework*. O pacote *Scheduler*, por sua vez, possui as classes que materializam as estratégias de resolução de conflitos, conforme descrito na Subseção 2.2.3. Por fim, o pacote *Core* é formado pelas classes que materializam as entidades colaborativas do PON, conforme apresentado na Figura 14.

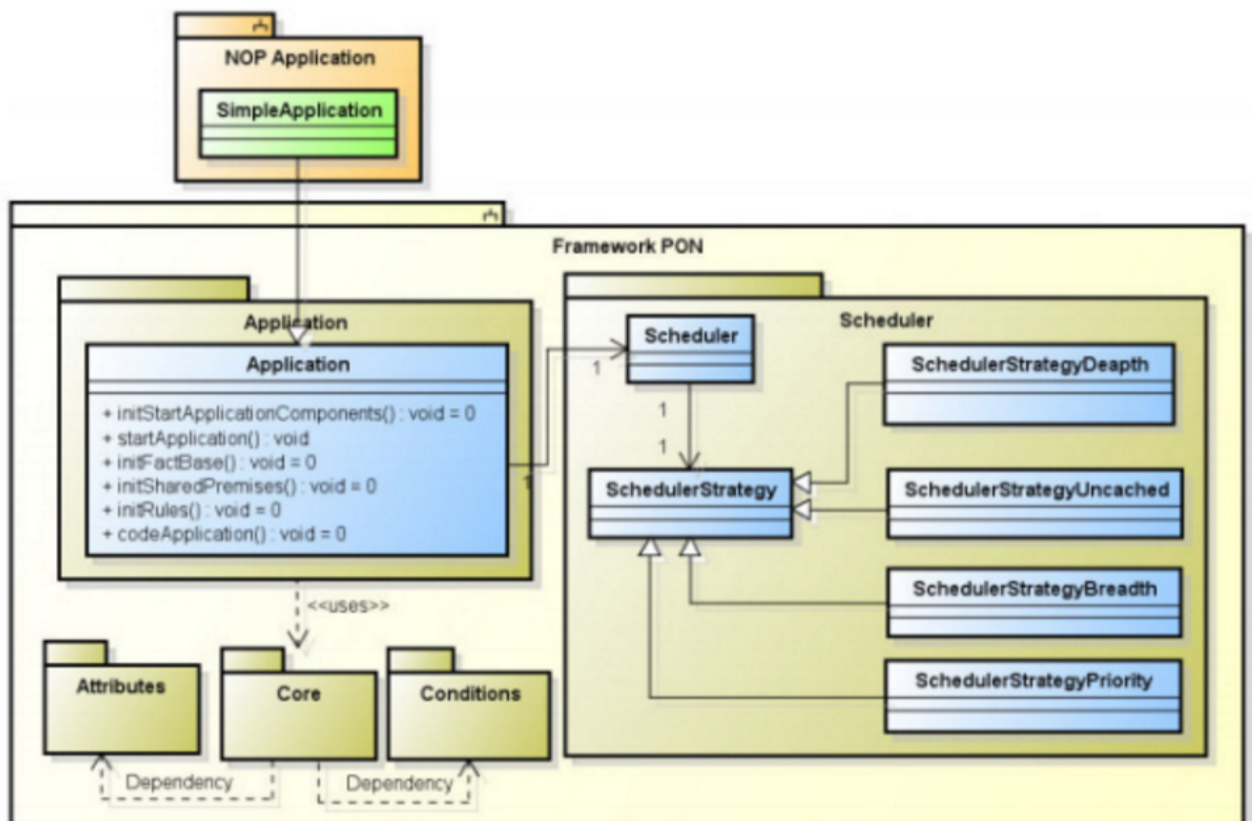

Figura 13: Estrutura do *Framework* PON [Linhares et al. 2011]

.

Conforme apresentado na Figura 14, as classes *Rule* e *FBE* se apresentam nas extremidades opostas do diagrama de classes e se relacionam através de classes ditas colaborativas, isto é *Attribute*, *Premise*, *Condition*, *Action*, *Instigation* e *Method*. Em

tempo, é a colaboração entre as instâncias destas classes que determina o fluxo de execução da aplicação PON.

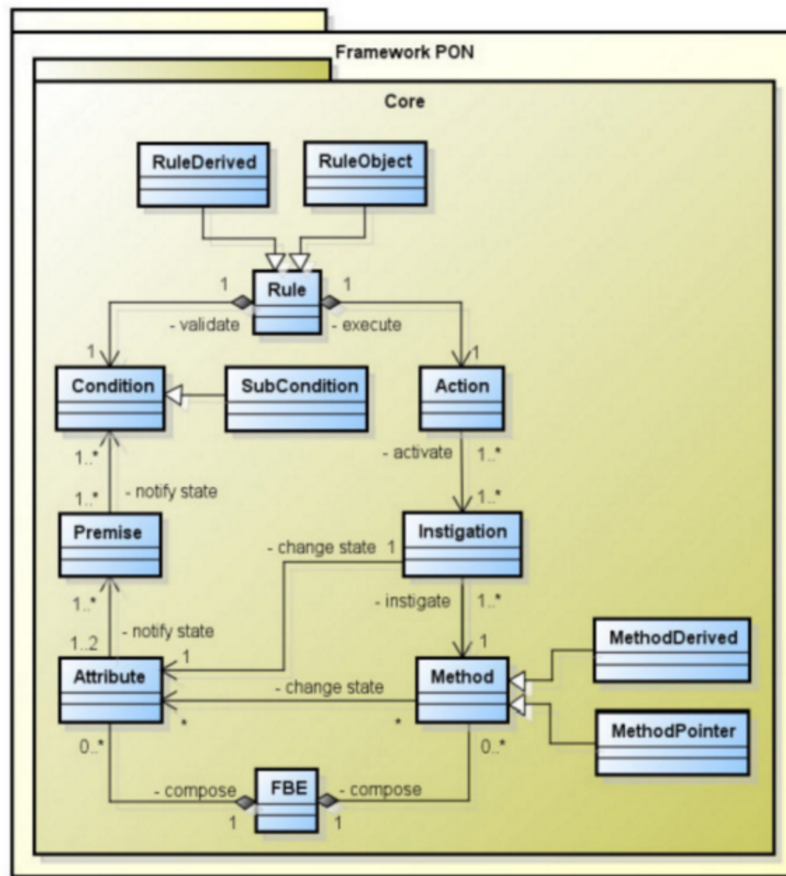

Figura 14: Diagrama de Classes do pacote *Core* [Linhares et al. 2011]
.

Ainda, o pacote *Core* é composto pelo subpacotes *Attributes* e *Conditions*. Conforme apresentado na Figura 15, o subpacote *Attributes* é composto pelas classes que encapsulam os tipos primitivos do POO (nomeadamente *Boolean*, *Char*, *Double*, *Integer* e *String*), introduzindo aos tipos primitivos reatividades de forma a permitir que estes façam parte da estrutura de notificações do PON. O subpacote *Conditions*, por sua vez, é composto pelas diferentes operações lógicas previstas (nomeadamente *Conjuction*, *Disjunction* e *Single*) que compõe a definição de uma *Condition* no PON.

Para desenvolver aplicações utilizando o *Framework* PON, inicialmente é necessário a criação de uma classe principal a qual estenda a classe *NOPApplication*. Conforme apresentado na Figura 16, a classe *NOPApplication* apresenta métodos abstratos, os quais constituem uma ponte entre a aplicação PON e o cerne do *Framework*.

A criação de entidades PON via *Framework* é dada pelo uso de uma fábrica de entidades, a qual possui como responsabilidade principal instanciar tais entidades adaptadas

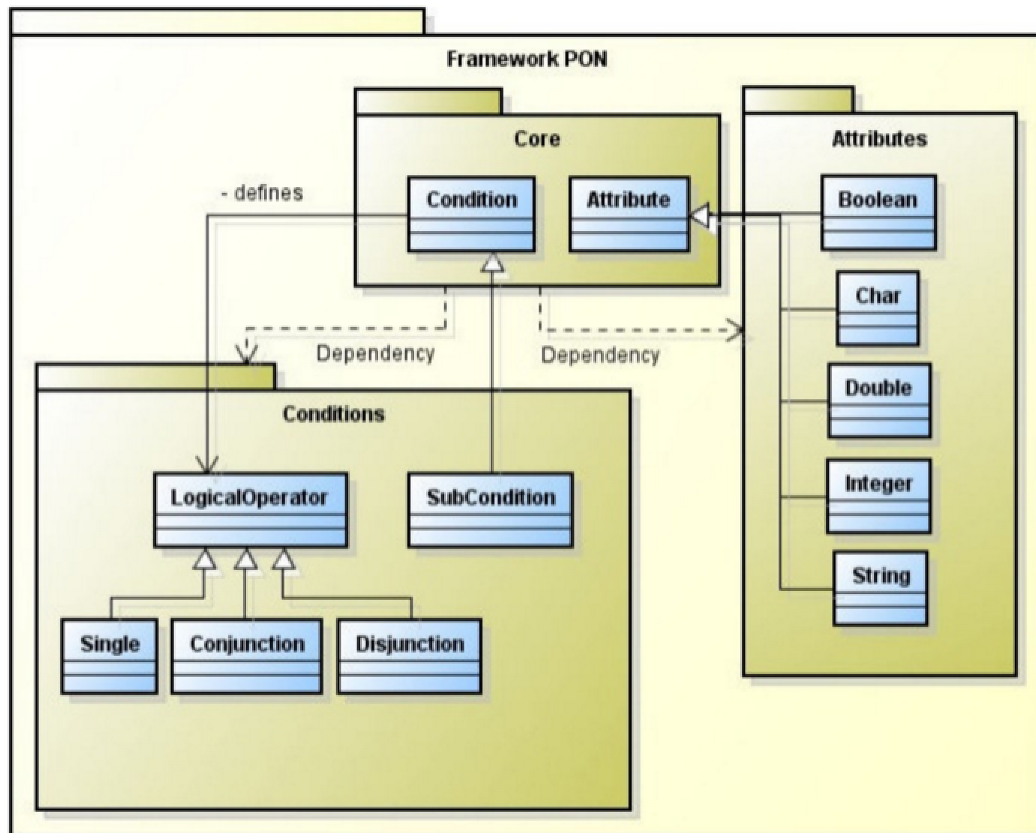

Figura 15: Diagrama de Classes dos subpacotes *Attributes* e *Conditions* [Linhares et al. 2011]

.

a uma estrutura de dados específica [Ronszcka 2012]. O Código 3 demonstra a inicialização dos componentes iniciais de uma aplicação PON, em especial a classe *SingleFactory* (linha 2), a qual é responsável pela instanciação das entidades PON. As possíveis fábricas a serem utilizadas na criação de entidades PON são NOP_LIST, NOP_VECTOR, NOP_HASH e STL_LIST.

Código 3: Inicialização dos componentes iniciais de uma aplicação PON

```
1 void Main::initStartApplicationComponents() {
2     SingletonFactory::changeStructure(SingletonFactory::NOPVECTOR);
3     SingletonLog::changeStream(SingletonLog::CONSOLE);
4     SingletonScheduler::changeScheduler(SchedulerStrategy::NO_ONE);
5     this->startApplication();
6 }
```

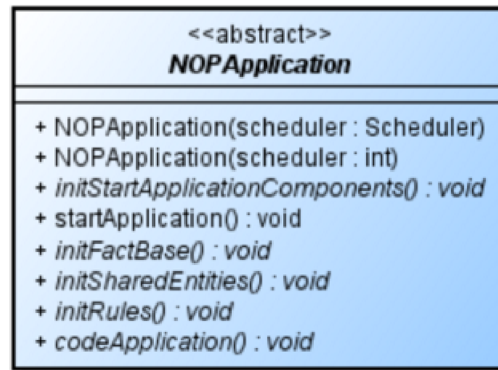

Figura 16: Diagrama de classes do procedimento inicial de uma aplicação PON [Ronszcka 2012]

.

Ademais, no Código 3 é mostrado que o método *initStartApplicationComponents* pode ser utilizado para inicializar o gerador de *logs* (linha 3), definir a estratégia de resolução de conflito de *Rules* (linha 4) e, por fim, o método *startApplication* deve ser invocado. Em tempo, o método *startApplication* é responsável por chamar ordenadamente os métodos de criação de uma aplicação PON, conforme apresentado pelo Código 4.

Código 4: Implementação do método *startApplication*

```
1 void NOPApplication::startApplication() {
2     initFactBase();
3     initSharedEntities();
4     initRules();
5     codeApplication();
6 }
```

A composição de entidades de *FBEs* no *Framework* PON se dá por meio da criação e instanciação de classes de *FBEs*. As classes que representam os modelos para criação de entidades *FBEs* pertinentes à aplicação podem conter *Attributes* e *Methods*.

De acordo com a sequência de criação de aplicações, o próximo método herdado da classe *NOPApplication* a ser implementado é o método *initFactBase*. Conforme apresetando no Código 5, nas linhas 2 e 3 são criadas duas instâncias de *FBEs* do tipo *Robot*.

Código 5: Implementação do método *initFactBase*

```
1 void Main::initFactBase() {
2     robot1 = new Robot();
3     robot2 = new Robot();
4 }
```

A composição de *Rules*, por sua vez, define o fluxo de execução de uma aplicação PON, uma vez que são as *Rules* que definem o relacionamento entre as diferentes entidades PON que compõem a aplicação. Conforme apresentado no Código 6, a configuração das *Rules* se dá através da implementação do método *initRules*. A *Rule* descrita no Código 6 foi apresentada na Subseção 1.1.2 e tem por objetivo controlar a movimentação de um robô em uma partida de futebol de robôs.

Código 6: Implementação do método *initFactBase*

```
1 void Main::initRules() {
2     Scheduler * scheduler = SingletonScheduler::getInstance();
3     RULE (rlRule1,scheduler,Condition::CONJUNCTION);
4         rlRule1->addPremise(prIsOn);
5         rlRule1->addPremise(prBallNotMoving);
6         rlRule1->addPremise(prGameStop);
7         rlRule1->addInstigation(inMoveRobot);
8         rlRule1->end();
9 }
```

Conforme apresentado no Código 6, uma *Rule* é criada a partir de três argumentos: o objeto do tipo *RuleObject* (rlRule1), o qual representa a *Rule* que está sendo criada, a referência ao *Scheduler* inicializado e configurado no método *initStartApplicationComponents* (*scheduler*) e o tipo de operação lógica desejado (*Condition::CONJUNCTION*). Entre as linhas 4 e 6, observa-se a associação de entidades *Premises* à entidade *Rule*. Por fim, na linha 7 é possível observar a adição de uma *Instigation* a *Rule*.

2.3.2 LINGUAGEM E COMPILADOR PARA O PON - LINGPON 1.0

A Linguagem e Compilador para o PON surgiram como resultado de esforços da disciplina “Linguagens e Compiladores” [3] ofertada pela UTFPR em 2013 e ministrada por Prof. Dr. João Alberto Fabro e Prof. Dr. Jean Marcelo Simão.

Em suma, a versão prototipal desenvolvida na disciplina foi uma primeira demonstração sobre a viabilidade de se desenvolver uma linguagem e compilador para o PON. Neste contexto, a gramática que define a linguagem de programação PON, doravante denominada LingPON 1.0, foi especificada segundo a *Backus-Naur Form* (BNF) apresentada no Anexo A deste trabalho. Em tempo, não raro e conforme o contexto, o termo LingPON se refere não só a Linguagem mas também a tecnologia de compilação pertinente. Em tempo, na dissertação de C. A. Ferreira esta versão prototipal de linguagem e compilador foi evoluída, atingindo a versão dita LingPON 1.0. Isto considerado, a subseção 2.3.2.1 apresenta a linguagem de programação PON e a subseção 2.3.2.2 apresenta o compilador PON.

2.3.2.1 LINGUAGEM DE PROGRAMAÇÃO PON

De modo geral, o código fonte de uma aplicação desenvolvida utilizando a LingPON segue um padrão de declarações, conforme apresentado pelo Código 7.

---

[3]Disciplina: Tópicos Avançados Em Engenharia De Software. Tema: LINGUAGENS E COMPILADORES. Código CAES101. Programa PPGCA/UTFPR - Prof João Alberto Fabro (em colaboração com Prof. Jean Marcelo Simão - CPGEI/UTFPR). 2 Trimestre de 2013.

Código 7: Padrão de declarações da linguagem PON

```
1  fbe Robot
2      . . .
3  end_fbe
4
5  fbe Ball
6      . . .
7  end_fbe
8
9  inst
10     . . .
11 end_inst
12
13 strategy
14     . . .
15 end_strategy
16
17 rule rlKickBall
18     . . .
19 end_rule
20
21 main {
22     . . .
23 }
```

Primeiramente, o desenvolvedor precisa definir os *FBEs* de seu programa. Em seguida, o desenvolvedor precisa declarar as instâncias de tais *FBEs*, bem como definir a estratégia de escalonamento das *Rules*. Subsequentemente, as *Rules* devem ser definidas para fins de avaliação lógico causal dos estados das instâncias de *FBEs* por meio de notificações. Por fim, é possível adicionar código específico da linguagem alvo escolhida no processo de compilação (e.g C ou C++) com a utilização do bloco de código *main* [Ferreira 2016].

Na gramática do PON, a palavra reservada **fbe** anuncia o início da estrutura de um *FBE*. Por padrão, todos os *FBEs* devem ser definidos no primeiro bloco de código.

Na sequência, as instâncias dos *FBEs* devem ser declaradas dentro do bloco de código iniciado pela palavra reservada **inst**. Em seguida, a estratégia de escalonamento das *Rules* deve ser declarada utilizando-se a palavra reservada **strategy**. Ainda, as *Rules* devem ser definidas fazendo uso da palavra reservada **rule**. Finalmente, a última estrutura a ser declarada é o bloco de código **main**, o qual possibilita a inserção de código nativo na linguagem alvo escolhida durante o período de compilação (i.e C ou C++).

O início do código fonte em LingPON é caracterizado pela declaração dos *FBEs*. Conforme apresentado no Código 8, um *FBE* é composto por dois blocos de código. O primeiro bloco representa a declaração dos *Attributes*, a qual é feita utilizando-se a palavra reservada **attributes**. O segundo representa a declaração dos *Methods*, a qual faz uso da palavra reservada **methods**.

Código 8: Exemplo de declaração de *FBEs* no LingPON

```
1  fbe Robot
2    attributes
3          boolean atConnected false
4          boolean atSetKick false
5          float atSpeed 0.0
6          integer atPosX 0
7          integer atPosY 0
8    end_attributes
9    methods
10       method mtKickBall (atSetKick = true)
11       method mtMoveForward (atPositionX = atPositionX + 1)
12       method mtSendCommand begin_method ... end_method
13   end_methods
14 end_fbe
```

A declaração dos *Attributes* na LingPON segue uma estrutura comumente utilizada pelas atuais linguagens de programação (*e.g. linguagem Java do PI*), formada pelo tipo do *Attribute*, seguido do seu nome e seu respectivo valor inicial. Neste âmbito, os tipos de dados suportados pela atual versão da linguagem são *boolean*, *integer*, *float*, *char* e *string* [Ferreira 2016].

Por convenção, mas não obrigatoriamente, os nomes dos *Attributes* seguem um padrão de nomenclatura, conforme apresentado em outros trabalhos do PON [Ronszcka 2012, Valença 2013], fazendo usdo do prefixo *at*, seguido de um mnemônico que define o propósito do *Attribute* em questão. Por fim, o valor inicial deve estar de acordo com o tipo de dados empregado na construção de tal *Attribute*.

Os *Methods*, por sua vez, apresentam uma construção particular em sua estrutura. Esta se dá pelo anúncio da abertura de um *Method* através da palavra reservada **method** seguida do nome e sua respectiva funcionalidade. Assim como os *Attributes*, os *Methods* também devem, por boas práticas, seguir um padrão de nomenclaturas, utilizando-se o prefixo *mt*. Neste âmbito, o Código 8, entre as linhas 9 e 11 apresenta as possíveis estruturas de construção válidas para os *Methods* na linguagem PON.

Na linha 8 do Código 8, a funcionalidade do *Method mtKickBall* está entre parênteses. Basicamente tal funcionalidade altera o valor do *Attribute atHasKicked* para *true*. Por sua vez, o método *mtMoveForward*, linha 9, apresenta a possibilidade de atribuição de uma operação (e.g soma) em um *Attribute*. Finalmente, o *Method mtSendCommand*, na linha 10, possibilita o desenvolvedor adicionar código nativo da linguagem alvo do compilador, tal como enviar um comando via *socket*, ao bloco de código do método entre as palavras reservadas **begin_method** e **end_method** .

O segundo bloco de código em um programa PON consiste na instanciação dos *FBEs* definidos no primeiro bloco. Para isso, o Código 9 apresenta como tais instanciações devem ser declaradas. No Código 9 é possível observar nas linhas 2 e 3 que duas instâncias do *FBEs Robot* são criadas.

Código 9: Exemplo de instanciações de *FBEs*

```
1 inst
2     Robot robot1
3     Robot robot2
4 end_inst
```

O terceiro bloco de código consiste na definição da estratégia de resolução de conflitos a ser utilizada. Conforme apresentado e explicado na Subseção 2.2.2, o PON apresenta três estratégias de resolução de conflitos (*NO_ONE*, *BREATH* e *DEPTH*). O Código 10 apresenta o padrão de implementação sugerido para adicionar a estratégia de

resolução de conflitos *NO_ONE* no código fonte em LingPON.

Código 10: Exemplo de definição de estratégia de escalonamento

```
1 strategy
2     no_one
3 end_strategy
```

O quarto bloco de código consiste na criação do conhecimento lógico-causal da aplicação através da definição das *Rules*. O Código 11 apresenta o padrão de implementação para a criação de uma *Rule*.

Código 11: Exemplo de criação de *Rules*

```
1  rule rlKickBallRobot1
2      condition
3          subcondition A1
4              premise prIsRobotStop robot1.atSpeed == 0.0 and
5              premise prIsPositioned robot1.atSetKick == true
6          end_subcondition
7      end_condition
8      action
9          instigation inKickBall robot1.mtKickBall();
10     end_action
11 end_rule
```

A definição de uma *Rule* é anunciada pela palavra reservada **rule** seguida de um identificador para a mesma. É importante ressaltar que o identificador da *Rule* nesse ponto é obrigatório, não podendo ser omitido.

Basicamente, cada *Rule* é composta por três blocos, que são as suas *Properties*, a *Condition* (expressão lógica) e *Action* (execução), sendo que o bloco *Properties* é opcional. O Código 12 apresenta um exemplo de código que pode ser utilizado no bloco *Properties* de uma *Rule*. O bloco *Properties* de uma *Rule* pode ser composto por duas propriedades: *Priority* e *Keeper*.

Código 12: Propriedades das *Rules*

```
1 properties
2     priority 1
3     keeper true
4 end_properties
```

A propriedade *priority* define uma ordem de prioridade e consequente execução de *Rules* quando duas ou mais *Rules* compartilham o mesmo *Exclusive Attribute* (cf. Subseção 2.2.3) em alguma de suas *Premises*. Portanto, a *Rule* que apresentar a maior prioridade terá sua execução priorizada frente as demais *Rules*.

Na atual versão da LingPON, é obrigatória a utilização de *SubConditions*, seguidas de um identificador, mesmo quando se trata de expressões com apenas uma *Premisse*. Como exemplo de tal construção, tem-se a linha 3 do Código 11. Seguindo o estado da arte do PON, a construção de uma *SubCondition* necessita de, ao menos, uma *Premise*. No caso da utilização de mais de uma *Premise*, estas devem estar conectadas por conjunções (**and**). Para utilização de disjunções (**or**), é necessário a criação de duas ou mais *SubConditions* na *Rule*.

Para a definição de *Premises* no LingPON, a palavra reservada **premise** deve ser utilizada seguida de um identificador (opcional) e uma avaliação lógica. Ainda, à uma *Premise* pode ser aplicada uma propriedade inerente dos padrões de execução do PON, conhecida como *Attribute* Impertinente (cf. Subseção 2.2.6). Para tal, basta adicionar a palavra reservada **imp** após o identificador da *Premise*, conforme apresentado no Código 13.

Código 13: Definição de uma *Premise* com *Attribute* impertinente

```
1     premise prIsConnected imp robot1.atIsConnected == true
```

A comparação em uma *Premise* é composta por três elementos: o valor de um *Attribute* vinculado a uma instância de um *FBE*, o operador de comparação (e.g ==) e o valor a ser comparado (e.g *true*). Este último pode ser tanto uma constante quanto o valor de um outro *Attribute*. Os operadores de comparação suportados pela atual versão

da LingPON são: “==”,“<”,“>”,“<=”,“>=”,“!=”.

O último bloco de código definido em uma *Rule* representa sua execução e é anunciada a partir da palavra reservada **action**. Este bloco consiste no vínculo de instigações a *Methods* definidos pelas *FBEs*. Conforme apresentado na linha 9 do Código 11, a estrutura das *Instigations* é composta pela palavra reservada **instigation** seguida de um identificador (opcional) e um *Method* de uma instância particular de um *FBE*. Dessa forma, cada *Instigation* está relacionada a execução de um único *Method* [Ferreira 2016].

Outrossim, o conjunto de padrões de nomenclaturas prevê, mas não obriga, os seguintes prefixos para as entidades PON apresentadas nessa etapa: *rl* para *Rules*, *cd* para *Conditions*, *sc* para *SubConditions* *pr* para *Premises*, *ac* para *Actions* e *in* para *Instigations*.

Finalmente, o último bloco de código consiste na criação do bloco de código principal (*main*). Este bloco de código permite ao desenvolvedor adicionar código específico para a linguagem alvo definida, conforme é apresentado no Código 14. É importante ressaltar que o código inserido dentro do bloco *main* não é avaliado pelas regras de compilação da LingPON 1.0 [Ferreira et al. 2013].

Código 14: Exemplo de definição do bloco *main*

```
1 main {
2     // Código específico em C++
3     robot1->setatConnected(true);
4     robot2->setatConnected(true);
5 }
```

Aliado à LingPON existe um compilador, o qual é objeto da próxima subseção. Em tempo, não raro a LingPON em si e seu compilador são chamados apenas de LingPON ou de tecnologia LingPON. Assim, na prática, o contexto que determinaria se está se referindo apenas a linguagem de programação em si ou se a ela e ao seu compilador.

2.3.2.2 COMPILADOR PARA O PON

Um compilador pode ser compreendido como um programa tradutor, o qual transforma uma linguagem fonte escrita em uma linguagem objeto [Aho et al. 1995]. Para isso, um compilador é constituído de fases que operam em sequência, nomeadamente análise léxica, análise sintática, análise semântica, otimização de código e geração de código.

O diagrama de atividades da Figura 17 apresenta as etapas existentes no compilador PON desenvolvida na LingPON 1.0. Cada etapa possui um objetivo específico e o resultado da fase precedente é utilizado pela fase posterior.

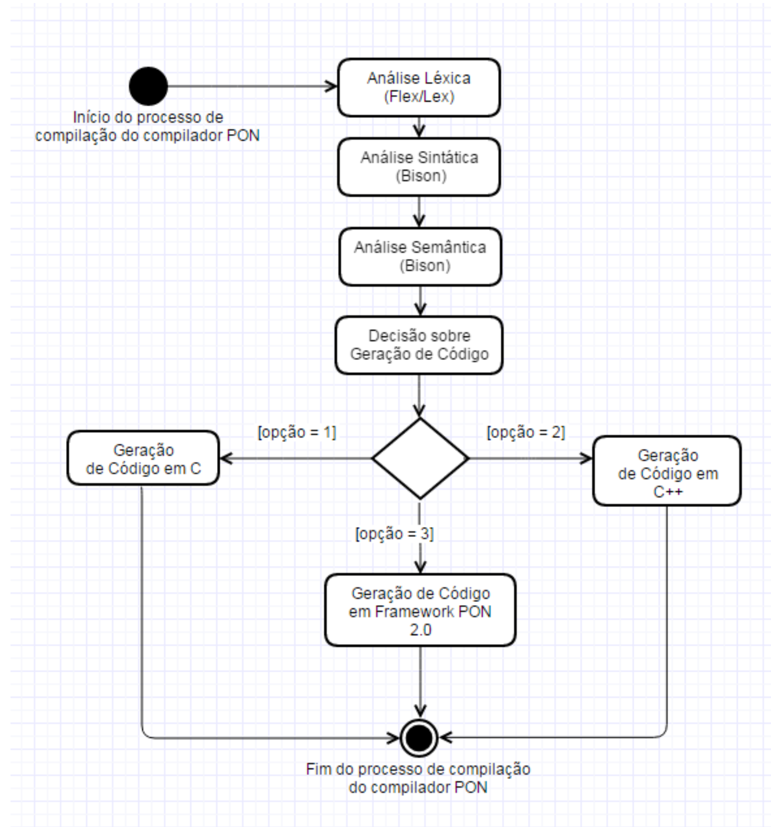

Figura 17: Diagrama de atividades UML do compilador PON.

Por se tratar da primeira fase de um compilador, o principal objetivo de um analisador léxico é ler os caracteres do código fonte, agrupá-los em *tokens* e produzir como saída uma sequência de *tokens*, sendo um para cada palavra ou símbolo encontrado no código fonte.

No caso do compilador PON, o analisar léxico foi gerado utilizando a ferramenta Flex/Lex [Grune et al. 2012]. A entrada do programa Flex/Lex é um arquivo contendo expressões regulares que definem os possíveis *tokens* de uma linguagem. Deste modo, a ferramenta Flex/Lex produz um módulo de *software* que pode ser compilado e acoplado ao módulo de análise sintática [Ferreira 2016].

A próxima fase no processo de compilação é a análise sintática. Um analisador sintático é compreendido como um programa computacional que determina se um código fonte está sintaticamente correto. Esta análise verifica se os *tokens* identificados pela análise léxica estão encadeados corretamente de acordo com a especificação gramatical da linguagem. Por fim, o analisador sintático agrupa os *tokens* em frases gramaticais que

serão usadas para sintetizar a saída.

Por sua vez, a fase de análise semântica verifica as frases gramaticas formadas pelo analisador sintático a fim de detectar possíveis erros semânticos. Diferentemente da análise sintática, a qual verifica se a estrutura da frase gramatical está correta, a análise semântica irá verificar sua significância. Como exemplo, o analisador semântico valida se um valor do tipo inteiro está sendo atribuído à uma variável do tipo inteiro. Em contrapartida, a análise sintática valida sua estrutura gramatical, ou seja, se os símbolos estão encadeados na ordem correta, de acordo com a especificação sintática da linguagem [Ferreira 2016]. A Figura 18 ilustra um fragmento de código no qual é possível perceber a diferença entre análise semântica e a análise sintática.

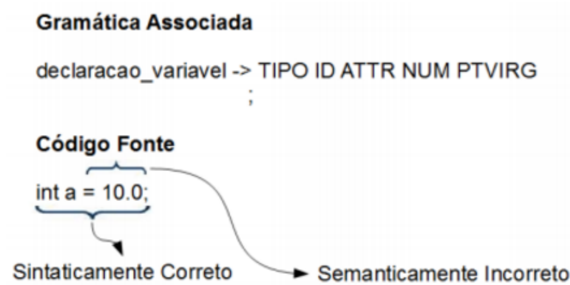

Figura 18: Exemplo de análise semântica [Ferreira 2016]

Os módulo de análise sintática e análise semântica foram gerados utilizando a ferramenta Bison [Grune et al. 2012]. O Bison é uma ferramenta que, baseado em uma gramática, constrói um programa em C/C++ que analisa uma sequência de *tokens* provenientes de um arquivo de código-fonte segundo as regras gramaticais.

A etapa de otimização de código é uma etapa usualmente presente no processo de compilação. Esta etapa tenta melhorar o código intermediário, de tal forma que venha resultar um código de máquina mais rápido em tempo de execução [Aho et al. 1995]. Entretanto, o compilador PON ainda não possui a etapa de otimização durante o seu processo de compilação.

Finalmente, a etapa de geração de código utiliza as informações interpretadas e validadas pelas etapas precedentes com o intuito de gerar código alvo. Por não existir uma ferramenta genérica para geração de código, um gerador de código específico para o PON foi desenvolvido em [Ferreira 2016]. O módulo de geração de código desenvolvido é capaz de gerar código C, C++ e *Framework* PON C++ 2.0 a partir do código-fonte LingPON.

De forma a abstrair as entidades PON durante o processo de compilação, algumas classes C++ foram criadas para armazenar as informações pertinentes a cada entidade

PON, conforme apresentado na Figura 19

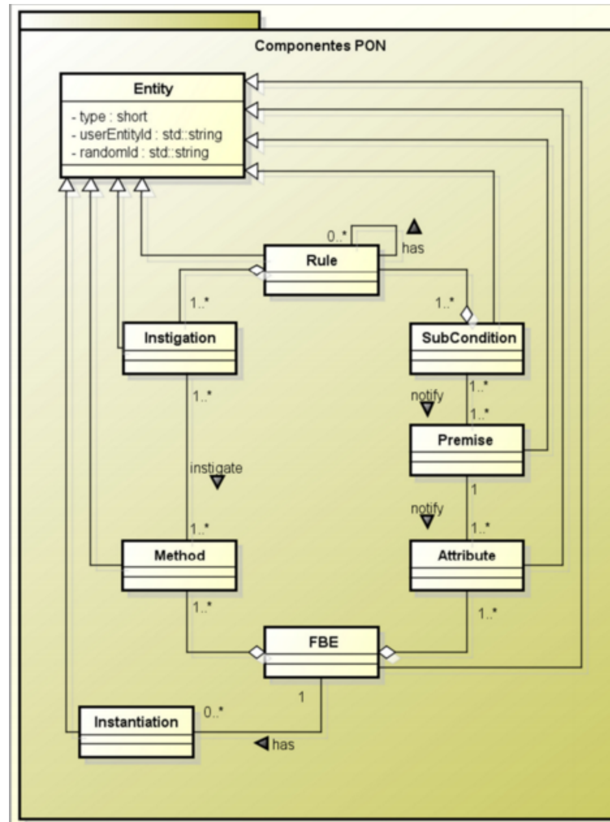

Figura 19: Diagrama de classes utilizada pelo compilador PON para representar entidades PON [Ferreira 2016].

Ao analisar um fragmento do código fonte, o módulo de análise sintática interage com a instância da classe *Compiler*. A classe *Compiler* é uma classe utilitária que foi criada com o objetivo de manipular, por meio de sua instância, a criação dos objetos que representam as entidades PON durante o processo de compilação.

Após a criação de um novo objeto, o *Compiler* o adiciona em uma estrutura de dados chamada tabela de símbolos, a qual é posteriormente utilizada para geração do código alvo. Atualmente, o compilador PON suporta a geração de código em C, C++ e *Framework*. Para maiores detalhes sobre a arquitetura da LingPON e seu respectivo compilador, sugere-se a leitura de [Ferreira 2016].

Em tempo, novas funcionalidades foram adicionadas à linguagem de programação PON durante a disciplina de "Linguagens e Compiladores"[4] ofertada pela UTFPR em 2015 e ministrada por Prof. Dr. João Alberto Fabro e Prof. Dr. Jean Marcelo Simão. As

[4]Disciplina: Tópicos Avançados Em Engenharia De Software. Tema: LINGUAGENS E COMPILADORES. Código CAES101. Programa PPGCA/UTFPR - Prof. João Alberto Fabro e Prof. Jean Marcelo Simão. 3 Trimestre de 2016.

funcionalidades adicionadas a LingPON foram suporte a regras de formação, compilação para *Notification Oriented Computer Architecture* [Linhares et al. 2015], compilação para *VHDL* e uma nova versão do gerador de código C++, no qual é possível gerar código estático (*static* classes).

2.3.3 OUTRAS MATERIALIZAÇÕES EM SOFTWARE DO PON

As materializações chamadas *Framework* PON 2.0 e LingPON 1.0 foram aqui detalhadas porque são objeto de estudo comparativo no âmbito dessa dissertação. Elas são objeto de estudo por serem as materializações mais estáveis no tocante a arquétipo/framework e linguagem-compilador para o PON, bem como por terem sido já comparadas para com o POO/PI ainda que em aplicações de envergadura assaz tímidas em geral. Isto considerado, apenas a título de elucidação, é pertinente salientar que há outras materializações prototipais.

O *Framework* prototipal PON em C++ foi adaptado por Weber para trabalhar com multi-threads de maneira bem prototipal também [Weber et al. 2010]. Subsequentemente, Belmonte adaptou o *Framework* PON 2.0 em C++ para trabalhar com Threads, aplicando-o para Multi-core a fim de demonstrar a capacidade multi-processada do PON já obtendo resultados positivos [Weber et al. 2010, Belmonte et al. 2016]. Ainda, Viana Melo adaptou o Framework PON 2.0 em C++ para trabalhar com aplicações Fuzzy o que se demonstrou funcional e útil [Melo et al. 2013]. Por fim, Schutz adaptou o *Framework* PON 2.0 em C++ para trabalhar com Redes Neurais o que se demonstrou pertinente sendo uma pesquisa em andamento [Schütz et al. 2015]. Ainda que interessante, esses *Frameworks* prototipais fogem do escopo deste trabalho que não trata de multi-thread/processamento-paralelo, *fuzzy* e redes neurais. Em todo caso, as aplicações ali desenvolvidas não tem a mesma envergadura em termos de números *Rules* e *FBES* que a considerada neste presente trabalho.

Além das adaptações dos *Frameworks* em C++, houve o desenvolvimento em disciplina *stricto senso* sobre o PON, em 2015 [5], de *Framework* em linguagem Java e C#. O *Framework* em Java foi uma adaptação do primeiro *Framework* em C++ sendo que, não obstante, em resultados preliminares o resultados dele se assemelharam ao do *Framework* C++ 2.0. O primeiro *Framework* C# segue o mesmo quadro e resultados do *Framework*

[5]Disciplina: - Tópicos Avançados Em Sistemas Embarcados. Tema: Paradigma Orientado a Notificaçõe. Código CASE102 . Programa PPGCA/UTFPR - Prof. Prof. Jean Marcelo Simão. 1 Trimestre de 2015. & Disciplina: - Tópicos Especiais Em EC: Paradigma Orientado a Notificações. Tema: LINGUAGENS E COMPILADORES. Código PGEID/PGEIM. Programa. CPGEI/UTFPR - Prof. Prof. Jean Marcelo Simão. 1 Trimestre de 2015.

Java. Entretanto, além dos resultados serem preliminares não houve comparações para com POO/PI. Em Mendonça, há o relato de uma aplicação PON feita neste *Framework* Java mas sem comparações com POO/PI [Mendonça 2016]. Ainda, na edição de 2016[6], foi desenvolvido uma evolução do *Framework* C#, o qual foi usado em aplicação híbrida com POO para simulação de ambiente IOT [7].

Tal qual o *Framework*, a LingPON também já tem derivações. Entretanto, estas derivações encontram-se em estágio prototipal, sem terem sido objetos de experimentações maiores e de bancas avaliadoras no âmbito de mestrado ou afins. Neste âmbito, na edição 2015 da disciplina de "Linguagens e Compiladores" dos Profs. J. A. Fabro e J. M. Simão, a LingPON 1.0 teve funcionalidades adicionadas. Tais funcionalidade são suporte a regras de formação, compilação para *Notification Oriented Computer Architecture* (NOCA ou ArqPON) e compilação para VHDL.

Ainda, na edição 2015 da disciplina de "Linguagens e Compiladores" dos Profs. J. A. Fabro e J. M. Simão surgiu uma nova versão do gerador de código C++, no qual é possível gerar código estático (*static classes*). Ainda que o código gerado pareça ser, nos resultados preliminares, mais rápido que o código gerado C++ e C pela tecnologia LingPON 1.0, ele ainda é prototipal e apresenta dificuldades maiores para conectar com código não estático, o que é problemático como no caso de aplicações de robôs. Por fim, na edição 2016 da disciplina de "Linguagens e Compiladores" dos Profs. J. A. Fabro e J. M. Simão, surge uma nova versão baseada em código orientados a *namespace* em C++ que seria mais rápida que as precedentes e sem problemas maiores de integração, entretanto isso é muito recente e largamente prototipal.

2.4 FUTEBOL DE ROBÔS - ROBOCUP

A *RoboCup* é uma iniciativa internacional e interdisciplinar que visa promover a pesquisa e desenvolvimento no campo da robótica, provendo tarefas comuns para a avaliação de diferentes teorias, Códigos e arquiteturas de robôs. Como tarefa comum, a *RoboCup* escolheu o futebol. Isto porque, para que um robô possa participar de uma partida de futebol, muitas tecnologias precisam ser integradas e uma série de avanços técnicos precisam ser alcançados, tais como mecânica de precisão, controle eletrônico e programação distribuída [Yoon 2015, Marling et al. 2003, Asada et al. 1998, Asada et al. 1999].

[6]Disciplina: - Tópicos Avançados Em Sistemas Embarcados. Tema: Paradigma Orientado a Notificações. Código CASE102 . Programa PPGCA/UTFPR - Prof. Prof. Jean Marcelo Simão. 1 Trimestre de 2016.

[7]Vide http://www.dainf.ct.utfpr.edu.br/ jeansimao/PON/PON.htm

Atualmente, a *RoboCup* está dividida em cinco categorias, sendo que cada uma possui seus próprios desafios a serem superados: *Small Size League* (SSL), *Middle Size League* (MSL), *Simulation League*, *Standard Platform League* e *Humanoid League*. Esse trabalho focará apenas na categoria SSL, a qual tem como principal objeto de estudo o desenvolvimento de sistemas inteligentes capazes de controlar robôs em ambientes altamente dinâmicos utilizando sistemas de controle híbridos (centralizado/distribuído) [Yoon 2015].

Na categoria *RoboCup SSL*, cada time é composto por no máximo seis robôs que disputam a partida de futebol utilizando uma bola de golfe alaranjada em um campo, cujas dimensões são apresentadas na Figura 20. O robô apresenta um formato cilíndrico, com 180 *mm* de diâmetro e 150 *mm* de altura e possui um marcador central, que pode ser amarelo ou azul, e marcadores diversos que identificam cada jogador de cada time.

As atividades durante a partida são capturadas por duas câmeras posicionadas acima do campo de jogo. As imagens então são processadas pelo SSL-Vision, um *software* executado em computador externo, de forma a identificar informações importantes sobre a partida, tais como posição de cada um dos robôs e da bola.

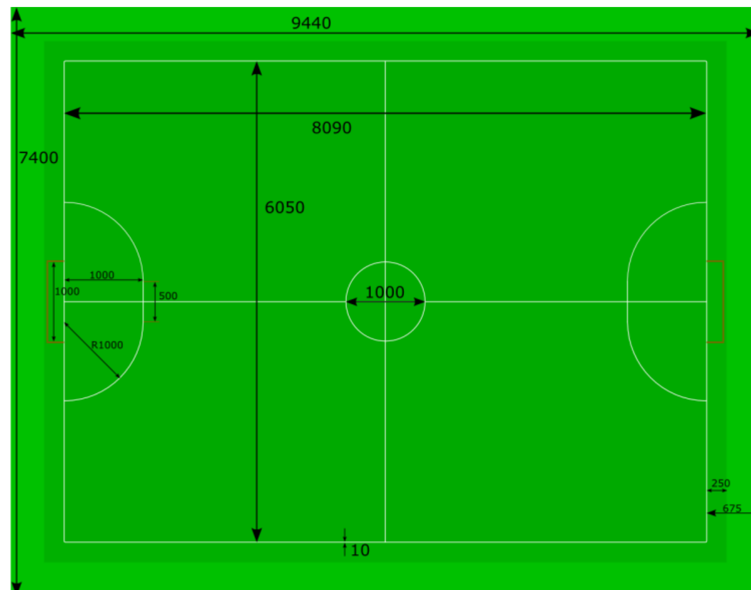

Figura 20: Dimensões do campo oficial da categoria SSL, em milímetros.

Utilizando as informações já processadas pelo SSL-Vision, um computador independente deve produzir as estratégias para as ações dos robôs e enviar comandos para cada um dos robôs através de uma conexão sem fio (*wireless*). O robô então processa o comando recebido e atua no ambiente, alterando o seu estado no campo de jogo. Tais mudanças serão capturadas pelas câmeras e todo o processo de controle se repete ao longo de toda a partida. A Figura 21 apresenta configuração de um ambiente *RoboCup* SSL.

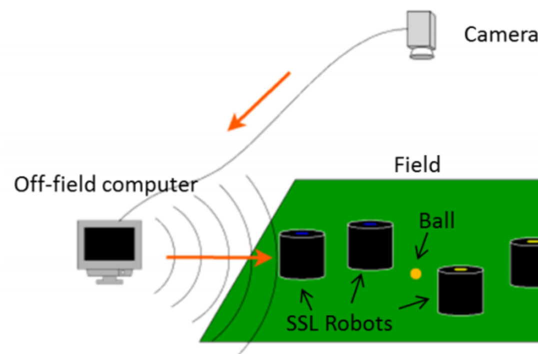

Figura 21: Sistema *RoboCup* SSL [Yoon 2015]

Como apresentado na Figura 21, o sistema de controle dos robôs na SSL é centralizado. Os robôs não possuem nenhuma estratégia ou capacidade para deliberar sobre qual ação deve ser executada. Eles simplesmente processam comandos recebidos a partir de uma unidade de controle onisciente e centralizada [Yoon 2015]

2.4.1 AMBIENTE SIMULADO *ROBOCUP* SSL

Visando permitir o desenvolvimento do software sem depender de robôs reais, um simulador do ambiente de competição da *RoboCup* pode ser utilizado. Esse ambiente é disponibilizado por pesquisadores que contribuem para a competição, sendo extensivamente utilizado por toda a comunidade, livremente, para suas pesquisas e desenvolvimento. Este ambiente é composto por duas aplicações: *grSim Simulator*[8] e *Referee Box*[9].

O *grSim Simulator* é um simulador funcional do ambiente de jogo *Robocup*, cuja interface é apresentada na Figura 22. Os robôs simulados possuem características muito próximas às reais, tais como dimensões, velocidade máxima, inércia e aceleração. Além disso, essa aplicação é responsável pelo envio de informações relativas aos objetos em campo seguindo o protocolo padrão do sistema de visão, SSL-Vision [Monajjemi et al. 2011].

[8]Disponível em http://www.parsianrobotic.ir/grsim/
[9]Disponível em http://robocupssl.cpe.ku.ac.th/referee:start

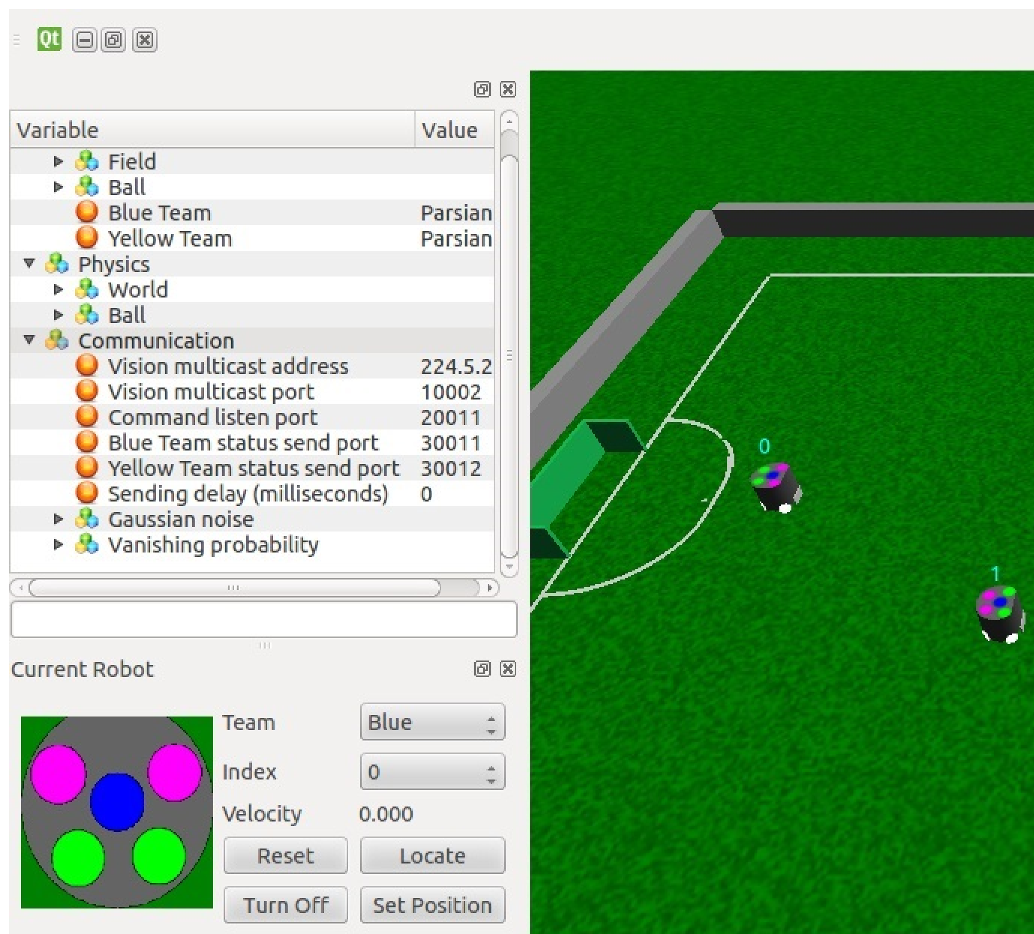

Figura 22: Interface gráfica da aplicação *grSim Simulator*.

A aplicação que irá controlar os robôs lê os dados referentes a cada um deles, os processa e envia comandos específicos à cada um dos robôs ao *grSim Simulator*, que irá executá-los, atualizar as ações usando a física simulada e obter as novas percepções do ambiente. Portanto, o *grSim Simulator* não possui lógica alguma de controle, apenas executa comandos recebidos e disponibiliza, através de uma interface de rede, informações sobre o estado atual do ambiente de jogo à quem interessar. Além disso, através de sua interface gráfica, é possível acompanhar em tempo real informações específicas de cada um dos robôs, tais como sua posição e velocidade atual.

A aplicação *Referee Box*, por sua vez, é um programa simples que permite a um operador neutro enviar ordens proferidas pelo árbitro diretamente ao software das equipes competidoras. Além de enviar os comandos referentes a ordens do árbitro, a *Referee Box* também atua como um utilitário auxiliar para o árbitro, na qual é possível acompanhar o tempo de jogo, gols marcados e os cartões amarelos e vermelhos.

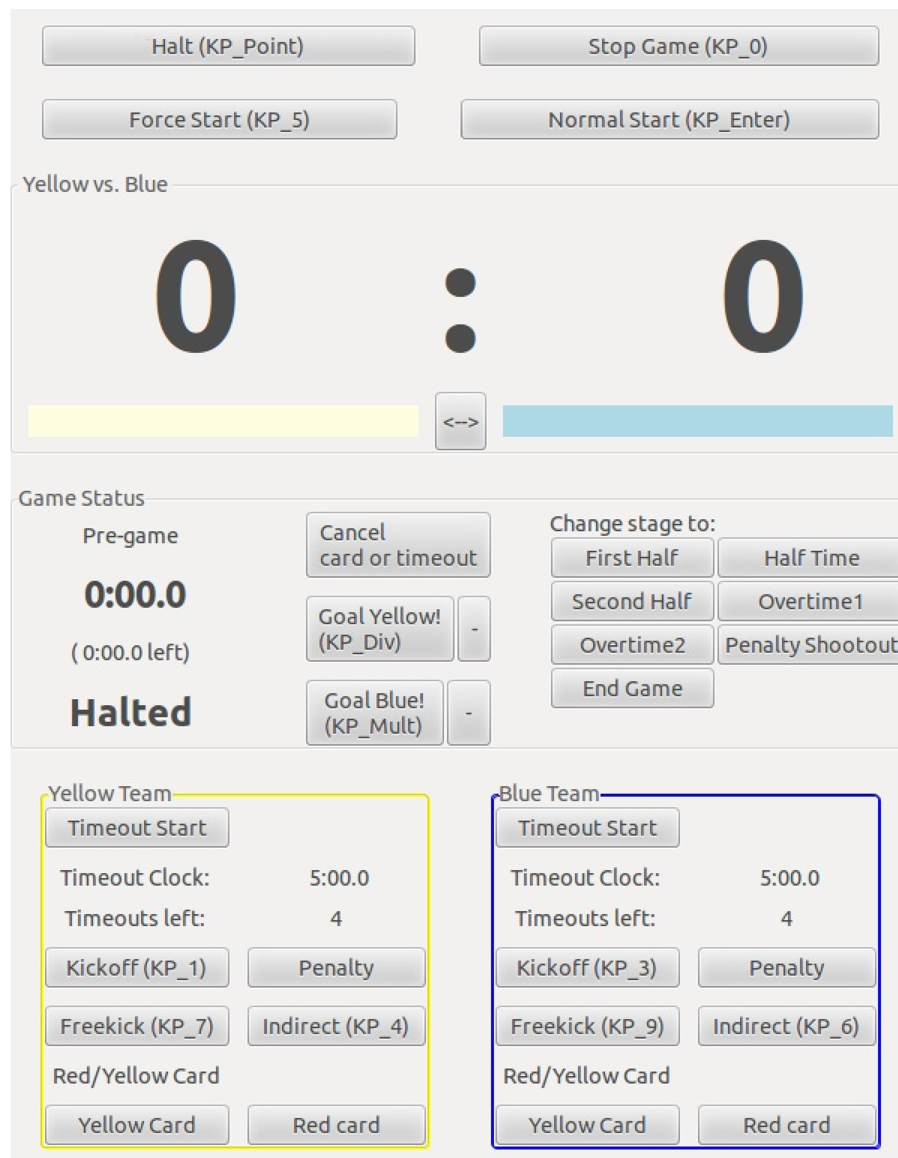

Figura 23: Interface gráfica da aplicação *Referee Box*.

A interface gráfica da aplicação *Referee Box* (Figura 23) apresenta quatro grandes botões em sua parte superior. O botão *Halt* determina que todos os robôs interrompam seus movimentos completamente de forma imediata. Esse comando normalmente é executado em casos de emergência, nos quais o árbitro ou as equipes precisem de um longo tempo para resolver a situação. O botão *Stop* interrompe a partida e ordena que todos os robôs se posicionem a pelo menos 50 cm da bola. Este botão é utilizado ao início da partida ou quando algum time sofre um gol, de forma a reiniciar a partida. Na sequência, o botão *Force Start* indica um reinício de partida no qual ambas as equipes estão autorizadas a aproximar-se e tocar na bola imediatamente. Finalmente, o botão *Normal Start* determina o início de partida permitindo apenas um dos times a tocar na bola.

Nos retângulos de controle referente a cada uma das equipes estão presentes os quatro botões de reinício favoráveis à respectiva equipe. O botão *Kickoff* ordena as equipes a se prepararem para um pontapé de saída. O botão *Penalty* ordena que os times se preparem para a cobrança de uma penalidade máxima. Ambos são comandos de preparação, ou seja, serão realmente executados somente após o comando *Normal Start*. Na sequência, o botão *Freekick* ordena o time a executar imediatamente um tiro livre direto. Esse comando é usado para cobranças de faltas. Finalmente, o botão *Indirect* ordena o time a imediatamente executar um tiro livre indireto, em tiros de meta e escanteios.

Uma partida de futebol de robôs inicia-se com o comando *Stop*. Nesse momento, os robôs começam a se mover pelo campo de jogo. O árbitro então escolhe uma equipe que dará início a partida. Finalmente, quando todos os robôs já estão em suas posições, o árbitro determina o início de partida, através do comando *Normal Start*. Durante a partida, uma série de interrupções e reinícios irão se suceder.

Nota-se ainda que, em uma partida de futebol de robôs, cada membro do time deve desempenhar uma função específica em campo, com responsabilidades muito bem definidas. Um goleiro, por exemplo, tem como responsabilidade evitar que os chutes do time adversário entrem em sua baliza. Já um atacante, não deve se preocupar em defender, mas sim em fazer gols na baliza adversária. Dessa forma, o sistema de controle deve também ser capaz de diferenciar as funções que cada robô pode assumir durante uma partida.

A Figura 24 apresenta uma visão geral sobre a responsabilidade de cada uma das aplicações que compõem o ambiente de simulação. A aplicação de controle dos robôs deve ser capaz de responder de maneira adequada aos diferentes comandos e informações recebidos de ambas as aplicações que compõem o ambiente de simulação, deliberando sobre qual ação deve ser executada por cada um dos robôs que compõem a equipe. A decisão sobre qual ação a ser executada é baseada em regras que permitem avaliar as atuais condições da partida para então decidir quais comandos serão enviados para cada um dos robôs.

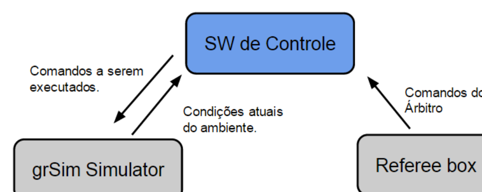

Figura 24: Diagrama representativo das aplicações que compõem o ambiente simulado Robocup SSL.

2.5 REFLEXÃO SOBRE A REVISÃO DO ESTADO DA ARTE

Este presente capítulo apresentou uma breve revisão acerca dos principais problemas encontrados nos paradigmas usuais de programação, nomeadamente o PI e PD. Ainda, uma revisão mais aprofundada sobre o PON foi apresentada, descrevendo suas principais características.

O PON é um paradigma emergente o qual se apresenta como uma alternativa aos paradigmas de programação vigentes (e.g. paradigma imperativo e declarativo). Visando solucionar algumas das principais deficiências dos atuais paradigmas de programação, tais como redundâncias estruturais e temporais, bem como os acoplamentos daí decorrentes, o PON apresenta uma nova forma de estruturar e executar artefatos de *software*.

Por meio da programação inspirada na forma declarativa (e.g fatos e regras), o PON permite criar aplicações de forma mais natural, com maior proximidade à cognição humana, em alto nível. Além disso, a execução de aplicações PON se dá através de pequenas entidades computacionais reativas, as quais colaboram por meio de notificações e formam o mecanismo de notificações do PON. Por se tratar de notificações precisas e pontuais, o PON permite execução otimizada e minimamente acoplada, características estas úteis tanto para 'mono-processamento' bem como para processamento distribuído [Simão e Stadzisz 2008].

Como apresentado na Subseção 2.2.8.1, o PON foi primeiramente materializado na forma de um *Framework*, o qual hoje se encontra em sua terceira versão, dado que houve a versão prototipal, a versão 1.0 e a versão 2.0. Tal materialização possibilitou a criação de aplicações PON e consequente validação dos conceitos relacionados a esse paradigma. Entretanto, as aplicações desenvolvidas utilizando o *Framework* não apresentaram desempenho satisfatório quando confrontados com a natureza do PON e comparados com o cálculo assintótico da sua estratégia de inferência apresentado na Subseção 2.2.5 [Ferreira et al. 2013].

Portanto, vislumbrou-se a necessidade da criação de uma linguagem de programação e respectivo compilador para o PON, nomeado de LingPON ou tecnologia LingPON. Dessa forma, a primeira versão da LingPON foi criada em [Ferreira 2016], a partir de um versão prototipal feita em uma disciplina de Linguagens e Compiladores [10], com o objetivo de abstrair as características do PON em uma gramática apropriada.

[10] Disciplina "Linguagens e Compiladores" ofertada pela UTFPR em 2014, ministrada por Prof. Dr. João Alberto Fabro e Prof. Dr. Jean Marcelo Simão e frequentada pelos alunos Adriano Francisco Ronszcka, Cleverson Avelino Ferreira, Priscila Ap. de Moraes Ioris e Clayton Kossoski.

A criação de uma linguagem nativa representou uma grande evolução para o estado da técnica do PON. No tocante ao desempenho, as aplicações desenvolvidas utilizando a LingPON apresentaram melhores resultados, relacionado a tempo de processamento, quando comparadas às suas versões desenvolvidas utilizando o *Framework* [Ferreira 2016]. No tocante a facilidade de programação, a LingPON se destaca pela sua simplicidade e clareza, tornando o desenvolvimento de aplicações PON mais simples nesse sentido, permitido estruturar o conhecimento de uma aplicação em mais alto nível de forma mais natural ao ser humano [Ferreira 2016].

Entretanto, mesmo com o surgimento de uma linguagem de programação nativa, apenas algumas poucas aplicações PON com certo nível de complexidade foram desenvolvidas até o presente momento utilizando ambas as materializações.

Em se tratando de *Framework*, a maioria das aplicações desenvolvidas apresentavam baixo nível de complexidade, conforme apresentado no Apêndice E. O principal motivo para isso é que muitas dessas aplicações tinham como principal objetivo demonstrar e validar as características do PON, sendo para tal suficiente o desenvolvimento de pequenas e médias aplicações aplicadas a cenários específicos de teste. Ademais, as aplicações um tanto mais complexas (como Controle de Sistema Manufatura em Framework prototipal C++ [Simão 2001]), não foram efetivamente comparadas com as respectivas aplicações em PI. Ainda, nenhuma aplicação foi comparada com o equvialente em PI totalmente feito por outrem, este diferente do desenvolvedor PON. Mesmo no caso do CTA as aplicações já existentes em PI eram corrigidas pelo desenvolvedor PON pertinente.

A LingPON, por sua vez, apresenta algumas limitações que dificultam o desenvolvimento de aplicações PON com escopo mais abrangente. Entre as principais limitações, pode-se destacar a redundância na declaração de *Rules* e impossibilidade de relacionar outros *FBEs* como *Attributes*. Em suma, falta a capacidade de *FBEs* agregarem outros *FBEs* e de *FBEs* agregarem outras *Rules*.

De forma a solucionar tais limitações, o próximo capítulo apresenta dois novos conceitos, no âmbito de relacionamento de agregação entre entidades PON, que podem facilitar o desenvolvimento de aplicações PON utilizando a LingPON.

3 DESENVOLVIMENTO

Esse capítulo apresenta as contribuições desse trabalho, as quais são relativas ao PON. Conforme apresentado na Seção 1.4, esse trabalho apresenta como parte do objetivo principal a evolução da LingPON e seu respectivo compilador de forma a facilitar a criação de aplicações PON. A partir disso, outra parte do objetivo principal é propor uma aplicação de complexidade reconhecida (futebol de robôs) comparando-a com a mesma aplicação desenvolvida sob o viés do PI e com outras materializações do PON.

Nesse sentido, as contribuições para a LingPON e seu respectivo compilador são apresentadas na Seção 3.1. Na sequência, a Seção 3.2 apresenta em detalhes o desenvolvimento da aplicação de controle para uma partida de futebol de robôs utilizando materializações do PON (*Framework 2.0*, atual versão da LingPON e a nova versão da LingPON aqui proposta) e a comparação das mesmas entre si e com uma solução equivalente desenvolvida sob o viés do PI.

3.1 CONTRIBUIÇÕES PARA A LINGPON

Esta seção apresenta as contribuições deste trabalho para com a LingPON e o seu respectivo compilador. Primeiramente, a subseção 3.1.1 apresenta em detalhes as alterações realizadas na LingPON para permitir o relacionamento de FBEs através de agregação. Na sequência, na seção 3.1.2 são apresentadas as alterações realizadas na LingPON e seu respectivo compilador para suportar a agregação de *Rules* em *FBEs* de forma a permitir a criação de *Rules* de forma mais direta e com menos redundância de linhas de código. Por fim, a seção 3.1.3 apresenta a correção (*bug fix*) aplicada ao processo de compilação da LingPON de forma a solucionar um erro encontrado na geração de código-alvo C++ com múltiplas instâncias de *FBEs* na atual versão do compilador.

As contribuições para a LingPON são aqui apresentadas de forma literal, com o objetivo de facilitar o entendimento das mesmas. Descrições mais detalhadas do ponto de vista de alterações realizadas no código-fonte do compilador PON, no tocante ao

analisador léxico, analisador sintático, analisador semântico e gerador de código, podem ser encontradas no Apêndice A deste trabalho.

3.1.1 AGREGAÇÃO ENTRE *FBES*

Conforme apresentado na Seção 1.3.2, a atual versão da LingPON e seu respectivo compilador suportam apenas *Attributes* de tipos primitivos, isto é, *boolean*, *integer*, *float*, *char* ou *string* [Ferreira 2016]. Entretanto, de forma a aumentar o nível de encapsulamento de *Attributes* e *Methods* no desenvolvimento de aplicações PON e, consequentemente, facilitar o desenvolvimento de aplicações PON, este trabalho propõe a possibilidade de declarar *Attributes* que sejam definidos a partir de outros *FBEs*.

Para exemplificar a vantagem de definir *Attributes* a partir de outro *FBE*, pode-se imaginar uma aplicação PON na qual seja necessário criar um *FBE* que represente o comportamento de um time de futebol. O time é composto por três jogadores, cada qual possuindo um nome, um número e uma posição de jogo. O Código 15 apresenta o código fonte criado para representar esse exemplo utilizando a versão original da LingPON.

Código 15: Criação do *FBE* Team na versão original do LingPON.

```
1  fbe Team
2      attributes
3          string atNamePlayer1 “ ”
4          integer atNumberPlayer1 0
5          string atPositionPlayer1 “ ”
6          string atNamePlayer2 “ ”
7          integer atNumberPlayer2 0
8          string atPositionPlayer2 “ ”
9          string atNamePlayer3 “ ”
10         integer atNumberPlayer3 0
11         string atPositionPlayer3 “ ”
12     end_attributes
13     methods
14         method mtExecuteActionPlayer1()
15         method mtExecuteActionPlayer2()
16         method mtExecuteActionPlayer3()
17     end_methods
18 end_fbe
```

Analisando o código apresentado em Código 15, construído utilizando a versão original da LingPON, nota-se a ocorrência de redundância na declaração dos *Attributes* (linhas 3 à 11) e *Methods* (linhas 14 à 16). Portanto, de forma a reduzir a necessidade de redundância de código e, consequentemente, facilitar o desenvolvimento de aplicações PON, a LingPON e seu respectivo compilador foram alterados de forma a suportar a declaração de *Attributes* que sejam definidos a partir de outros *FBEs*.

Para suportar a declaração de *Attributes* não primitivos, algumas mudanças no analisador sintático utilizado pelo compilador foram propostas. Em um primeiro momento, o arquivo de configuração do analisador sintático (Bison) foi alterado para que o mesmo reconhecesse a declaração desse tipo de *Attribute*. Entretanto, isso ainda não foi suficiente, uma vez que a tabela de símbolos utilizada pelo compilador na geração de código alvo deveria gerenciar de maneira correta a relação entre *FBEs* e seus *Attributes* não primitivos. Nesse sentido, o gerenciador da tabela de símbolos do compilador foi alterado. Com as alterações feita no analisador sintático e na tabela de símbolos, viabilizou-se a agregação

entre FBEs. Neste sentido, dado que um *FBE Team* possui três *Attributes* do tipo *Player*, para cada nova instância do *FBE Team*, três novas instâncias de *Player* são criadas, adicionadas à tabela de símbolos e relacionadas com o *FBE* que a definiu (*Team*).

Utilizando a nova versão do *LingPON*, foi possível reescrever o código apresentado no Código 15 de forma muito mais legível, conforme apresentado no Código 16. Entre as linhas 1 e 10, foi definido uma *FBE Player*, o qual possui os *Attributes* e *Methods* referente ao jogador de futebol. A partir da linha 11, foi declarado o *FBE Team*, o qual possui três *Attributes* do tipo *Player*.

Código 16: Criação do *FBE* Team na nova versão do LingPON.

```
1  fbe Player
2    attributes
3      string atNamePlayer “ ”
4      integer atNumberPlayer 0
5      string atPositionPlayer “ ”
6    end_attributes
7    methods
8      method mtExecuteActionPlayer()
9    end_methods
10 end_fbe
11 fbe Team
12   attributes
13     Player atPlayer1 ;
14     Player atPlayer2 ;
15     Player atPlayer3 ;
16   end_attributes
17   methods
18     method mtTeamInfo()
19   end_methods
20 end_fbe
```

Caso o requisito da aplicação fosse alterado para o time suportar quatro jogadores,

na aplicação desenvolvida na versão original da LingPON, seria necessário adicionar três novos *Attributes* (*atNamePlayer4*, *atNumberPlayer4* e *atPositionPlayer4*) e um novo *Method* (*mtExecuteActionPlayer4*). A mesma mudança de requisito poderia ser feita, na nova versão da LingPON, simplesmente adicionando um novo *Attribute* (*atPlayer4*) do tipo *Player* no *FBE Team* e realizando as devidas alterações no método *mtTeamInfo*.

3.1.2 AGREGAÇÃO DE *RULES* EM *FBES*

Conforme apresentado na Seção 1.3.1, a atual versão da LingPON faz com que, no código fonte, as *Rules* sejam relacionadas a instâncias de *FBEs*. Com isso, para cada nova instância criada, novas *Rules* devem ser adicionadas ao sistema, criando assim redundância de código e consequente aumento de complexidade do código fonte.

Para exemplificar esse problema, pode-se imaginar uma aplicação PON na qual seja necessário controlar os movimentos de um robô em uma partida de futebol. De forma a facilitar o entendimento do exemplo, o robô deverá simplesmente correr atrás da bola quando sua posição for diferente da posição da bola.

Nesse sentido, os *FBEs Robot* e *Ball* apresentados no Código 17, poderiam ser criados para representar, respectivamente, os robôs e a bola em uma partida de futebol.

Código 17: Exemplo de *FBE Robot*.

```
1  fbe Robot
2      attributes
3              float atRobotPos 0.0
4      end_attributes
5      methods
6          method mtRunToBall ( ... )
7      end_methods
8  end_fbe
9
10 fbe Ball
11     attributes
12             float atBallPos 0.0
13     end_attributes
14     methods
15         method mtGetPosition ( ... )
16     end_methods
17 end_fbe
```

Conforme mencionado anteriormente, na atual versão da LingPON, cada uma das *Rules* que determinam o comportamento lógico-causal do sistema está diretamente relacionada a uma instância de *FBE*. Neste sentido, caso seja necessário controlar dois robôs, ou seja, duas instâncias do *FBE Robot* (*robot1* e *robot2*), será necessário criar duas *Rules*, conforme apresentado em Código 18.

Código 18: Exemplo de declaração de *Rules* para controle de robôs na versão original do LingPON.

```
1  rule rlRunToBallRobot1
2   condition
3      subcondition condRobot1NotBallPos
4          premise prRb1NotBallPos robot1.atRobotPos != ball.atBallPos
5      end_subcondition
6    end_condition
7    action
8       instigation inRb1Move robot1.mtRunToBall();
9    end_action
10 end_rule
11
12 rule rlRunToBallRobot2
13  condition
14     subcondition condRobot2NotBallPos
15         premise prRb2NotBallPos robot2.atRobotPos != ball.atBallPos
16     end_subcondition
17   end_condition
18   action
19      instigation inRb2Move robot2.mtRunToBall();
20   end_action
21 end_rule
```

Ao analisar o código apresentado no Código 18, é possível observar que as duas *Rules* são muito semelhantes em suas declarações. A única real diferença entre elas é a instância do *FBE Robot* com a qual cada uma delas se relaciona. Enquanto a primeira *Rule* está fazendo referência para a instância *robot1*, a segunda *Rule* referencia a instância *robot2*. Neste sentido, é clara a redundância de código para a construção das *Rules* na atual versão do LingPON.

Para facilitar o desenvolvimento de aplicações PON complexas, isto é, com múltiplas instâncias de *FBEs* e múltiplas *Rules*, um novo conceito foi inserido na programação de aplicações PON: *FBE Rule*. Uma *FBE Rule* é definida como uma *Rule* que, ao invés de estar relacionada a uma instância de *FBE*, está relacionada a uma classe de

FBE. Dessa forma, para cada nova instância de *FBE* criada, todo o conjunto de *FBE Rules* associado ao *FBE* será criado de forma automática, pelo próprio compilador PON. Neste quadro, cada *FBE Rule* trataria dessa instância em específico. Ademais, uma *FBE Rule* pode referenciar instâncias outras ao próprio *FBE*, por exemplo a instância *ball* do *FBE Ball*.

Para tornar possível a criação de um conjunto de *Rules* para cada nova instância de *FBE*, foi necessário alterar o processo de compilação da LingPON. Conforme apresentado na Figura 25, um pré-compilador foi inserido no processo de compilação de código-fonte PON. O pré-compilador utiliza o mesmo analisador léxico e sintático utilizado pelo compilador PON, diferenciando-se apenas pelo gerador de código.

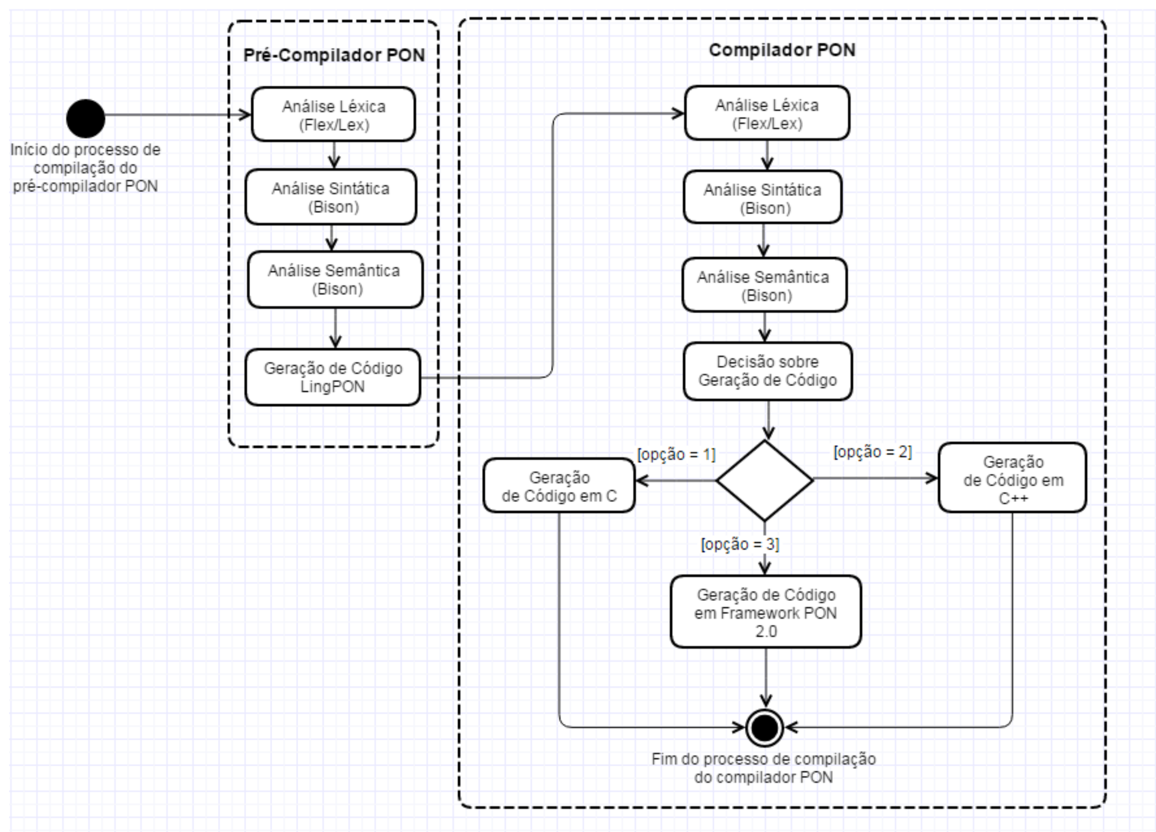

Figura 25: Fluxo de compilação de código utilizando o pré-compilador PON.

Primeiramente, o pré-compilador executa a etapa de análise léxica, na qual os caracteres presentes no código-fonte são lidos e agrupados em *tokens*. Essa etapa gera como saída uma sequência de *tokens*. Utilizando a sequência de *tokens* gerada pelo analisador léxico, o pré-compilador executa a análise semântica, de forma a identificar se os *tokens* provenientes do código-fonte estão na sequência correta.

Caso não seja encontrado nenhum erro semântico, o pré-compilador inicia a etapa

de geração de código. Nesta última etapa, o pré-compilador identifica as *FBE Rules* presentes no código-fonte e os *FBEs* a elas associados. Para cada instância de um *FBE* que possui *FBE Rule* associada, o pré-compilador irá criar uma entidade *Rule* e associá-la a uma instância do *FBE*.

A etapa de pré-compilação tem como saída um código-fonte PON pré-compilado. Esse código-fonte apresenta todas as *Rules* associadas a cada uma das instâncias de *FBE* presentes no código-fonte original.

Dessa forma, esse novo conceito diminui a necessidade de redundância de código na declaração de *Rules*, fazendo com que o número de *Rules* a serem declaradas não seja mais diretamente proporcional ao número de instâncias presentes em uma aplicação PON.

Utilizando o conceito de *FBE Rules*, presente na nova versão da LingPON, foi possível reescrever as *Rules* apresentadas no Código 18 sem a necessidade de redundâncias na declaração das mesmas, conforme apresentado no Código 19.

Código 19: Exemplo de declaração de *FBE Rule* para controle de robôs.

```
1  fbe Robot
2      attributes
3          string atRobotPos 0.0
4      end_attributes
5      methods
6        method mtRunToBall ( ... )
7      end_methods
8      fbeRule rlRunToBall
9        condition
10           subcondition condRobotNotBallPos
11             premise prRbNotBallPos Robot.atRobotPos != ball.atBallPos
12           end_subcondition
13       end_condition
14       action
15           instigation inMoveRobot Robot.mtRunToBall();
16       end_action
17     end_fbeRule
18 end_fbe
```

Caso três instâncias do *FBE Robot*, apresentado no Código 19, forem criadas e nomeadas respectivamente como "*robot1*", "*robot2*" e "*robot3*", o pré-compilador PON criará três *Rules* no código PON pré-compilado, diferenciando-as apenas pela instância associada. Nota-se que a instância de Ball (*ball*) foi referenciada na *Premisse* da *FBE Rule*. Dessa forma, todas as *Rules* geradas no processo de compilação farão referência à mesma instância do *FBE* Ball. O resultado da pré-compilação, presente no código PON pré-compilado, pode ser observado no Código 20

Código 20: *Rules* criadas a partir do processo de pré-compilação

```
1  rule rlRunToBall0
2    condition
3      subcondition condRobotNotBallPos0
4        premise prRbNotBallPos robot1.atRobotPos != ball.atBallPos
5      end_subcondition
6    end_condition
7    action
8      instigation inMoveRobot0 robot1.mtRunToBall();
9    end_action
10 end_rule
11 rule rlRunToBall1
12   condition
13     subcondition condRobotNotBallPos1
14       premise prRbNotBallPos robot2.atGameStarted != ball.atBallPos
15     end_subcondition
16   end_condition
17   action
18     instigation inMoveRobot1 robot2.mtRunToBall();
19   end_action
20 end_rule
21 rule rlRunToBall2
22   condition
23     subcondition condRoboNotBallPos2
24       premise prRbNotBallPos robot3.atGameStarted != ball.atBallPos
25     end_subcondition
26   end_condition
27   action
28     instigation inMoveRobot2 robot3.mtRunToBall();
29   end_action
30 end_rule
```

Caso seja necessário alterar os requisitos do sistema de forma a controlar mais três robôs, na aplicação desenvolvida utilizando a versão original da LingPON, seria necessário

declarar três novas instâncias do *FBE Robot* (*robot3*, *robot4* e *robot5*) e adicionar três novas *Rules* ao código fonte. Essas alterações resultariam no acréscimo de 30 linhas de código ao código fonte.

A mesma mudança de requisito, utilizando a nova versão da LingPON, com suporte a declaração de *FBE Rules* não exigiria nenhuma alteração na declaração do *FBE* ou de *Rules*. A única alteração no código fonte seria a declaração de três novas instâncias do *FBE Robot*, a qual não resultaria em acréscimo de linhas no código.

3.1.3 CORREÇÃO DE ERRO: GERAÇÃO DE CÓDIGO-ALVO C++ COM MÚLTIPLAS INSTÂNCIAS DE *FBE*

Durante o desenvolvimento da aplicação de controle para partida de futebol de robôs, a qual é utilizada como estudo de caso deste trabalho, foi encontrado um erro (*bug*) que compromete o desenvolvimento de aplicações PON utilizando a atual versão do compilador PON (versão 1.0).

De forma sucinta, a atual versão do compilador não é capaz de distinguir diferentes instâncias de *FBEs* durante o processo de geração de código-alvo em C++. Para o compilador, todas as instâncias de um determinado *FBE* são representados por uma mesma classe C++. Este erro pode ser facilmente reproduzido pelo código PON apresentado no Código 21.

Código 21: Código utilizado para reproduzir erro de geração de código-alvo C++ na atual versão do LingPON.

```
1  fbe Robot
2    attributes
3          int atFunction 0
4    end_attributes
5    methods
6        method mtDefense (atFunction = 2)
7    end_methods
8  end_fbe
9
10 inst
11   Robot robot1
12   Robot robot2
13 end_inst
14
15 rule rlRobot1
16   condition
17       subcondition condition1
18           premise prRobotFunction1 robot1.atFunction == 1
19       end_subcondition
20   end_condition
21   action
22       instigation inRobot1Defense robot1.mtDefense();
23   end_action
24 end_rule
25 rule rlRobot2
26   condition
27       subcondition condition2
28           premise prRobotFunction1 robot2.atFunction == 1
29       end_subcondition
30   end_condition
31   action
32       instigation inRobot2Defense robot2.mtDefense();
33   end_action
34 end_rule
```

Ao analisar o Código 21, observa-se que duas instâncias do *FBE Robot* foram criadas (*robot1* e *robot2*) e, para cada instância, uma *Rule* foi associada. Dessa forma, a instância *robot1* deve relacionar-se apenas com a *Rule rlRobot1* e a instância *robot2* com *rlRobot2*.

Ao compilar este código-fonte com a opção de geração de código em C++, a atual versão do compilador irá gerar os arquivos de saída com sucesso. Entretanto, ao analisar o conteúdo do arquivo C++ "*Robot.cpp*", é possível observar que tanto *robot1* quanto *robot2* relacionam-se com as duas *Rules* , conforme apresentado no Código 22.

Código 22: Código C++ gerado pela atual versão do compilador.

```
1 Robot::Robot(rlRobot1 * rlRobot1, rlRobot2 * rlRobot2)
2 {
3     this->rlRobot1 = rlRobot1;
4     this->rlRobot2 = rlRobot2;
5     ...
6 }
```

Isto ocorre porque a atual versão do compilador PON transforma cada *FBE* presente no código-fonte PON em uma classe no código gerado em C++. Com isso, não existe diferenciação das *Rules* associadas às instâncias *robot1* e *robot2*. Este problema não fora observado em [Ferreira 2016] porque as aplicações PON desenvolvidas pelo mesmo, a saber Mira ao Alvo e Sistema de Vendas, apresentavam apenas uma instância de cada *FBE*.

A solução para este problema utilizando a estrutura e fluxo de compilação do atual compilador PON não é trivial, conforme fora discutido em reunião do grupo de pesquisa PON da UTFPR. Entretanto, com o advento do pré-compilador, a solução para tal problema tornou-se de certa forma simples. Durante o processo de pré-compilação, o pré-compilador é capaz de criar tantas cópias de um determinado *FBE* quantas forem o número de suas instâncias. Por exemplo, dado o Código 21, o pré-compilador criaria

dois *FBEs* no código-fonte PON pré-compilado (*Robotrobot1* e *Robotrobot2*), conforme apresentado no Código 23.

Código 23: Código PON pré-compilado gerado pelo pré-compilador PON para solucionar problema com múltiplas instâncias de *FBEs*.

```
1  fbe Robotrobot1
2  ...
3  end_fbe
4
5  fbe Robotrobot2
6  ...
7  end_fbe
8
9  inst
10     Robotrobot1 robot1
11     Robotrobot2 robot2
12 end_inst
13
14 rule rlRobot1
15     condition
16         subcondition condition1
17             premise prRobotFunction1 robot1.atFunction == 1
18         end_subcondition
19     end_condition
20     action
21         instigation inRobot1Defense robot1.mtDefense();
22     end_action
23 end_rule
24 rule rlRobot2
25     condition
26         subcondition condition2
27             premise prRobotFunction1 robot2.atFunction == 1
28         end_subcondition
29     end_condition
30     action
31         instigation inRobot2Defense robot2.mtDefense();
32     end_action
33 end_rule
```

Dessa forma, o atual compilador interpretaria cada *FBE* e sua instância relacionada separadamente, gerando assim os arquivos C++ de maneira correta.

Esta solução foi aplicada com sucesso e, utilizando o pré-compilador, foi possível criar a aplicação para controle de futebol de robôs em PON, a qual utiliza múltiplas instâncias de um dado *FBE*.

3.2 ESTUDO DE CASO - SOFTWARE DE CONTROLE PARA O FUTEBOL DE ROBÔS

O objetivo principal desta seção é apresentar o desenvolvimento do sistema de controle para uma partida de futebol de robôs em PON, segundo regras e características da categoria SSL, utilizando o *Framework* PON versão 2.0, a atual versão da LingPON e a nova versão da LingPON apresentada neste trabalho, bem como compará-las entre elas e também com uma solução funcionalmente equivalente construída sobre o PI por outrem.

Primeiramente, a aplicação de controle do Futebol de Robôs foi codificada em linguagem de programação C++ POO/PI por outro desenvolvedor. Na sequência, tal aplicação serviu como base para o desenvolvimento da aplicação sob o viés do PON, utilizando as materializações Framework 2.0, tecnologia LingPON 1.0 (dita original) e a nova versão desta tecnologia desenvolvida neste trabalho, aqui nomeada de LingPON 1.2.

Isto considerado, a subseção 3.2.1 apresenta uma descrição, em alto nível, de como o sistema de controle deve funcionar em uma partida de futebol de robôs. Na sequência, a subseção 3.2.2 apresenta os detalhes da aplicação desenvolvida sob o viés do POO/PI. A seção 3.2.3 apresenta detalhes das aplicações desenvolvidas sob o viés do PON. A seção 3.2.4 apresenta as comparações realizadas utilizando as diferentes implementações. Por fim, a Seção 3.2.5 apresenta as considerações finais sobre as diferentes soluções apresentadas no presente capítulo.

3.2.1 ESPECIFICAÇÃO DO *SOFTWARE* DE CONTROLE PARA UMA PARTIDA DE FUTEBOL DE ROBÔS

Baseando-se em três variáveis, nomeadamente *cmdReferee*, *lastCmd* e *teamColor*, o sistema de controle deve ser capaz de avaliar e determinar o comportamento para cada um dos robôs que estão sendo controlados. A variável *cmdReferee* representa o último

comando enviado pela aplicação *Referee Box*, descrita na subseção 2.4.1. A variável *lastCmd*, por sua vez, representa o valor predecessor enviado pela aplicação *Referee Box*, ou seja, o penúltimo comando recebido. A variável *TeamColor* representa a cor do time que está sendo controlado, podendo assumir o valor Amarelo (*Yellow*) ou Azul (*Blue*).

A Figura 26 apresenta o diagrama de atividades em UML para algumas poucas atividades recorrentes durante uma partida de futebol de robôs alcançaveis a partir das três variáveis recém mencionadas. Um diagrama de atividades contendo todos as atividades possíveis torna-se inviável de ser apresentado, devido ao grande número de condições a serem avaliadas pelas regras que regem o comportamento do sistema, as quais são apresentadas no Apêndice C.

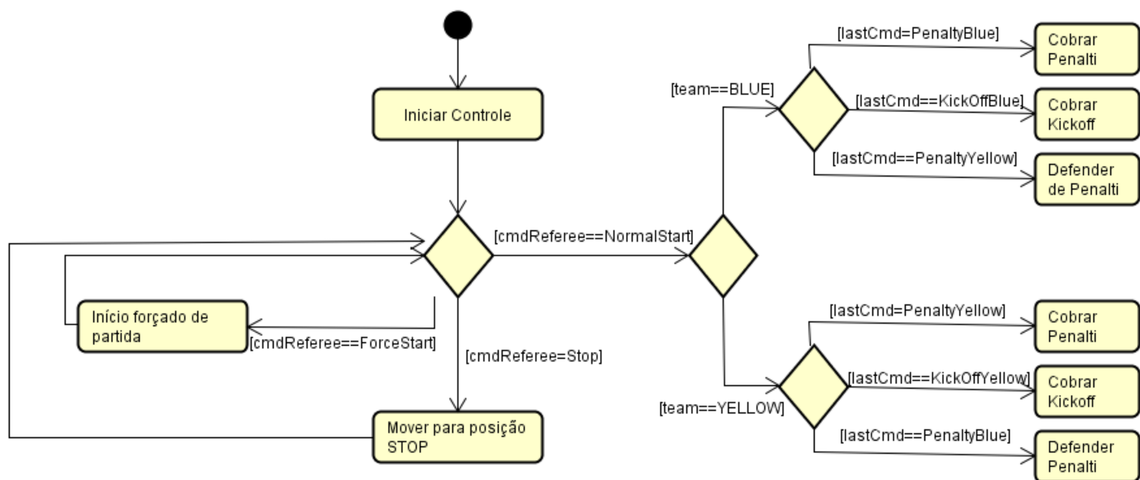

Figura 26: Diagrama de atividades simplificado de uma partida de futebol de robôs.

Como demonstrado em [Wiecheteck et al. 2011,Batista 2013], a partir da análise de um diagrama de atividades pode-se levantar as regras lógico causais que regem a aplicação. O levantamento de regras pode ser realizado respondendo as seguintes perguntas:

- Qual o objetivo da regra?
- O que precisa acontecer para que a regra seja executada?
- O que acontece se a regra for executada?

Dessa forma, aqui se enumeram duas regras que foram extraídas do diagrama de atividade apresentado na Figura 26 utilizando as perguntas descritas acima:

1. Nome da regra: Parar robôs

- Qual o objetivo da regra? Mover robôs para posições de início/reínicio de partida.
- O que precisa acontecer para que a regra seja executada? O comando enviado pelo *Referee Box* deve ser *Stop*.
- O que acontece se a regra for executada? Cada robô deve ir para sua posição, respeitando a distância mínima de 50 cm da bola.

2. Nome da regra: Cobrar pênalti (time azul)

   - Qual o objetivo da regra? Cobrar penalidade máxima
   - O que precisa acontecer para que a regra seja executada? O comando enviado pelo *Referee Box* deve ser *NormalStart*, o time controlado deve ser *Blue* e o penúltimo comando enviado pelo árbitro deve ser *PenaltyBlue*.
   - O que acontece se a regra for executada? Um robô do time azul, que está sendo controlado, deve cobrar a penalidade máxima.

A decisão sobre qual ação deve ser executada por cada um dos robôs apresenta uma certa variação de complexidade entre os possíveis cenários (decorrentes de atividades) em uma partida de futebol. Para alguns comandos enviados pela *Referee Box*, a decisão sobre qual ação executar é realizada com facilidade. Por exemplo, quando a aplicação de controle recebe o comando *Stop*, não é necessário verificar qual time está sendo controlado, pois, independente do time que está sendo controlado, todos os robôs deverão se posicionar-se de forma a defender o seu gol, a pelo menos 50 *cm.* da bola.

Entretanto, para alguns comandos essa decisão torna-se mais complexa. Conforme apresentado na Regra 2, quando o comando *NormalStart* é recebido, deve-se avaliar o penúltimo comando recebido, o qual é nomeado comando de preparação, e qual time está sendo controlado pelo sistema. Isso é necessário devido a diferença que determinados comandos geram em cada uma das equipes. Por exemplo, quando o comando de preparação *PenaltyBlue* é recebido, o time *Blue* deve se preparar para cobrança de penalidade máxima, enquanto o time *Yellow* deve se preparar para defender a cobrança.

Observa-se que os resultados das regras apresentadas acima geram um novo fluxo de avaliações. Por exemplo, apenas um robô integrante do time pode efetuar, de fato, a cobrança de penalidade máxima. Desse modo, o sistema deve ser capaz de julgar e escolher qual robô irá executar a cobrança, conforme apresentado no diagrama de atividades da Figura 27. Essa escolha pode ser realizada com base na função que cada um dos robôs

exerce durante a partida. Por exemplo, pode-se determinar que o robô que atua como meio-campo é quem deverá executar a cobrança de penalidade máxima.

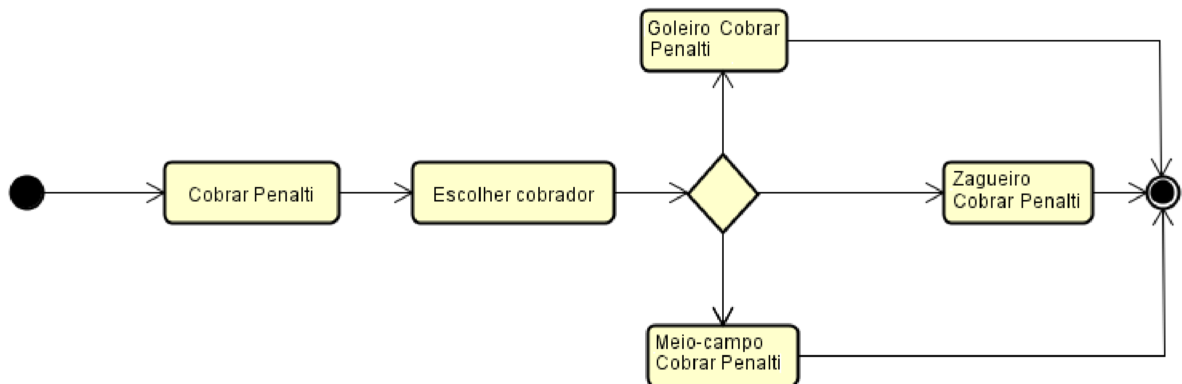

Figura 27: Diagrama de atividades para escolha do batedor de penalidade máxima.

Após a decisão sobre qual ação deve ser executada, o *software* de controle deve ser capaz de transformar essa decisão em comandos de controle a serem executados por cada um dos robôs. Cada robô possui motores utilizados para deslocamento e um dispositivo chutador utilizado para lançar a bola, isto é, realizar comandos de chute e passe.

Dessa forma, o *software* de controle deve calcular as ações a serem realizadas por cada robô de forma a atingir o objetivo desejado. Por exemplo, quando a ação a ser executada é cobrar uma penalidade máxima, o *software* deve calcular qual a velocidade e ângulo com que o robô deve dirigir-se à bola e qual a potência de chute deve ser utilizada para que a cobrança seja executada com sucesso.

Portanto, os sistemas de controle construídos sob o viés do PI e PON, a serem apresentados nas seções subsequentes, devem ser capazes de materializar as diversas regras que modelam o comportamento dos robôs em uma partida de futebol de robôs, realizar os cálculos necessários, incluindo os lógico-causais. Maiores detalhes sobre as especificações técnicas da categoria SSL da Robocup e informações sobre os requisitos funcionais desta aplicação podem ser encontrado no Apêndice B deste trabalho.

3.2.2 SOLUÇÃO DESENVOLVIDA SOB O VIÉS DO PARADIGMA IMPERATIVO - PROGRAMAÇÃO ORIENTADA A OBJETOS

Esta seção tem por objetivo apresentar em detalhes a codificação do sistema de controle para uma partida de futebol de robôs sob o viés do PI, utilizando a linguagem de programação C++. O código apresentado foi desenvolvido por prof. João A. Fabro e o então estudante de Engenharia da Computação André Botta, que foi bolsista de iniciação

científica da UTFPR [Botta 2012]. Apesar de ter sido desenvolvida por outrem, esta aplicação é aqui apresentada por ter sido utilizada como base para o desenvolvimento das três aplicações PON apresentadas neste trabalho.

Seguindo o modelo de desenvolvimento orientado a objetos, cada entidade que compõe o sistema de controle foi representada em termos de classes, como apresentado no diagrama de classes da Figura 28[1].

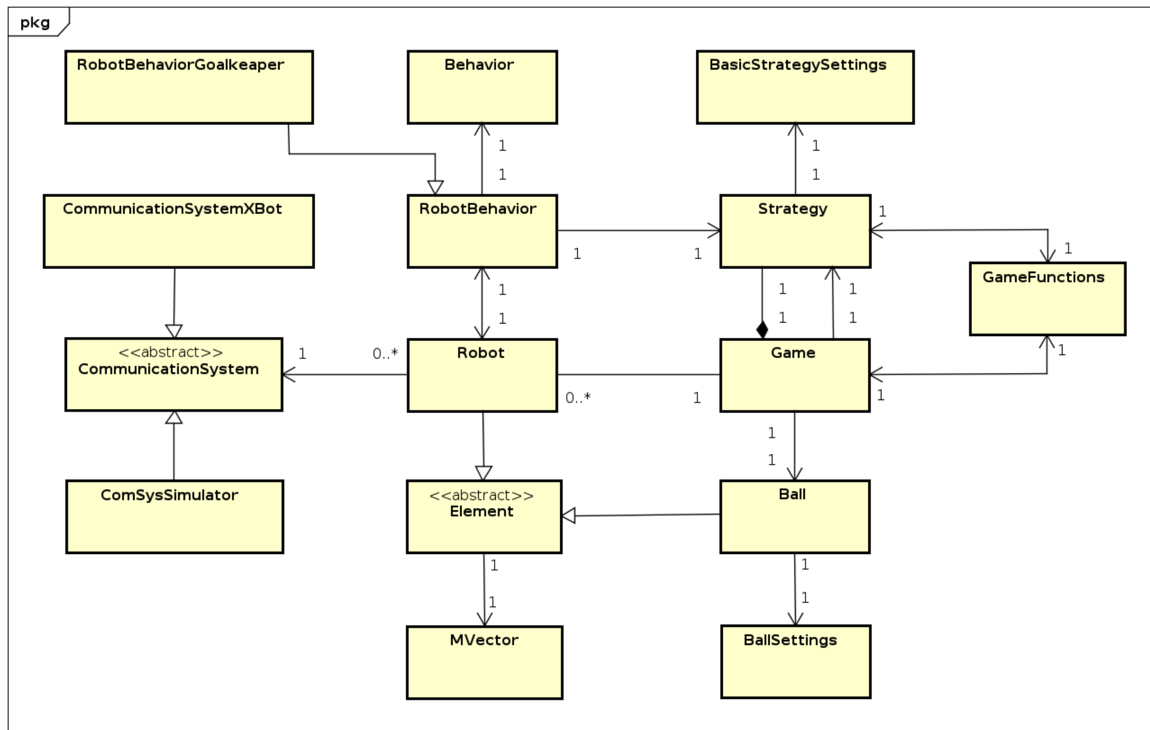

Figura 28: Diagrama de classes simplificado da solução desenvolvida em PI.

Cada um dos robôs é representado por um objeto da classe *Robot*. Essa classe possui atributos que representam as principais características de um robô, tais como número de identificação, posição atual e velocidades normal e angular. De forma similar, uma instância da classe *Ball* representa a bola que está sendo utilizada na partida. Por sua vez, a classe *CommunicationSystem* é utilizada como interface de comunicação entre o *software* de controle e cada um dos robôs que estão sendo controlados.

Cada instância da classe *RobotBehavior* contém as funções de cálculo utilizadas pelas instâncias da classe *Robot* para controlar seus movimentos. Um exemplo de tais funções é aquela que calcula a velocidade com que o robô deve deslocar-se em direção a bola. Neste sentido, os métodos e atributos presentes na classe *RobotBehavior* são utilizados apenas pelas instâncias da classe *Robot*. Portanto, não haveria necessidade de

[1] O diagrama de classes apresentado foi criado a partir dos esforços de André Botta.

separar a lógica relacionada aos robôs em duas classes.

A classe *GameFunctions*, por sua vez, apresenta métodos utilizados para obter informações do time como um todo. Um exemplo de método presente nesta classe é o utilizado para calcular qual robô está mais perto da bola em um determinado momento.

A classe *Game*, instanciada no início da execução da aplicação, é responsável por instanciar todos os objetos que serão utilizados durante a execução do *software* de controle. Em sua inicialização, o número necessário de instâncias da classe *Robot* é criado, como também uma respectiva instância da classe *RobotBehavior*. Para cada ciclo de execução, isto é, cada ciclo de leitura dos pacotes recebidos a partir do sistema *SSL-Vision* (descrita na subseção 2.4.1) a classe *Game* aciona a execução da classe *Strategy*.

Utilizando as informações obtidas através das instâncias da classe *Robot* e *GameFunctions*, a instância da classe *Strategy* irá determinar qual ação cada um dos robôs deverá executar. É importante salientar que, devido a necessidade do objeto da classe *Strategy* conhecer o estado de todos os robôs, existe apenas uma instância da classe *Strategy*, a qual é referenciada pelas diferentes instâncias de *Robot*. Portanto, a classe *Strategy* desempenha o papel de controlador central para todos as instâncias de *Robot*.

Entretanto, apesar de a instância da classe *Strategy* representar o conhecimento lógico causal do sistema, não é fácil entender quais são as as regras de controle que a mesma aplica sobre os robôs que estão sendo controlados. Isto ocorre devido a limitações do paradigma imperativo, utilizado na concepção do sistema de controle.

Conforme apresentado na seção 2.1, a concepção de *software* sob o viés do PI é realizada utilizando sequências de instruções, as quais realizam buscas sobre entidades passivas (dados e comandos). Nesse sentido, as regras que regem o sistema foram implementadas com a utilização de estruturas de controle, tais como *switch-case* (escolha caso) e *if-then-else* (se então senão) e estruturas de repetição como *for* (para passo) e *while* (enquanto). Essas estruturas apresentam-se dispersas em diferente métodos e classes, fato este que dificulta o entendimento das regras que atuam sobre o sistema.

Em um primeiro momento, a instância da classe *Strategy* avalia qual foi o último comando enviado pela aplicação *Referee Box*, a qual atua como árbitro da partida. A avaliação foi codificada utilizando a estrutura de controle *switch-case*, conforme apresentado no Código 24.

Código 24: Estrutura de controle *switch-case* utilizada pela classe *Strategy* (*Strategy.cpp*) para determinar qual ação será executada.

```
1  void Strategy::makeStrategy ()
2  {
3      switch ( game->getCmdReferee() )
4      {
5          case Stop:
6          {
7              strategyStop();
8              break;
9          }
10         case Ready:
11         {
12             char lastCmd = game->getLastCmdReferee();
13             if ( gameFunctions->foulForEnemy() )
14                 game->setReadyEnemy ( true );
15
16             if ( game->getTeamColor() == YELLOW )
17             {
18                 if ( lastCmd == PenaltyYellow )
19                 {
20                     readyPenalty();
21                 }
22                 else if ( lastCmd == KickOffYellow )
23                 {
24                     readyKickoff ();
25                 }
26                 else
27                 {
28                     if ( lastCmd == PenaltyBlue )
29                         penaltyEnemy ();
30                     else
31                     {
32                         strategyStart();
33                     }
34                 }
35             }
36             else
37             {
38                 if ( lastCmd == PenaltyBlue )
39                 {
40                     readyPenalty();
41                 }
42                 else if ( lastCmd == KickOffBlue )
43                 {
44                     readyKickoff ();
45                 }
46                 else
47                 {
48                     if ( lastCmd == PenaltyYellow )
49                         penaltyEnemy ();
50                     else
51                     {
52                         strategyStart();
53                     }
54                 }
55             }
56             break;
57         }
58         case Start:
59         {
60             strategyStart();
61             break;
62         }
63
64         case KickOffYellow:
65         {
66             if ( game->getTeamColor() == YELLOW )
67             {
68                 strategyKickoff ();
69             }
70             else
71             {
72                 strategyStop ();
73             }
```

```
74                  break;
75              }
76              case PenaltyYellow :
77              {
78                  if ( game->getTeamColor() == YELLOW )
79                  {
80                      penaltyTeam();
81                  }
82                  else
83                  {
84                      penaltyEnemy();
85                  }
86                  break;
87              }
88              case DirectFreeKickYellow:
89              {
90                  if ( game->getTeamColor() == YELLOW )
91                  {
92                      strategyDirectKick ();
93                  }
94                  else
95                  {
96                      strategyStop();
97                  }
98                  break;
99              }
100             case IndirectFreeKickYellow:
101             {
102                 if ( game->getTeamColor() == YELLOW )
103                 {
104                     strategyIndirectKick();
105                 }
106                 else
107                 {
108                     strategyStop();
109                 }
110                 break;
111             }
112             case KickOffBlue:
113             {
114                 if ( game->getTeamColor() == BLUE )
115                 {
116                     strategyKickoff ();
117                 }
118                 else
119                 {
120                     strategyStop ();
121                 }
122                 break;
123             }
124             case PenaltyBlue :
125             {
126                 if ( game->getTeamColor() == BLUE )
127                 {
128                     penaltyTeam();
129                 }
130                 else
131                 {
132                     penaltyEnemy();
133                 }
134                 break;
135             }
136             case DirectFreeKickBlue:
137             {
138                 if ( game->getTeamColor() == BLUE )
139                 {
140                     strategyDirectKick ();
141                 }
142                 else
143                 {
144                     strategyStop();
145                 }
146                 break;
```

```
147             }
148             case IndirectFreeKickBlue:
149             {
150                 if ( game->getTeamColor() == BLUE )
151                 {
152                     strategyIndirectKick();
153                 }
154                 else
155                 {
156                     strategyStop();
157                 }
158                 break;
159             }
160             default:
161             {
162                 strategyHalt ();
163                 break;
164             }
165         }
166     }
```

Conforme mencionado na seção 3.2.1, a decisão sobre qual ação deve ser executada por cada um dos robôs apresenta diferentes níveis de complexidade entre os possíveis cenários em uma partida de futebol. Quando o comando recebido é '*Stop*', a classe *Strategy* não precisa processar mais nenhuma informação para determinar qual estratégia deve ser executada, pois esse comando define por si só qual ação deve ser executada.

Entretanto, quando o comando recebido é '*Ready*', o qual pode representar o reinício da partida após a marcação de uma penalidade máxima, a classe *Strategy* deve avaliar o comando de preparação recebido anteriormente e a cor do time que está sendo controlado. Isso é necessário porque o comportamento esperado para o reinício da partida após uma penalidade máxima a favor do time que está sendo controlado é diferente daquele quando a penalidade é assinalada a favor do time adversário.

Nesse sentido, a classe *Strategy* avalia o comando de preparação, com base na cor do time que está sendo controlado, utilizando a estrutura de controle *if-then-else*, conforme apresentado entre as linhas 16 e 55 do Código 24. Uma vez avaliado o comando enviado pela aplicação *Referee Box* e, quando necessário, o comando de preparação, a instância da classe *Strategy* define então qual estratégia será executada pelos robôs que compõem a equipe que está sendo controlada.

Entretanto, em alguns casos, a classe *Strategy* deve ainda determinar qual dos robôs irá executar uma determinada ação. Por exemplo, quando o árbitro determina a paralisação da partida, representado pelo comando *'Stop'*, a estratégia adotada é posicionar alguns robôs próximos à bola. Para isso, a aplicação deve escolher quais robôs devem deslocar-se para próximo à bola.

O método responsável por executar a estratégia *'Stop'* é apresentado no Código

25. Nas linha 8 e 9 do Código 25, o método avalia a posição da bola em relação ao gol que está sendo defendido. Se a bola estiver próxima ao gol que está sendo defendido e o ângulo entre a bola e o gol for maior que 80°, os robôs que possuem a função "MIDFIELD_ONLY", "STRIKER_LEFT" e "STRIKER_RIGHT" devem deslocar-se para junto à posição da bola. Caso contrário, além destes três robôs se deslocarem para próximo à bola, os defensores devem movimentar-se para próximo à área de defesa.

Código 25: Método da classe *Strategy* utilizado para executar a estratégia *Stop*.

```
1   void Strategy::stop()
2   {
3       Behavior behavior1 = DEFENDER_LEFT;
4       Behavior behavior2 = DEFENDER_RIGHT;
5       double angle = game->getOurGoal()->calculateAngleTo( game->getBall()->getPosition() );
6       gameFunctions->oppositeAngle ( angle );
7
8       if (    game->getBall()->getPosition()->getDistanceTo ( game->getOurGoal() ) <= bss->distanceMinStop
9               || abs ( angle * 180 / PI ) > 80 )
10      {
11          NDefense ( 3,   gameFunctions->getRobot ( MIDFIELD_ONLY ),
12                          gameFunctions->getRobot ( STRIKER_LEFT ),
13                          gameFunctions->getRobot ( STRIKER_RIGHT ));
14      }
15      else
16      {
17          NDefense ( 3,   gameFunctions->getRobot ( MIDFIELD_ONLY ),
18                          gameFunctions->getRobot ( STRIKER_LEFT ),
19                          gameFunctions->getRobot ( STRIKER_RIGHT ));
20
21          auxStopGoal2Pl (gameFunctions->getRobot( behavior1 ),
22                          gameFunctions->getRobot( behavior2 ));
23      }
24  }
```

Com a estratégia a ser executada e o robô que irá a executar definidos, a classe *Strategy* aciona a execução de diferentes métodos das classes *Robot*, *RobotBehavior* e *GameFunctions*. Esses métodos realizam cálculos matemáticos diversos que são utilizados pela classe *Strategy* para enviar os comandos necessários para o robô executar a ação desejada. Tais cálculos estão relacionados ao campo de estudo da robótica, o qual não faz parte do escopo principal deste trabalho e por esse motivo não serão abordados.

Os Códigos 24 e 25 demonstram como as regras que regem o comportamento do sistema são apresentadas de forma difusa no código-fonte PI/POO. Esta característica dificulta o entendimento do funcionamento da aplicação em uma primeira vista, exigindo grande esforço intelectual de quem lê o código-fonte. Isto acaba por dificultar a manutenção do código-fonte, uma vez que pequenas alterações em determinados métodos podem causar consequências inesperadas para o comportamento da aplicação.

3.2.3 SOLUÇÕES DESENVOLVIDAS SOB O VIÉS DO PARADIGMA ORIENTADO A NOTIFICAÇÕES

Essa seção apresenta o desenvolvimento de aplicações de controle para uma partida de futebol de robôs desenvolvidas sob o viés do PON, sendo uma desenvolvida a partir do *Framework* PON C++ 2.0, outra utilizando a LingPON 1.0 e outra ainda utilizando a LingPON 1.2. Todas as soluções foram construídas reaproveitando parte do código não efetivamente lógico-causal da solução apresentado na seção anterior, principalmente as funções de cálculo referentes a movimentação dos robôs.

Apesar dos avanços e contribuições para a LingPON apresentados neste trabalho, a LingPON ainda apresentava certas limitações. Isto é assaz natural em linguagem que se encontra em fase de evoluções. Entretanto, o fato é que algumas limitações dificultavam o desenvolvimento de aplicações complexas, como é o caso do sistema de controle *Robocup*. Entre essas limitações pode-se destacar a dificuldade de realizar cálculos matemáticos complexos no código fonte LingPON e a incapacidade de utilizar bibliotecas externas, tais como bibliotecas para comunicação via *socket*.

Por esse motivo, esse trabalho propõe a construção de uma aplicação modularizada, isto é, composta por um módulo construído sob o viés do PON e outro módulo construído segundo o PI para aqueles tipos de especificidades citadas. Dessa forma, por exemplo, o código dedicado referente à comunicação via *socket* foi mantido em PI. Entretanto, todas as regras lógico-causais da aplicação foram transcritas para as estruturas computacionais do PON. A razão pela qual foi utilizada essa abordagem é permitir que o mesmo modelo estrutural da solução pudesse ser aplicado no desenvolvimento das aplicações PON utilizando o *Framework* PON C++ 2.0, a LingPON 1.0 e a LingPON 1.2.

Dessa forma, a Figura 29 apresenta o modelo estrutural PON obtido como solução para esse estudo de caso. O modelo em PON é demonstrado por meio de um diagrama de classes UML. Em tempo, neste diagrama identifica-se os elementos que compõem a base de fatos *FBE* com o estereótipo $\ll NOP_FBE \gg$.

Conforme apresentado no diagrama de classes da Figura 29, o *FBE Robot* representa cada um dos robôs que estão sendo controlados pela aplicação. Nas aplicações desenvolvidas utilizando o *Framework* PON e LingPON 1.2, as *Rules* que regem o comportamento de cada um dos robôs são definidas no escopo do próprio *FBE Robot*. Em tempo, por não suportar a declaração de *Rules* no escopo de *FBEs*, as *Rules* foram declaradas fora do escopo do *FBE Robot* na solução desenvolvida utilizando a LingPON 1.0. Detalhes específicos sobre cada uma das implementações são apresentados nas próximas seções.

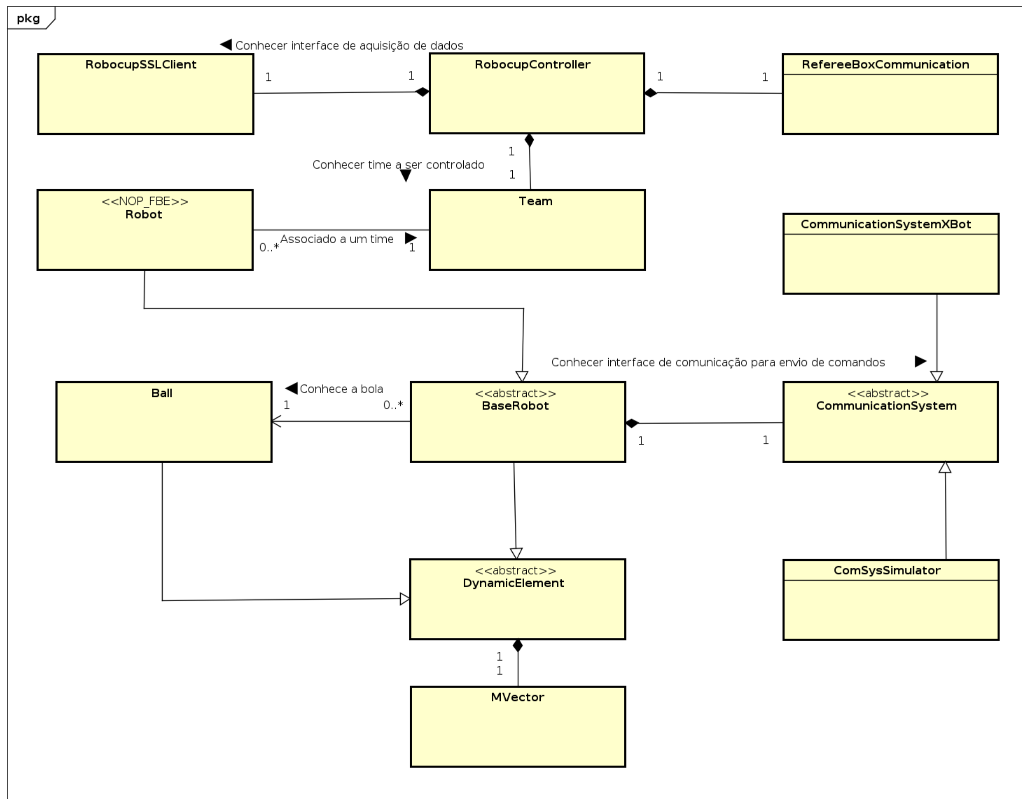

Figura 29: Diagrama de classes do *software* de controle para partida de futebol de robôs em PON.

As demais classes apresentadas no diagrama da Figura 29 assemelham-se muito às classes utilizadas para a construção da solução sob o viés do PI e foram desenvolvidas utilizando apenas a linguagem de programação C++, sem nenhum conceito relacionado ao PON. A instância da classe *RobocupController* é responsável por instanciar a classe *Team*. A instância da classe *Team*, por sua vez, é responsável por criar o número de instâncias necessárias do *FBE Robot* a serem controladas pela aplicação. Ademais, a instância da classe *Team* é responsável por atualizar o estado dos *Attributes* das instâncias do *FBE Robot* com as informações recebidas a partir do sistema *SSL-Vision* e da aplicação *RefereeBox*. Neste sentido, a classe *Team* foi criada de forma a permitir um melhor gerenciamento de todos os robôs que estão sendo controlados pelo sistema.

A instância da classe *CommunicationSystem*, por sua vez, representa a interface de comunicação entre a aplicação de controle e os robôs. A instância da classe *Ball* representa a bola que está sendo utilizada na partida. Em um primeiro momento, *Ball* foi modelada para ser um *FBE*. Dessa forma, seria possível a criação de *Rules* a partir da correlação dos *FBEs Robot* e *Ball*. Um exemplo de *Rule* que poderia ser definida a partir da correlação destes *FBEs* seria a *Rule* que define que o robô deve deslocar-se para bola quando a

distância entre eles for menor que 50 cm. De forma literal, esta *Rule* pode ser definida como:

Se ((robot1.atPositionX - ball.atPositionX) <0.5) Então robot1.mtMoveToBall()

Entretanto, as materializações em *software* do PON (*Framework* PON 2.0, LingPON 1.0 e LingPON 1.2) não suportam a declaração de operações aritméticas na declaração de *Premises*. Por esse motivo, *Ball* foi definida como uma classe PI/POO convencional ao invés de ser definida como um *FBE*.

Desse modo, o *FBE Robot* possui alguns *Attributes booleanos* (*atIsReady*, *atBallEnemyField*, *atClosestToBall* e *atEnemyOnGoalLine*) que definem sua percepção em relação ao ambiente, incluido percepção sobre outros robôs e sobre a bola. Assim, ao receber uma nova percepção do ambiente, cada *Robot* atualiza o estado destes *Attributes*. E, quando um *Attribute* do *FBE Robot* tem seu estado alterado, o mesmo notifica as *Premises* relacionadas às *Rules* do *FBE Robot*.

Dessa forma, o fluxo de execução da aplicação é controlado pela entidade *PON FBE* Robot presente na aplicação. Assim, evita-se reavaliações desnecessárias de expressões lógico-causais presente na solução desenvolvida sob o viés do PI, tais como a avaliação do último valor enviado pela aplicação *Referee Box*, a cada ciclo de execução, mesmo quando o mesmo não foi alterado.

Outrossim, o cerne de uma aplicação PON está na construção de *Rules* a partir da associação de *Conditions* com *Premises* e *Instigations* com *Actions*. Dessa forma, as *Rules* podem ser apresentadas em uma tabela, conforme apresentado na Tabela 1, a qual apresenta um pequeno grupo de *Rules* criadas para controlar os robôs durante uma partida de futebol.

Rule	**Nome**	**Condition e suas Premises**	**Action e suas Instigations**
1	rlMOStop	RobotPON.atPlayerRole == “MIDFIELD_ONLY” RobotPON.atCmdReferee == Stop	RobotPON->mtStopAtacante
2	rlMOBlueKickoff	RobotPON.atPlayerRole== “MIDFIELD_ONLY” RobotPON.atCmdReferee == KickoffBlue RobotPON.atTeam == Blue	RobotPON->mtMovePositionToKick
3	rlMOBlueReadyKickoff	RobotPON.atPlayerRole== “MIDFIELD_ONLY” RobotPON.atCmdReferee == Start RobotPON.atLastCmdReferee == KickoffBlue RobotPON.atTeam == Blue	RobotPON->mtReadyKickoff

Tabela 1: *Rules*, *Conditions* e suas *Premises* e *Actions* instigadas do software de controle PON para futebol de robôs.

A primeira *Rule* apresentada na Tabela 1 determina que o robô, cuja função é “MIDFIELD_ONLY”, mova-se a para a posição de reínicio de partida quando o comando

recebido por parte da aplicação *RefereeBox* for "Stop". De forma semelhante, a *Rule* número 2 determina que o robô, cuja função é "MIDFIELD_ONLY", mova-se para a posição que o permita chutar a bola em direção ao gol adversário quando o comando recebido por parte da aplicação *RefereeBox* for "KickoffBlue" e a cor do seu time for azul (*Blue*). Por fim, a *Rule* número 3 determina que o robô, cuja função é "MIDFIELD_ONLY", execute a ação de chutar a bola quando a cor do seu time for azul (*Blue*) e os dois últimos comandos recebidos por parte da aplicação *RefereeBox* forem, respectivamente, "KickoffBlue" e "Start".

Assim, um total de 74 *Rules* foram cridas a partir da associação de *Premises* e *Instigations* dentro do escopo do *FBE PONRobot*, conforme apresentado no Apêndice C e aplicadas nas soluções desenvolvidas sob o viés do PON. No mais, as seções posteriores apresentam maiores detalhes sobre a implementação de cada uma das aplicações desenvolvidas a partir das materializações visadas do PON em *software*.

3.2.3.1 SOLUÇÃO DESENVOLVIDA SOB O VIÉS DO FRAMEWORK PON 2.0

Essa seção apresenta a aplicação de controle para uma partida de futebol de robôs construída utilizando o *Framework* PON C++ 2.0.

Além de possuir *Attributes* e *Methods*, o *FBE Robot* foi criado apresentando também em sua definição um conjunto de *Rules* que definem qual será seu comportamento durante a execução da aplicação. Essa abordagem garante uma programação mais modular, na qual cada *FBE* possui um conjunto de *Rules*. Dessa forma, ao criar uma nova instância do *FBE PONRobot*, não somente as entidades *Attributes* e *Methods* serão criadas, como também as demais entidades que compõem uma aplicação PON, tais como *Premises*, *Conditions*, *Rules*, *Actions* e *Instigations* [Ronszcka 2012].

Dessa forma, o fragmento de código extraído do código-fonte da solução desenvolvida e apresentado no Código 26 mostra como a *Rule* 3 da Tabela 1 foi construída com a utilização do *Framework* C++ 2.0 do PON. No código apresentado, o operador *this* utilizado na declaração das *Premises* refere-se à própria instância do *FBE Robot*. A *Rule* apresentada determina que o robô, cuja função atribuída é *"MIDFIELD_ONLY"*, inicie a partida no meio de campo após um gol do time adversário.

Código 26: Código de uma *Rule* presente em PONRobot.cpp

```
1  PREMISE (prRoleMidfieldOnly, this->atRole,
2            new String(this, "MIDFIELD_ONLY"),
3            Premise::EQUAL, Premise::STANDARD, false);
4
5  PREMISE (prRefereeCmdStartGame, this->atRefereeCmd,
6            new Char(this, Referee::Ready),
7            Premise::EQUAL, Premise::STANDARD, false);
8
9  PREMISE (prLastRefereeCmdKickoffBlue, this->atLastRefereeCmd,
10           new Char(this, Referee::KickOffBlue),
11           Premise::EQUAL, Premise::STANDARD, false);
12
13 PREMISE (prTeamBlue, this->atTeamColor,
14           new String(this, "BLUE"),
15           Premise::EQUAL, Premise::STANDARD, false);
16
17 RULE (rlMOBlueReadyKickoff, scheduler, Condition::CONJUNCTION) ;
18 rlMOBlueReadyKickoff->addPremise(prRoleMidfieldOnly) ;
19 rlMOBlueReadyKickoff->addPremise(prRefereeCmdStartGame) ;
20 rlMOBlueReadyKickoff->addPremise(prLastRefereeCmdKickoffBlue) ;
21 rlMOBlueReadyKickoff->addPremise(prTeamBlue) ;
22 rlMOBlueReadyKickoff->addInstigation(this->mtReadyKickoff) ;
```

3.2.3.2 SOLUÇÃO DESENVOLVIDA SOB O VIÉS DO PARADIGMA ORIENTADO A NOTIFICAÇÕES - LINGPON 1.0

Essa seção apresenta a aplicação de controle para uma partida de futebol de robôs desenvolvida utilizando a atual versão da LingPON (versão 1.0).

Conforme apresentado na seção 3.1, na atual versão da LingPON não é possível agregar *Rules* à nível de *FBEs*. Dessa forma, todas as *Rules* que regem o comportamento de um determinado robô em uma partida de futebol de robôs foram criadas referenciando cada uma das instâncias do *FBE Robot*, conforme apresentado no Código 27.

Código 27: Código de uma *Rule* desenvolvida utilizando a atual versão da LingPON presente em robocup.pon

```
1  rule rlMOBlueReadyKickoffBlue
2   condition
3     subcondition condition41
4       premise prMidfieldOnly robot1.atRole == “MIDFIELD_ONLY” and
5       premise prStartGame robot1.atRefereeCmd == ‘ ’ and
6       premise prLastCmdKickoffBlue robot1atLastRefereeCmd == ‘K’ and
7       premise prTeamBlue robot1.atTeamColor == “BLUE”
8     end_subcondition
9   end_condition
10  action
11    instigation inMOBlueReadyKickoffBlue robot1.mtReadyKickoff();
12  end_action
13 end_rule
```

Neste sentido, o conjunto de 74 *Rules* criadas para a instância “*robot1*” do *FBE* Robot foi replicado para as outras 5 instâncias do *FBE* Robot existentes na aplicação. Portanto, ao invés de declarar 74 *Rules*, foi necessário declarar 444 *Rules*, fato este que notoriamente influenciou no tamanho e na manutenibilidade do código-fonte desenvolvido e no tempo necessário para o desenvolvimento da aplicação.

3.2.3.3 SOLUÇÃO DESENVOLVIDA SOB O VIÉS DO PARADIGMA ORIENTADO A NOTIFICAÇÕES - LINGPON 1.2

Essa seção apresenta a aplicação de controle para uma partida de futebol de robôs desenvolvida utilizando a nova versão da LingPON (versão 1.2), desenvolvida e apresentada neste trabalho.

O *FBE Robot* foi construído utilizando o conceito de *FBE Rules*, o qual permite declarar *Rules* dentro do escopo do *FBE* na LingPON. Assim, cada nova instância do *FBE Robot* possuirá em seu escopo *Attributes*, *Methods* e todas as *Rules* necessárias para seu controle durante uma partida de futebol de robôs.

Neste sentido, o Código 28 apresenta como a *Rule* 3 da Tabela 1, cuja implementação utilizando o *Framework* PON 2.0 foi apresentado no Código 26, foi codificada segundo as definições da LingPON. Essa *Rule* determina que o robô, cuja função atribuída é *"MIDFIELD_ONLY"*, inicie a partida no meio de campo após um gol do time adversário.

Código 28: Código de uma *Rule* desenvolvida utilizando a nova versão da LingPON presente em robocup.pon

```
1  fbe Robot
2   attributes
3    string atRole "OFF"
4    char atRefereeCmd 'N'
5    char atLastRefereeCmd 'N'
6    string atTeamColor "none"
7    ...
8   end_attributes
9   methods
10   method begin_method readyKickoff(); end_method'
11   ...
12  end_methods
13  fbeRule rlMOBlueReadyKickoffBlue
14   condition
15     subcondition condition41
16       premise prMidfieldOnly Robot.atRole == "MIDFIELD_ONLY" and
17       premise prStartGame Robot.atRefereeCmd == ' ' and
18       premise prLastCmdKickoffBlue Robot.atLastRefereeCmd == 'K' and
19       premise prTeamBlue Robot.atTeamColor == "BLUE"
20    end_subcondition
21  end_condition
22  action
23     instigation inMOBlueReadyKickoffBlue Robot.mtReadyKickoff();
24  end_action
25  end_fbeRule
26  ...
27 end_fbe
```

A utlização de *FBE Rules*, adicionada à LingPON no desenvolvimento deste trabalho, permitiu a criação do código-fonte mais compacto, em termos de número de linhas de código, quando comparado à solução desenvolvida a partir da LingPON 1.0. Isto porque apenas 74 *FBE Rules* foram declaradas no código-fonte, ao invés de 444 *Rules* que foram declaradas no código-fonte LingPON 1.0.

3.2.4 COMPARAÇÕES ENTRE A APLICAÇÃO DE CONTROLE DE FUTEBOL DE ROBÔS DESENVOLVIDA NOS PARADIGMA ORIENTADO A OBJETOS E NO PARADIGMA ORIENTADO A NOTIFICAÇÕES

Nesta seção são apresentadas algumas comparações entra a solução desenvolvida sob o viés do POO/PI e as soluções desenvolvidas sob o viés do PON; Tais comparações são no tocante a medidas de complexidade de código (*i.e.* quantidade de linhas de código e *tokens* presente no código-fonte) e facilidade de manutenibilidade. Por fim, são apresentadas as reflexões sobre as comparações.

3.2.4.1 COMPARAÇÕES DE COMPLEXIDADE DE CÓDIGO-FONTE ENTRE A APLICAÇÃO DE CONTROLE DE FUTEBOL DE ROBÔS DESENVOLVIDA EM PI/POO E PON

Essa seção apresenta o experimento realizado para comparar a complexidade do código-fonte das aplicações de controle para uma partida de futebol de robôs. Esse experimento é relativo ao objetivo específico deste trabalho que visa comparar a aplicação desenvolvida utilizando a nova versão da LingPON com soluções semelhantes desenvolvidas sob o PI/POO e outras materializações do PON (nomeadamente versão anterior da LingPON e *Framework* PON 2.0).

As métricas utilizadas para comparações em complexidade de código-fonte foram número de linhas de código (*LOC - lines of code*) e número de *tokens* na linguagem (*i.e* medidas em C++ puro, *Framework* PON 2.0, LingPON atual e LingPON 1.2). O objetivo é comparar a complexidade de código entre programas similares (*i.e.* que solucionam o mesmo problema) desenvolvidos em diferentes paradigmas de programação e com técnicas diferentes.

Para a contagem de linhas de código-fonte foi utilizado a ferramenta *cloc* [Danial 2006]. Essa ferramenta foi escolhida por ser de licença livre e *open-source*. Para a

contagem da quantidade de *tokens* presentes no código-fonte, foi criado um analisar léxico (implementado com a ferramenta *flex*), cujo código-fonte é apresentado no Apêndice D.

A contagem do número de linhas de código e quantidade de *tokens* presentes no código-fonte da aplicação desenvolvida utilizando o *Framework* PON foram realizadas desconsiderando o código-fonte do *Framework* PON. Foi utilizada essa abordagem porque o desenvolvedor não necessita ter conhecimento sobre o código interno do *Framework* PON, ele deve apenas utilizar as funções que o mesmo disponibiliza.

A Figura 30 apresenta os dados da medição global do número de linhas de código. Essa medição mostra uma diferença considerável de quantidade de linhas de código entre as aplicações desenvolvidas utilizando a atual versão da LingPON e a nova versão apresentada neste trabalho. Essa grande diferença se deve ao fato de, na atual versão da LingPON, o desenvolvedor ter que declarar o mesmo conjunto de *Rules* para cada uma das instâncias do *FBE* Robot. Utilizando o conceito de *FBE Rules* presente na nova versão da LingPON, a redundância na declaração de *Rules* é mitigada e faz com que o número de linhas de código se aproxime das soluções desenvolvidas a partir do PI/POO e a partir do *Framework* PON.

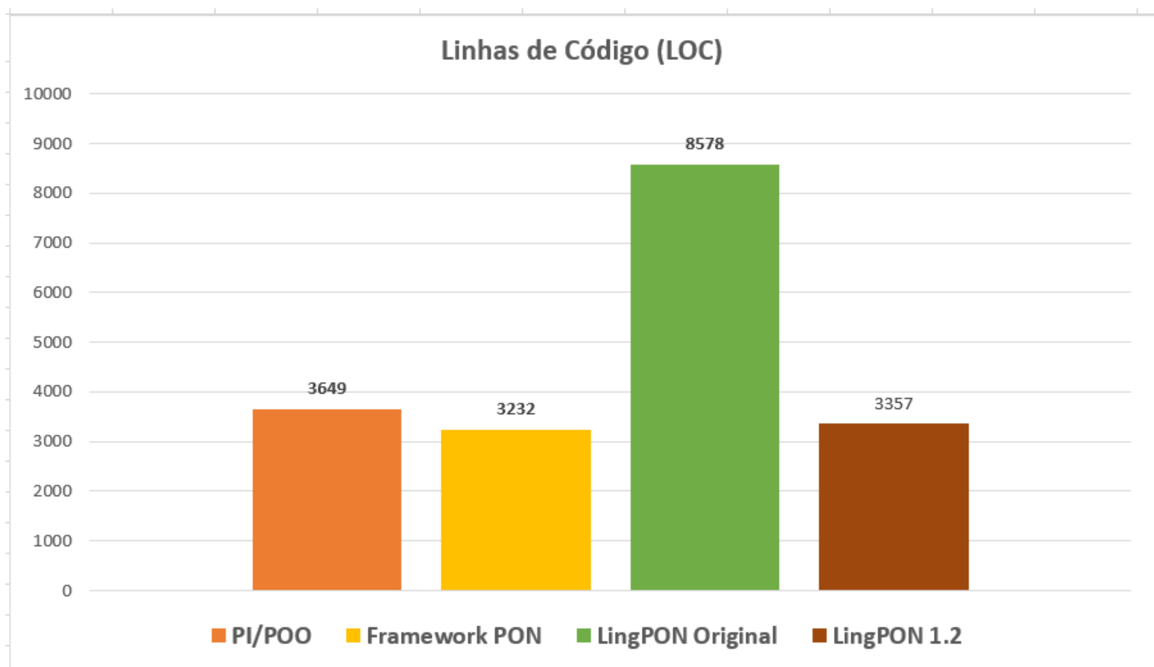

Figura 30: Gráfico linhas de código-fonte para cada uma das soluções apresentadas.

Nota-se que a solução desenvolvida utilizando a nova versão da LingPON apresenta

um número maior de linhas de código quando comparado com a solução desenvolvida a partir do *Framework PON*. Isso se deve principalmente ao fato de o *Framework* PON utilizar pseudônimos para evitar redundâncias nas declaração de entidades PON, os quais possuem a capacidade de realizar um conjunto de instruções com apenas uma única chamada, conforme apresentado em [Ronszcka 2012].

A Figura 31 apresenta os dados obtidos através da medição do número de *tokens* presentes no código-fonte para cada uma das aplicações desenvolvidas. Apesar de a solução desenvolvida utilizando a nova versão da LingPON apresentar mais linhas de código em seu código-fonte quando comparada com a solução *Framework* PON, o número de *tokens* presentes no código é ligeiramente menor.

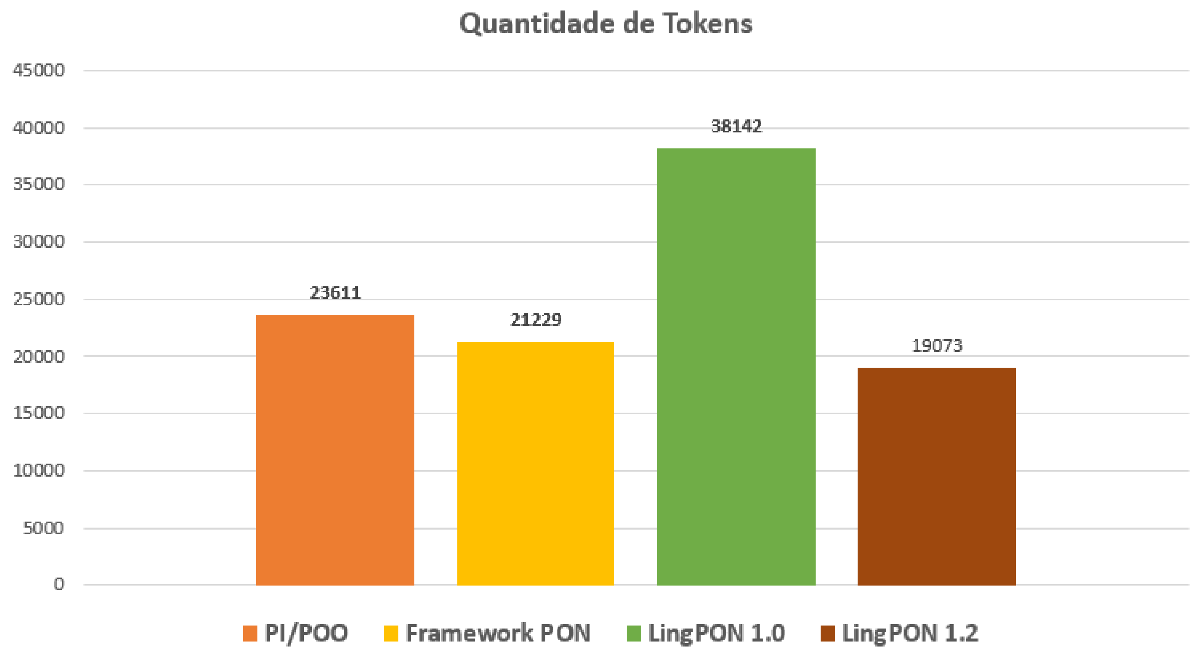

Figura 31: Gráfico quantidade de *tokens* presentes no código-fonte de cada uma das soluções apresentadas.

Ademais, ao contrário do que ocorre na aplicação desenvolvida a partir do PI/POO, observa-se através dos fragmentos de código apresentados nas subseções 3.2.3.1, 3.2.3.2 e 3.2.3.3 que as soluções desenvolvidas a partir do PON apresentam a vantagem de ter o conhecimento lógico-causal da aplicação explícito através de *Rules* e centralizados em uma única região do código-fonte. Na aplicação desenvolvida segundo o PI/POO, o conhecimento lógico-causal encontra-se disperso em diferentes classes, fato este que dificulta o entendimento da aplicação e sua manutenção, conforme será explorado na subseção 3.2.4.2.

3.2.4.2 COMPARAÇÕES DE MANUTENIBILIDADE ENTRE A APLICAÇÃO DE CONTROLE DE FUTEBOL DE ROBÔS DESENVOLVIDA EM PI/POO E PON

Essa seção apresenta o experimento realizado para comparar o nível de manutenibilidade das aplicações de controle para uma partida de futebol de robôs. Esse experimento é relativo ao objetivo específico deste trabalho que visa verificar propriedades do PON de facilidade de programação por meio de suas materializações.

As atividades de manutenção de *software* são atividades que visam modificar o código-fonte de um sistema pré existente [Souza 2005]. Neste sentido, as atividades de manutenção podem ser funcionalmente classificadas em três categorias [Arthur 1988]:

- Manutenção corretiva: Visa consertar defeitos de um sistema para que o mesmo fique em conformidade com o requisitos sobre os quais fora inicialmente desenvolvido.
- Manutenção adaptativa: Adapta o sistema de forma que o mesmo atenda novos requisitos, sejam eles relacionados a necessidades de usuários ou ambiente.
- Manutenção perfectiva: Modifica o sistema de modo a aumentar a qualidade do código-fonte do *software*, sem alterar sua funcionalidade.

Visando comparar o nível de manutenibilidade do código-fonte de cada uma das aplicações desenvolvidas neste trabalho, foi proposta uma atividade de manutenção adaptativa. Um novo requisito funcional deveria ser adicionado em cada um dos *softwares* desenvolvidos. O requisito funcional escolhido para essa comparação determina que, quando o comando recebido a partir da aplicação *Referee Box* for '*Halt*' (i.e suspender partida), dois robôs devem se deslocar até próximos a bola e permanecer efetuando troca de passes até que a partida seja reiniciada. Este requisito foi proposto apenas como forma de avaliar a manutenibilidade, não sendo uma regra real do futebol de robôs.

As métricas de comparação utilizadas foram o tempo despendido para desenvolver o código necessário para o novo requisito e a quantidade de linhas no código-fonte alteradas. O tempo utilizado para a manutenção está diretamente ligado ao custo do processo de manutenção de um *software* e, portanto, define a viabilidade da manutenção a ser realizada [Ferreira et al. 2008]. De forma a tornar a comparação justa e imparcial, o estudante de Engenharia da Computação André Botta[2], autor e desenvolvedor da solução de controle desenvolvida sob o viés do PI/POO, foi convidado a contribuir com essa

[2]Currículo Lattes: http://lattes.cnpq.br/5735332297553810

atividade. Dessa forma, uma possível curva de aprendizado para entender o código-fonte já existente não interferiria na comparação.

Segundo [Arthur 1988], qualquer manutenção adaptativa deve ser realizada a partir das seguintes atividades: identificar partes da arquitetura envolvidas, elaborar alternativas, avaliar alternativas e implementar a alternativa escolhida. Dessa forma, o mesmo processo foi aplicado para a adição do novo requisito funcional nos quatro sistemas desenvolvidos.

Após uma semana de atividades e algumas tentativas de codificar o novo requisito funcional, o estudante André Botta relatou que seria necessário alterar a arquitetura do código-fonte da solução desenvolvida em PI/POO de forma a permitir que a mesma suportasse esse novo comportamento. Isso se deve principalmente a forma com a qual a lógica que rege o comportamento dos robôs está dispersa entre diferentes classes, tais como *Strategy*, *RobotBehavior* e *GameFunctions*, conforme apresentado na subseção 3.2.2. Sendo assim, a adição de uma nova regra lógico-causal para atender o novo requisito funcional exigiria profundas alterações nestas três classes. Isto tornou-se inviável uma vez que as alterações necessárias para atender o novo requisito afetariam o funcionamento das regras já existentes no sistema.

Não satisfeito apenas com a tentativa do estudante André Botta, o autor[3] deste trabalho também tentou realizar a adição deste novo requisito funcional. Por não ter sido o autor do código-fonte e o mesmo não apresentar uma documentação técnica, um certo tempo foi investido a fim de entender o funcionamento do sistema de controle desenvolvido a partir do PI/POO. A partir do momento que o funcionamento da aplicação foi compreendido, iniciou-se os esforços para adicionar o novo requisito funcional.

Após cerca de 3 horas de trabalho, uma solução foi proposta. A solução consistia em adicionar alterações nas três classes responsáveis pela definição de qual estratégia e movimento será adotada pelo robô, a saber as classes *Strategy*, *RobotBehavior* e *GameFunctions*. Ao executar a aplicação com o objetivo de validar o novo requisito, verificou-se que seu funcionamento não estava como o esperado. Ao receber o comando '*Halt*', dois robôs deslocaram-se até próximo a bola e efetuaram a primeira troca de passes.

Entretanto, após realizar o primeiro passe, o robô desloca-se em sentido ao gol adversário. Isto se deu provavelmente à alguma regra que já estava definida previamente no sistema de controle. Todavia, por apresentar regras dispersas em diferentes classes, não foi possível localizar a regra (ou regras) que poderia estar interferindo no funcionamento

[3]Currículo Lattes: http://lattes.cnpq.br/8972298057414510

deste novo requisito.

Posteriormente, o professor João Fabro[4], especialista e professor de programação em PI/POO, que foi o orientador do estudante André Botta durante o desenvolvimento do software de controle de robôs em PI/POO apresentada neste trabalho, também foi convidado a participar do experimento. Segundo seus relatos, foram necessárias 4 horas de intensa programação para o desenvolvimento de uma possível solução. Entretanto, assim como no caso do estudante André Botta, foi relatada a impossibilidade da solução atender completamente aos requisitos.

Apesar de a construção do algoritmo para atender este novo requisito ser relativamente simples, sua construção sob o PI/POO não é trivial. Conforme relatado por prof. Fabro, a dificuldade de programar este novo comportamento está na dinamicidade necessária para que os robôs executem o passe e, na sequência, se posicionem para receber o passe de seu companheiro. Utilizando o PI/POO, o programador deve explicitar a ação que deve ser executada e quando deve ser executada, de forma sequencial. Entretanto, o futebol de robôs é um ambiente dinâmico, no qual é difícil prever a sequência em que diferentes situações ocorrerão.

Neste sentido, verificou-se que a solução desenvolvida pelo prof. Fabro aproximou-se, de certo modo, do comportamento esperado pelo novo requisito. Quando recebido o comando '*Halt*', o robô *kicker* se desloca em direção à bola. Entretanto, o passe não é executado na direção do outro robô (*partner*). Além disso, após o primeiro passe, os robôs começam a deslocar-se para posições aleatórias. Desse modo, a solução apresentada pelo prof. Fabro não foi capaz de atender o novo requisito funcional proposto.

Em contrapartida, a adição do novo requisito foi realizada com certa facilidade nas três aplicações PON. Isso se deve principalmente à forma com a qual o comportamento dos robôs é expresso através de *Rules* nas aplicações PON, facilitando a atividade de identificar as partes da arquitetura que deveriam ser alteradas para acomodar o novo requisito.

Dessa forma, o trabalho em adicionar o novo requisito ao sistema se concentrou em criar *Rules* que fossem capazes de executar a ação descrita pelo requisito. Por ser um comportamento dinâmico, que envolve tanto a percepção espacial do campo e movimentos, três novas *Rules* para cada um dos dois robôs foram criadas. Essas *Rules* tem como objetivo posicionar cada um dos robôs próximo a bola e fazer com que o robô que esteja mais próximo a bola se locomova até a mesma e efetue o passe para um companheiro.

[4]Currículo Lattes: http://lattes.cnpq.br/6841185662777161

O tempo despendido para elaborar a solução, isto é, elaborar as *Rules* que deveriam ser adicionadas ao sistema e implementá-las foi de aproximadamente uma hora. Primeiramente, essa solução foi desenvolvida no código-fonte LingPON 1.2. Posteriormente, o mesmo conjunto de *Rules* foi aplicado às soluções desenvolvidas a partir do *Framework* PON e da LingPON 1.0.

A primeira *Rule* adicionada determina que, dado a condição '*Halt*' de partida, o robô deve movimentar-se para a linha que divide os lados do campo da partida. A segunda *Rule* determina que, se o robô for o jogador que está mais próximo à bola e se não houver nenhum obstáculo entre ele e o robô que irá receber o passe, o robô deve deslocar-se em direção à bola. Por fim, a terceira *Rule* determina que o robô deve tocar a bola para seu companheiro quando estiver próximo a bola. As *Rules* aqui descritas são apresentadas em *Framework* PON e LingPON 1.2, respectivamente, nos Código 29, 30 e 31.

O mesmo conjunto de *Rules* foi aplicado à solução desenvolvida utilizando a LingPON 1.0. Entretanto, dada suas limitações, cada uma das *Rules* criadas foi relacionada a uma instância do *FBE* Robot. Portanto, foi necessário adicionar 18 novas *Rules* ao sistema.

Código 29: Código das *Premises* criadas em *Framework* PON para atender o novo requisito funcional.

```
1   PREMISE (prRoleStrickerRight, this->atRole,
2                 new String(this, "STRIKER_RIGHT"),
3                 Premise::EQUAL, Premise::STANDARD, false);
4
5   PREMISE (prRefereeCmdHalt, this->atRefereeCmd,
6                 new Char(this, 'H'),
7                 Premise::EQUAL, Premise::STANDARD, false);
8
9   PREMISE (prNotClosestToBall, this->atClosestToBall,
10                new Boolean(this, false),
11                Premise::EQUAL, Premise::STANDARD, false);
12
13  PREMISE (prClosestToBall, this->atClosestToBall,
14                new Boolean(this, true),
15                Premise::EQUAL, Premise::STANDARD, false);
16
17  PREMISE (prNotSetPassBall, this->atSetPassBall,
18                new Boolean(this, false),
19                Premise::EQUAL, Premise::STANDARD, false);
20
21  PREMISE (prSetPassBall, this->atSetPassBall,
22                new Boolean(this, true),
23                Premise::EQUAL, Premise::STANDARD, false);
24
```

Código 30: Código das *Rules* criadas em *Framework* PON para atender o novo requisito funcional.

```
1  RULE (rlSRHaltNotClose, scheduler, Condition::CONJUNCTION) ;
2  rlSRHaltNotClose->addPremise(prRoleStrickerRight) ;
3  rlSRHaltNotClose->addPremise(prRefereeCmdHalt) ;
4  rlSRHaltNotClose->addPremise(prNotClosestToBall) ;
5  rlSRHaltNotClose->addInstigation(this->mtMoveRightWarmUp);
6  RULE (rlSRHaltClose, scheduler, Condition::CONJUNCTION) ;
7  rlSRHaltClose->addPremise(prRoleStrickerRight) ;
8  rlSRHaltClose->addPremise(prRefereeCmdHalt) ;
9  rlSRHaltClose->addPremise(prClosestToBall) ;
10 rlSRHaltClose->addPremise(prNotSetPassBall) ;
11 rlSRHaltClose->addInstigation(this->mtMoveIndirectKick);
12
13 RULE (rlSRHaltCloseReady, scheduler, Condition::CONJUNCTION)
14 rlSRHaltCloseReady->addPremise(prRoleStrickerRight)
15 rlSRHaltCloseReady->addPremise(prRefereeCmdHalt)
16 rlSRHaltCloseReady->addPremise(prClosestToBall)
17 rlSRHaltCloseReady->addPremise(prSetPassBall)
18 rlSRHaltCloseReady->addInstigation(this->mtMoveIndirectKick)
```

Código 31: Código das *Rules* criadas em LingPON para atender o novo requisito funcional.

```
1  fbeRule rlSRHaltNotClose
2   condition
3       premise prStrickerRight Robot.atRole == “STRIKER_RIGHT” and
4       premise prRefereeCmdHalt Robot.atRefereeCmd == ‘H’ and
5       premise prNotClosestToBall Robot.atClosestToBall == false
6   end_condition
7   action
8     instigation inSRHaltNotClose Robot.mtMoveRightWarmUp();
9   end_action
10 end_fbeRule
11
12 fbeRule rlSRHaltClose
13  condition
14      premise prStrickerRight Robot.atRole == “STRIKER_RIGHT” and
15      premise prRefereeCmdHalt Robot.atRefereeCmd == ‘H’ and
16      premise prNotClosestToBall Robot.atClosestToBall == true and
17      premise prFreePartner Robot.atPartnerFreeID >=0 and
18      premise prNotSetPassBall Robot.atSetPassBall == false and
19  end_condition
20  action
21    instigation inSRHaltClose Robot.mtMoveIndirectKick();
22  end_action
23 end_fbeRule
24
25 fbeRule rlSRHaltCloseReady
26  condition
27      premise prStrickerRight Robot.atRole == “STRIKER_RIGHT” and
28      premise prRefereeCmdHalt Robot.atRefereeCmd == ‘H’ and
29      premise prNotClosestToBall Robot.atClosestToBall == true and
30      premise prFreePartner Robot.atPartnerFreeID >=0 and
31      premise prNotSetPassBall Robot.atSetPassBall == true
32  end_condition
33  action
34    instigation inSRHaltCloseReady Robot.mtPassBallPartner();
35  end_action
36 end_fbeRule
```

Sendo assim, 80 novas linhas de código foram adicionadas ao código-fonte LingPON 1.2, 34 novas linhas ao código-fonte *Framework* PON e 534 novas linhas na solução desenvolvida utilizando a LingPON 1.0. Novamente, o menor número de linhas de código necessárias para a solução no código-fonte da solução desenvolvida a partir do *Framework* PON se dá devido a utilização de pseudônimos para evitar redundâncias nas declarações de entidades PON, conforme apresentado em [Ronszcka 2012]. Ademais, a LingPON, em sua origem, foi projetada para ser mais verbosa com o objetivo de ser mais facilmente entendida.

3.2.5 REFLEXÃO SOBRE AS COMPARAÇÕES

Como reflexões finais desta seção de comparações, é pertinente ressaltar que as aplicações foram comparadas a partir da solução de um problema comum, a saber, um sistema de controle para partida de futebol de robôs.

Em um primeiro momento, foi comparada a complexidade de código-fonte entre as quatro soluções apresentadas, utilizando como critérios a quantidade de linhas de código e número de *tokens*. Ao comparar as aplicações desenvolvidas utilizando a LingPON, é possível observar reais avanços da LingPON em sua nova versão. O código-fonte desenvolvido utilizando a nova versão da LingPON (1.2) apresentou 60.87% menos linhas de código e 50% menos *tokens* do que na versão 1.0.

Em termos de linhas de código e *tokens* presentes no código-fonte, as soluções desenvolvidas sob o viés do PON, utilizando o *Framework* PON e a LingPON 1.2, apresentaram resultados melhores em relação aos obtidos a partir da aplicação PI/POO. Isso indica que o PON apresenta uma sintaxe tão concisa quanto a encontrada na linguagem C++, utilizada no desenvolvimento da aplicação PI/POO.

Entre as soluções PON, a aplicação desenvolvida utilizando a LingPON 1.2 apresentou maior quantidade de linhas de código e menos *tokens* em seu código-fonte. Conforme apresentado, o *Framework* PON utiliza pseudônimos criados através da linguagem de programação C++ para reduzir o número de instruções utilizadas na criação de entidades PON, conforme apresentado em [Ronszcka 2012]. Neste sentido, acredita-se que a linguagem de programação PON pode ser ainda evoluída no sentido de tornar as declarações mais concisas, semelhante ao que foi feito no *Framework* PON com a utilização de pseudônimos.

Com isso, a LingPON apresentaria uma capacidade de representação ainda maior.

Entretanto, as métricas de complexidade de código não estão necessariamente relacionadas a facilidade de programação inerente a cada um dos paradigmas de programação e suas linguagens de programação. Ao desenvolver exatamente a mesma classe, utilizando duas linguagens de programação diferentes (*i.e* C++ e Java), a quantidade de linhas de código e *tokens* presentes em cada uma das soluções será certamente diferente, uma vez que a sintaxe utilizada na construção da classe é diferente para cada linguagem de programação. Entretanto, por se tratar da mesma classe, com métodos e atributos idênticos, a complexidade em ambas será a mesma. Isso justifica o fato de tais métricas não apresentarem significado completo quando utilizadas de forma independente.

Por esse motivo, decidiu-se comparar o nível de manutenibilidade entre as soluções de forma prática, isto é, através da adição de um novo requisito funcional ao sistema. O requisito funcional foi escolhido de forma a não favorecer nenhuma das soluções. Neste âmbito, visando apresentar uma comparação imparcial, isto é, sem favorecimento do objeto de estudo (PON), o autor da aplicação desenvolvida a partir do PI/POO foi convidado a participar do experimento.

Esse experimento destacou a dificuldade em adicionar um novo requisito funcional ao sistema desenvolvido segundo o PI/POO, o qual acabou por não ser adicionado devido à constatação de que tal alteração resultaria em alterações em diversas classes da aplicação. Por outro lado, a adição de um novo requisito funcional nos sistemas desenvolvidos sob o viés do PON se deu de maneira direta, através da adição de algumas *Rules* que fizeram o sistema atender o novo requisito funcional.

Ao comparar a dificuldade em adicionar um novo requisito nas aplicações desenvolvidas a partir da LingPON é possível observar que a LingPON 1.0 necessita de muito mais linhas de código para resolver o mesmo problema quando comparada a LingPON 1.2. Ao utilizar o conceito de *FBE Rule*, introduzido na nova versão da LingPON, foi possível solucionar o problema com apenas 80 novas linhas de código contra as 534 necessárias na atual versão da LingPON. Isso representa uma economia de aproximadamente 85% de linhas de código e tem impacto direto na velocidade de desenvolvimento e do nível de manutenibilidade da aplicação, uma vez que é mais fácil manter 80 linhas de código ao invés de 534.

A Tabela 2 apresenta de forma resumida os resultados obtidos a partir dos dois experimentos apresentados neste capítulo.

Tabela 2: Resultados obtidos a partir dos experimentos de contabilidade de linhas de código e quantidade de *tokens* presentes no código fonte e nível de manutenibilidade.

Aplicação	**Nº de linhas**	**Nº de tokens**	**Foi possível adicionar o novo requisito?**
PI/POO= C++	3649	23611	Não
FW PON C++ 2.0	3232	21229	Sim
LingPON 1.0	8578	38142	Sim
LingPON 1.2	3357	19073	Sim

4 CONCLUSÃO E TRABALHOS FUTUROS

Este capítulo apresenta as conclusões deste trabalho e indica possíveis trabalhos futuros. Desta forma, a seção 4.1 apresenta a conclusão dessa dissertação de mestrado, relacionando as contribuições da mesma. Por fim, a seção 4.2 vislumbra trabalhos futuros que poderão contribuir para a consolidação da LingPON e, consequentemente, do Paradigma Orientado a Notificações (PON).

4.1 CONCLUSÃO

Este trabalho teve como um dos objetivos a evolução da linguagem de programação LingPON e seu respectivo compilador (*i.e.* da tecnologia LingPON) no tocante a facilidade de programação, visando facilitar a criação de aplicações PON. Ademais, este trabalho teve também como objetivo a elaboração de uma aplicação PON de complexidade reconhecida (futebol de robôs), utilizando diferentes materializações (*i.e.* implementações) em *software* do PON, a serem comparadas com a mesma aplicação desenvolvida sob o viés do PI por outrem.

Inicialmente, a questão de paradigmas de programação foi apresentada de maneira sucinta, a luz de trabalhos anteriores desenvolvidos pelo grupo de pesquisa PON da UTFPR. Subsequentemente, tanto o PON quanto as suas duas principais materializações em *software* (então vigentes), nomeadamente *Framework* PON 2.0 e LingPON 1.0, foram detalhadas. Posteriormente, foram apresentadas algumas limitações da atual versão da LingPON (versão 1.0), as quais tornam o desenvolvimento de aplicações PON lento e complicado.

Visando solucionar tais limitações e permitir a criação de aplicações PON com menor esforço, foram propostos dois avanços à LingPON. O primeiro avanço proposto permite maior encapsulamento do âmbito de *FBEs* por meio de declaração de *Attributes* de tipos não primitivos, isto é, *Attributes* que sejam definidos por outros *FBEs*. O segundo avanço proposto permite ao desenvolvedor declarar *Rules* no âmbito de *FBE*, garantindo

assim que todas as instâncias de um dado *FBE* possuam um determinado conjunto de *Rules*. Portanto, os avanços são à luz de relacionamentos de agregação dado que nesta nova versão da tecnologia LingPON *FBEs* podem agregar outros *FBEs* e/ou *Rules*.

Inclusive para validar a evolução da LingPON apresentada neste trabalho, foi proposto o desenvolvimento de aplicações de controle para uma partida de futebol de robôs, segundo características da categoria SSL da *Robocup*. Na verdade, tais aplicações foram desenvolvidas utilizando o *Framework* PON 2.0, a LingPON 1.0 e a nova versão da LingPON (chamada de LingPON 1.2). Como apresentado na subseção 2.3.2.2 (e mesmo no Apêndice D), nenhuma aplicação com tamanha complexidade, em termos de número de *Rules* (que demandassem reflexão[1]) e instâncias de *FBEs*, foi desenvolvida até o presente momento utilizando a LingPON. Dessa forma, a aplicação de controle para uma partida de futebol de robôs, desenvolvida e apresentada neste trabalho, representa um grande avanço para a tecnologia LingPON, contribuindo para seu avanço como linguagem de programação. Por consequência, contribui também com o PON em si, não só pela complexidade da aplicação, mas também por haver comparações com aplicação funcionalmente equivalente desenvolvida em POO por outrem.

De fato, as soluções desenvolvidas para aplicação de controle de futebol de robôs LingPON 1.0, LingPON 1.2 e *Framework* PON 2.0 foram submetidas a comparações com uma aplicação funcionalmente equivalente, construída por outro desenvolvedor, sobre o PI POO C++. Este desenvolvimento em PI, feito por outrem, foi realizado antes da elaboração e dos esforços dessa presente dissertação [Botta 2012]. Em tempo, tais comparações se deram segundo indicadores de complexidade de código, especificamente linhas de código e quantidade de *tokens* presentes no código-fonte, e nível de manutenibilidade. O número de linhas de código presente no código-fonte de cada uma das aplicações apresentadas foi mensurado utilizando a ferramenta *cloc* [Danial 2006]. Para calcular o número de *tokens* presente no código-fonte em cada uma das aplicações apresentadas foi desenvolvido um analisador léxico a partir da ferramenta *flex*. O nível de manutenibilidade, por sua vez, foi medido através do tempo necessário para que um programador pudesse adicionar um requisito funcional comum às aplicações PON e PI.

No tocante a complexidade de código-fonte, foi possível observar que a nova versão da LingPON (versão 1.2), apresentada neste trabalho, permitiu a criação de uma aplicação funcionalmente idêntica a desenvolvida a partir da atual versão da LingPON (versão 1.0) mas utilizando 61% menos linhas de código e 50% menos *tokens*. Isto influencia

[1] *Rules* que não fossem nas *Premises* apenas combinação simples de avaliação de instâncias de *FBEs*.

diretamente na velocidade de desenvolvimento de aplicações PON através da LingPON e fez com que o código-fonte LingPON apresentasse indicadores de complexidade de código muito próximos aos apresentados pela solução em PI e a solução desenvolvida utilizando o *Framework* PON 2.0.

Através do experimento de nível de manutenibilidade, foi possível observar que o PON apresenta indicadores de maior expressividade em programação quando comparado ao PI, pois apresenta de forma mais explícita e coesa o conjunto de regras lógico-causais que regem o comportamento do sistema. Isto se deve principalmente pela forma na qual o conhecimento lógico-causal da aplicação PON é representado. Em PON, isso se dá na forma de regras explícitas coesas em base de regras, as quais, entretanto, são tratadas por entidades orientadas a notificação chamadas *Rules* gerando o devido desacoplamento. Essa característica contribui para um fácil entendimento do código-fonte e consequente aumento do nível de manutenibilidade. De fato, pelo experimento, percebeu-se que alterar e/ou adicionar requisitos se torna mais fácil nas aplicações PON quando comparadas a aplicação PI.

Conforme os resultados apresentados, este trabalho contribuiu para a redução da complexidade de código em aplicações desenvolvidas utilizando a LingPON. Ao utilizar os novos conceitos apresentados neste trabalho, em suma *FBEs* que podem agregar outros *FBEs* e/ou *Rules*, é possível desenvolver aplicações PON com código-fonte mais enxuto. Desse modo, isso corrobora para o desenvolvedor encontrar ainda maior facilidade em adicionar ou alterar as entidades PON presentes no código-fonte. Isto foi demonstrado através do experimento de nível de manutenibilidade, no qual foi possível observar ser necessário utilizar 85% menos linhas de código na LingPON 1.2, quando comparada a LingPON 1.0, para solucionar o problema proposto.

Tudo isto considerado, é possível concluir que os avanços relativos a LingPON apresentados neste trabalho mostram-se promissores e certamente contribuem no caminho para ajudar a consolidar o PON e sua respectiva linguagem de programação como alternativa para o desenvolvimento de *software*. Utilizando a versão 1.2 da LingPON, proposta, implementada e avaliada neste trabalho, foi possível desenvolver um *software* complexo, com maior número de *Rules* (com conhecimento não 'repetitivo'), instâncias de *FBEs* e afins do que as aplicações que haviam sido desenvolvidas anteriormente. Ademais, a nova versão da LingPON oferece maior concisão e facilidade de programação ao desenvolvedor que deseja utilizar a LingPON, permitindo assim maior manutenibilidade e simplicidade de programação, tanto em relação à versão 1.0 da LingPON quanto até mesmo com relação

à uma implementação em PI/POO.

Uma vez apresentada as conclusões deste trabalho nesta presente sub-seção, na próxima são apresentados possíveis trabalhos futuros, os quais podem contribuir para a evolução da tecnologia LingPON.

4.2 TRABALHOS FUTUROS

Apesar de bem apresentar características que facilitam o desenvolvimento e principalmente a manutenibilidade de *softwares*, o desenvolvimento de aplicações LingPON é ainda comprometido por algumas limitações remanescentes de sua materialização.

Neste sentido, esta seção apresenta perspectivas de pesquisas que possam contribuir para avanços do estado da técnica do PON.

4.2.1 SUPORTE A MÚLTIPLOS ARQUIVOS DE CÓDIGO-FONTE

Atualmente, o compilador PON suporta apenas a compilação de um único arquivo de código-fonte por vez. Dessa forma, o desenvolvedor deve escrever o código necessário para sua aplicação em um único arquivo código-fonte. Com isso, o arquivo de código-fonte tende a ser demasiadamente extenso.

Uma possível solução para tal problema seria alterar o compilador de forma que o mesmo permitisse referenciar arquivos externos. Dessa forma, seria possível separar a declaração de diferentes entidades PON em diferentes arquivos, tais como um arquivo para cada *FBE*. Isso contribuiria para a redução da extensão dos arquivos de código fonte e, consequentemente, melhor organização.

4.2.2 UTILIZAÇÃO DE BIBLIOTECAS EXTERNAS

Por se tratar de uma linguagem de programação recém criada, a LingPON ainda não suporta algumas funcionalidades acessórias para o desenvolvimento de certas aplicações, como comunicação via *socket* e interação com interfaces de usuário. Para contornar essa limitação, o desenvolvedor deve utilizar mecanismos não convencionais, tais como alterar o código-fonte gerado pelo compilador PON de forma a adicionar as funcionalidades necessárias em sua aplicação.

Essa limitação da LingPON foi um dos fatores críticos que influenciaram no desenvolvimento das aplicações apresentadas no estudo de caso. Por não permitir importar

e utilizar bibliotecas externas, o código-alvo gerado pelo compilador PON a partir do código-fonte LingPON teve de ser manualmente alterado para que funcionasse corretamente.

Para solucionar essa limitação, uma possível solução seria permitir ao desenvolvedor referenciar e executar métodos de bibliotecas externas utilizando a própria sintaxe da LingPON. Isto seria algo semelhante ao que acontece na programação utilizando a linguagem de programação C++.

4.2.3 SIMPLIFICAÇÃO DA SINTAXE DA LINGPON

Em sua origem, a LingPON foi concebida com o objetivo de ser didática em termos de entidades PON. Por esse motivo, ao desenvolver uma aplicação utilizando a LingPON é notório a percepção, por parte do desenvolvedor, de redundância de *keywords* ao se declarar as entidades PON, principalmente *Rules*. Isso se deve a necessidade em ter que declarar seções de *conditions*, *subConditions*, *premises*, *actions* e *instigations* para cada *Rule*.

Essa ideia segue no mesmo sentido dos avanços apresentados por linguagens de programação consolidadas na indústria de *software*, tais como C++ e Java, as quais buscam permitir a construção de código-fonte com o menor número de instruções e linhas de código possíveis. Portanto, um possível avanço para a LingPON seria a redefinição de sua sintaxe de forma a reduzir o número de instruções e *keywords* necessárias para a construção das entidades PON, algo semelhante ao que foi feito no *Framework* PON 2.0 com a utilização de *pseudônimos* [Ronszcka 2012].

Neste âmbito, muito recentemente os discentes da disciplina de Linguagens e Compiladores de 2016 do PPGCA/UTPFR[2] criaram uma nova versão do LingPON, ainda de todo prototipal, chamada (temporariamente ao menos) de *LigPON_NameSpace*. Nesta versão, em temos de linguagens, já se pode obter alguma redução ou simplicidade do escrever do código, conforme pode ser observado no Anexo E [3]. Estes esforços podem ser reaproveitados nos avanços subsequentes da LingPON.

4.2.4 TESTE DE UNIDADE PARA O COMPILADOR

Por se tratar de um projeto em equipe, no qual diversos pesquisadores estão trabalhando simultaneamente e o utilizando em suas pesquisas, faz-se necessário garantir

[2]Disciplina "Linguagens e Compiladores" ofertada pelo PPGCA da UTFPR em 2016, ministrada por Prof. Dr. João Alberto Fabro e Prof. Dr. Jean Marcelo Simão.

[3]O relatório técnico apresentado no Anexo E foi entregue em Março de 2017.

que o compilador PON esteja sempre funcionado de acordo com seus requisitos.

Uma forma simples e fácil de garantir essa condição é a utilização correta de testes automatizados que visam verificar se todos os requisitos continuam funcionando corretamente após cada alteração no código-fonte. Isso pode ser alcançado através de testes de unidade (*unit tests*), os quais testam se todos os métodos envolvidos no processo de compilação estão funcionando corretamente.

Caso uma alteração mude o comportamento esperado do compilador, isso pode ser identificado de forma mais rápida, sem afetar os demais usuários do compilador.

4.2.5 MELHORAR EXPERIMENTO DE NÍVEL DE MANUTENIBILIDADE

Este trabalho apresentou uma comparação de nível de manutenibilidade entre aplicações PON, desenvolvidas utilizando o *Framework* PON 2.0, a LingPON 1.0 e a LingPON 1.2, e uma aplicação desenvolvida em linguagem C++ POO/PI.

Por se tratar de uma aplicação complexa e voltada a uma situação muito específica (partida de futebol de robôs), essa comparação foi realizada com a participação de apenas três desenvolvedores, a saber o estudante de Engenharia da Computação André Botta, o autor deste trabalho e o prof. Dr. João Fabro.

Essa comparação foi útil para o desenvolvimento deste trabalho e permitiu observar que o PON apresenta indicadores de um código-fonte mais expressivo, legível e com maior nível de manutenibilidade.

Entretanto, de forma a validar essa observação, seria interessante repetir tal experimento utilizando outras aplicações e envolvendo mais pessoas, de forma a melhorar a confiabilidade dos resultados.

4.2.6 ESTUDO DE AGENTES APLICADOS AO PON

Em [Simão 2001] foi apresentada uma arquitetura de controle dinâmico e a eventos discretos de sistemas flexíveis de manufatura (FMS), a qual era baseada em regras e agentes. Em suma essa arquitetura acabou por se tornar o chamado Controle Orientado a Notificações (CON). Nos anos subsequentes, essa arquitetura de CON evoluiu até se tornar um paradigma de programação, nomeadamente o PON.. Portanto, o PON encontrou inspiração em conceitos oriundos da programação orientada a agentes, particularmente nos agentes reativos [Simão et al. 2001, Simão et al. 2001, Simão e Stadzisz 2002, Simão et

al. 2003].

A ideia geral de Programação Orientada a Agentes (AOP) não é algo novo em si. O primeiro trabalho a trazer esse conceito foi publicado em 1993 [Shoham 1993]. O objetivo da Programação Orientada a Agentes introduzida por Shoham foi a apresentação de um novo nível de abstração pós orientação a objetos, fornecendo recursos de mais alto nível [Ricci e Santi 2011].

Entretanto, nota-se que a programação orientada a agentes ainda não teve impacto significativo nas pesquisas em linguagens de programação e desenvolvimento de *software* até hoje [Ricci e Santi 2011]. Isto se deve possivelmente ao fato de que os maiores esforços ainda se concentraram em questões teóricas relacionadas a agentes, ao invés de focar em sua aplicabilidade para a computação prática [Ricci e Santi 2011] [4].

Outrossim, a aplicação de Futebol de Robôs também atrai a comunidade que trabalha no assunto de agentes. Em tempo, há também aplicações outras de agentes em outros domínios, como em aplicações de cunho industrial [Banaszewski 2009]. Enfim, mesmo que haja ainda estudos de cunho teórico ou acadêmico no tocante a agentes, salientando aqui a Programação Orientada a Agentes (POA), haveria uma tendência natural a se estudar aplicação de agentes em situação cada vez mais reais.

Neste sentido, um trabalho futuro seria pesquisar quais seriam as sinergias entre PON e as temáticas de agentes, sublinhando LingPON e AOP. Talvez agentes e POA permitam aprimorar o PON/LingPON ou reciprocamente o PON/LingPON possa ser um viabilizador teórico-tecnológico do POA e afins.

[4]Enquanto material de suporte, o Apêndice E traz um resumo sobre Programação Orientada a Agentes e afins

REFERÊNCIAS

AHO, A. V.; SETHI, R.; ULLMAN, J. D. Compiladores: Princıpios, técnicas e ferramentas. **LTC, Rio de Janeiro, Brasil**, 1995.

ARTHUR, L. J. **Software evolution: the software maintenance challenge**. [S.l.]: Wiley-Interscience, 1988.

ASADA, M.; ICHINODA, S.; HOSODA, K. Action-based sensor space segmentation for soccer robot learning. **Applied Artificial Intelligence**, Taylor & Francis, v. 12, n. 2-3, p. 149–164, 1998.

ASADA, M. et al. Robocup: Today and tomorrow. **Experimental Robotics VI**, Springer Science & Business Media, v. 250, p. 369, 1999.

BANASZEWSKI, R. F. Paradigma orientado a notificações: avanços e comparações. 2009. Master in Science Thesis, Graduate School in Electrical Engineering and Industrial Computer Science (CPGEI) at the Federal University of Technology - Paraná (UTFPR). Curitiba, Brazil. Disponível em: `http://files.dirppg.ct.utfpr.edu.br/cpgei/Ano_2009/dissertacoes/Dissertacao_500_2009.pdf`.

BANASZEWSKI, R. F. Proposta e implementaÇÃo de um modelo cognitivo para agentes de software baseado no paradigma orientado a notificaÇÕes. 2009. Plano de Doutorado. Curitiba - PR Brasil: CPGEI/UTFPR.

BATISTA, M. V. Proposta de um método de aplicação da teoria de projeto axiomático ao desenvolvimento de software pon-por. Curitiba, 2013. Dissertação (Mestrado em Engenharia Elétrica e Informática Industrial) - Universidade Tecnológica Federal do Paraná. Disponível em: `http://repositorio.utfpr.edu.br/jspui/handle/1/613`.

BELMONTE, D.; SIMAO, J. M.; STADZISZ, P. C. Proposta de um método para distribuição de carga de trabalho usando o paradigma orientado a notificações (pon). **Revista SODEBRAS**, v. 8, n. 84, 2012.

BELMONTE, D. L. et al. A new method for dynamic balancing of workload and scalability in multicore systems. **IEEE Latin America Transactions**, IEEE, v. 14, n. 7, p. 3335–3344, 2016.

BOTTA, A. L. C. Modelos artificiais integrados ao software de controle do time de futebol de robôs da utfpr. In: **Sicite 2012**. [S.l.: s.n.], 2012.

BROOKSHEAR, J. G. **Computer science: an overview**. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 2002.

BROOKSHEAR, J. G. **Computer science: an overview**. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 2012.

DANIAL, A. **CLOC - Count Lines of Code**. 2006. Disponível em: <http://cloc.sourceforge.net/>.

DÖLLNER, J. et al. Illustrative visualization of 3d city models. In: INTERNATIONAL SOCIETY FOR OPTICS AND PHOTONICS. **Electronic Imaging 2005**. [S.l.], 2005. p. 42–51.

EUGSTER, P. T. et al. The many faces of publish/subscribe. **ACM computing surveys (CSUR)**, ACM, v. 35, n. 2, p. 114–131, 2003.

FAISON, T. **Event-Based Programming**. [S.l.]: Springer, 2006.

FERG, S. Event-driven programming: introduction, tutorial, history. 2006. Disponível em: `http://sourceforge.net/projects/eventdrivenpgm/files/event_driven_programming.pdf` Acessado em 03/02/2017.

FERREIRA, C. A. Linguagem e compilador para o paradigma orientado a notificações (pon): avanços e comparações. 2016. Dissertação de Mestrado, PPGCA UTFPR. Disponível em: `http://www.utfpr.edu.br/curitiba/estrutura-universitaria/diretorias/dirppg/programas/ppgca/edital-de-defesas/2015/ppgca-mestrado-cleverson-avelino-ferreira`.

FERREIRA, C. A. et al. Compilador para o paradigma orientado a notificações. 2013.

FERREIRA, K. A. M.; BIGONHA, M. A.; BIGONHA, R. S. Reestruturação de software dirigida por conectividade para redução de custo de manutenção. **Revista de Informática Teórica e Aplicada**, v. 15, n. 2, p. 155–180, 2008.

FORGY, C. L. Rete: A fast algorithm for the many pattern/many object pattern match problem. **Artificial intelligence**, Elsevier, v. 19, n. 1, p. 17–37, 1982.

GABBRIELLI, M.; MARTINI, S. **Programming languages: principles and paradigms**. [S.l.]: Springer Science & Business Media, 2010.

GENESERETH, M. R.; KETCHPEL, S. P. Software agents. **Commun. ACM**, v. 37, n. 7, p. 48–53, 1994.

GREGORI, R. H. et al. Analysis of a triangle mesh slicing algorithm under the notification oriented and imperative paradigms. 2012. Framework NOP/PON C++ 2.0, Mestrando PPGCA/UTFPR, 2012. Disciplina sobre Paradigma Orientado a Notificações (PON), CPGEI-PPGCA/UTFPR (Prof. J. M. Simão), Curitiba - PR, Brasil.

GRUNE, D. et al. **Modern compiler design**. [S.l.]: Springer Science & Business Media, 2012.

HANSEN, S.; FOSSUM, T. Event based programming. 2010. Disponível em: `http://www.cs.uwp.edu/staff/hansen/EventsWWW/`.

JASINSKI, R. P. Framework para geração de hardware em vhdl a partir de modelos em pon (paradigma orientado a notificações. 2012. Relatório da disciplina de Lógica Reconfigurável por Hardware. Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial. Universidade Tecnológica Federal do Paraná.

KAISLER, S. H. **Software paradigms**. [S.l.]: John Wiley & Sons, 2005.

KERSCHBAUMER, R. et al. Paradigma orientado a notificações para a síntese de lógica reconfigurável. 2015. LA-CCI/CBIC. ISBN: 9788569972006. Disponível em: `https://www.researchgate.net/publication/283018125_Paradigma_Orientado_a_Notificaes_para_a_Sntese_de_Lgica_Reconfigurvel`.

KOSSOSKI, C. Jogo em 2d desenvolvido em c++ pon e allegro. 2013. Disciplina sobre Paradigma Orientado a Notificações (PON), CPGEI-PPGCA/UTFPR (Prof. J. M. Simão), Curitiba - PR, Brasil.

KOSSOSKI, C.; STADZISZ, P. C.; M., S. J. Introdução ao teste funcional de software no paradigma orientado a notificações. In: **VI Congresso Intern. de Computación y Telecom.-COMTEL, Lima, Peru**. [S.l.: s.n.], 2014.

KRACHINSKI, V. et al. Analysis of a triangle mesh slicing algorithm under the notification oriented and imperative paradigms. 2015. Ling-PON-pilador 0.7, Mestrando PPGCA/UTFPR, 2015. Disciplina sobre Paradigma Orientado a Notificações (PON), CPGEI-PPGCA/UTFPR (Prof. J. M. Simão), Curitiba - PR, Brasil.

KRUG, D. L. Torre de hanÓi com lingpon – paradigma orientado a notificações. 2016. Aplicação em Ling PON 1.0/1.5. IFPR (Professor) – Mestrando PPGCA/UTFPR, 2016. Disciplina sobre Paradigma Orientado a Notificações (PON), CPGEI-PPGCA/UTFPR (Prof. J. M. Simão e Prof. H. Panetto [visitante CPGEI e UL-França]), Curitiba - PR, Brasil.

LEE, P.-Y.; CHENG, A. M. K. Hal: A faster match algorithm. **IEEE Transactions on Knowledge and Data Engineering**, IEEE, v. 14, n. 5, p. 1047–1058, 2002.

LIBALLEG. **Allegro: A game programming library**. 2004. Disponível em: <http://liballeg.org/index.html/>.

LINHARES, R. R. **Contribuição para o desenvolvimento de uma arquitetura de computação própria ao paradigma orientado a notificações**. Tese (Doutorado) — Graduate School in Electrical Engineering and Industrial Computer Science (CPGEI) at the Federal University of Technology - Paraná (UTFPR). Curitiba, Brazil., 2015. Disponível em: `http://repositorio.utfpr.edu.br/jspui/handle/1/1324`.

LINHARES, R. R. et al. Comparações entre o paradigma orientado a objetos e o paradigma orientado a notificações sob o contexto de um simulador de sistema telefônico. In: **III Congresso Intern. de Computación y Telecom.-COMTEL, Lima, Peru**. [S.l.: s.n.], 2011.

LINHARES, R. R.; SIMAO, J. M.; STADZISZ, P. C. Noca - a notification-oriented computer architecture. **IEEE Latin America Transactions**, IEEE, v. 13, n. 5, p. 1593–1604, 2015.

MARLING, C. et al. Case-based reasoning for planning and world modeling in the robocup small size league. In: **IJCAI Workshop on issues in designing physical agents for dynamic real-time environments**. [S.l.: s.n.], 2003. p. 1–2.

MEDONçA, I. T. M. et al. Método para desenvolvimento de sistemas orientados a regras utilizando o paradigma orientado a notificações. 2015. LA-CCI/CBIC, 2015. ISBN: 9788569972006. Disponível em: `https://www.researchgate.net/profile/Igor_Mendonca/publication/282818387_Mtodo_para_Desenvolvimento_de_Sistemas_Orientados_a_Regras_utilizando_o_Paradigma_Orientado_a_Notificaes`".

MELO, L. C. V.; FABRO, J. A.; SIMãO, J. M. RelatÓrio da adaptaÇÃo do paradigma orientado a notificaÇÕes - pon para suporte a desenvolvimento de sistemas de lÓgica fuzzy. 2013. Framework NOP/PON 2.0 C++ (adaptado para fuzzy), Mestrando CPGEI/UTFPR, 2013. Disciplina sobre Paradigma Orientado a Notificações (PON), CPGEI-PPGCA/UTFPR (Prof. J. M. Simão), Curitiba - PR, Brasil.

MENDONçA, I. T. M. Metodologia de projeto de software orientado a notificações. 2016. Qualificação de Doutorado, Programa de Pós-Graduação em Engenharia Elétrica e Informática Industria da Universidade Tecnológica Federal do Paraná (UTFPR). Curitiba – Paraná (PR), Brazil.

MIRANDA, F. S. Um estudo comparativo entre o paradigma orientado a notificações (pon) e o paradigma orientado a objetos (poo) em um problema de uma cidade virtual. 2016. Disciplina sobre Paradigma Orientado a Notificações (PON), CPGEI-PPGCA/UTFPR (Prof. J. M. Simão e Prof. H. Panetto [visitante CPGEI e UL-França]), Curitiba - PR, Brasil.

MIRANKER, D. P. Treat: A better match algorithm for ai production systems. **Sixth National Conference on Artificial Intelligence - AAAI'87**, p. 42–47, 1987.

MIRANKER, D. P. et al. On the performance of lazy matching in production systems. In: **AAAI**. [S.l.: s.n.], 1990. v. 90, p. 685–692.

MIRANKER, D. P.; LOFASO, B. J. The organization and performance of a treat-based production system compiler. **IEEE Transactions on Knowledge and Data Engineering**, IEEE, v. 3, n. 1, p. 3–10, 1991.

MONAJJEMI, V.; KOOCHAKZADEH, A.; GHIDARY, S. S. grsim–robocup small size robot soccer simulator. In: **RoboCup 2011: Robot Soccer World Cup XV**. [S.l.]: Springer, 2011. p. 450–460.

MONTE-ALTO, H. H. L. C. Desenvolvendo o jogo pac-man com o paradigma orientado a notificações. 2015. Disciplina sobre Paradigma Orientado a Notificações (PON), CPGEI-PPGCA/UTFPR (Prof. J. M. Simão), Curitiba - PR, Brasil.

NWANA, H. S. Software agents: An overview. **The knowledge engineering review**, Cambridge Univ Press, v. 11, n. 03, p. 205–244, 1996.

PANESCU, D.; PASCAL, C.; OLAERU, R. M. A rule-based approach for a multi-robot application. In: IEEE. **System Theory, Control and Computing (ICSTCC), 2015 19th International Conference on**. [S.l.], 2015. p. 75–80.

PETERS, E. Coprocessador para aceleração de aplicações desenvolvidas utilizando paradigma orientado a notificações. 2012. Master in Science Thesis, Graduate School in

Electrical Engineering and Industrial Computer Science (CPGEI) at the Federal University of Technology Paraná (UTFPR). Curitiba – Paraná (PR), Brazil. Disponível em: http://repositorio.utfpr.edu.br/jspui/handle/1/325.

PETERS, E. et al. A new hardware coprocessor for accelerating notification-oriented applications. In: IEEE. **Field-Programmable Technology (FPT), 2012 International Conference on**. [S.l.], 2012. p. 257–260.

POO, D.; KIONG, D.; ASHOK, S. **Object-oriented programming and Java**. [S.l.]: Springer Science & Business Media, 2007.

PORDEUS, L. F. Notification oriented paradigm (nop): Cta simulator. 2015. Disciplina sobre Paradigma Orientado a Notificações (PON), CPGEI-PPGCA/UTFPR (Prof. J. M. Simão), Curitiba - PR, Brasil.

PORDEUS, L. F. Contribuição para avaliação e melhoria de uma arquitetura de computação própria ao paradigma orientado a notificações. 2016. Qualificação de Mestrado. CPGEI/UTFPR, Curitiba - PR, Brasil, 01/Julho 2016.

PORDEUS, L. F. et al. Trabalho e manual lingpon versão 2015. 2015. Relatório da disciplina 'Linguagens e Compiladores', Programa de Pós-Graduação em Engenharia Elétrica e Informática Industria da Universidade Tecnológica Federal do Paraná (UTFPR). Curitiba – Paraná (PR), Brazil.

PRESSMAN, R.; MAXIM, B. **Engenharia de Software-8ª Edição**. [S.l.]: McGraw Hill Brasil, 2016.

RESNICK, M. **Turtles, termites, and traffic jams: Explorations in massively parallel microworlds**. [S.l.]: Mit Press, 1997.

RICCI, A.; SANTI, A. Agent-oriented computing: Agents as a paradigm for computer programming and software development. In: CITESEER. **Proc. of the 3rd Int'l Conf. on Future Computational Technologies and Applications. Wilmington: Xpert Publishing Services**. [S.l.], 2011. p. 42–51.

RILEY, G.; GIARRATANO, J. Expert systems principles and practice. **Massachusetts, PWS**, 1993.

RONSZCKA, A. F. Contribuição para a concepção de aplicações no paradigma orientado a notificações (pon) sob o viés de padrões. 2012. Master in Science Thesis, Graduate School in Electrical Engineering and Industrial Computer Science (CPGEI) at the Federal University of Technology Paraná (UTFPR). Curitiba – Paraná (PR), Brazil. Disponível em: http://repositorio.utfpr.edu.br/jspui/handle/1/327.

RONSZCKA, A. F. et al. Comparações quantitativas e qualitativas entre o paradigma orientado a objetos e o paradigma orientado a notificações sobre um simulador de jogo. In: **III Congresso Intern. de Computación y Telecom.-COMTEL, Lima, Peru**. [S.l.: s.n.], 2011.

ROY, P. V. et al. Programming paradigms for dummies: What every programmer should know. **New computational paradigms for computer music**, IRCAM/Delatour France, v. 104, 2009.

RUSSELL, S.; NORVIG, P. Artificial intelligence: a modern approach. 2009.

SCHüTZ, F. et al. Training of an artificial neural network with backpropagation algorithm using notification oriented paradigm. 2015. LA-CCI/CBIC, October.

SHOHAM, Y. Agent-oriented programming. **Artificial intelligence**, Elsevier, v. 60, n. 1, p. 51–92, 1993.

SIMÃO, J. et al. Rule and agent oriented software architecture for controlling automated manufacturing systems. **Frontiers in Artificial Intelligence and Applications (Advances on Logic Artificial Intelligence and Robotics). Amsterdam, The Netherlands: IOS PRESS BOOKS v. 71**, p. 224–231, 2001.

SIMÃO, J. M. **Proposta de uma arquitetura de controle para sistemas flexíveis de manufatura baseada em regras e agentes**. 2001. Master in Science Thesis, Graduate School in Electrical Engineering and Industrial Computer Science (CPGEI) at the Federal University of Technology Paraná (UTFPR). Curitiba – Paraná (PR), Brazil.

SIMÃO, J. M. **A Contribution to the Development of a HMS simulation tool and Proposition of a Meta-Model for Holonic Control**. Tese (Doutorado) — School in Electrical Engineering and Industrial Computer Science (CPGEI) at Federal University of Technology - Paraná (UTFPR, Brazil) and Research Center For Automatic Control of Nancy (CRAN) - Henry Poincaré University (UHP, France), 2005. Disponível em: http://arquivos.cpgei.ct.utfpr.edu.br/Ano_2005/teses/Tese_012_2005.pdf.

SIMÃO, J. M. et al. Notification oriented and object oriented paradigm comparison via sale system. **Journal of Software Engineering and Applications**, Scientific Research Publishing, v. 5, n. 09, p. 695–710, 2012. ISSN 1945-3116. DOI 10.4236/jsea.2012.56047. Disponível em: http://www.scirp.org/journal/PaperInformation.aspx?paperID=22362.

SIMÃO, J. M. et al. A game comparative study: Object-oriented paradigm and notification-oriented paradigm. **Journal of Software Engineering and Applications**, Scientific Research Publishing, v. 5, n. 09, p. 722–736, 2012. ISSN 1945-3116. DOI 10.4236/jsea.2012.59085. Disponível em: http://www.scirp.org/journal/PaperInformation.aspx?paperID=22364.

SIMÃO, J. M. et al. Evaluation of the notification oriented paradigm applied to sentient computing. In: IEEE. **Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2014 IEEE 17th International Symposium on**. [S.l.], 2014. p. 253–260.

SIMÃO, J. M. et al. Arquitetura de software de controle orientada a regras e agentes para sistemas automatizados de manufatura. **Simpósio Brasileiro de Automaç ao Inteligente (SBAI), Canela-RS, Anais SBAI**, 2001.

SIMÃO, J. M.; STADZISZ, P. C. An agent-oriented inference engine applied for supervisory control of automated manufacturing systems. **Frontiers in Artificial Intelligence and Applications (Advances in Logic, Art. Int. and Robotics - LAPTEC 2002 Edited by Abe J. M., Silva Filho J. I..), IOS Press Books, Amsterdam - The Netherlands, Vol. 85**, v. 85, p. 234–241, 2002. ISBN: 1 58603 292 5, 2002.

SIMÃO, J. M.; STADZISZ, P. C. Paradigma orientado a notificações (pon)–uma técnica de composição e execução de software orientado a notificações. 2008. PEDIDO DE PATENTE: Privilégio de Inovação. Número do registro: PI08055181, data de depósito: 26/11/2008, INPI - Instituto Nacional da Propriedade Industrial. Universidade Tecnológica Federal do Paraná - UTFPR (Demanda Agência de Inovação, 2007). Disponível em: http://bit.ly/1SAQod3.

SIMÃO, J. M.; STADZISZ, P. C. Inference based on notifications: a holonic metamodel applied to control issues. **Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on**, IEEE, v. 39, n. 1, p. 238–250, 2009.

SIMÃO, J. M.; STADZISZ, P. C.; KÜNZLE, L. A. Rule and agent-oriented architecture to discrete control applied as petri net players. **Frontiers in Artificial Intelligence and Applications (FAAI)-Advances in Intelligent Systems and Robotics"LAPTEC 2003**, IOS Press, Amsterdam-The Netherlands, p. 121–129, 2003. ISBN 4 274 90624 8 C3055 (Ohmsha).

SIMÃO, J. M. et al. Mecanismo de inferência otimizado do paradigma orientado a notificações (pon) e mecanismos de resolução de conflitos para ambientes monoprocessados e multiprocessados aplicados ao pon. 2010. Patent pending submitted to INPI/Brazil (Instituto Nacional de Propriedade Industrial) in 03/2010 and Innovation Agency of UTFPR in 2010. INPI Number: PI1003736-5. Disponível em: http://bit.ly/1SgQMeK.

SIMÃO, J. M.; TACLA, C. A.; STADZISZ, P. C. Holonic control metamodel. **Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on**, IEEE, v. 39, n. 5, p. 1126–1139, 2009.

SIMÃO, J. M. et al. Notification oriented paradigm (nop) and imperative paradigm: A comparative study. 2012. Journal of Software Engineering and Applications (JSEA), p.402-416, v.5, n.6, 2012. ISSN: 1945-3116. DOI 10.4236/jsea.2012.59083. Disponível em: https://www.researchgate.net/publication/272666606_Notification_Oriented_Paradigm_%28NOP%29_and_Imperative_Paradigm_A_Comparative_Study.

SIMãO, J. M. et al. Paradigma orientado a notificações em hardware digital. 2012. [Pedido de Proteção Industrial e Pedido de Patente enviados à Agência de Inovação da UTFPR respectivamente em 11/05/2012 e 17/07/2012] Patent pending INPI/Brazil and UTFPR, 2012. Patent INPI: BR 10 2012 026429 3. http://www.google.com/patents/WO2014059497A1?cl=pt.

SIMãO, J. M. et al. Comparações entre duas materializações do paradigma orientado a notificações (pon): Framework pon prototipal versus framework pon primário. 2012. IV Congreso Internacional de Computación y Telecomunicaciones, COMTEL 2012, Lima, Peru.

SOUZA, T. B. A. Um modelo para avaliação de manutenibilidade de código-fonte orientado a objeto. 2005. Trabalho de Graduação em Engenharia de Software, Universidade Federal de Pernambuco.

VALENÇA, G. Z. Contribuição para materialização do paradigma orientado a notificações (pon) via framework e wizard. 2013. Dissertação (Mestrado em Computação Aplicada - PPGCA) – Universidade Tecnológica Federal do Paraná (UTFPR), Curitiba. Disponível em: http://repositorio.utfpr.edu.br/jspui/handle/1/393.

VAN-ROY, P.; HARIDI, S. **Concepts, techniques, and models of computer programming**. [S.l.]: MIT press, 2004.

VISSER, U.; BURKHARD, H.-D. Robocup: 10 years of achievements and future challenges. **AI magazine**, v. 28, n. 2, p. 115, 2007.

VOLPATO, N. **Prototipagem rápida: tecnologias e aplicações**. [S.l.]: Edgard Clucher, 2007.

WATT, D. A. **Programming language design concepts**. [S.l.]: John Wiley & Sons, 2004.

WEBER, L. et al. Viabilidade de controle orientado a notificações (con) em ambiente concorrente baseado em threads. 2010. In: XV Seminário de Iniciação Científica e Tecnológica (XV SICITE), Campus UTFPR, Cornélio Procópio, PR, Brasil. Anais do XV Seminário de Iniciação Científica e Tecnológica da UTFPR.

WIECHETECK, L. V. B. Método para projeto de software usando o paradigma orientado a notificações–pon. 2012. Master in Science Thesis, Graduate School in Electrical Engineering and Industrial Computer Science (CPGEI) at the Federal University of Technology – Paraná (UTFPR). Curitiba – Paraná (PR), Brazil. Disponível em: `http://repositorio.utfpr.edu.br/jspui/handle/1/212`.

WIECHETECK, L. V. B.; STADZISZ, P. C.; SIMÃO, J. M. Um perfil uml para o paradigma orientado a notificações (pon). In: **"Um Perfil UML para o Paradigma Orientado a Notificações (PON)". III Internacional Congress of Computationm and Telecomunications (Congreso Internacional de Computación y Telecomunicaciones - COMTEL), Lima, Peru**. [S.l.: s.n.], 2011.

WITT, F. A. et al. Comparação entre o paradigma orientado a objetos (poo) e o paradigma orientado a notificações (pon) em um controle discreto em lógica reconfigurável. 2011. Em: XVI SICITE - Seminário de Iniciação Científica e Tecnológica da UTFPR, 2011, Ponta Grossa - PR. Anais do XVI SICITE.

WOOLDRIDGE, M.; JENNINGS, N. R. et al. Intelligent agents: Theory and practice. **Knowledge engineering review**, Cambridge Univ Press, v. 10, n. 2, p. 115–152, 1995.

WOOLDRIDGEY, M.; CIANCARINI, P. Agent-oriented software engineering: The state of the art. In: SPRINGER. **Agent-oriented software engineering**. [S.l.], 2001. p. 1–28.

XAVIER, R. D. **Paradigmas de desenvolvimeto de software: Comparação entre abordagens orientada a eventos e orientada a notificações**. Tese (Doutorado) — Master Thesis, Universidade Tecnológica Federal do Paraná-UTFPR, Graduate School of Electrical Engineering and Computer Science-CPGEI, Curitiba/PR, 2014.

YOON, M. **Developing basic soccer skills using reinforcement learning for the RoboCup Small Size League**. Tese (Doutorado) — Stellenbosch University, 2015.

APÊNDICE A – DESCRIÇÃO DAS ALTERAÇÕES REALIZADAS NA LINGPON

Este apêndice apresenta as contribuições deste trabalho para a LingPON. Primeiramente, a seção A.1 apresenta em detalhes as alterações realizadas na LingPON para permitir o relacionamento de *FBEs* através de agregação. Na sequência, na seção A.2 são apresentadas as alterações realizadas na LingPON para suportar a agregação de *Rules* em *FBEs*, a qual foi nomeada *FBE Rules*, de forma a permitir a criação de *Rules* de forma mais simples e com menor redundância de linhas de código. Por fim, na seção A.3 são apresentadas as alterações realizadas na LingPON para solucionar o problema encontrado na geração de código C++ com múltiplas instâncias de um dado *FBE*.

A.1 AGREGAÇÃO DE *FBES*

Em sua versão original, o analisador léxico utilizado pelo compilador já suportava a declaração de *FBEs* como *Attributes* de outros *FBEs*. Entretanto, os módulos de análise sintática e geração de código não eram capazes de identificar a relação de agregação de *FBEs* na declaração de *Premises* e *Instigations*, conforme apresentado no Código 32, causando exceção e consequente encerramento prematuro do processo de compilação.

Código 32: Exemplo de agregação de *FBEs* não suportada na versão original da LingPON

```
1  fbe Arm
2      attributes
3          float atSize 0.0
4      end_attributes
5      methods
6          method mtExecuteAction()
7      end_methods
8  end_fbe
9
10 fbe Robot
11     attributes
12         Arma atArm ;
13     end_attributes
14 ...
15 end_fbe
16
17 rule rlMoveArm
18     condition
19         subcondition condition1
20             premise prRobotArmSize robot.atArm.size == 10.0
21         end_subcondition
22     end_condition
23     action
24         instigation inMoveRobotArm robot.atArm.mtExecuteAction();
25     end_action
26 end_rule
```

Através do fragmento de código apresentado no Código 33, é possível observar que, na versão original do compilador, uma *Instigation* poderia ser definida pela palavra chave *INSTIGATION*, um *Method* (*method_use*), o qual representa uma ação a ser executada e, opcionalmente, um nome para essa *Instigation*. O *Method*, por sua vez, deveria ser definido por um “id”, o qual representa um *Method* de um *FBE* que deverá ser executado.

Código 33: Definição de *Instigation* extraído do arquivo de configuração do analisador sintático (*Bison*) utilizado pela atual versão do compilador PON.

```
1  instigation    : INSTIGATION method_use
2                 | INSTIGATION method_use
3                 ;
4
5  method_use     : id LP RP SEMICOLON
6                 ;
7
8  id             : ID
9                 | ID POINT ID
10                ;
```

De forma simplista, o analisador sintático da atual versão do compilador considera a sequência de caracteres anterior ao ponto como sendo o nome de uma instância de *FBE* e o restante como sendo o nome do *Method* referente à esta instância. No exemplo apresentado no Código 32, o nome da instância a ser interpretada pelo compilador na *Instigation* seria "*robot*" e o nome do *Method* "*atArm.mtExecuteAction*". Como o *FBE Robot não possui nenhum Method* com esse nome, o processo de compilação do código-fonte seria abortado.

Visando solucionar este problema, algumas mudanças foram realizadas no analisador sintático do compilador PON. Em um primeiro momento, uma sutil alteração no arquivo de configuração do analisador sintático foi efetuada de forma a permitir a correta separação do nome da instância que deverá executar a ação e qual método deverá ser invocado pela *Instigation.* Esta alteração é apresentada no Código 34.

Código 34: Nova definição da regra "id" no arquivo de configuração do analisador sintático (*Bison*).

```
1  id             : ID
2                 | ID POINT id
3                 ;
```

Utilizando essa nova definição da regra “id” do analisador sintático no exemplo apresentado no Código 32, o nome da instância interpretada pelo compilador na *Instigation* passa a ser “*robot.atArm*” e o nome do *Method* “*mtExecuteAction*”.

Com o problema de separação de nomes de instância e *Method* relacionados a *Instigation* solucionado, deu-se início às alterações no código relacionado ao gerenciamento da tabela de símbolos utilizada pelo compilador.

Dado que um *FBE Robot* possui um *Attribute* do tipo *Arm*, para cada nova instância do *FBE Robot* criada, a nova versão do compilador deverá criar uma nova instância do *FBE Arm*, adicioná-la à tabela de símbolos do compilador e relacionar essa nova instância com a correta instância do *FBE Robot*. O controle de quais instâncias devem ser criadas para criar a correta correlação entre os *FBEs* deve ser feito de maneira recursiva, de forma a permitir a agregação de *FBEs* em múltiplos níveis.

Desse modo, o método *createInstantiation* da classe *Compiler* do compilador PON foi alterado. As alterações realizadas são apresentadas entre as linhas 13 e 24 do Código 35.

Código 35: Método da classe *Compiler* alterado para permitir a criação de instâncias de *FBEs* de forma recursiva.

```
1  void createInstantiation(string fbeName, list<string>idList)
2  {
3    Entity *fbeFound = semanticAnalyser.getEntity(fbeName);
4    if (fbeFound == 0) {return;}
5
6    for (list<string>::iterator it = idList.begin(); it != idList.end(); ++it)
7    {
8      Entity *entityFound = semanticAnalyser.getEntity(*it);
9      if (entityFound == 0)
10     {
11       Instantiation *instantiation = new Instantiation(*it);
12       instantiation->fbe = (Fbe*)fbeFound;
13       list<Attribute*>attrs = ((Fbe*)(fbeFound))->attributes;
14
15       for (list<Attribute *>::iterator itAttr = attrs.begin();
16         itAttr != attrs.end(); ++itAttr)
17       {
18         if ((*itAttr)->aType == Attribute::A_ID)
19         {
20           list<string> lsFbeAttribute;
21           lsFbeAttribute.push_back(*it + "_" + (*itAttr)->userEntityId);
22           createInstantiation((*itAttr)-textgreatervalue, lsFbeAttribute);
23         }
24       }
25       semanticAnalyser.addEntity(instantiation);
26     }
27
28 }
```

A.2 *FBE RULES*

Conforme mencionado na seção 3.1.2, um pré-compilador foi desenvolvido de forma a possibilitar a criação de *FBE Rules*. Este pré-compilador utiliza os mesmos módulos de análise léxica e sintática utilizados pelo compilador PON, diferenciando-se apenas pelo módulo de geração de código, o qual gera como saída um código-fonte PON intermediário. Dessa forma, ambos os módulos foram alterados para que fossem capazes de reconhecer e interpretar as *FBE Rules*.

A primeira alteração foi realizada no analisador léxico, uma vez que o mesmo deve ser capaz de reconhecer a sequência de caracteres que identificam as palavras chave *fbeRule* e *end_fbeRule* como *tokens*. Dessa forma, o arquivo "*lex_pon.l*" foi alterado conforme apresentado no Código 36.

Código 36: Alteração realizada no arquivo "*lex_pon.l*" para suportar *FBE Rules*.

```
1 fbeRule                return FBERULE;
2 end_fbeRule            return END_FBE_RULE;
```

Posteriormente, o analisador sintático do compilador PON foi alterado de forma a suportar a declaração de *FBE Rules* dentro o escopo de declaração de *FBEs*. Para isto, duas alterações foram realizadas no arquivo "*bison_pon.y*". A primeira alteração realizada foi a criação da regra para a correta interpretação de *FBE Rules*, conforme é apresentado entre as linhas 1 e 3 do Código 37. A segunda alteração foi realizada na regra que define como um *FBE* pode ser declarado. A regra foi alterada para suportar, além de *Attributes* e *Methods*, também a declaração de *FBE Rules*, conforme é apresentado entre as linhas 5 e 8 do Código 37

Código 37: Alterações realizadas no arquivo "*bison_pon.y*" para suportar *FBE Rules.*

```
1 fbeRules      : fbeRule
2               | fbeRule fbeRules
3               ;
4
5 fbe_body      : decl_attributes decl_methods fbeRules
6               | decl_attributes decl_methods
7               | decl_attributes
8               ;
```

Na sequência, foi necessário aplicar alterações sobre a classe *NOPCompiler*, a qual representa a classe responsável pela geração de código do pré-compilador PON. Para isso, foi criado um novo método, nomeado *createFbeRules*. Este método é responsável por identificar o *FBE* associado à uma *FBE Rule* e criar uma entidade *Rule* para cada instância declarada do *FBE*.

Primeiramente o pré-compilador verifica qual o *FBE* relacionado à entidade *FBE Rule* que está sendo processada. Uma vez identificado, o pré-compilador percorre sua tabela de símbolos afim de identificar todas as instâncias declaradas do *FBE*.

Conhecendo a lista de instâncias declaradas do *FBE*, o pré-compilador inicia um laço de repetição para criar uma nova entidade *Rule* para cada uma das instâncias presentes na lista. Dessa forma, uma *Rule* é criada no código-fonte intermediário para cada uma das instâncias do *FBE*.

A.3 CORREÇÃO DE ERRO: GERAÇÃO DE CÓDIGO-ALVO C++ COM MÚLTIPLAS INSTÂNCIAS DE *FBE*

Conforme apresentado na seção 3.1.3, um *bug* existente na atual versão do compilador PON foi solucionado com auxílio do pré-compilador PON.

A solução desenvolvida faz com que cada instância de um dado *FBE* seja transformado em um *FBE* no código-fonte PON intermediário, durante o processo de pré-compilação. Para isso, o método *parseFbeByInstances*, apresentado no Código 38, foi adicionado à classe NOPCompiler.

Código 38: Código-fonte do método *parseFbeByInstances* adicionado à classe NOPCompiler.

```
1  void NOPCompiler::parseFbeByInstances() {
2    for (map<string, Instantiation*>::iterator it = mapInstantiations.begin();
3          it != mapInstantiations.end(); ++it)
4     {
5       Fbe *fbe = (it->second)->fbe;
6
7       std::string fbeName = fbe->userEntityId;
8       std::string instanceName = (it->second)->userEntityId;
9
10      Fbe *newFbe = new Fbe(fbeName + instanceName);
11      newFbe->methods = fbe->methods;
12      newFbe->premises = fbe->premises;
13      newFbe->attributes = copyAttributes(fbe->attributes);
14
15      for (list<Attribute *>::iterator itAttribute = newFbe-¿attributes.begin();
16            itAttribute != newFbe->attributes.end(); ++itAttribute)
17      {
18        if ((*itAttribute)->aType == Attribute::A_ID)
19        {
20            (*itAttribute)->value = (*itAttribute)->value +
21                                    instanceName +
22                                    (*itAttribute)->userEntityId;
23        }
24      }
25      (it->second)->fbe = newFbe;
26      mapFBEs.erase(fbe->userEntityId);
27      mapFBEs[newFbe->userEntityId] = newFbe;
28    }
29 }
```

Para cada instância de *FBE* presente no código-fonte PON, o método *parseF-*

beByInstances cria uma cópia do *FBE* original, isto é, um *FBE* com os mesmos *Methods*, *Attributes* e *Premises* associadas, conforme apresentado entre as linhas 10 e 24 do Código 38. Entretanto, conforme apresentado na linha 10 do Código 38, o nome do novo *FBE* é uma composição do nome do *FBE* original e da instância à ele associada. Por exemplo, caso o nome do *FBE* seja "*Robot*" e o nome das instâncias "*robot1*" e "*robot2*", o pré-compilador criará dois novos *FBEs*: *Robotrobot1* e *Robotrobot2*.

Posteriormente, uma vez que o novo *FBE* esteja criado, o *FBE* antigo é removido da tabela de símbolos utilizada pelo compilador (linha 26) e o novo objeto *FBE* é então adicionado à tabela de símbolos (linha 27).

Desse modo, a atual versão do módulo de compilação e geração de código para C++ é capaz de interpretar e criar as classes e objetos em C++ de maneira correta, relacionando à cada instância de *FBE* apenas as *Rules* à ela associada.

APÊNDICE B – ESPECIFICAÇÃO TÉCNICA - ROBOCUP SMALL SIZE LEAGUE

B.1 DESCRIÇÃO DO AMBIENTE DE JOGO

Uma partida de futebol de robôs na categoria SSL(Small Size League) ocorre em um campo de jogo retangular, sobre um piso plano recoberto com um feltro ou carpete fino, na cor verde escuro. As dimensões das linhas que demarcam o campo definem uma área de jogo retangular, de 9000mm x 6000 mm (9 x 6 metros). Deve haver uma área lateral que permita aos jogadores saírem da área delimitada do campo de jogo(por exemplo para a cobrança de um arremesso lateral). Esta área lateral deve ter 700 mm em todos os lados, sendo que a 300 mm das linhas delimitantes, deve ser construída uma parede de proteção de 100 mm de altura, para impedir que robôs ou a bola saiam da área de jogo. Os gols devem ter 160mm de altura, fixos ao piso.

As dimensões exatas do campo, bem como pinturas de área de goleiros, meio de campo, e círculo central, são apresentadas na Figura 32. Todas as linhas são pintadas em cor branca, e devem ter 10 mm de espessura. As paredes laterais, e os gols, devem também ser pintados de branco. Os gols não podem possuir tetos, pois deve ser possível ver o jogador que atua como goleiro mesmo se o mesmo entrar parcial ou totalmente dentro do gol.

Entretanto, deve haver uma trave superior na parte frontal do gol, composta por uma barra de aço com diâmetro não maior que 10 mm, porém forte o suficiente para rebater a bola e evitar dúvida sobre se a bola entrou no gol, ou passou por cima do mesmo. Esta barra deve ficar a 155 mm de altura do solo. A parte superior do gol deve ser coberta por uma rede, que não impeça a visão superior dos marcadores do jogador goleiro, mas impeça a bola de sair do gol pela parte superior do mesmo.

A bola a ser utilizada deve ser uma bola de golfe, esférica, com aproximadamente 43 mm de diâmetro, aproximadamente 46 g de massa, e na cor alaranjada.

A partida é disputada por dois times de robôs totalmente autônomos. Cada time

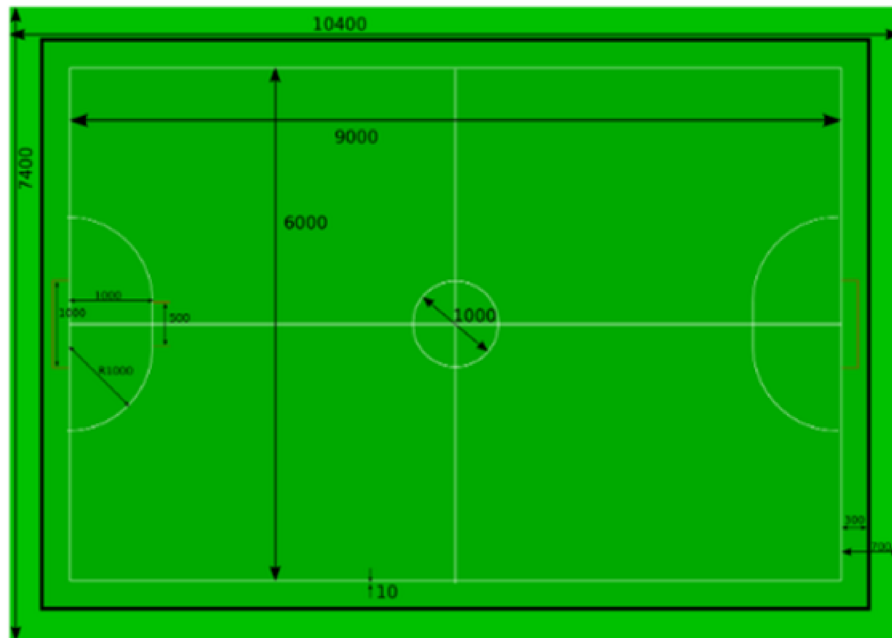

Figura 32: Campo de jogo (dimensões em milímetros).

é composto de não mais que 6 (seis) robôs, claramente identificados por números, para que seja possível ao juiz identificar cada robô individualmente. O robô que irá atuar como goleiro deve ser previamente informado ao juiz, antes do início da partida. A partida não pode ter início a não ser que ambos os times tenham ao menos um robô funcionando.

Substituição de robôs: É possível substituir qualquer robô por outro (robôs reservas) quantas vezes for necessário, durante as interrupções da partida, mediante informação e autorização do árbitro. Não há limite para o número de substituições. Sempre que há uma substituição, primeiramente o robô a ser substituído deve deixar o campo de jogo, e posteriormente, após autorização do juiz, um novo robô entra no campo de jogo, sendo posicionado no centro de campo, próximo à linha lateral de um lado ou de outro. Se um robô deixar de funcionar, após autorização, um integrante humano da equipe de desenvolvimento pode adentrar ao campo para retirar este robô defeituoso, somente em uma interrupção da partida, e com autorização prévia do árbitro, após a autorização da substituição.

B.2 DESCRIÇÃO DOS ROBÔS

Cada robô deve ser construído de forma a estar contido em um cilindro de 180 mm de diâmetro e 150 mm de altura, conforme apresentado na Figura 33. A parte superior do robô deve ser plana, de forma a ser possível colar um Identificador Padrão, descrito na

seção “Uniformes”.

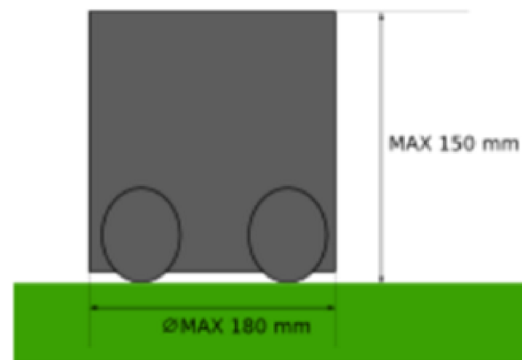

Figura 33: Dimensões máxima do robô (em milímetros).

B.2.1 UNIFORMES

Cada robô deve possuir um identificador único, de forma que seja possível identificá-lo durante a partida de futebol de robôs. Esse identificador é composto 5 marcadores que estão localizados na parte superior do robôs, conforme apresentado na Figura 34.

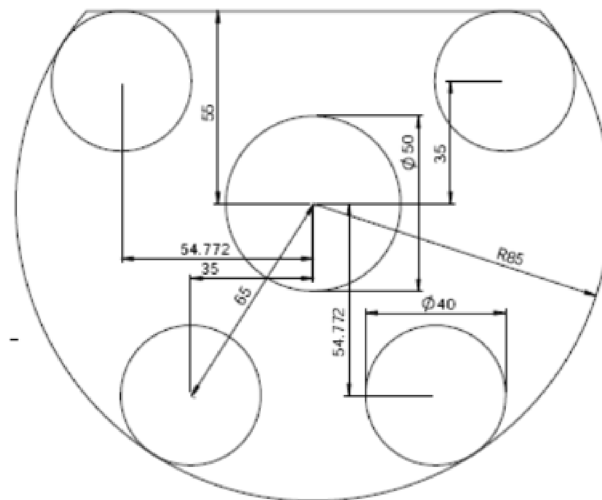

Figura 34: Posição do marcador central (azul ou amarelo), e dos 4 marcadores laterais coloridos (magenta ou verde claro).

Utilizando este padrão de identificação, cada time pode ter até 12 robôs, dos quais no máximo 6 poderão ser utilizado ao mesmo tempo durante a partida de futebol, conforme apresentado na Figura 35.

Como apresentado na Figura 36, o funcionamento de uma partida de futebol de robôs na categoria Small Size League(SSL) é o seguinte: um software padronizado,

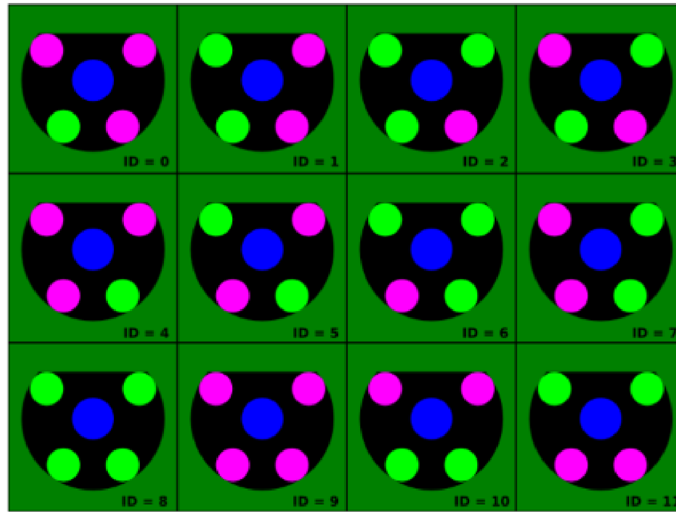

Figura 35: Marcadores coloridos utilizados para a identificação.

denominado SSL-Vision [Zickler et al. 2009], continuamente processa imagens de vídeo obtidas através de câmeras posicionadas superiormente ao campo, e a cada 1/60 segundos (60 vezes por segundo) executa a captura das posições dos jogadores em campo (de ambos os times) e da bola, e repassa estas informações através de uma rede cabeada, usando protocolo UDP, para os computadores de controle tanto da equipe Azul quanto da Equipe Amarela. Cabe aos sistemas de controle utilizar estas informações para definir o comportamento individual de cada robô de seu time, e repassar estes comandos através de algum mecanismo de comunicação sem fio aos respectivos robôs para execução.

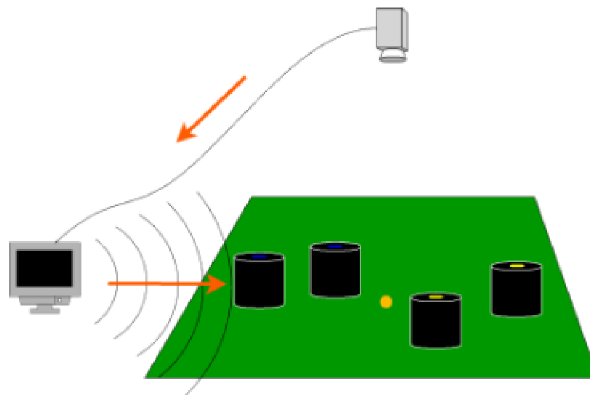

Figura 36: Esquema geral de funcionamento do Futebol de Robôs da categoria SSL. Fonte: http://wiki.robcup.org/Small_Size_League

As regras de um jogo da categoria SSL procuram seguir o que se esperaria de um jogo de futebol humano, com algumas diferenças básicas. As regras básicas são descritas a seguir:

- O jogo é disputado em dois tempos de 10 minutos cada, com um intervalo de não

mais de 5 minutos entre os tempos. Cada equipe pode requerer até 4 “tempos” durante a partida, desde que o tempo total somado não ultrapasse 5 minutos. Por exemplo, um time pode pedir dois tempos de 1 minuto cada, e mais um tempo de 3 minutos depois. Os tempos só podem ser pedidos durante uma parada do jogo;

- Em caso de empate, são disputados outros 2 tempos de jogo, que podem ser de 10 minutos, ou de 5 minutos cada; Em caso de continuidade do empate, pode haver uma decisão através de pênaltis alternados;

- Início de partida, ou reinício de partida após um gol: o juiz posiciona a bola no centro do campo, informa qual time deverá reiniciar a partida (time azul ou amarelo), e envia o comando para reiniciar a partida; O time responsável por reiniciar a partida deve enviar um robô para que o mesmo toque na bola, reiniciando a partida;

Há também um outro programa, denominado SSL-RefereeBox (cuja interface é apresentada na Figura 37), que permite a um árbitro auxiliar informar, também através do protocolo de rede local de comunicação (desta vez TCP) sobre as decisões do árbitro a cada momento. Os comandos que podem ser enviados pelo árbitro aos times são listados a seguir:

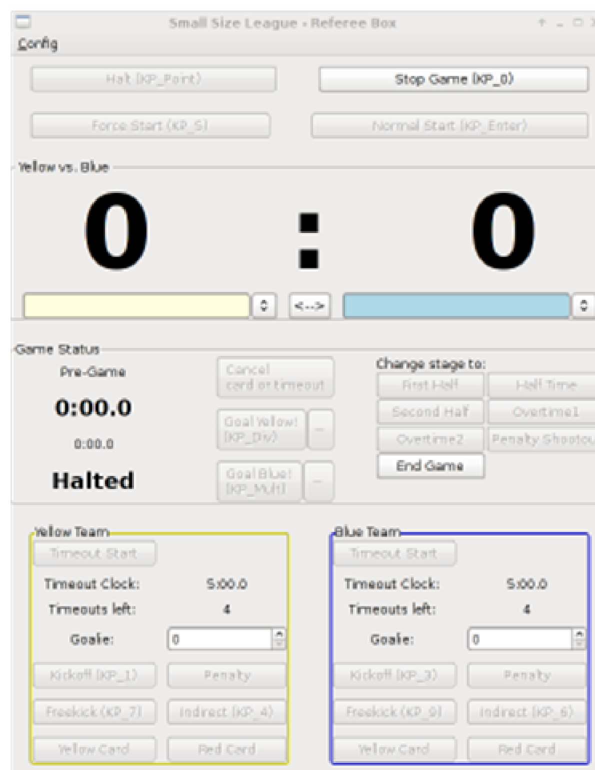

Figura 37: Interface do programa SSL Referee Box, para envio de comandos do árbitro.

- Halt: informa a todos os robôs de ambos os times para pararem de se mover imediatamente, pois há alguma situação de emergência a ser tratada;
- Stop Game: ocorreu uma interrupção da partida (ou um gol, ou a bola saiu da área de jogo, ou foi marcada uma falta, ou pênalti). Este comando também é executado imediatamente antes do início de cada tempo de jogo, ou após uma parada de emergência (comando Halt), para que os robôs se preparem para o reinício da partida;
- Normal Start: este comando é enviado somente após o envio de um comando de início de partida (KickOff), ou de cobrança de pênalti, após os jogadores terem tido tempo de se posicionar adequadamente (os casos de início de partida ocorrem nos inícios dos tempos de jogo, ou após um time marcar um gol; no caso da marcação de um pênalti, o árbitro primeiro envia um comando "Stop Game", depois informa qual time vai bater e qual vai defender, enviando o comando "Penalty Blue", por exemplo, em que um jogador do time azul se posiciona para bater o pênalti, e o goleiro do time amarelo se posiciona para defender, e todos os outros jogadores de ambos os times se posicionam atrás da linha de meio de campo, no campo contrário ao que será utilizado para a cobrança do pênalti);
- Force Start: este comando só é enviado pelo árbitro quando um time não consegue realizar sua ação (iniciar a partida, bater um tiro indireto-por exemplo um tiro lateral, ou bater um tiro direto, ou bater um pênalti) por um período de 5 segundos; então o árbitro envia um comando "Stop Game", e envia em seguida um comando "Force Start" informando que ambos os times podem ir em direção à bola;
- Kickoff: este comando enviado pelo árbitro para indicar o reinício da partida no início da partida, após um gol, no início do segundo tempo ou no início de cada período de prorrogação. Ainda, um gol pode ser marcado a partir da cobrança direta de um kickoff.
- Indirect Freekick: comando enviado pelo árbitro para indicar o reinício da partida após marcação de certas irregularidades durante a partida, tais como segurar a bola por mais de 15 segundos ou segurar a bola após soltá-la sem que nenhum outro robô tenha tocado na bola dentro de sua área de defesa. Além disso, ocorre a marcação de "Indirect Kick" quando um robô encosta no robô "goleiro" do time adversário dentro de sua área de defesa, ou então quando um robô percorre mais de 1000 mm

mantendo a posse da bola ou quando um robô chuta a bola com velocidade maior que 8 m/s.

- Penalty: comando enviando pelo árbitro quando um robô defensor realiza as seguintes irregularidades dentro da área de defesa: choque de forma acentuada com um robô do time adversário, segurar um robô adversário impossibilitando-o de se locomover ou segurar a posse de bola de forma que impossibilite a continuidade da partida.

B.3 REQUISITOS FUNCIONAIS DA APLICAÇÃO DE CONTROLE

Os seguintes requisitos funcionais devem ser seguidos por um software que controle o comportamento de um time de futebol de robôs

Requisitos Funcionais relativos à quantidade de robôs em campo em um determinado momento:

- RF01: Se 6 robôs estiverem presentes no campo, o de menor número de camisa deve ser o goleiro, os 3 seguintes, em ordem de número de camisa, devem se comportar como zagueiros, e os dois seguintes devem se comportar como atacantes;
- RF02: Se 5 robôs estiverem presentes no campo, o de menor número de camisa deve ser o goleiro, os 3 seguintes, em ordem de número de camisa, devem se comportar como zagueiros, e seguinte deve se comportar como atacante;
- RF03: Se 4 robôs estiverem presentes no campo, o de menor número de camisa deve ser o goleiro, os 2 seguintes, em ordem de número de camisa, devem se comportar como zagueiros, e seguinte deve se comportar como atacante;
- RF04: Se 3 robôs estiverem presentes no campo, o de menor número de camisa deve ser o goleiro, e os 2 seguintes, em ordem de número de camisa, devem se comportar como zagueiros;
- RF05: Se 2 robôs estiverem presentes no campo, o de menor número de camisa deve ser o goleiro, e o outro deve se comportar como zagueiro;
- RF06: Se apenas 1 robô estivere presente no campo, o mesmo deve se comportar como goleiro;

Requisito Funcional relativo à movimentação dos robôs:

•RF07: O sistema de controle deve ser capaz de enviar comandos para mover cada um dos robôs, de forma individual, até uma posição determinada pelas coordenadas do plano cartesiano (x, y) e um determinado ângulo.

Requisitos Funcionais comuns à todos os jogadores:

•RF08: Ao receber o comando "Halt", todos os robôs devem parar imediatamente;

•RF09: Ao receber o comando "Stop", os robôs devem manter a distância mínima de 500mm. em relação à bola.

•RF10: Quando o último comando recebido foi "Normal Start" e a bola estiver em movimento, o robô deve tocar a bola para um companheiro do seu time caso a bola esteja localizada em seu campo de defesa e haja algum companheiro sem marcação adversária.

•RF11: Quando o último comando recebido foi "Normal Start" e a bola estiver em movimento, o robô deve chutar a bola em direção ao gol adversário caso a bola esteja localizada em seu campo de defesa e não haja nenhum companheiro sem marcação adversária.

•RF12: Quando o último comando recebido foi "Normal Start" e a bola estiver em movimento, o robô deve chutar a bola em direção ao gol adversário caso a bola esteja localizada no campo adversário e não haja nenhum adversário entre o robô e o gol adversário.

•RF13: Quando o último comando recebido foi "Normal Start" e a bola estiver em movimento, o robô deve tocar a bola para um companheiro de seu time caso a bola esteja localizada no campo adversário e haja ao menos um adversário entre o robô e o gol adversário.

Requisitos Funcionais relativos ao jogador que está com o comportamento de "Goleiro" em um determinado momento:

•RF14: Ao receber o comando "Stop Game", o robô que está na posição "Goleiro" deve se posicionar próximo à linha de fundo, em frente ao seu próprio gol, de forma a estar centralizado sobre a linha imaginária que liga a posição atual da bola ao fundo do gol, caso a distância entre a bola e o gol defendido seja menor que 1800 mm.;

- RF15: Ao receber o comando “Stop Game”, o robô que está na posição “Goleiro” deve se posicionar junto à linha de sua área de defesa, em frente ao seu próprio gol, de forma a estar centralizado sobre a linha imaginária que liga a posição atual da bola ao fundo do gol caso a distância entre a bola e o gol defendido seja maior que 1800 mm.;

- RF16: Ao receber o comando “Normal Start”, o robô que está na posição “Goleiro” deve chutar a bola em direção ao gol adversário caso a bola esteja à uma distância menor que 800 mm em relação ao gol que está sendo defendido e o robô seja o jogador mais próximo à bola.

- RF17: Ao receber o comando “Normal Start”, o robô que está na posição “Goleiro” deve se posicionar próximo à linha de fundo, em frente ao seu próprio gol caso a bola esteja à uma distância menor que 800 mm. em relação ao gol que está sendo defendido e o robô não seja o jogador mais próximo à bola.

- RF18: Ao receber o comando “Penalty” favorável à equipe adversária, o jogador que está na posição “Goleiro” deve se posicionar sobre a linha do gol que está sendo defendido de forma a bloquear a cobrança do time adversário.

- RF19: Ao receber o comando “Kickoff” favorável à equipe adversária, o jogador que está na posição “Goleiro” deve se deve se posicionar próximo à linha de fundo, em frente ao seu próprio gol, de forma a estar centralizado sobre a linha imaginária que liga a posição atual da bola ao fundo do gol.

- RF20: Ao receber o comando “Free Kick” favorável à equipe adversária, o jogador que está na posição “Goleiro” deve se deve se posicionar próximo à linha de fundo, em frente ao seu próprio gol, de forma a estar centralizado sobre a linha imaginária que liga a posição atual da bola ao fundo do gol.

- RF21: Ao receber o comando “Indirect Kick” favorável à equipe adversária, o jogador que está na posição “Goleiro” deve se deve se posicionar próximo à linha de fundo, em frente ao seu próprio gol, de forma a estar centralizado sobre a linha imaginária que liga a posição atual da bola ao fundo do gol.

Requisitos Funcionais relativos ao jogador que está com o comportamento de “Defensor Direito” em um determinado momento:

- RF22: Ao receber o comando “Stop Game”, o robô que está na posição “Defensor Direito” deve se posicionar em frente à linha de sua área de defesa e na direção da

trave do lado direito do gol que está sendo defendido caso a distância entre a bola e o gol defendido seja maior que 1800 mm;

- RF23: Ao receber o comando "Stop Game", o robô que está na posição "Defensor Direito" deve se posicionar em frente à bola, de forma a estar posicionado entre a bola e o gol defendido, caso a distância entre a bola e o gol defendido seja menor que 1800 mm;

- RF24: Ao receber o comando "Free Kick" para sua própria equipe, o robô que está na posição "Defensor Direito" deve se posicionar em frente à linha de sua área de defesa e na direção da trave do lado direito do gol que está sendo defendido.

- RF25: Ao receber o comando "Free Kick" para a equipe adversária, o robô que está na posição "Defensor Direito" deve se posicionar fora da linha imaginária que liga a bola ao fundo do gol que está sendo defendido.

- RF26: Ao receber o comando "Indirect Kick", independente para qual equipe, o robô que está na posição "Defensor Direito" deve se posicionar em frente à linha de sua área de defesa e na direção da trave do lado direito do gol que está sendo defendido.

- RF27: Ao receber o comando "Penalty" favorável à equipe adversária, o robô que está na posição "Defensor Direito" deve se posicionar fora da área de defesa e ao lado direito da bola, respeitando a distância mínima para a cobrança da penalidada máxima.

- RF28: Ao receber o comando "Normal Start", o robô que está na posição "Defensor Direito" deve se posicionar em frente à linha de sua área de defesa e na direção da trave do lado direito do gol que está sendo defendido.

Requisitos Funcionais relativos ao jogador que está com o comportamento de "Defensor Esquerdo" em um determinado momento:

- RF29: Ao receber o comando "Stop Game", o robô que está na posição "Defensor Esquerdo" deve se posicionar em frente à linha de sua área de defesa e na direção da trave do lado esquerdo do gol que está sendo defendido caso a distância entre a bola e o gol defendido seja maior que 1800 mm.;

- RF30: Ao receber o comando "Stop Game", o robô que está na posição "Defensor Esquerdo" deve se posicionar em frente à bola, de forma a estar posicionado entre a

bola e o gol defendido, caso a distância entre a bola e o gol defendido seja menor que 1800 mm;

- RF31: Ao receber o comando “Free Kick” para sua própria equipe, o robô que está na posição “Defensor Esquerdo” deve se posicionar em frente à linha de sua área de defesa e na direção da trave do lado esquerdo do gol que está sendo defendido.

- RF32: Ao receber o comando “Free Kick” para a equipe adversária, o robô que está na posição “Defensor Esquerdo” deve se posicionar fora da linha imaginária que liga a bola ao fundo do gol que está sendo defendido.

- RF33: Ao receber o comando “Indirect Kick”, independente para qual equipe, o robô que está na posição “Defensor Esquerdo” deve se posicionar em frente à linha de sua área de defesa e na direção da trave do lado esquerdo do gol que está sendo defendido.

- RF34: Ao receber o comando “Penalty” favorável à equipe adversária, o robô que está na posição “Defensor Esquerdo” deve se posicionar fora da área de defesa e ao lado esquerdo da bola, respeitando a distância mínima para a cobrança da penalidada máxima.

- RF35: Ao receber o comando “Normal Start”, o robô que está na posição “Defensor Esquerdo” deve se posicionar em frente à linha de sua área de defesa e na direção da trave do lado esquerdo do gol que está sendo defendido.

Requisitos Funcionais relativos ao jogador que está com o comportamento de “Meio-Campo” em um determinado momento:

- RF36: Ao receber o comando “Stop Game”, o robô que está na posição “Meio-Campo” deve se posicionar sobre a linha imaginária que liga a posição atual da bola ao fundo do gol que está sendo defendido por ele;

- RF37: Ao receber o comando “Kickoff” favorável à equipe adversária, o robô que está na posição “Meio-Campo” deve se posicionar sobre a linha imaginária que liga a posição atual da bola ao fundo do gol que está sendo defendido por ele;

- RF38: Ao receber o comando “Kickoff” favorável à sua equipe, o robô que está na posição “Meio-Campo” deve se posicionar para executar a cobrança em direção ao gol adversário.

- RF39: Ao receber o comando "Normal Start" após o comando "Kickoff" favorável à sua equipe, o robô que está na posição "Meio-Campo" deve executar a cobrança em direção ao gol adversário.

- RF40: Ao receber o comando "Direct Kick" favorável à sua equipe, o robô que está na posição "Meio-Campo" deve se posicionar para executar a cobrança em direção ao gol adversário.

- RF41: Ao receber o comando "Normal Start" após o comando "Direck Kick" favorável à sua equipe, o robô que está na posição "Meio-Campo" deve executar a cobrança em direção ao gol adversário.

- RF42: Ao receber o comando "Indirect Kick" favorável à equipe adversária, o robô que está na posição "Meio-Campo" deve se posicionar sobre a linha imaginária que liga a posição atual da bola ao fundo do gol que está sendo defendido por ele;

- RF43: Ao receber o comando "Indirect Kick" favorável à sua equipe, o robô que está na posição "Meio-Campo" deve se posicionar para executar a cobrança em direção à um de seus companheiros.

- RF44: Ao receber o comando "Normal Start" após o comando "Indireck Kick" favorável à sua equipe, o robô que está na posição "Meio-Campo" deve executar a cobrança em direção à um de seus companheiros.

- RF45: Ao receber o comando "Penalty" para a equipe adversária, o robô que está na posição "Meio-Campo" deve se posicionar atrás do cobrador da penalida máxima, respeitando a distância mínima.

- RF46: Ao receber o comando "Penalty" favorável à sua equipe, o robô que está na posição "Meio-Campo" deve se posicionar para executar a cobrança da penalida máxima.

- RF47: Ao receber o comando "Normal Start" após o comando "Penalty" favorável à sua equipe, o robô que está na posição "Meio-Campo" deve executar a cobrança da penalidade máxima em direção ao gol adversário.

Requisitos Funcionais relativos ao jogador que está com o comportamento de "Atacante Esquerdo" em um determinado momento:

- RF48: Ao receber o comando “Stop Game”, o robô que está na posição “Atacante Esquerdo” deve se posicionar sobre a linha imaginária que liga a posição atual da bola ao fundo do gol que está sendo defendido por ele;
- RF49: Ao receber o comando “Direct Kick”, o robô que está na posição “Atacante Esquerdo” deve se posicionar próximo à trave esquerda do gol adversário.
- RF50: Ao receber o comando “Indirect Kick”, o robô que está na posição “Atacante Esquerdo” deve se posicionar próximo à trave esquerda do gol adversário.
- RF51: Ao receber o comando “Penalty” favorável à sua equipe, o robô que está na posição “Atacante Esquerdo” deve se posicionar junto à linha da área de defesa do time adversário e ao lado esquerdo do robô que está efetuando a cobrança da penalidade máxima.
- RF52: Ao receber o comando “Penalty” favorável à equipe adversária, o robô que está na posição “Atacante Esquerdo” deve se posicionar próximo à trave esquerda do gol adversário.
- RF53: Ao receber o comando “Normal Start”, o robô que está na posição “Atacante Esquerdo” deve se posicionar próximo à trave esquerda do gol adversário caso não seja o robô da equipe que está sendo controlada mais próximo à bola.

Requisitos Funcionais relativos ao jogador que está com o comportamento de “Atacante Direito” em um determinado momento:

- RF54: Ao receber o comando “Stop Game”, o robô que está na posição “Atacante Direito” deve se posicionar sobre a linha imaginária que liga a posição atual da bola ao fundo do gol que está sendo defendido por ele;
- RF55: Ao receber o comando “Direct Kick”, o robô que está na posição “Atacante Direito” deve se posicionar próximo à trave direita do gol adversário.
- RF56: Ao receber o comando “Indirect Kick”, o robô que está na posição “Atacante Direito” deve se posicionar próximo à trave direita do gol adversário.
- RF57: Ao receber o comando “Penalty” favorável à sua equipe, o robô que está na posição “Atacante Direito” deve se posicionar junto à linha da área de defesa do time adversário e ao lado direito do robô que está efetuando a cobrança da penalidade máxima.

- RF58: Ao receber o comando “Penalty” favorável à equipe adversária, o robô que está na posição “Atacante Direito” deve se posicionar próximo à trave direita do gol adversário.
- RF59: Ao receber o comando “Normal Start”, o robô que está na posição “Atacante Direito” deve se posicionar próximo à trave direita do gol adversário caso não seja o robô da equipe que está sendo controlada mais próximo à bola.

APÊNDICE C – CONJUNTO DE *RULES* APLICADAS À SOLUÇÃO DESENVOLVIDA SOB O VIÉS DO PON

Este apêndice apresenta o conjunto de *Rules* que foram elaboradas e aplicadas às aplicações PON de controle para uma partida de futebol de robôs utilizando diagrama de *Rules* e a sintaxe definida pela LingPON.

C.1 DIAGRAMA DE OBJETOS PON

A Figura 38 apresenta a notação proposta por [Kossoski et al. 2014] para representar os objetos PON. Um FBE está representado como um objeto retangular. O Attribute é representado como um triângulo. A Premise é representada como um losango simbolizando uma decisão. Uma Rule é representada com um objeto que desencadeia um fluxo de execução em um sentido. Considerou-se Condition como parte integrante da respectiva Rule. O Method é representado como uma engrenagem. O sentido das notificações é representado como uma seta [Kossoski et al. 2014].

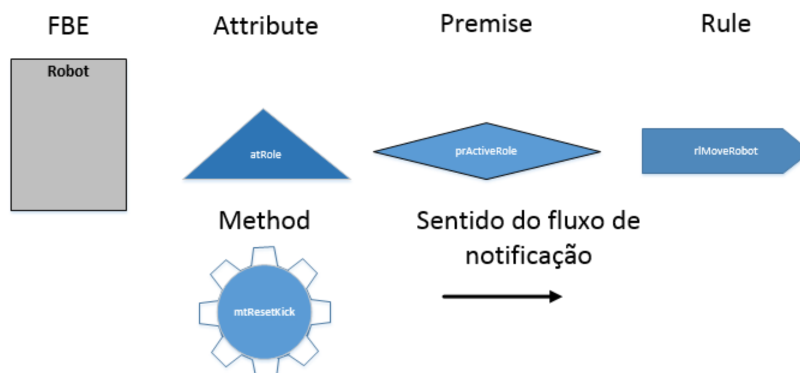

Figura 38: Elementos do diagrama de objetos PON [Kossoski et al. 2014].

Para facilitar o entendimento dos diagrama de objetos PON apresentados, a Tabela 3 apresenta a definição de todas as *Premises* referenciadas pelos diagramas de objetos PON.

Tabela 3: Definição das *Premises* utilizadas pelas *Rules* na aplicação de controle de futebol de robôs.

Premise	Código da Premise
prRobotMoveX	Robot.atPosX != Robot.atPosToGoX
prRobotMoveY	Robot.atPosY != Robot.atPosToGoY
prAngleMove	Robot.atAngle != Robot.atAngleToGo
prRobotIsReady	Robot.atIsReady == TRUE
prRobotIsNotReady	Robot.atIsReady == FALSE
prBallEnemyField	Robot.atBallEnemyField == TRUE
prBallTeamField	Robot.atBallEnemyField == FALSE
prClosestToBall	Robot.atClosestToBall == TRUE
prNotClosestToBall	Robot.atClosestToBall == FALSE
prEnemyOnLineGoal	Robot.atEnemyOnGoalLine == TRUE
prNoEnemyOnLineGoal	Robot.atEnemyOnGoalLine == FALSE
prBallIsClose	Robot.atDistanceToBall $<$ 300.0
prBallIsFar	Robot.atDistanceToBall $\geq$ 300.0
prBallCloseTeamGoal	Robot.atBallDistanceToTeamGoal $\leq$ 1800.0
prBallFarTeamGoal	Robot.atBallDistanceToTeamGoal $>$ 1800.0
prBallInsideGoalArea	Robot.atBallDistanceToTeamGoal $\leq$ 800
prBallNotInsideGoalArea	Robot.atBallDistanceToTeamGoal $>$ 800
prRefereeCmdStop	Robot.atRefereeCmd == ‘S’
prRefereeCmdKickoffBlue	Robot.atRefereeCmd == ‘K’
prRefereeCmdKickoffYellow	Robot.atRefereeCmd == ‘k’
prRefereeCmdStartGame	Robot.atRefereeCmd == ‘ ’
prRefereeCmdDirectKickBlue	Robot.atRefereeCmd == ‘F’
prRefereeCmdDirectKickYellow	Robot.atRefereeCmd == ‘f’
prRefereeCmdPenaltyBlue	Robot.atRefereeCmd == ‘P’
prRefereeCmdPenaltyYellow	Robot.atRefereeCmd == ‘p’
prRefereeCmdIndirectKickBlue	Robot.atRefereeCmd == ‘I’
prRefereeCmdIndirectKickYellow	Robot.atRefereeCmd == ‘i’
prLastRefereeCmdKickoffBlue	Robot.atRefereeCmd == ‘I’
prLastRefereeCmdKickoffYellow	Robot.atRefereeCmd == ‘i’
prLastRefereeCmdPenaltyBlue	Robot.atLastRefereeCmd == ‘P’
prLastRefereeCmdPenaltyYellow	Robot.atLastRefereeCmd == ‘p’
prActiveRole	Robot.atRole != “ ”
prLinePlayerRole	Robot.atRole != “GOALKEEPER”
prRoleGoalkeeper	Robot.atRole == “GOALKEEPER”
prRoleDefenderLeft	Robot.atRole == “DEFENDER_LEFT”
prRoleDefenderRight	Robot.atRole == “DEFENDER_RIGHT”
prRoleMidfieldOnly	Robot.atRole == “MIDFIELD_ONLY”
prRoleStrickerLeft	Robot.atRole == “STRIKER_LEFT”
prRoleStrickerRight	Robot.atRole == “STRIKER_RIGHT”
prTeamLeftSide	Robot.atTeamSide = “LEFT”
prTeamRightSide	Robot.atTeamSide = “RIGHT”
prTeamBlue	Robot.atTeamColor == “BLUE”
prTeamYellow	Robot.atTeamColor == “YELLOW”
prFreePartner	Robot.atPartnerFreeID $\geq$ 0
prNoFreePartner	Robot.atPartnerFreeID $<$ 0

A seguir, todas as *Rules* são apresentadas segundo o diagrama de objetos PON.

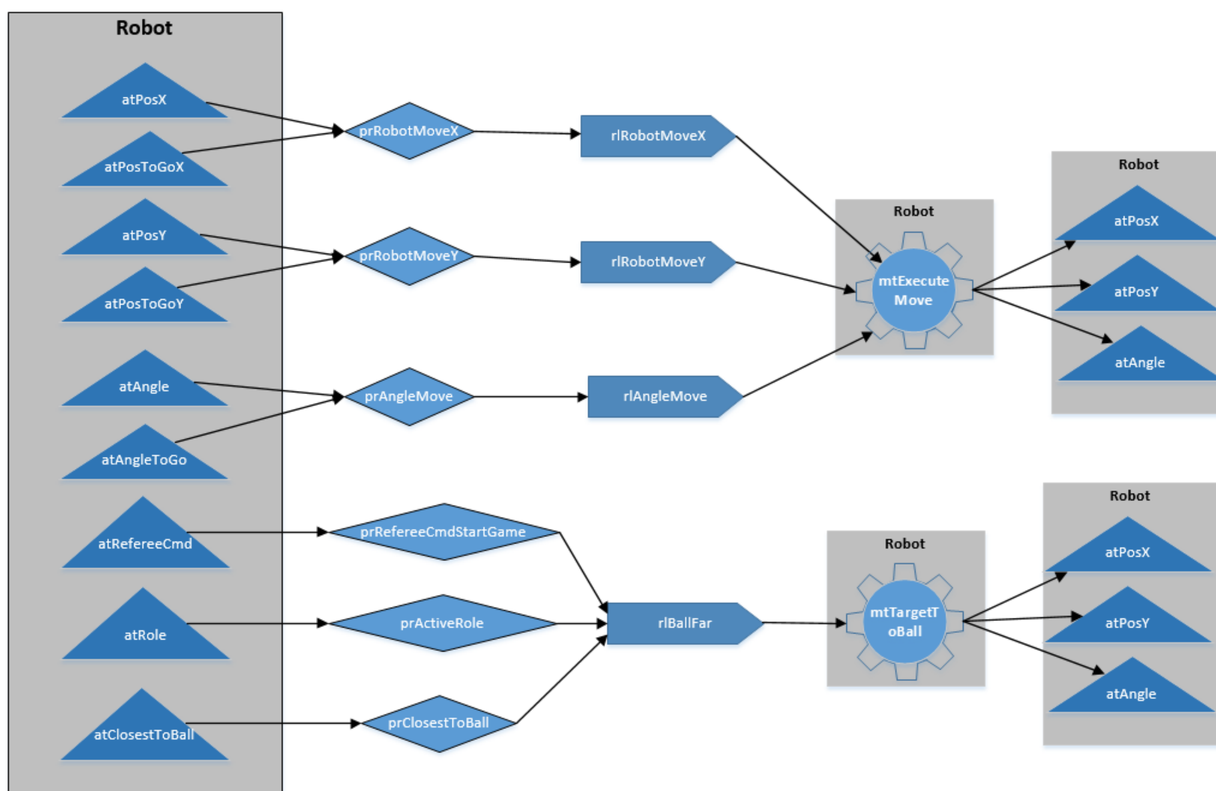

Figura 39: Diagrama de objetos PON das *Rules* rlRobotMoveX, rlRobotMoveY, rlAngle-Move e rlBallFar.

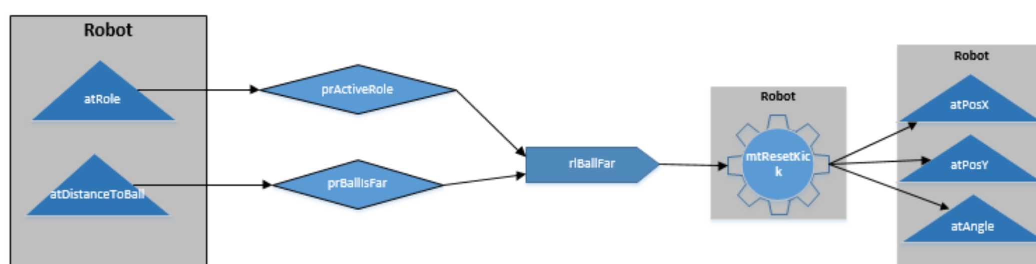

Figura 40: Diagrama de objetos PON da *Rule* rlStartTargetToBall.

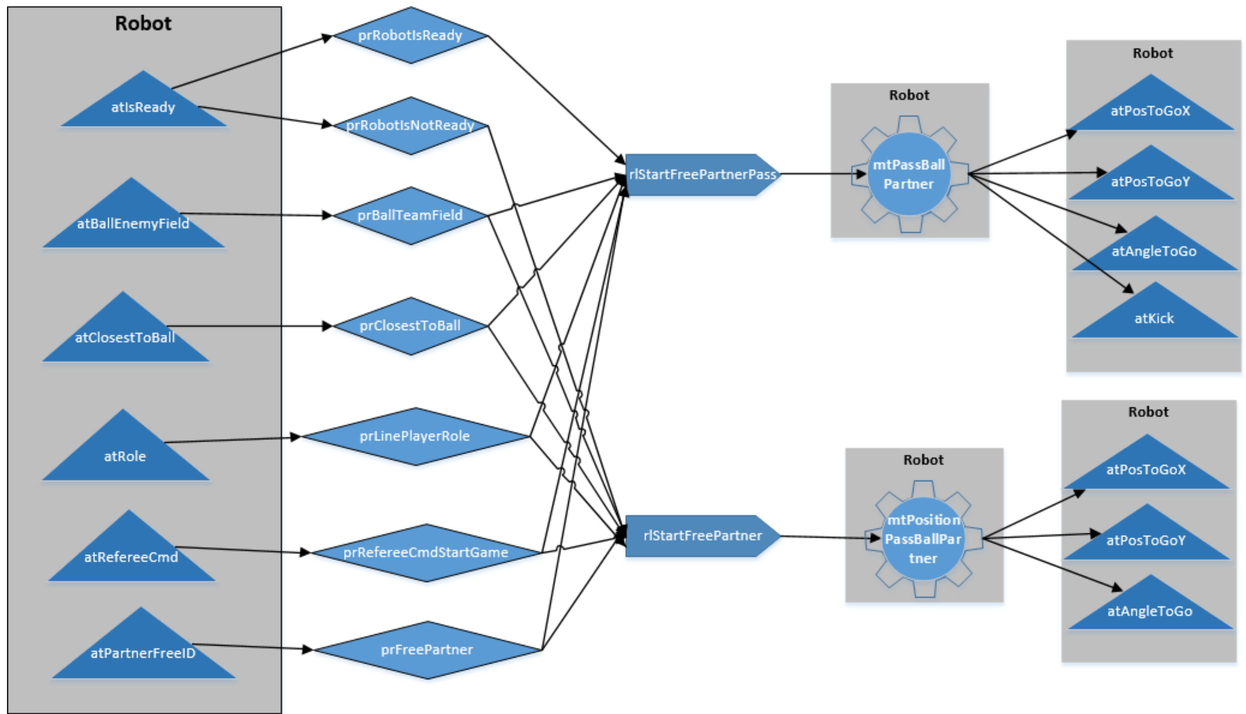

Figura 41: Diagrama de objetos PON das *Rules* rlStartFreePartner e rlStartFreePartner-Pass.

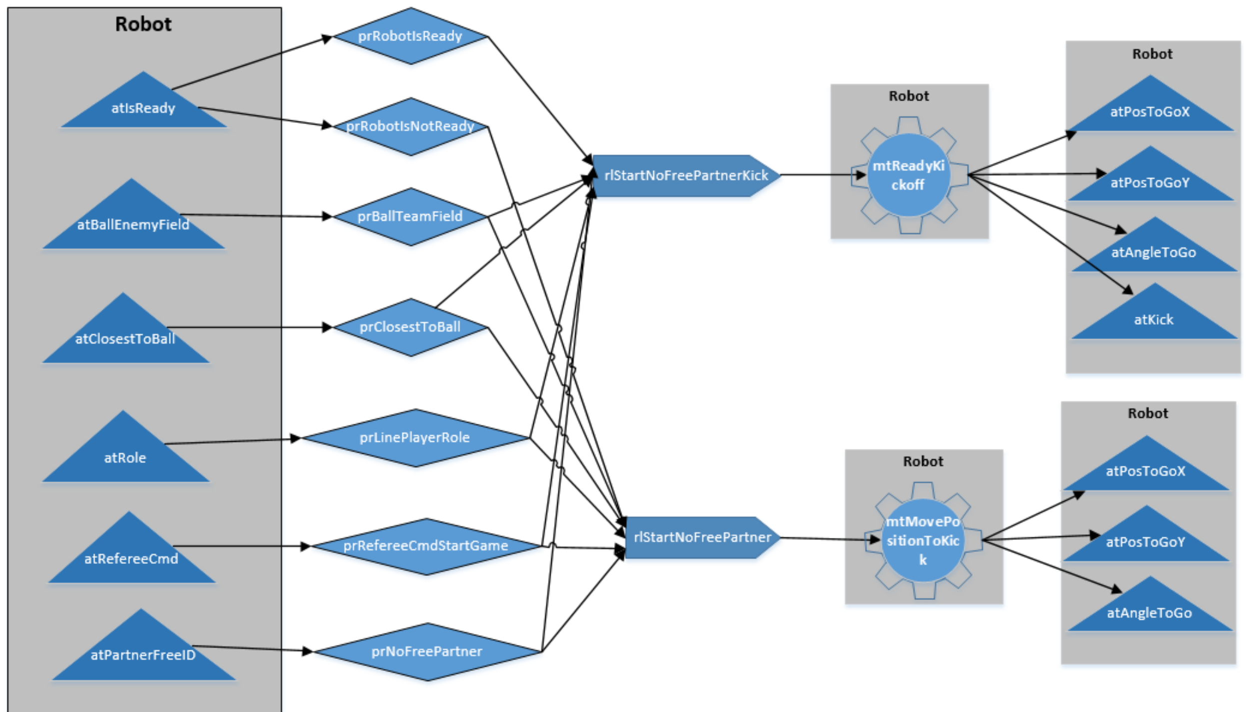

Figura 42: Diagrama de objetos PON das *Rules* rlStartNoFreePartner e rlStartNoFree-PartnerKick.

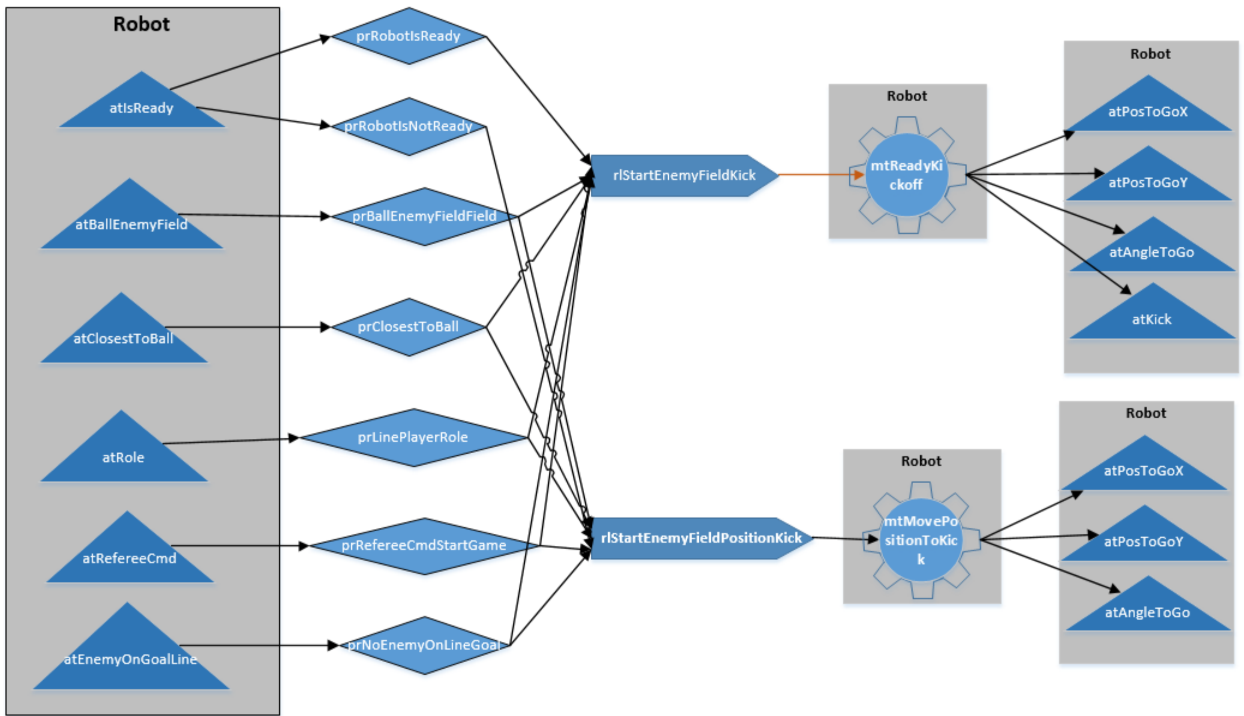

Figura 43: Diagrama de objetos PON das *Rules* rlStartEnemyPositionKick e rlStartEnemyFieldKick

.

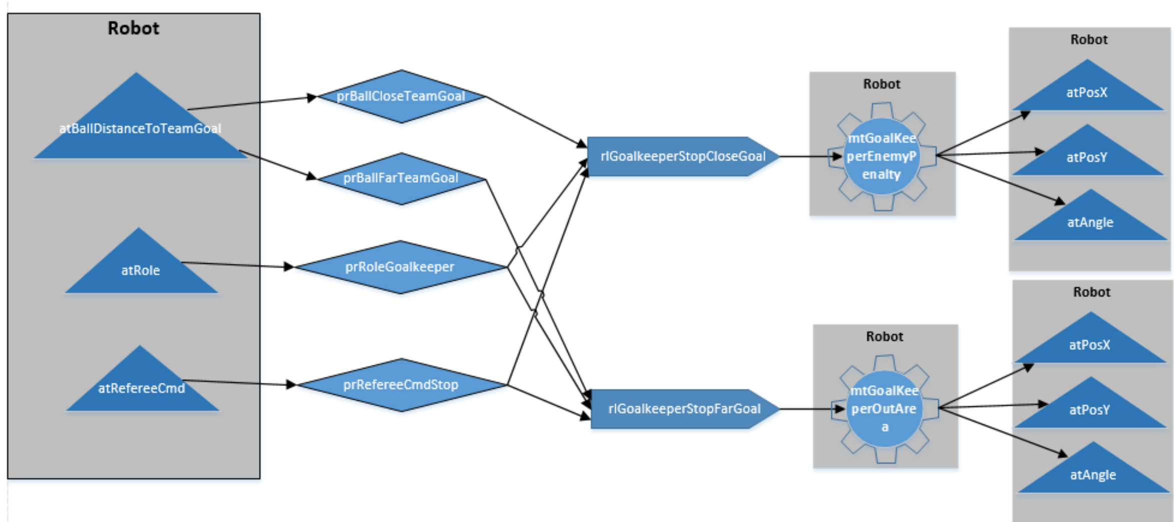

Figura 44: Diagrama de objetos PON das *Rules* rlGoalkeeperStopCloseGoal e rlGoalKeeperStopFarGoal.

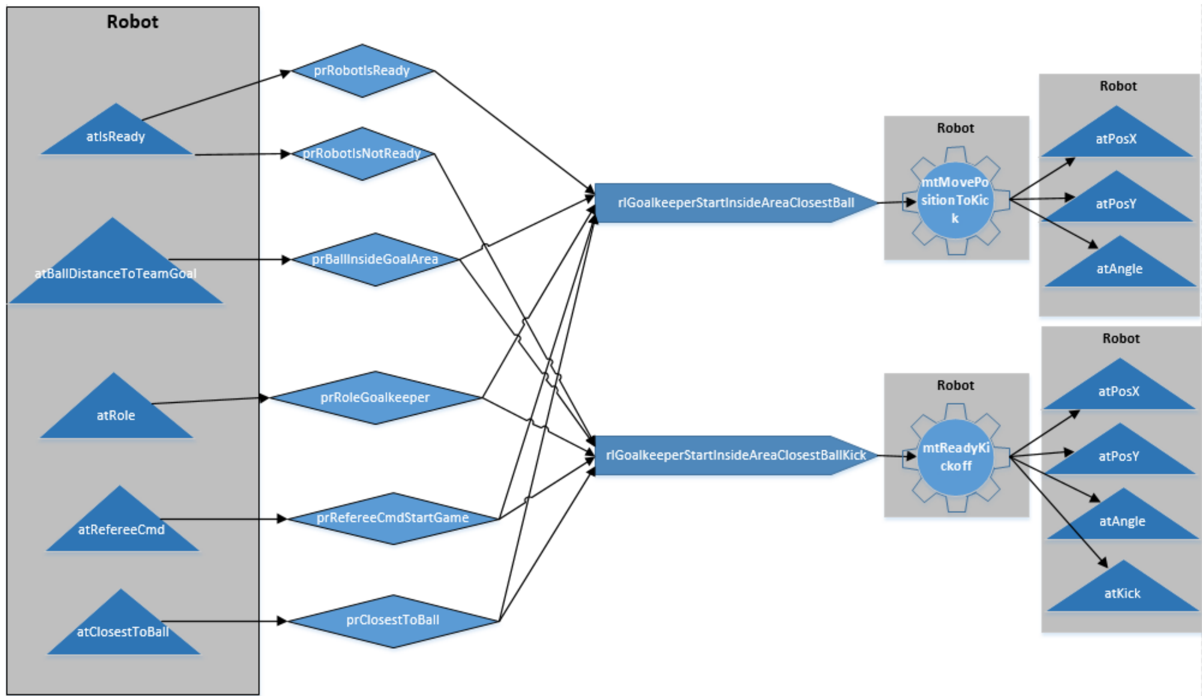

Figura 45: Diagrama de objetos PON das *Rules* rlGoalkeeperStartInsideAreaClosestBall e rlGoalkeeperStartInsideAreaClosestBallKick.

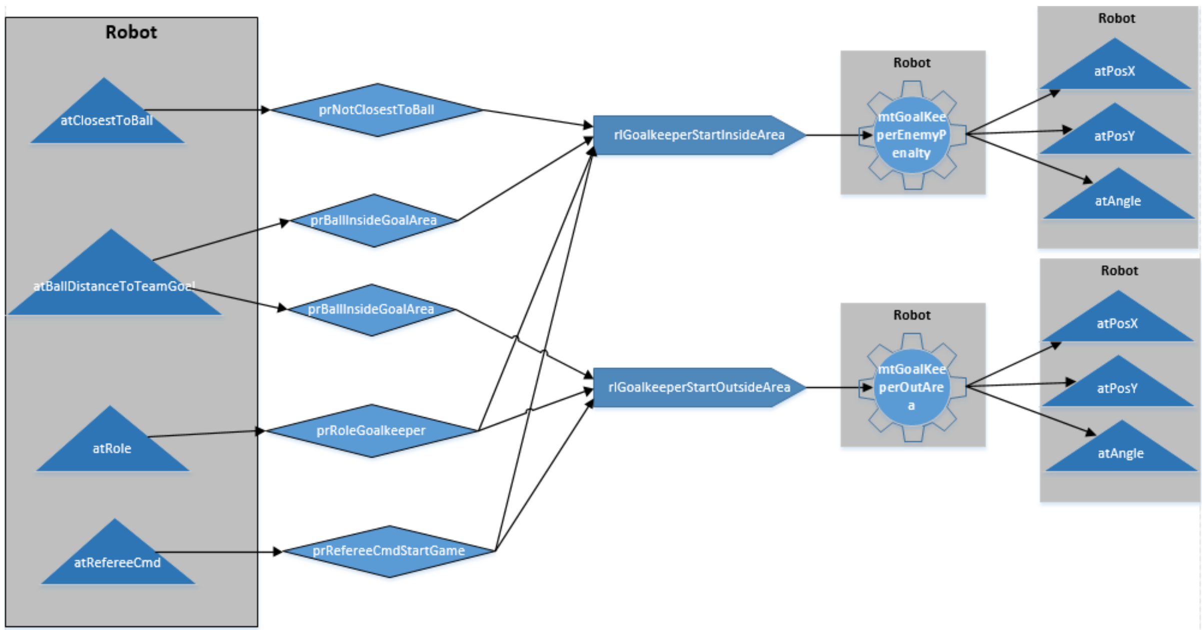

Figura 46: Diagrama de objetos PON das *Rules* rlGoalkeeperStartInsideArea e rlGoalkeeperStartOutsideArea.

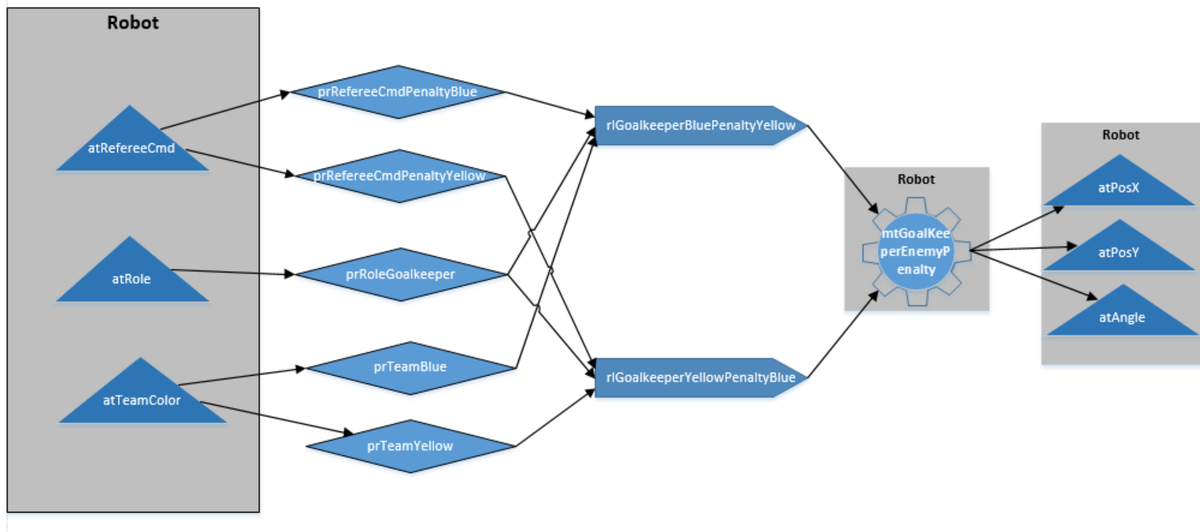

Figura 47: Diagrama de objetos PON das *Rules* rlGoalkeeperBluePenaltyYellow e rlGoalkeeperYellowPenaltyBlue.

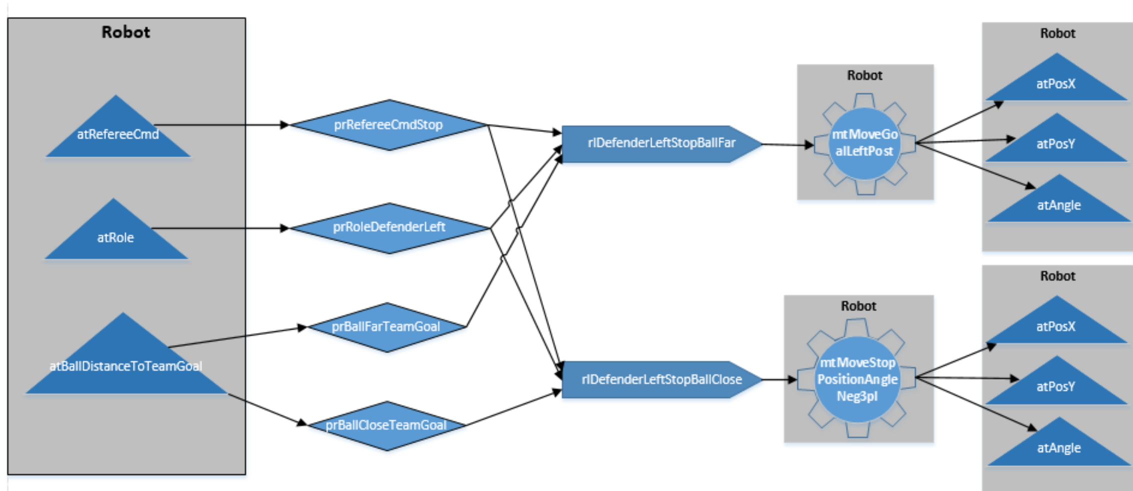

Figura 48: Diagrama de objetos PON das *Rules* rlDefenderLeftStopBallFar e rlDefender-LeftStopBallClose.

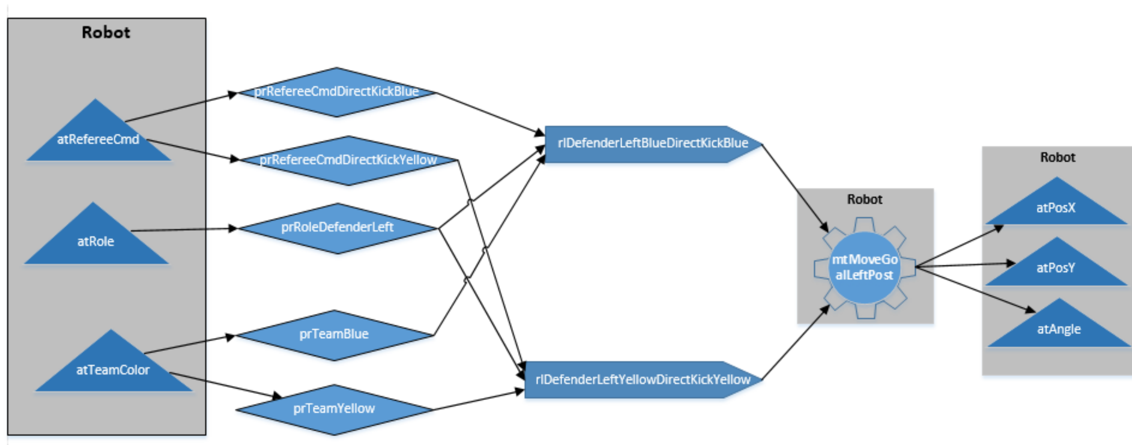

Figura 49: Diagrama de objetos PON das *Rules* rlDefenderLeftBlueDirectKickBlue e rlDefenderLeftYellowDirectKickYellow.

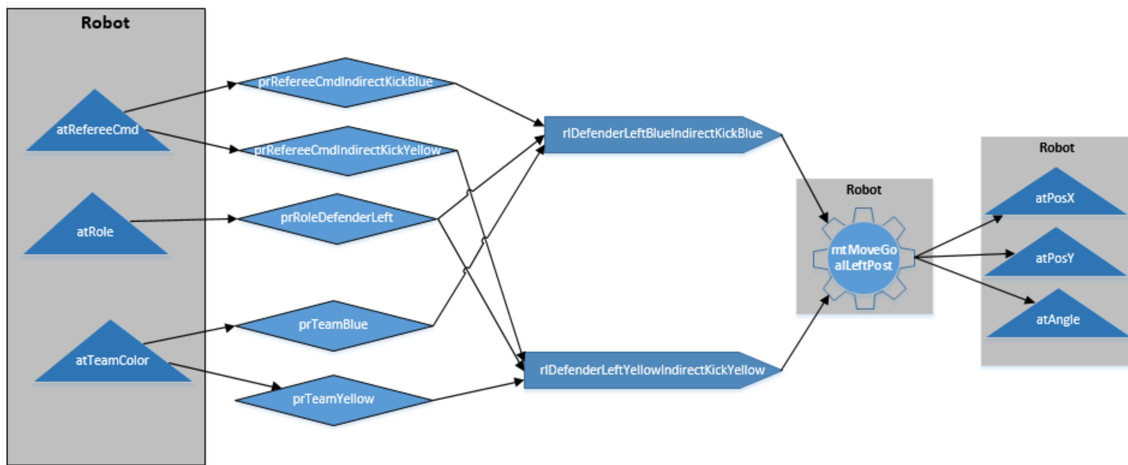

Figura 50: Diagrama de objetos PON das *Rules* rlDefenderLeftBlueIndirectKickBlue e rlDefenderLeftYellowIndirectKickYellow.

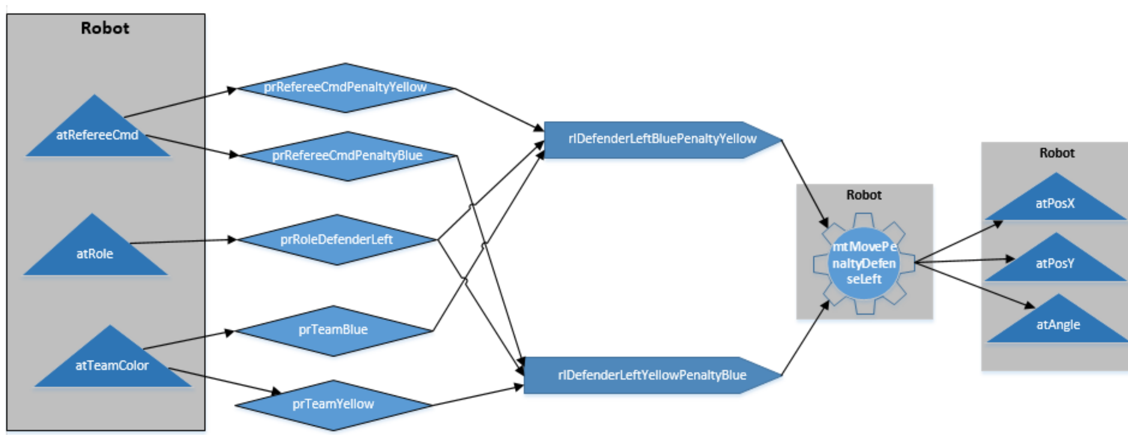

Figura 51: Diagrama de objetos PON das *Rules* rlDefenderLeftBluePenaltyYellow e rlDefenderLeftYellowPenaltyBlue.

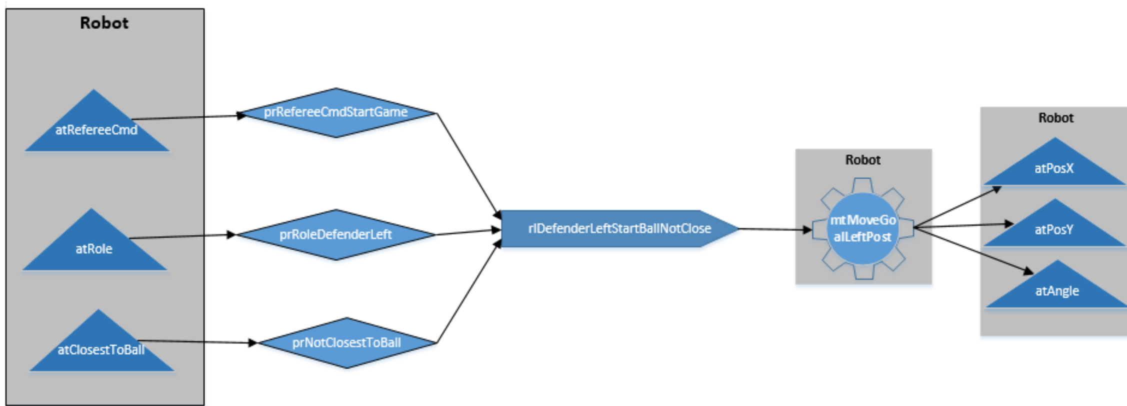

Figura 52: Diagrama de objetos PON da *Rule* rlDefenderLefStartBallNotClose.

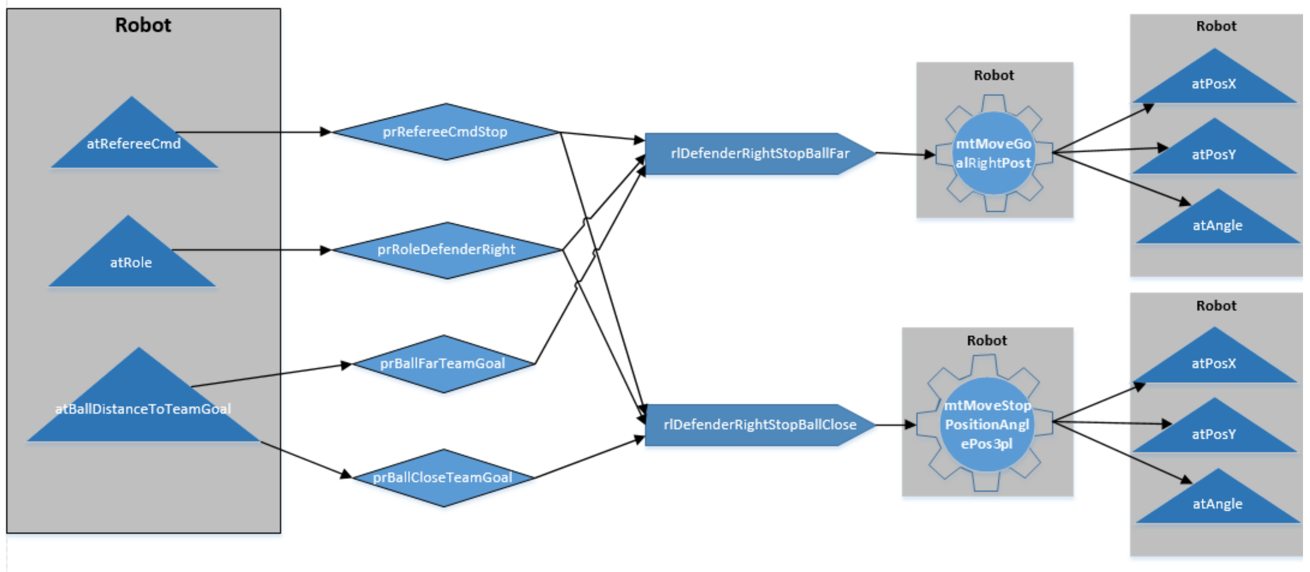

Figura 53: Diagrama de objetos PON das *Rules* rlDefenderRightStopBallFar e rlDefenderRightStopBallClose.

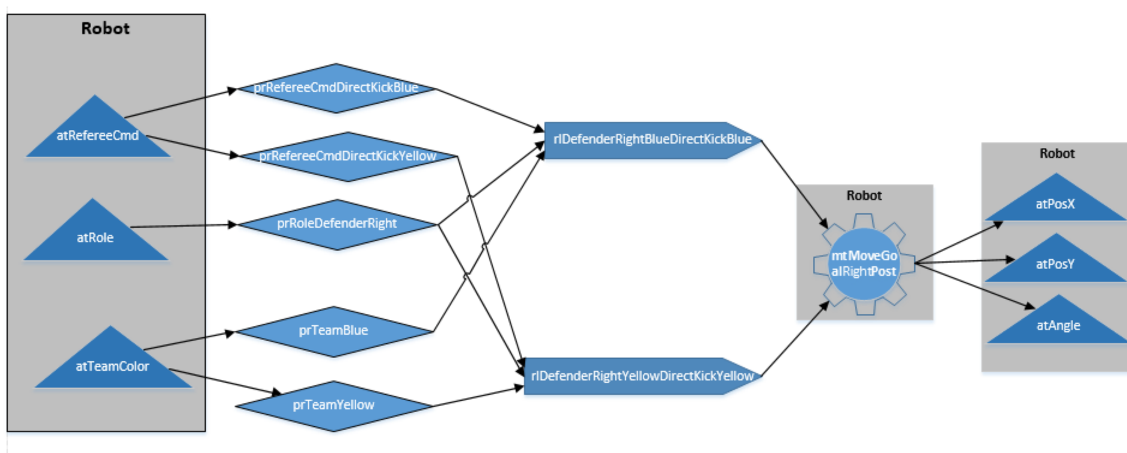

Figura 54: Diagrama de objetos PON das *Rules* rlDefenderRightBlueDirectKickBlue e rlDefenderRightYellowDirectKickYellow.

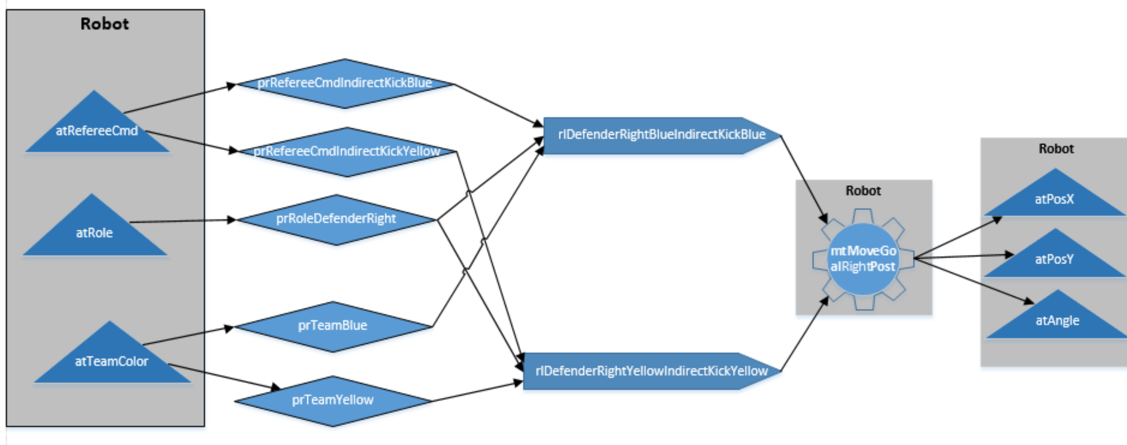

Figura 55: Diagrama de objetos PON das *Rules* rlDefenderRightBlueIndirectKickBlue e rlDefenderRightYellowIndirectKickYellow.

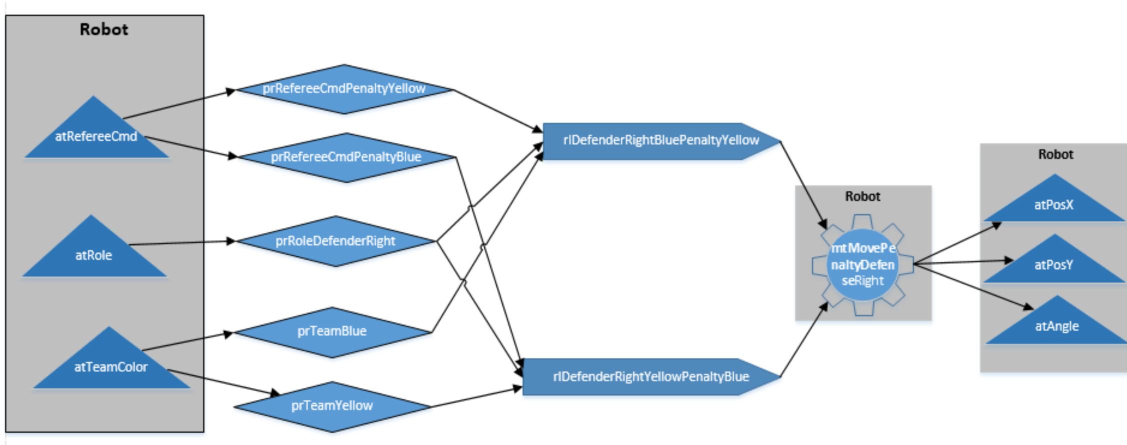

Figura 56: Diagrama de objetos PON das *Rules* rlDefenderRightBluePenaltyYellow e rlDefenderRightYellowPenaltyBlue.

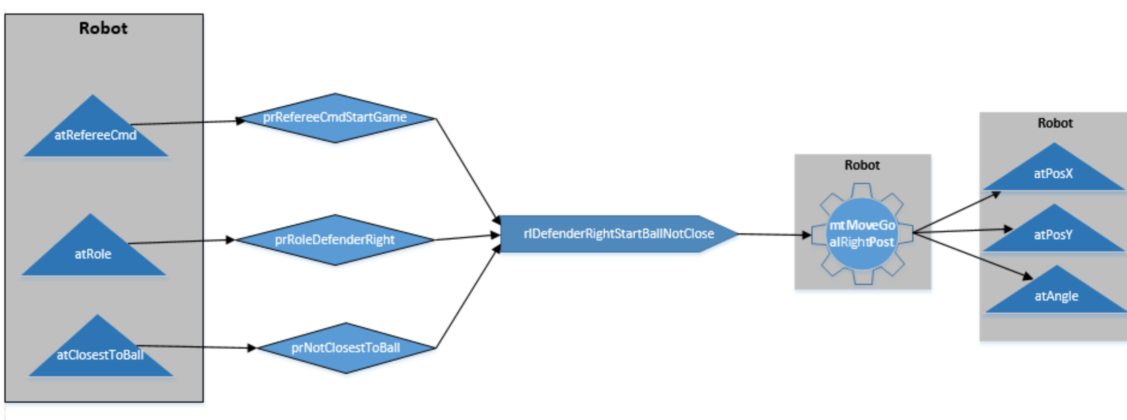

Figura 57: Diagrama de objetos PON da *Rule* rlDefenderRightStartBallNotClose.

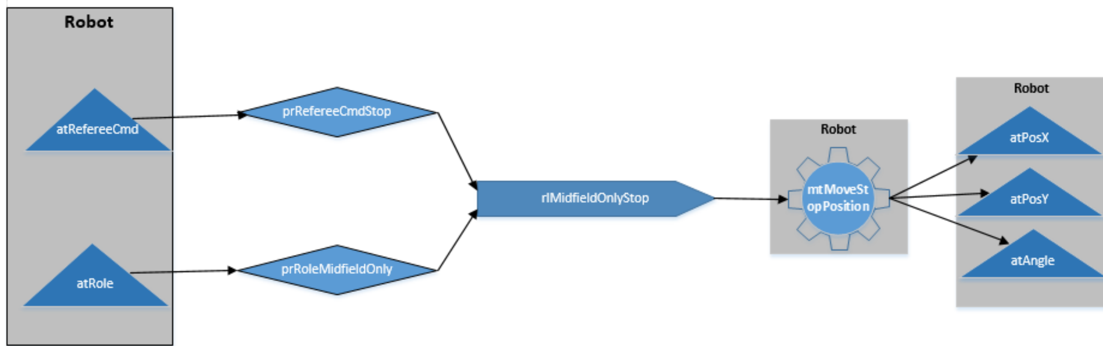

Figura 58: Diagrama de objetos PON da *Rule* rlMidfieldOnlyStop.

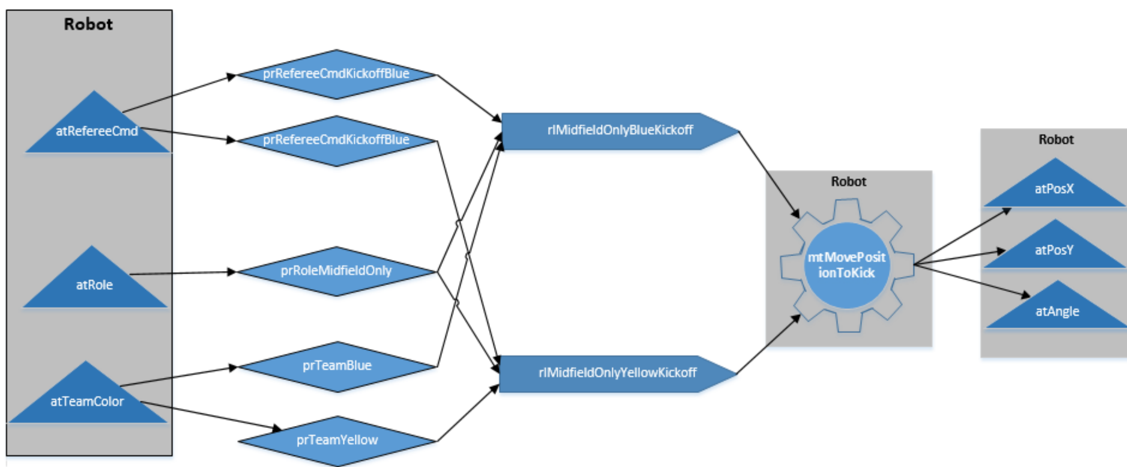

Figura 59: Diagrama de objetos PON das *Rules* rlMidfieldOnlyBlueKickoff e rlMidfieldOnlyYellowKickoff.

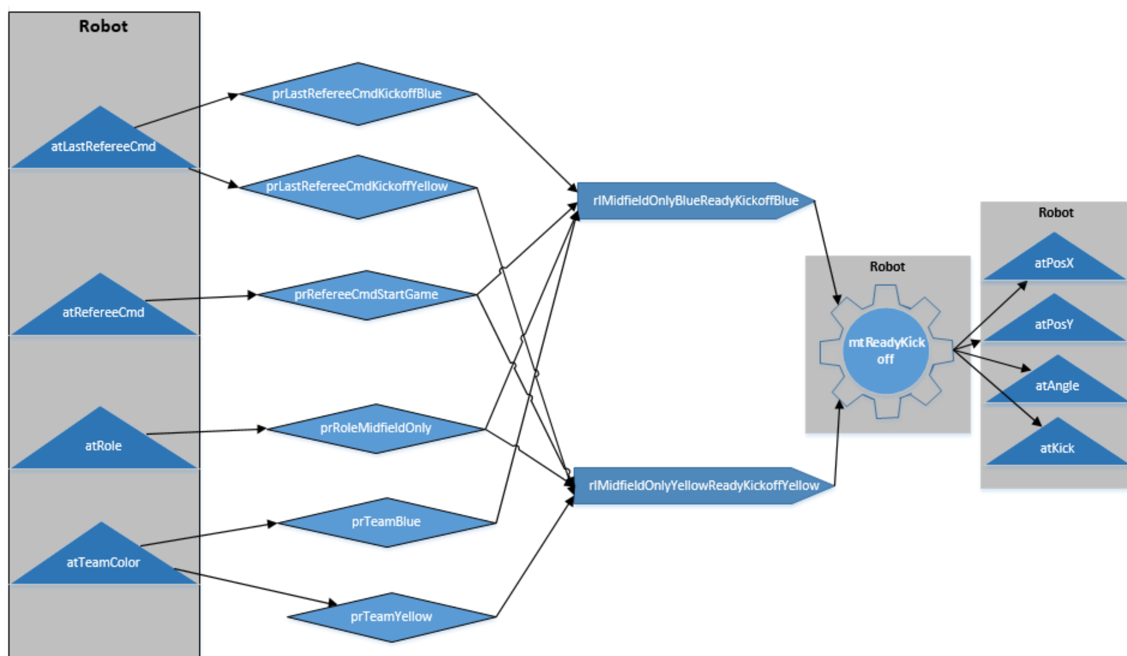

Figura 60: Diagrama de objetos PON das *Rules* rlMidfieldOnlyBlueReadyKickoffBlue e rlMidfieldOnlyYellowReadyKickoffYellow.

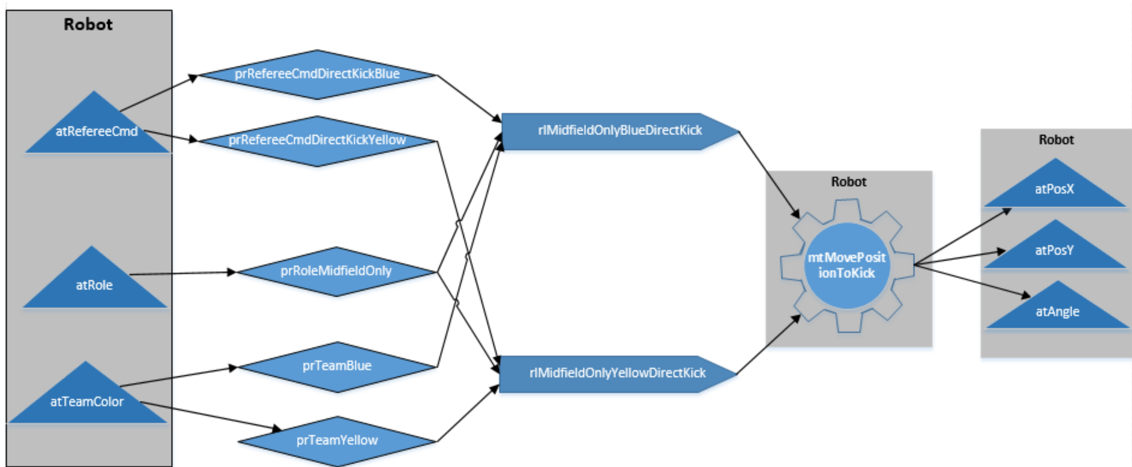

Figura 61: Diagrama de objetos PON das *Rules* rlMidfieldOnlyBlueDirectKick e rlMidfieldOnlyYellowDirectKick.

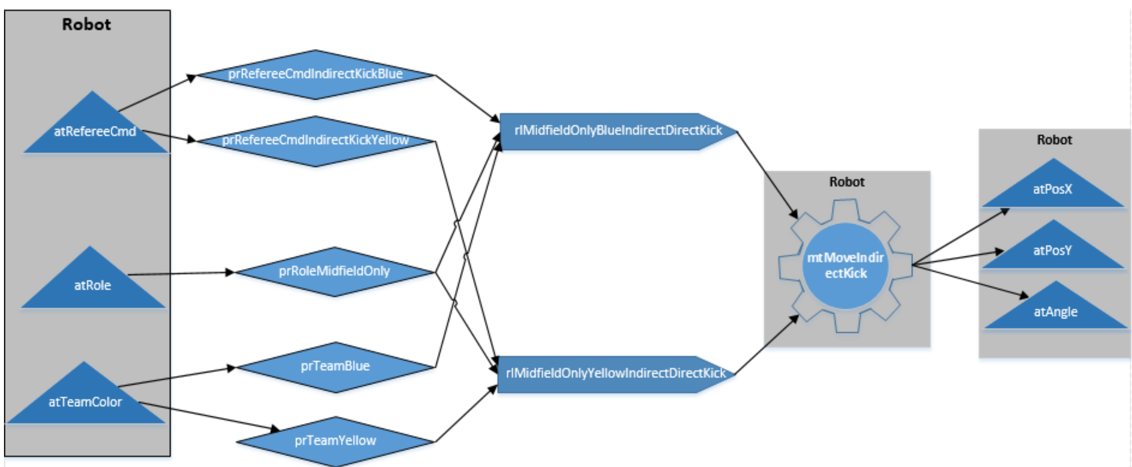

Figura 62: Diagrama de objetos PON das *Rules* rlMidfieldOnlyBlueIndirectKick e rlMidfieldOnlyYellowIndirectKick.

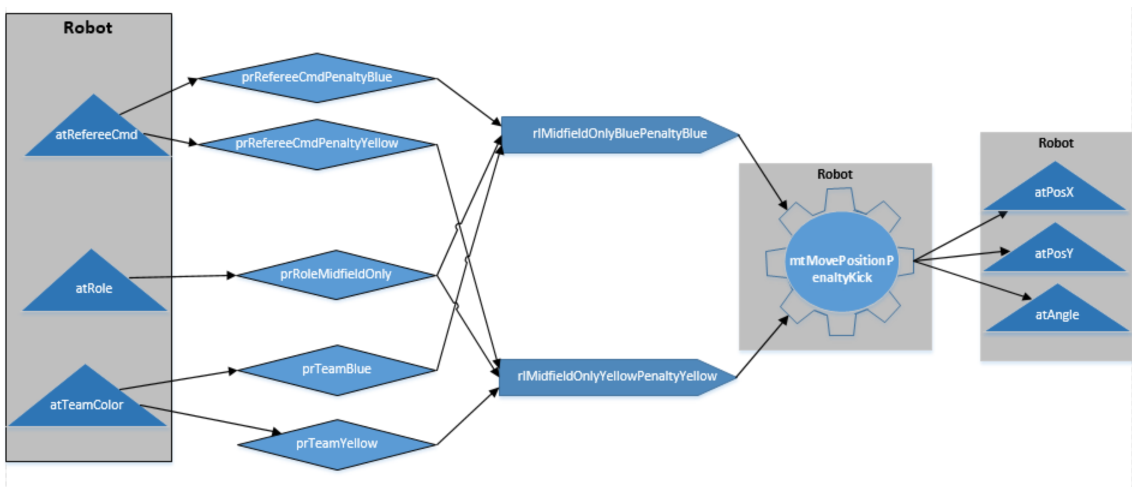

Figura 63: Diagrama de objetos PON das *Rules* rlMidfieldOnlyBluePenaltyBlue e rlMidfieldOnlyYellowPenaltyYellow.

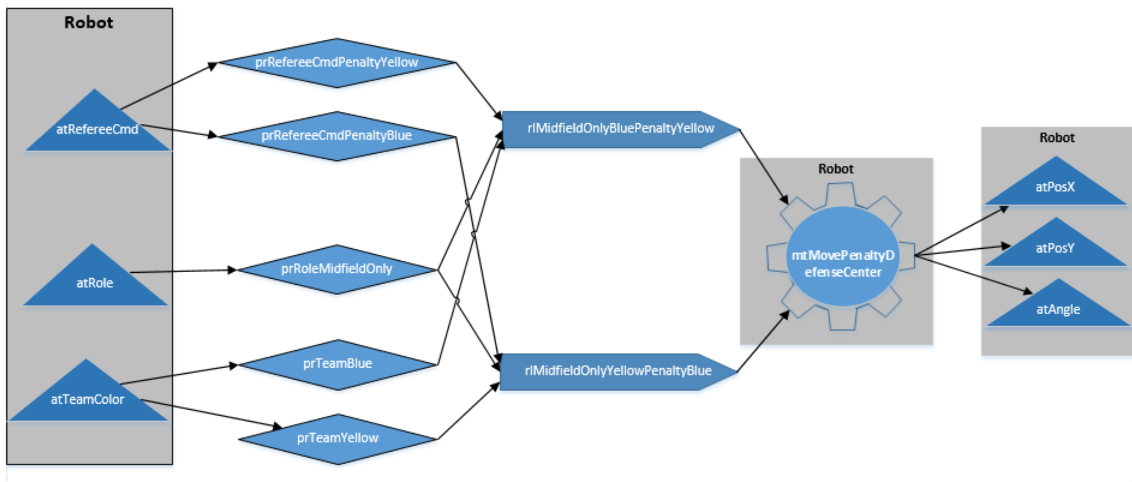

Figura 64: Diagrama de objetos PON das *Rules* rlMidfieldOnlyBluePenaltyYellow e rlMidfieldOnlyYellowPenaltyBlue.

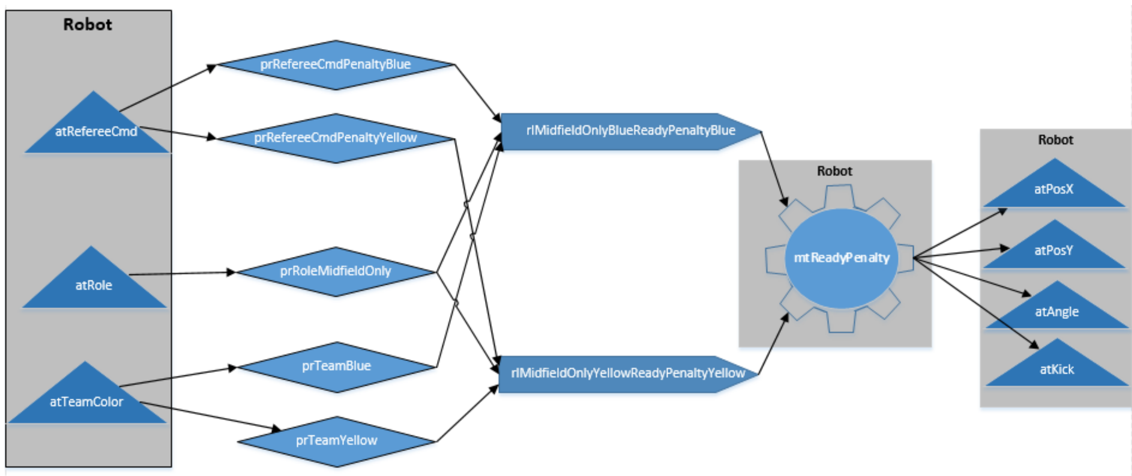

Figura 65: Diagrama de objetos PON das *Rules* rlMidfieldOnlyBlueReadyPenaltyBlue e rlMidfieldOnlyYellowReadyPenaltyYellow.

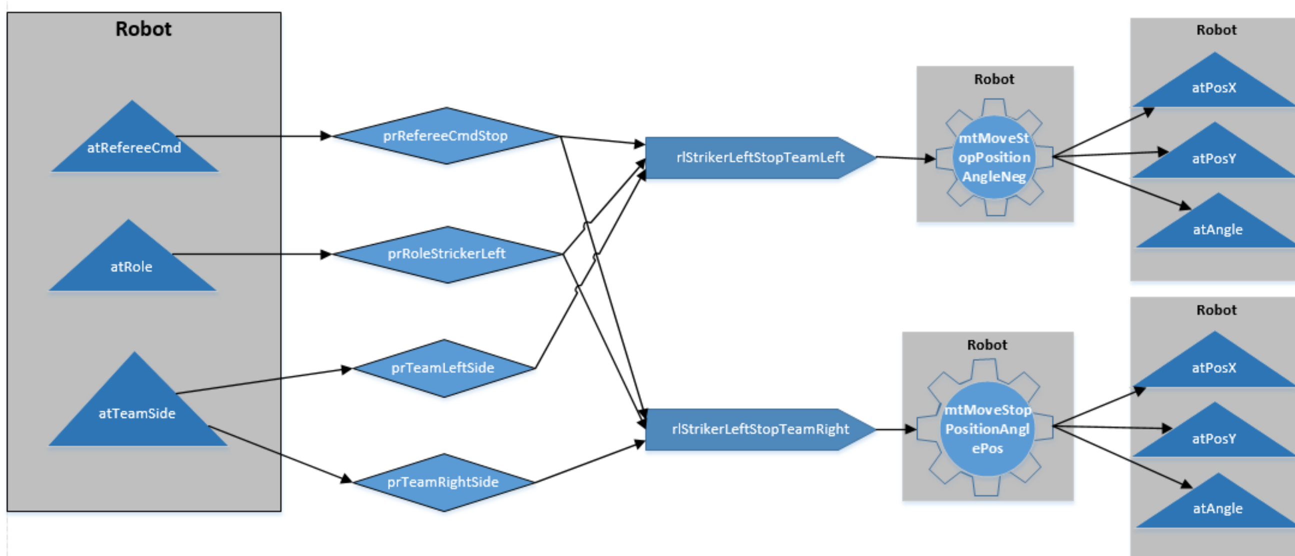

Figura 66: Diagrama de objetos PON das *Rules* rlStrikerLeftStopTeamLeft e rlStrikerLeftStopTeamRight.

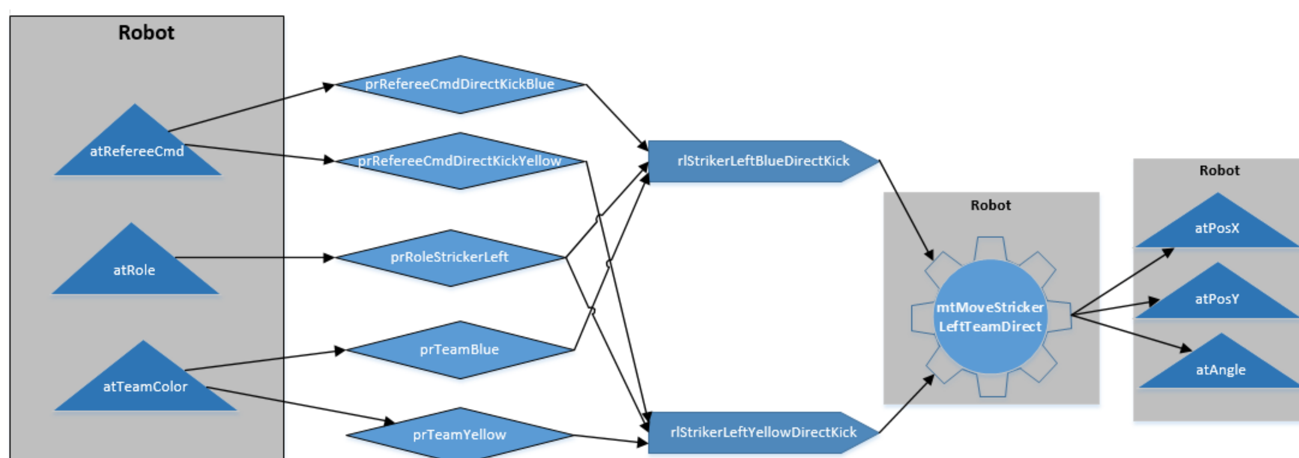

Figura 67: Diagrama de objetos PON das *Rules* rlStrikerLeftBlueDirectKick e rlStrikerLeftYellowDirectKick.

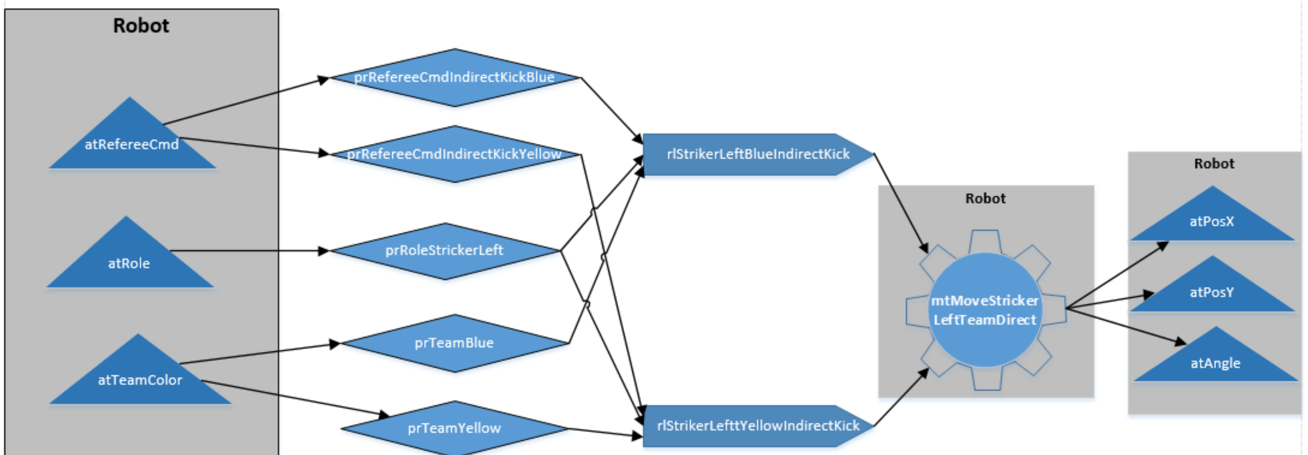

Figura 68: Diagrama de objetos PON das *Rules* rlStrikerLeftBlueIndirectKick e rlStrikerLeftYellowIndirectKick.

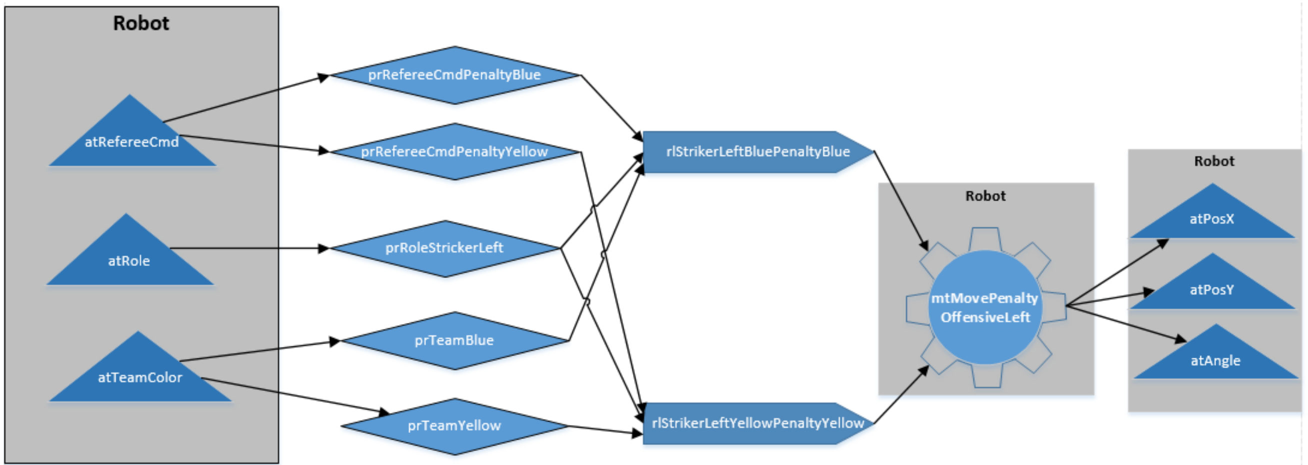

Figura 69: Diagrama de objetos PON das *Rules* rlStrikerLeftBluePenaltyBlue e rlStrikerLeftYellowPenaltyYellow.

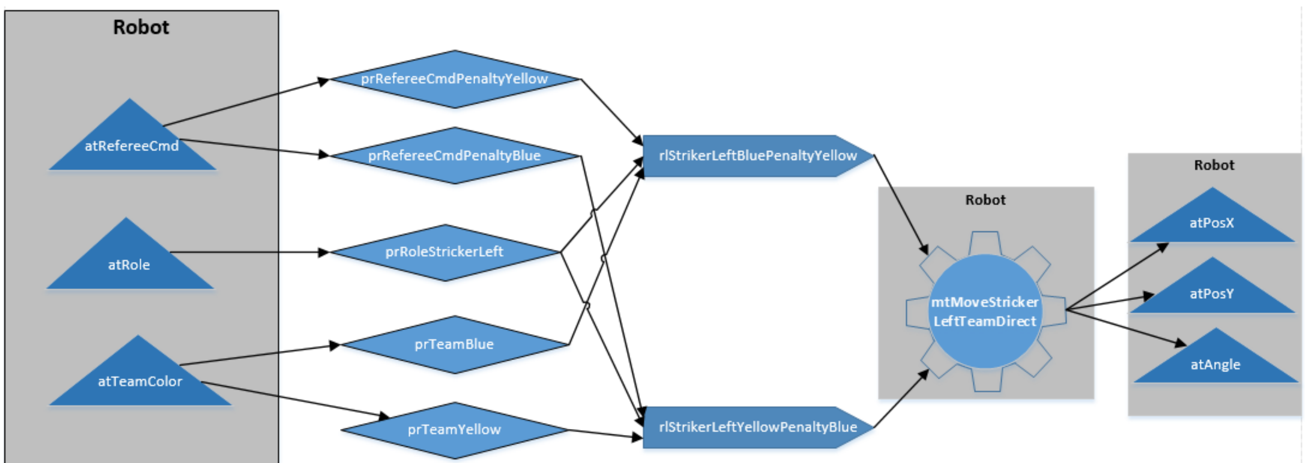

Figura 70: Diagrama de objetos PON das *Rules* rlStrikerLeftBluePenaltyYellow e rlStrikerLeftYellowPenaltyBlue.

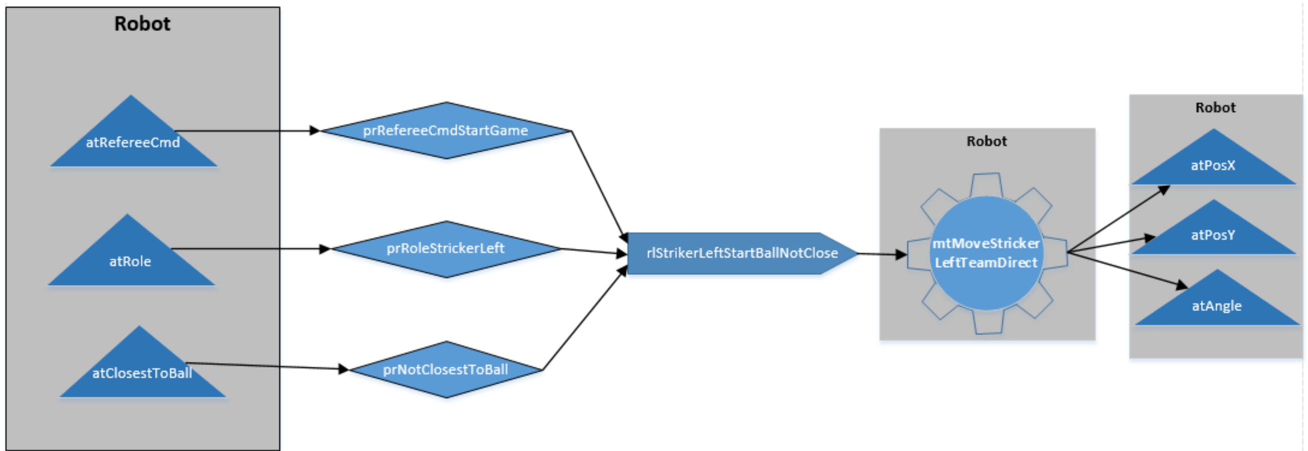

Figura 71: Diagrama de objetos PON da *Rule* rlStrikerLeftStartBallNotClose.

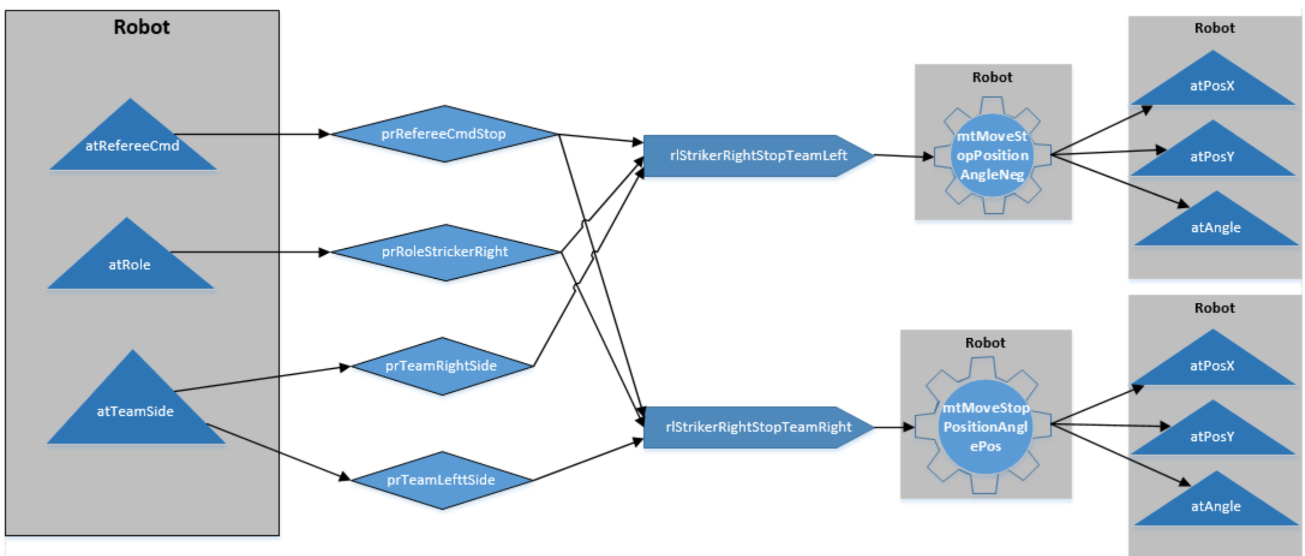

Figura 72: Diagrama de objetos PON das *Rules* rlStrikerRightStopTeamLeft e rlStrikerRightStopTeamRight.

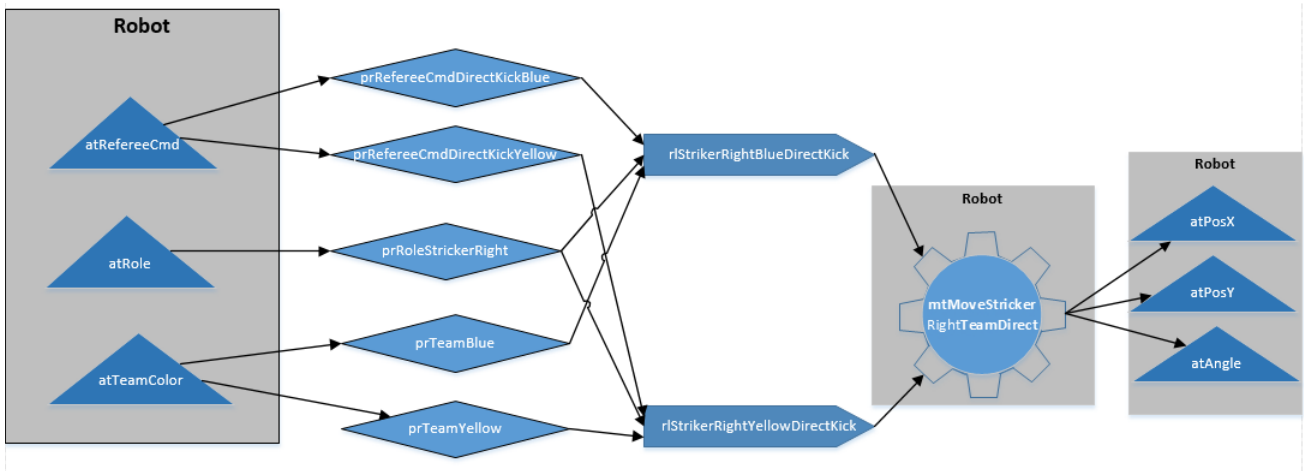

Figura 73: Diagrama de objetos PON das *Rules* rlStrikerRightBlueDirectKick e rlStrikerRightYellowDirectKick.

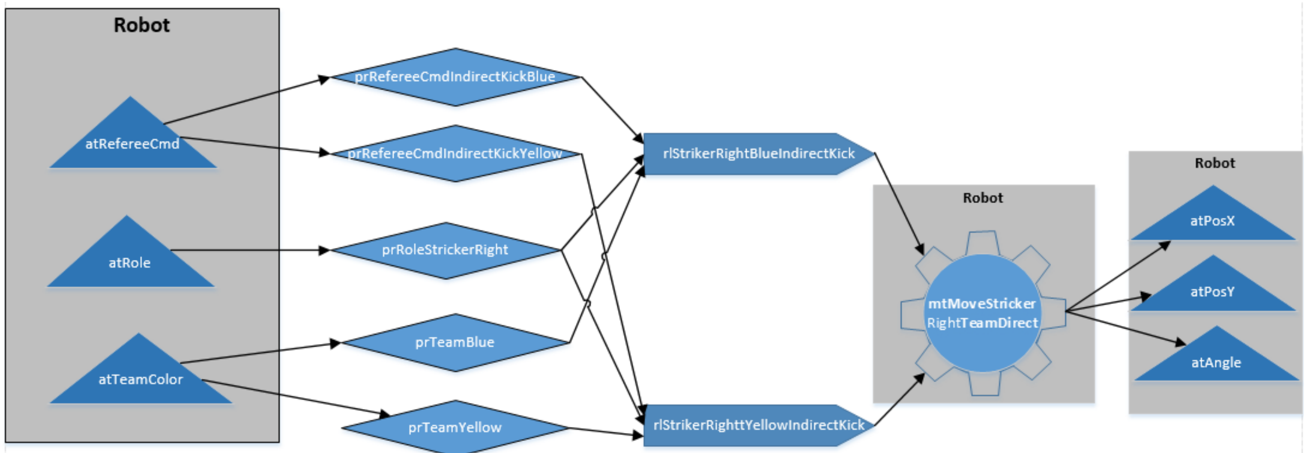

Figura 74: Diagrama de objetos PON das *Rules* rlStrikerRightBlueIndirectKick e rlStrikerRightYellowIndirectKick.

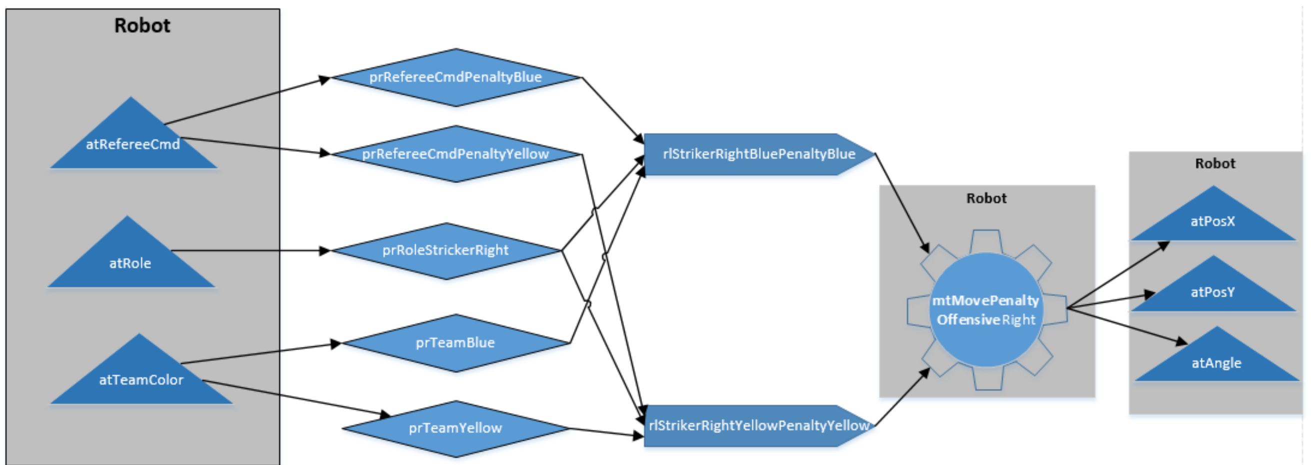

Figura 75: Diagrama de objetos PON das *Rules* rlStrikerRightBluePenaltyBlue e rlStrikerRightYellowPenaltyYellow.

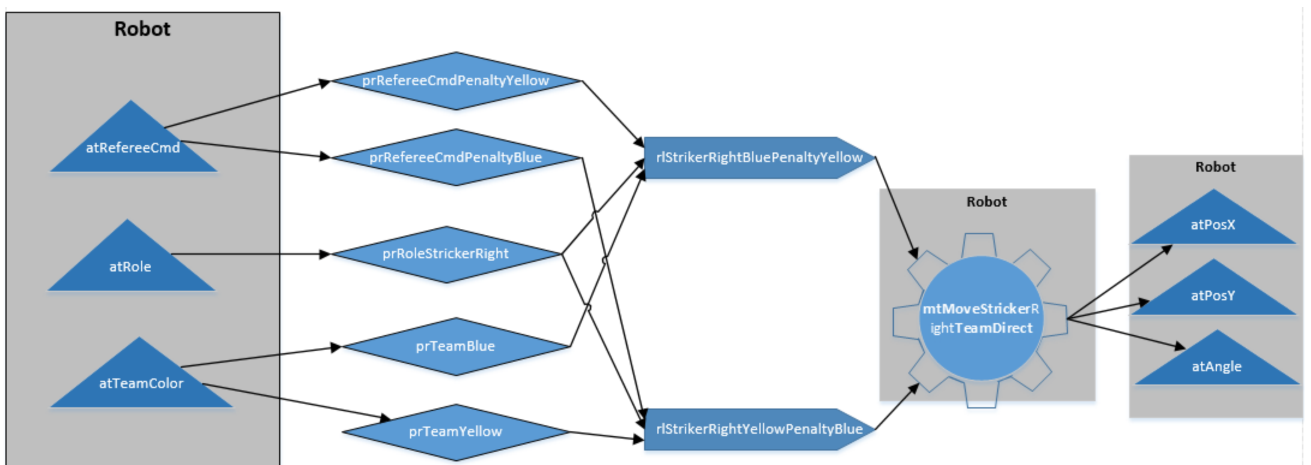

Figura 76: Diagrama de objetos PON das *Rules* rlStrikerRightBluePenaltyYellow e rlStrikerRightYellowPenaltyBlue.

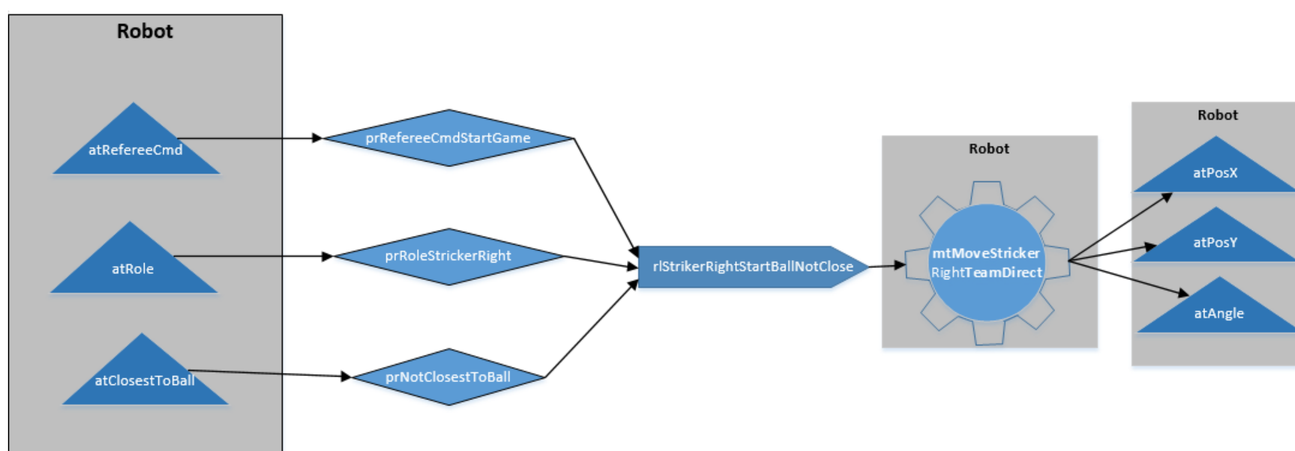

Figura 77: Diagrama de objetos PON das *Rules* rlStrikerRightStartBallNotClose.

C.2 CÓDIGO-FONTE DAS *RULES* DESENVOLVIDAS PARA O *SOFTWARE* PON

De forma a facilitar o entendimento para o leitor, a primeira *Rule* será apresentada em um formato textual e relacionada ao código LingPON referente.

A *Rule* rlMOBlueReadyKickoffBlue define que se a função do robô (atRole) for meio-campo (MIDFIELD_ONLY), o comando atual enviado pelo árbitro (atRefereeCmd) for “Inicie a jogada” (Start), o comando predecessor enviado pelo árbitro (atLastRefereeCmd) for posse de bola para o time azul (‘K’) e a cor do time do robô (atTeamColor) for azul (Blue) então o robô deve chutar a bola em direção ao gol adversário (Robot.mtReadyKickoff). Esta *Rule* é executada no início da partida no meio do campo (kickoff) pelo time Azul (Blue), ou após um gol do time adversário, no reinício da partida.

O código LingPON referente a essa *Rule* é apresentado pela *Rule* 1.

Rule 1: Código da *Rule* rlMOBlueReadyKickoffBlue.

```
1  fbeRule rlMOBlueReadyKickoffBlue
2   condition
3       premise prRoleMidfieldOnly Robot.atRole == “MIDFIELD_ONLY”
    and
4       premise prRefereeCmdStartGame Robot.atRefereeCmd == ‘ ’ and
5       premise prLastCmdKickoffBlue Robot.atLastRefereeCmd == ‘K’ and
6       premise prTeamBlue Robot.atTeamColor == “BLUE”
7   end_condition
8   action
9     instigation inMOBlueReadyKickoffBlue Robot.mtReadyKickoff();
10  end_action
11 end_fbeRule
```

As demais 73 *Rules* que compõem o conhecimento lógico-causal completo da aplicação de controle para uma partida de futebol de robôs são apresentadas na sequência, para fins de eventuais consultas, e por motivos de completude da descrição da aplicação.

Rule 2: Código da *Rule* rlRobotMoveX.

```
1 fbeRule rlRobotMoveX
2  condition
3      premise prRobotMoveX Robot.atPosX != Robot.atPosToGoX
4  end_condition
5  action
6    instigation inMoveX Robot.mtExecuteMove();
7  end_action
8 end_fbeRule
```

Rule 3: Código da *Rule* rlRobotMoveY.

```
1 fbeRule rlRobotMoveY
2  condition
3      premise prRobotMoveY Robot.atPosY != Robot.atPosToGoY
4  end_condition
5  action
6    instigation inMoveY Robot.mtExecuteMove();
7  end_action
8 end_fbeRule
```

Rule 4: Código da *Rule* rlAngleMove.

```
1 fbeRule rlAngleMove
2  condition
3      premise prAngleMove Robot.atAngle != Robot.atAngleToGo;
4  end_condition
5  action
6    instigation inAngleMove Robot.mtAngleMove();
7  end_action
8 end_fbeRule
```

Rule 5: Código da *Rule* rlBallFar.

```
1  fbeRule rlBallFar
2   condition
3       premise prBallIsFar Robot.atDistanceToBall >= 300 and
4       premise prActiveRole Robot.atRole != " " and
5   end_condition
6   action
7     instigation inReset Robot.mtResetKick();
8   end_action
9  end_fbeRule
```

Rule 6: Código da *Rule* rlStartFreePartner.

```
1  fbeRule rlStartFreePartner
2   condition
3       premise prRefereeCmdStartGame Robot.atRefereeCmd == ' ' and
4       premise prLinePlayerRole Robot.atRole != "GOALKEEPER" and
5       premise prBallTeamField Robot.atBallEnemyField == false and
6       premise prClosestToBall Robot.atClosestToBall == true and
7       premise prFreePartner Robot.atPartnerFreeID >= 0 and
8       premise prRobotIsNotReady Robot.atIsReady == false
9   end_condition
10  action
11    instigation inPosPassBallPartner Robot.mtPosPassBall();
12  end_action
13 end_fbeRule
```

Rule 7: Código da *Rule* rlStartFreePartnerPass.

```
1  fbeRule rlStartFreePartnerPass
2   condition
3       premise prRefereeCmdStartGame Robot.atRefereeCmd == ' ' and
4       premise prLinePlayerRole Robot.atRole != "GOALKEEPER" and
5       premise prBallTeamField Robot.atBallEnemyField == false and
6       premise prClosestToBall Robot.atClosestToBall == true and
7       premise prFreePartner Robot.atPartnerFreeID >= 0 and
8       premise prRobotIsNotReady Robot.atIsReady == true
9   end_condition
10  action
11     instigation inPassBallPartner Robot.mtPassBallPartner()
12  end_action
13 end_fbeRule
```

Rule 8: Código da *Rule* rlStartNoFreePartner.

```
1  fbeRule rlStartNoFreePartner
2   condition
3       premise prRefereeCmdStartGame Robot.atRefereeCmd == ' ' and
4       premise prLinePlayerRole Robot.atRole != "GOALKEEPER" and
5       premise prBallTeamField Robot.atBallEnemyField == false and
6       premise prClosestToBall Robot.atClosestToBall == true and
7       premise prFreePartner Robot.atPartnerFreeID <= 0 and
8       premise prRobotIsNotReady Robot.atIsReady == false
9   end_condition
10  action
11     instigation inPosKickNoFreePartner Robot.mtMovePositionToKick()
12  end_action
13 end_fbeRule
```

Rule 9: Código da *Rule* rlStartNoFreePartnerKick.

```
1  fbeRule rlStartNoFreePartnerKick
2   condition
3       premise prRefereeCmdStartGame Robot.atRefereeCmd == ‘ ’ and
4       premise prLinePlayerRole Robot.atRole != “GOALKEEPER” and
5       premise prBallTeamField Robot.atBallEnemyField == false and
6       premise prClosestToBall Robot.atClosestToBall == true and
7       premise prFreePartner Robot.atPartnerFreeID <= 0 and
8       premise prRobotIsNotReady Robot.atIsReady == true
9   end_condition
10  action
11    instigation inKickNoFreePartner Robot.mtReadyKickoff()
12  end_action
13 end_fbeRule
```

Rule 10: Código da *Rule* rlStartEnemyFieldPositionKick.

```
1  fbeRule rlStartEnemyFieldPositionKick
2   condition
3       premise prRefereeCmdStartGame Robot.atRefereeCmd == ‘ ’ and
4       premise prLinePlayerRole Robot.atRole != “GOALKEEPER” and
5       premise prBallEnemyField Robot.atBallEnemyField == true and
6       premise prNoEnemyOnLineGoal Robot.atEnemyOnGoalLine == true
   and
7       premise prClosestToBall Robot.atClosestToBall == true and
8       premise prRobotIsNotReady Robot.atIsReady == false and
9   end_condition
10  action
11    instigation inPosKickEnemyField Robot.mtMovePositionToKick()
12  end_action
13 end_fbeRule
```

Rule 11: Código da *Rule* rlStartEnemyFieldKick.

```
1  fbeRule rlStartEnemyFieldKick
2   condition
3       premise prRefereeCmdStartGame Robot.atRefereeCmd == ‘ ’ and
4       premise prLinePlayerRole Robot.atRole != “GOALKEEPER” and
5       premise prBallEnemyField Robot.atBallEnemyField == true and
6       premise prNoEnemyOnLineGoal Robot.atEnemyOnGoalLine == true
   and
7       premise prClosestToBall Robot.atClosestToBall == true and
8       premise prRobotIsNotReady Robot.atIsReady == true and
9   end_condition
10  action
11    instigation inKickEnemyField Robot.mtReadyKickoff()
12  end_action
13 end_fbeRule
```

Rule 12: Código da *Rule* rlStartTargetToBall.

```
1  fbeRule rlStartTargetToBall
2   condition
3       premise prRefereeCmdStartGame Robot.atRefereeCmd == ‘ ’ and
4       premise prActiveRole Robot.atRole != “ ” and
5       premise prNotClosestToBall Robot.atClosestToBall == false and
6   end_condition
7   action
8     instigation inTargetToBall Robot.mtTargetToBall()
9   end_action
10 end_fbeRule
```

Rule 13: Código da *Rule* rlGkStopCloseGoal.

```
1  fbeRule rlGkStopCloseGoal
2    condition
3        premise prRefereeCmdStop Robot.atRefereeCmd == 'S' and
4        premise prRoleGoalkeeper Robot.atRole == "GOALKEEPER" and
5        premise prBallCloseTeamGoal Robot.atBallDistTeamGoal <= 1800
6    end_condition
7    action
8      instigation inGkStopClose Robot.mtGkEnemyPenalty()
9    end_action
10 end_fbeRule
```

Rule 14: Código da *Rule* rlGkStopFarGoal.

```
1  fbeRule rlGkStopFarGoal
2    condition
3        premise prRefereeCmdStop Robot.atRefereeCmd == 'S' and
4        premise prRoleGoalkeeper Robot.atRole == "GOALKEEPER" and
5        premise prBallCloseTeamGoal Robot.atBallDistTeamGoal >1800
6    end_condition
7    action
8      instigation inGkStopFarGoal Robot.mtGkOutArea()
9    end_action
10 end_fbeRule
```

Rule 15: Código da *Rule* rlGkStartInsideAreaClosestBall.

```
1  fbeRule rlGkStartInsideAreaClosestBall
2   condition
3       premise prRefereeCmdStartGame Robot.atRefereeCmd == ‘ ’ and
4       premise prRoleGoalkeeper Robot.atRole == “GOALKEEPER” and
5       premise prBallInGoalArea Robot.atBallDistTeamGoal <= 800 and
6       premise prClosestToBall Robot.atClosestToBall == true and
7       premise prRobotIsNotReady Robot.atIsReady == false
8   end_condition
9   action
10    instigation inGkStartInside Robot.mtMovePositionToKick()
11  end_action
12 end_fbeRule
```

Rule 16: Código da *Rule* rlGkStartInsideAreaClosestBallKick.

```
1  fbeRule rlGkStartInsideAreaClosestBallKick
2   condition
3       premise prRefereeCmdStartGame Robot.atRefereeCmd == ‘ ’ and
4       premise prRoleGoalkeeper Robot.atRole == “GOALKEEPER” and
5       premise prBallInGoalArea Robot.atBallDistTeamGoal <= 800 and
6       premise prClosestToBall Robot.atClosestToBall == true and
7       premise prRobotIsNotReady Robot.atIsReady == true
8   end_condition
9   action
10    instigation inGkAreaKick Robot.mtReadyKickoff()
11  end_action
12 end_fbeRule
```

Rule 17: Código da *Rule* rlGkStartInsideArea.

```
1  fbeRule rlGkStartInsideArea
2   condition
3       premise prRefereeCmdStartGame Robot.atRefereeCmd == ‘ ’ and
4       premise prRoleGoalkeeper Robot.atRole == “GOALKEEPER” and
5       premise prBallInsideGoalArea Robot.atBallDistTeamGoal <= 800
   and
6       premise prNotClosestToBall Robot.atClosestToBall == false
7   end_condition
8   action
9     instigation inGkStartInArea Robot.mtGkEnemyPenalty()
10  end_action
11 end_fbeRule
```

Rule 18: Código da *Rule* rlGkStartOutsideArea.

```
1  fbeRule rlGkStartOutsideArea
2   condition
3       premise prRefereeCmdStartGame Robot.atRefereeCmd == ‘ ’ and
4       premise prRoleGoalkeeper Robot.atRole == “GOALKEEPER” and
5       premise prBallNotInGoalArea Robot.atBallDistTeamGoal >800
6   end_condition
7   action
8     instigation inGkStartOutArea Robot.mtGkOutArea()
9   end_action
10 end_fbeRule
```

Rule 19: Código da *Rule* rlGkBluePenaltyYellow.

```
1  fbeRule rlGkBluePenaltyYellow
2   condition
3       premise prRefereeCmdPenaltyYellow Robot.atRefereeCmd == ‘p’
   and
4       premise prRoleGoalkeeper Robot.atRole == “GOALKEEPER” and
5       premise prTeamBlue Robot.atTeamColor == “BLUE”
6   end_condition
7   action
8     instigation inGkEnemyPenalty1 Robot.mtGkEnemyPenalty()
9   end_action
10 end_fbeRule
```

Rule 20: Código da *Rule* rlGkYellowPenaltyBlue.

```
1  fbeRule rlGkYellowPenaltyBlue
2   condition
3       premise prRefereeCmdPenaltyYellow Robot.atRefereeCmd == ‘P’
   and
4       premise prRoleGoalkeeper Robot.atRole == “GOALKEEPER” and
5       premise prTeamYellow Robot.atTeamColor == “YELLOW”
6   end_condition
7   action
8     instigation inGkEnemyPenalty2 Robot.mtGkEnemyPenalty()
9   end_action
10 end_fbeRule
```

Rule 21: Código da *Rule* rlDLtStopBallFar.

```
1  fbeRule rlDLtStopBallFar
2   condition
3       premise prRefereeCmdStop Robot.atRefereeCmd == 'S' and
4       premise prRoleDefLeft Robot.atRole == "DEFENDER_LEFT" and
5       premise prBallFarTeamGoal Robot.atBallDistTeamGoal >1800
6   end_condition
7   action
8     instigation inStopDLFar Robot.mtMoveLeftPost()
9   end_action
10 end_fbeRule
```

Rule 22: Código da *Rule* rlDLStopBallClose.

```
1  fbeRule rlDLStopBallClose
2   condition
3       premise prRefereeCmdStop Robot.atRefereeCmd == 'S' and
4       premise prRoleDefLeft Robot.atRole == "DEFENDER_LEFT" and
5       premise prBallCloseTeamGoal Robot.atBallDistTeamGoal == 800
6   end_condition
7   action
8     instigation inDLStopBallClose Robot.mtMovePosAngleNeg3pl()
9   end_action
10 end_fbeRule
```

Rule 23: Código da *Rule* rlDLBlueDirectKickBlue.

```
1  fbeRule rlDLBlueDirectKickBlue
2   condition
3       premise prRefCmdDirectKickBlue Robot.atRefereeCmd == 'F' and
4       premise prRoleDefLeft Robot.atRole == "DEFENDER_LEFT" and
5       premise prTeamBlue Robot.atTeamColor == "BLUE"
6   end_condition
7   action
8     instigation inDLBlueDirectKickBlue Robot.mtMoveLeftPost()
9   end_action
10 end_fbeRule
```

Rule 24: Código da *Rule* rlDLYellowDirectKickYellow.

```
1  fbeRule rlDLYellowDirectKickYellow
2   condition
3       premise prRefCmdDirectKickYellow Robot.atRefereeCmd == 'f' and
4       premise prRoleDefLeft Robot.atRole == "DEFENDER_LEFT" and
5       premise prTeamYellow Robot.atTeamColor == "YELLOW"
6   end_condition
7   action
8     instigation inDLYellowDirectKickYellow Robot.mtMoveLeftPost()
9   end_action
10 end_fbeRule
```

Rule 25: Código da *Rule* rlDLtBlueIndirectBlue.

```
1  fbeRule rlDLtBlueIndirectBlue
2   condition
3       premise prRefCmdIndirectKickBlue Robot.atRefereeCmd == 'I' and
4       premise prRoleDefLeft Robot.atRole == "DEFENDER_LEFT" and
5       premise prTeamBlue Robot.atTeamColor == "BLUE"
6   end_condition
7   action
8     instigation inDLtBlueIndirectBlue Robot.mtMoveLeftPost()
9   end_action
10 end_fbeRule
```

Rule 26: Código da *Rule* rlDLYellowIndirectYellow.

```
1  fbeRule rlDLYellowIndirectYellow
2   condition
3       premise prRefCmdIndirectKickYellow Robot.atRefereeCmd == ‘i’
   and
4       premise prRoleDefLeft Robot.atRole == “DEFENDER_LEFT” and
5       premise prTeamYellow Robot.atTeamColor == “YELLOW”
6   end_condition
7   action
8     instigation inDLYellowIndirectYellow Robot.mtMoveLeftPost()
9   end_action
10 end_fbeRule
```

Rule 27: Código da *Rule* rlDLtBluePenaltyYellow.

```
1  fbeRule rlDLtBluePenaltyYellow
2   condition
3       premise prRefCmdPenaltyYellow Robot.atRefereeCmd == ‘p’ and
4       premise prRoleDefLeft Robot.atRole == “DEFENDER_LEFT” and
5       premise prTeamBlue Robot.atTeamColor == “BLUE”
6   end_condition
7   action
8     instigation inDLtBluePenaltyYellow Robot.mtMovePenaltyDefLeft()
9   end_action
10 end_fbeRule
```

Rule 28: Código da *Rule* rlDLYellowPenaltyBlue.

```
1  fbeRule rlDLYellowPenaltyBlue
2   condition
3       premise prRefCmdPenaltyBlue Robot.atRefereeCmd == ‘P’ and
4       premise prRoleDefLeft Robot.atRole == “DEFENDER_LEFT” and
5       premise prTeamYellow Robot.atTeamColor == “YELLOW”
6   end_condition
7   action
8     instigation inDLYellowPenaltyBlue Robot.mtMovePenaltyDefLeft()
9   end_action
10 end_fbeRule
```

Rule 29: Código da *Rule* rlDLStartNotClose.

```
1  fbeRule rlDLStartNotClose
2   condition
3       premise prRefCmdStartGame Robot.atRefereeCmd == ‘ ’ and
4       premise prRoleDefLeft Robot.atRole == “DEFENDER_LEFT” and
5       premise prNotClosestToBall Robot.atClosestToBall == false
6   end_condition
7   action
8     instigation inDLStartNotClose Robot.mtMoveLeftPost()
9   end_action
10 end_fbeRule
```

Rule 30: Código da *Rule* rlDRStopBallFar.

```
1  fbeRule rlDRStopBallFar
2   condition
3       premise prRefereeCmdStop Robot.atRefereeCmd == ‘S’ and
4       premise prDefRight Robot.atRole == “DEFENDER_RIGHT” and
5       premise prBallFarTeamGoal Robot.atBallDistTeamGoal >1800
6   end_condition
7   action
8     instigation inDRStopBallFar Robot.mtMoveRightPost()
9   end_action
10 end_fbeRule
```

Rule 31: Código da *Rule* rlDRStopBallClose.

```
1  fbeRule rlDRStopBallClose
2   condition
3       premise prRefereeCmdStop Robot.atRefereeCmd == 'S' and
4       premise prDefRight Robot.atRole == "DEFENDER_RIGHT" and
5       premise prBallCloseTeamGoal Robot.atBallDistTeamGoal <= 1800
6   end_condition
7   action
8     instigation inDRStopBallClose Robot.mtMovePosAnglePos3pl()
9   end_action
10 end_fbeRule
```

Rule 32: Código da *Rule* rlDRBlueDirectKickBlue.

```
1  fbeRule rlDRBlueDirectKickBlue
2   condition
3       premise prRefCmdDirectKickBlue Robot.atRefereeCmd == 'F' and
4       premise prDefRight Robot.atRole == "DEFENDER_RIGHT" and
5       premise prTeamBlue Robot.atTeamColor == "BLUE"
6   end_condition
7   action
8     instigation inDRBlueDirectKickBlue Robot.mtMoveRightPost()
9   end_action
10 end_fbeRule
```

Rule 33: Código da *Rule* rlDRYellowDirectKickYellow.

```
1  fbeRule rlDRYellowDirectKickYellow
2   condition
3       premise prRefCmdCmdDirectKickYellow Robot.atRefereeCmd == ‘f’
   and
4       premise prDefRight Robot.atRole == “DEFENDER_RIGHT” and
5       premise prTeamYellow Robot.atTeamColor == “YELLOW’
6   end_condition
7   action
8     instigation inDRYellowDirectKickYellow Robot.mtMoveRightPost()
9   end_action
10 end_fbeRule
```

Rule 34: Código da *Rule* rlDRBlueIndirectKickBlue.

```
1  fbeRule rlDRBlueIndirectKickBlue
2   condition
3       premise prRefCmdIndirectKickBlue Robot.atRefereeCmd == ‘I’ and
4       premise prDefRight Robot.atRole == “DEFENDER_RIGHT” and
5       premise prTeamBlue Robot.atTeamColor == “BLUE”
6   end_condition
7   action
8     instigation inDRBlueIndirectKickBlue Robot.mtMoveRightPost()
9   end_action
10 end_fbeRule
```

Rule 35: Código da *Rule* rlDRtYellowIndirectKickYellow.

```
1  fbeRule rlDRtYellowIndirectKickYellow
2   condition
3       premise prRefCmdIndirectKickYellow Robot.atRefereeCmd == 'i'
   and
4       premise prDefRight Robot.atRole == "DEFENDER_RIGHT" and
5       premise prTeamYellow Robot.atTeamColor == "YELLOW"
6   end_condition
7   action
8     instigation inDRtYellowIndirectKickYellow Robot.mtMoveRightPost()
9   end_action
10 end_fbeRule
```

Rule 36: Código da *Rule* rlDRBluePenaltyYellow.

```
1  fbeRule rlDRBluePenaltyYellow
2   condition
3       premise prRefCmdPenaltyYellow Robot.atRefereeCmd == 'p' and
4       premise prDefRight Robot.atRole == "DEFENDER_RIGHT" and
5       premise prTeamBlue Robot.atTeamColor == "BLUE"
6   end_condition
7   action
8     instigation inDRBluePenaltyYellow Robot.mtMovePenaltyDefRight()
9   end_action
10 end_fbeRule
```

Rule 37: Código da *Rule* rlDRYellowPenaltyBlue.

```
1  fbeRule rlDRYellowPenaltyBlue
2   condition
3      premise prRefCmdPenaltyBlue Robot.atRefereeCmd == 'P' and
4      premise prDefRight Robot.atRole == "DEFENDER_RIGHT" and
5      premise prTeamYellow Robot.atTeamColor == "YELLOW"
6   end_condition
7   action
8     instigation inDRYellowPenaltyBlue Robot.mtMovePenaltyDefRight()
9   end_action
10 end_fbeRule
```

Rule 38: Código da *Rule* rlDRStartBallNotClose.

```
1  fbeRule rlDRStartBallNotClose
2   condition
3      premise prRefCmdStartGame Robot.atRefereeCmd == ' ' and
4      premise prDefRight Robot.atRole == "DEFENDER_RIGHT" and
5      premise prNotClosestToBall Robot.atClosestToBall == false
6   end_condition
7   action
8     instigation inDRStartBallNotClose Robot.mtMoveRightPost()
9   end_action
10 end_fbeRule
```

Rule 39: Código da *Rule* rlMOStop.

```
1  fbeRule rlMOStop
2   condition
3      premise prRefereeCmdStop Robot.atRefereeCmd == 'S' and
4      premise prMidfieldOnly Robot.atRole == "MIDFIELD_ONLY"
5   end_condition
6   action
7     instigation inMOStop Robot.mtMoveStopPosition()
8   end_action
9  end_fbeRule
```

Rule 40: Código da *Rule* rlMOBlueKickoff.

```
1  fbeRule rlMOBlueKickoff
2   condition
3       premise prRefCmdKickoffBlue Robot.atRefereeCmd == 'K' and
4       premise prMidfieldOnly Robot.atRole == "MIDFIELD_ONLY" and
5       premise prTeamBlue Robot.atTeamColor == "BLUE"
6   end_condition
7   action
8     instigation inMOBlueKickoff Robot.mtMovePositionToKick()
9   end_action
10 end_fbeRule
```

Rule 41: Código da *Rule* rlMOYellowKickoff.

```
1  fbeRule rlMOYellowKickoff
2   condition
3       premise prRefCmdKickoffYellow Robot.atRefereeCmd == 'k' and
4       premise prMidfieldOnly Robot.atRole == "MIDFIELD_ONLY" and
5       premise prTeamYellow Robot.atTeamColor == "YELLOW"
6   end_condition
7   action
8     instigation inMOYellowKickoff Robot.mtMovePositionToKick()
9   end_action
10 end_fbeRule
```

Rule 42: Código da *Rule* rlMOYellowReadyKickoffYellow.

```
1  fbeRule rlMOYellowReadyKickoffYellow
2   condition
3       premise prRefCmdStartGame Robot.atRefereeCmd == ‘ ’ and
4       premise prMidfieldOnly Robot.atRole == “MIDFIELD_ONLY” and
5       premise prLastRefCmdKickoffYellow Robot.atLastRefCmd == ‘k’
   and
6       premise prTeamYellow Robot.atTeamColor == “YELLOW”
7   end_condition
8   action
9     instigation inMOYellowReadyKickoffYellow Robot.mtReadyKickoff()
10  end_action
11 end_fbeRule
```

Rule 43: Código da *Rule* rlMOBlueDirectKick.

```
1  fbeRule rlMOBlueDirectKick
2   condition
3       premise prRefCmdDirectKickBlue Robot.atRefereeCmd == ‘K’ and
4       premise prMidfieldOnly Robot.atRole == “MIDFIELD_ONLY” and
5       premise prTeamBlue Robot.atTeamColor == “BLUE”
6   end_condition
7   action
8     instigation inMOBlueDirectKick Robot.mtMovePositionToKick()
9   end_action
10 end_fbeRule
```

Rule 44: Código da *Rule* rlMOYellowDirectKick.

```
1  fbeRule rlMOYellowDirectKick
2   condition
3       premise prRefCmdDirectKickYellow Robot.atRefereeCmd == 'f' and
4       premise prMidfieldOnly Robot.atRole == "MIDFIELD_ONLY" and
5       premise prTeamYellow Robot.atTeamColor == "YELLOW"
6   end_condition
7   action
8     instigation inMOYellowDirectKick Robot.mtMovePositionToKick()
9   end_action
10 end_fbeRule
```

Rule 45: Código da *Rule* rlMOBlueIndirectKick.

```
1  fbeRule rlMOBlueIndirectKick
2   condition
3       premise prRefCmdIndirectKickBlue Robot.atRefereeCmd == 'I' and
4       premise prMidfieldOnly Robot.atRole == "MIDFIELD_ONLY" and
5       premise prTeamBlue Robot.atTeamColor == "BLUE"
6   end_condition
7   action
8     instigation inMOBlueIndirectKick Robot.mtMoveIndirectKick()
9   end_action
10 end_fbeRule
```

Rule 46: Código da *Rule* rlMOYellowIndirectKick.

```
1  fbeRule rlMOYellowIndirectKick
2   condition
3       premise prRefCmdIndirectKickYellow Robot.atRefereeCmd == ‘i’
   and
4       premise prMidfieldOnly Robot.atRole == “MIDFIELD_ONLY” and
5       premise prTeamYellow Robot.atTeamColor == “YELLOW”
6   end_condition
7   action
8     instigation inMOYellowIndirectKick Robot.mtMoveIndirectKick()
9   end_action
10 end_fbeRule
```

Rule 47: Código da *Rule* rlMOBluePenaltyBlue.

```
1  fbeRule rlMOBluePenaltyBlue
2   condition
3       premise prRefCmdPenaltyBlue Robot.atRefereeCmd == ‘P’ and
4       premise prMidfieldOnly Robot.atRole == “MIDFIELD_ONLY” and
5       premise prTeamBlue Robot.atTeamColor == “BLUE”
6   end_condition
7   action
8     instigation inMOBluePenaltyBlue Robot.mtMovePenaltyKick()
9   end_action
10 end_fbeRule
```

Rule 48: Código da *Rule* rlMOBluePenaltyYellow.

```
1  fbeRule rlMOBluePenaltyYellow
2   condition
3       premise prRefCmdPenaltyYellow Robot.atRefereeCmd == 'p' and
4       premise prMidfieldOnly Robot.atRole == "MIDFIELD_ONLY" and
5       premise prTeamBlue Robot.atTeamColor == "BLUE"
6   end_condition
7   action
8     instigation inMOBluePenaltyYellow Robot.mtPenaltyDefCenter()
9   end_action
10 end_fbeRule
```

Rule 49: Código da *Rule* rlMOYellowPenaltyYellow.

```
1  fbeRule rlMOYellowPenaltyYellow
2   condition
3       premise prRefCmdPenaltyYellow Robot.atRefereeCmd == 'p' and
4       premise prMidfieldOnly Robot.atRole == "MIDFIELD_ONLY" and
5       premise prTeamYellow Robot.atTeamColor == "YELLOW"
6   end_condition
7   action
8     instigation inMOYellowPenaltyYellow Robot.mtMovePenaltyKick()
9   end_action
10 end_fbeRule
```

Rule 50: Código da *Rule* rlMOYellowPenaltyBlue.

```
1  fbeRule rlMOYellowPenaltyBlue
2   condition
3       premise prRefCmdPenaltyBlue Robot.atRefereeCmd == 'P' and
4       premise prMidfieldOnly Robot.atRole == "MIDFIELD_ONLY" and
5       premise prTeamYellow Robot.atTeamColor == "YELLOW"
6   end_condition
7   action
8     instigation inMOYellowPenaltyBlue Robot.mtPenaltyDefCenter()
9   end_action
10 end_fbeRule
```

Rule 51: Código da *Rule* rlMOBlueReadyPenaltyBlue.

```
1  fbeRule rlMOBlueReadyPenaltyBlue
2   condition
3       premise prRefCmdStartGame Robot.atRefereeCmd == ‘ ’ and
4       premise prMidfieldOnly Robot.atRole == “MIDFIELD_ONLY” and
5       premise prLastRefCmdPenaltyBlue Robot.atLastRefCmd == ‘P’ and
6       premise prTeamBlue Robot.atTeamColor == “BLUE”
7   end_condition
8   action
9     instigation inMOBlueReadyPenaltyBlue Robot.mtReadyPenalty()
10  end_action
11 end_fbeRule
```

Rule 52: Código da *Rule* rlMOYellowReadyPenaltyYellow.

```
1  fbeRule rlMOYellowReadyPenaltyYellow
2   condition
3       premise prRefCmdStartGame Robot.atRefereeCmd == ‘ ’ and
4       premise prMidfieldOnly Robot.atRole == “MIDFIELD_ONLY” and
5       premise prLastRefCmdPenaltyYellow Robot.atLastRefCmd == ‘p’
   and
6       premise prTeamYellow Robot.atTeamColor == “YELLOW”
7   end_condition
8   action
9     instigation inMOYellowReadyPenaltyYellow Robot.mtReadyPenalty()
10  end_action
11 end_fbeRule
```

Rule 53: Código da *Rule* rlSLStopTeamLeft.

```
1  fbeRule rlSLStopTeamLeft
2   condition
3       premise prRefCmdStop Robot.atRefereeCmd == 'S' and
4       premise prStrickerLeft Robot.atRole == "STRIKER_LEFT" and
5       premise prTeamLeftSide Robot.atTeamSide == "LEFT"
6   end_condition
7   action
8     instigation inSLStopTeamLeft Robot.mtMovePosAngleNeg()
9   end_action
10 end_fbeRule
```

Rule 54: Código da *Rule* rlSLStopTeamRight.

```
1  fbeRule rlSLStopTeamRight
2   condition
3       premise prRefCmdStop Robot.atRefereeCmd == 'S' and
4       premise prStrickerLeft Robot.atRole == "STRIKER_LEFT" and
5       premise prTeamRightSide Robot.atTeamSide == "RIGHT"
6   end_condition
7   action
8     instigation inSLStopTeamRight Robot.mtMovePosAnglePos()
9   end_action
10 end_fbeRule
```

Rule 55: Código da *Rule* rlSLBlueDirectKick.

```
1  fbeRule rlSLBlueDirectKick
2   condition
3       premise prRefCmdDirectKickBlue Robot.atRefereeCmd == 'F' and
4       premise prStrickerLeft Robot.atRole == "STRIKER_LEFT" and
5       premise prTeamBlue Robot.atTeamColor == "BLUE"
6   end_condition
7   action
8     instigation inSLBlueDirectKick Robot.mtStrickerLeftDirect()
9   end_action
10 end_fbeRule
```

Rule 56: Código da *Rule* rlSLYellowDirectKick.

```
1  fbeRule rlSLYellowDirectKick
2   condition
3       premise prRefCmdDirectKickYellow Robot.atRefereeCmd == ‘f’ and
4       premise prStrickerLeft Robot.atRole == “STRIKER_LEFT” and
5       premise prTeamYellow Robot.atTeamColor == “YELLOW”
6   end_condition
7   action
8     instigation inSLYellowDirectKick Robot.mtStrickerLeftDirect()
9   end_action
10 end_fbeRule
```

Rule 57: Código da *Rule* rlSLBlueIndirectKick.

```
1  fbeRule rlSLBlueIndirectKick
2   condition
3       premise prRefCmdIndirectKickBlue Robot.atRefereeCmd == ‘I’ and
4       premise prStrickerLeft Robot.atRole == “STRIKER_LEFT” and
5       premise prTeamBlue Robot.atTeamColor == “BLUE”
6   end_condition
7   action
8     instigation inSLBlueIndirectKick Robot.mtStrickerLeftDirect()
9   end_action
10 end_fbeRule
```

Rule 58: Código da *Rule* rlSLYellowIndirectKick.

```
1  fbeRule rlSLYellowIndirectKick
2   condition
3       premise prRefCmdIndirectKickYellow Robot.atRefereeCmd == 'i'
    and
4       premise prStrickerLeft Robot.atRole == "STRIKER_LEFT" and
5       premise prTeamYellow Robot.atTeamColor == "YELLOW"
6   end_condition
7   action
8     instigation inSLYellowIndirectKick Robot.mtStrickerLeftDirect()
9   end_action
10 end_fbeRule
```

Rule 59: Código da *Rule* rlSLBluePenaltyBlue.

```
1  fbeRule rlSLBluePenaltyBlue
2   condition
3       premise prRefCmdPenaltyBlue Robot.atRefereeCmd == 'P' and
4       premise prStrickerLeft Robot.atRole == "STRIKER_LEFT" and
5       premise prTeamBlue Robot.atTeamColor == "BLUE"
6   end_condition
7   action
8     instigation inSLBluePenaltyBlue Robot.mtPenaltyAttackLeft()
9   end_action
10 end_fbeRule
```

Rule 60: Código da *Rule* rlSLBluePenaltyYellow.

```
1  fbeRule rlSLBluePenaltyYellow
2   condition
3       premise prRefCmdPenaltyYellow Robot.atRefereeCmd == 'p' and
4       premise prStrickerLeft Robot.atRole == "STRIKER_LEFT" and
5       premise prTeamBlue Robot.atTeamColor == "BLUE"
6   end_condition
7   action
8     instigation inSLBluePenaltyYellow Robot.mtStrickerLeftDirect()
9   end_action
10 end_fbeRule
```

Rule 61: Código da *Rule* rlSLYellowPenaltyYellow.

```
1  fbeRule rlSLYellowPenaltyYellow
2   condition
3       premise prRefCmdPenaltyYellow Robot.atRefereeCmd == 'p' and
4       premise prStrickerLeft Robot.atRole == "STRIKER_LEFT" and
5       premise prTeamYellow Robot.atTeamColor == "YELLOW"
6   end_condition
7   action
8     instigation inSLYellowPenaltyYellow Robot.mtPenaltyAttackLeft()
9   end_action
10 end_fbeRule
```

Rule 62: Código da *Rule* rlSLYellowPenaltyBlue.

```
1  fbeRule rlSLYellowPenaltyBlue
2   condition
3       premise prRefCmdPenaltyBlue Robot.atRefereeCmd == 'P' and
4       premise prStrickerLeft Robot.atRole == "STRIKER_LEFT" and
5       premise prTeamYellow Robot.atTeamColor == "YELLOW"
6   end_condition
7   action
8     instigation inSLYellowPenaltyBlue Robot.mtStrickerLeftDirect()
9   end_action
10 end_fbeRule
```

Rule 63: Código da *Rule* rlSLStartBallNotClose.

```
1  fbeRule rlSLStartBallNotClose
2   condition
3      premise prRefCmdStartGame Robot.atRefereeCmd == ‘ ’ and
4      premise prStrickerLeft Robot.atRole == “STRIKER_LEFT” and
5      premise prNotClosestToBall Robot.atClosestToBall == false
6   end_condition
7   action
8     instigation inSLStartBallNotClose Robot.mtStrickerLeftDirect()
9   end_action
10 end_fbeRule
```

Rule 64: Código da *Rule* rlSRStopTeamLeft.

```
1  fbeRule rlSRStopTeamLeft
2   condition
3      premise prRefCmdStop Robot.atRefereeCmd == ‘S’ and
4      premise prStrikerRight Robot.atRole == “STRIKER_RIGHT” and
5      premise prTeamLeftSide Robot.atTeamSide == “LEFT”
6   end_condition
7   action
8     instigation inSRStopTeamLeft Robot.mtMovePosAnglePos()
9   end_action
10 end_fbeRule
```

Rule 65: Código da *Rule* rlSRStopTeamRight.

```
1  fbeRule rlSRStopTeamRight
2   condition
3      premise prRefCmdStop Robot.atRefereeCmd == ‘S’ and
4      premise prStrikerRight Robot.atRole == “STRIKER_RIGHT” and
5      premise prTeamRightSide Robot.atTeamSide == “RIGHT”
6   end_condition
7   action
8     instigation inSRStopTeamRight Robot.mtMovePosAngleNeg()
9   end_action
10 end_fbeRule
```

Rule 66: Código da *Rule* rlSRBlueDirectKick.

```
1  fbeRule rlSRBlueDirectKick
2   condition
3       premise prRefCmdDirectKickBlue Robot.atRefereeCmd == 'F' and
4       premise prStrikerRight Robot.atRole == "STRIKER_RIGHT" and
5       premise prTeamBlue Robot.atTeamColor == "BLUE"
6  end_condition
7  action
8     instigation inSRBlueDirectKick Robot.mtStrickerRightDirect()
9  end_action
10 end_fbeRule
```

Rule 67: Código da *Rule* rlSRYellowDirectKick.

```
1  fbeRule rlSRYellowDirectKick
2   condition
3       premise prRefCmdDirectKickYellow Robot.atRefereeCmd == 'f' and
4       premise prStrikerRight Robot.atRole == "STRIKER_RIGHT" and
5       premise prTeamYellow Robot.atTeamColor == "YELLOW"
6  end_condition
7  action
8     instigation inSRYellowDirectKick Robot.mtStrickerRightDirect()
9  end_action
10 end_fbeRule
```

Rule 68: Código da *Rule* rlSRBlueIndirectKick.

```
1  fbeRule rlSRBlueIndirectKick
2   condition
3       premise prRefCmdIndirectKickBlue Robot.atRefereeCmd == 'I' and
4       premise prStrikerRight Robot.atRole == "STRIKER_RIGHT" and
5       premise prTeamBlue Robot.atTeamColor == "BLUE"
6  end_condition
7  action
8     instigation inSRBlueIndirectKick Robot.mtStrickerRightDirect()
9  end_action
10 end_fbeRule
```

Rule 69: Código da *Rule* rlSRYellowIndirectKick.

```
1  fbeRule rlSRYellowIndirectKick
2   condition
3      premise prRefCmdIndirectKickYellow Robot.atRefereeCmd == 'i'
   and
4      premise prStrikerRight Robot.atRole == "STRIKER_RIGHT" and
5      premise prTeamYellow Robot.atTeamColor == "YELLOW"
6   end_condition
7   action
8     instigation inSRYellowIndirectKick Robot.mtStrickerRightDirect()
9   end_action
10 end_fbeRule
```

Rule 70: Código da *Rule* rlSRBluePenaltyBlue.

```
1  fbeRule rlSRBluePenaltyBlue
2   condition
3      premise prRefCmdPenaltyBlue Robot.atRefereeCmd == 'P' and
4      premise prStrikerRight Robot.atRole == "STRIKER_RIGHT" and
5      premise prTeamBlue Robot.atTeamColor == "BLUE"
6   end_condition
7   action
8     instigation inSRBluePenaltyBlue Robot.mtPenaltyAttackRight()
9   end_action
10 end_fbeRule
```

Rule 71: Código da *Rule* rlSRBluePenaltyYellow.

```
1  fbeRule rlSRBluePenaltyYellow
2   condition
3       premise prRefCmdPenaltyYellow Robot.atRefereeCmd == 'p' and
4       premise prStrikerRight Robot.atRole == "STRIKER_RIGHT" and
5       premise prTeamBlue Robot.atTeamColor == "BLUE"
6   end_condition
7   action
8     instigation inSRBluePenaltyYellow Robot.mtStrickerRightDirect()
9   end_action
10 end_fbeRule
```

Rule 72: Código da *Rule* rlSRYellowsPenaltyYellow.

```
1  fbeRule rlSRYellowsPenaltyYellow
2   condition
3       premise prRefCmdPenaltyYellow Robot.atRefereeCmd == 'p' and
4       premise prStrikerRight Robot.atRole == "STRIKER_RIGHT" and
5       premise prTeamYellow Robot.atTeamColor == "YELLOW"
6   end_condition
7   action
8     instigation inSRYellowsPenaltyYellow Robot.mtPenaltyAttackRight()
9   end_action
10 end_fbeRule
```

Rule 73: Código da *Rule* rlSRYellowsPenaltyBlue.

```
1  fbeRule rlSRYellowsPenaltyBlue
2   condition
3       premise prRefCmdPenaltyBlue Robot.atRefereeCmd == 'P' and
4       premise prStrikerRight Robot.atRole == "STRIKER_RIGHT" and
5       premise prTeamYellow Robot.atTeamColor == "YELLOW"
6   end_condition
7   action
8     instigation inSRYellowsPenaltyBlue Robot.mtStrickerRightDirect()
9   end_action
10 end_fbeRule
```

Rule 74: Código da *Rule* rlSRStartBallNotClose.

```
1  fbeRule rlSRStartBallNotClose
2   condition
3       premise prRefCmdStartGame Robot.atRefereeCmd == ‘ ’ and
4       premise prStrikerRight Robot.atRole == “STRIKER_RIGHT” and
5       premise prNotClosestToBall Robot.atClosestToBall == false
6   end_condition
7   action
8     instigation inSRStartBallNotClose Robot.mtStrickerRightDirect()
9   end_action
10 end_fbeRule
```

APÊNDICE D – ANALISADOR LÉXICO DESENVOLVIDO PARA A CONTAGEM DO NÚMERO DE ***TOKENS***

Para auxiliar a contagem de *tokens* presente em cada um dos códigos-fonte apresentados neste trabalho, foi desenvolvido um analisador léxico a partir da ferramenta *Flex/Lex*, a qual permite criar analisadores léxicos de maneira fácil e rápida. Este analisador léxico foi desenvolvido com auxílio do prof. Fabro, o qual vislumbrou a possibilidade de utilizar a ferramenta *Flex/Lex* para a contagem de *tokens*.

O código-fonte do analisador léxico desenvolvido é apresentado no Código 38.

Código 38: Código-fonte do analisador léxico desenvolvido para efetuar a contagem de *Tokens* das aplicações apresentadas.

```
1  % {
2        #include < stdio.h >
3        int tokens = 0;
4  % }
5
6  %%
7  [a-zA-Z0-9_]+              { tokens++; }
8  \“                         { tokens++; }
9  \{                         { tokens++; }
10 \}                         { tokens++; }
11 \(                         { tokens++; }
12 \)                         { tokens++; }
13 \[                         { tokens++; }
14 \]                         { tokens++; }
15 ;                          { tokens++; }
16 =                          { tokens++; }
17 “==”                       { tokens++; }
18 “!=”                       { tokens++; }
19 “>”                        { tokens++; }
20 “>=”                       { tokens++; }
21 “<”                        { tokens++; }
22 “<=”                       { tokens++; }
23 “->”                       { tokens++; }
24 “*”                        { tokens++; }
25 \n                                     { }
26 “\\”([a-z]|[0-9]|[A-Z]|“ ”)*           { }
27
28 int main (int argc, char **argv)
29 {
30       yyin = fopen( argv[1], “r”);
31       yylex();
32       printf(“Quantidade de Tokens: %8d”, tokens);
33       return tokens;
34 }
```

APÊNDICE E – CENSO DAS APLICAÇÕES PON

Este Apêndice apresenta um levantamento das aplicações PON desenvolvidas até o presente momento, incluindo detalhes tais como materialização PON utilizada para o desenvolvimento, número de *Rules*, quantidade de instâncias de *FBEs* e resultados obtidos com cada aplicação.

Os dados para cada uma das aplicações foram obtidos através de pesquisa realizada com pessoas que trabalham ou já trabalharam com o PON. Cada pessoa enviou informações sobre as aplicações por elas desenvolvidas e todas as informações recebidas foram agrupadas em forma de uma única tabela. Nesta tabela, as aplicações são apresentadas em ordem cronológica.

Esta tabela é de grande valor para pesquisadores que trabalham na linha de pesquisa do PON. Isto porque, até o presente momento não se tinha contabilizado a quantidade de aplicações que haviam sido desenvolvidas sob o viés do PON e tão pouco a complexidade, em termos de entidades PON, de cada uma dessas aplicações.

Posteriormente, algumas das aplicações PON desenvolvidas em *softwares* são brevemente descritas, destacando a quantidade de entidades *FBEs* e *Rules* presente na aplicação bem como resultados obtidos a partir de cada aplicação.

Autor(es)	Ano	Nome da Aplicação	Materialização	Contexto	Rules	Sub Conditions	Premises	Instigations	FBE Classes	Attributes	Methods	Instâncias de FBE	Comparado com o que?	Resumo Resultado
Jean Simão	2000	Construção do Framework CON acoplado ao Analytice II	-	Dissertação de Mestrado	1	1	1	1	1	1	1	1	N/A	Demonstrou viabilidade com um pequeno exemplo
Jean Simão	2000	CON Célula - Célula Robotizada - Analytice II	Framework CON 1.0	Dissertação de Mestrado	25	0	43	88	8	1	1	10	N/A	Permitiu demonstrar o conceito de Controle Orientado a Notificações em ANALYTICE II. Permitiu uma simulação em Analytice II.
Jardel Lucca	2008	Wizard para criação de controles de célula de manufatura sobre Analytice II	Framework CON 2.0	Iniciação Científica	8	-	15	15	-	14	15	3	N/A	Primeiras implementações de controles CON via aplicação em alto nível com interface gráfica amigável.
Roni Banaszewski	2009	Mira ao Alvo - Segundo Cenário	Framework C++ 1.0	Dissertação de Mestrado	5	0	17	6	4	6		202	PD	-
Roni Banaszewski	2009	Sistema de Condicionamento de Ar	Framework C++ 1.0	Dissertação de Mestrado	16	0	42	18	3	3		48	N/A	-
Adriano Ronszcka	2010	Pacman	Framework C++ 1.0/2.0	Disciplina Programação Avançada	99	0	32	98	11	15	5	330	FW PON 1.0/2.0 e PI/POO	O FW 2.0 apresentou desempenho 3x melhor em relação ao 1.0. Entretanto a versão em POO C++ ficou mais rápida que ambos (5x mais que o 2.0).
Luciana Wiecheteck	2011	Portão Eletrônico	Framework PON 1.0	Dissertação	8	4	8	8	3	3	1	1	N/A	Caso de estudo utilizado para validar o método DON proposto.
Robson Linhares	2011	Central telefônica PON	Framework C++ 1.0	Tese de Doutorado	21	0	18	18	1	2	9	2	POO	A versão utilizando POO convencional apresenta resultados de desempenho significativamente melhores do que os da versão utilizando PON.
Luciana Wiecheteck	2011	Simulador de Portão Eletrônico (SPE)	Não me recordo	Dissertação	8	4	8	8	3	-	-	-	N/A	Caso de estudo utilizado para validar o método DON proposto.
Glauber Valença	2011	Sistema de controle de vendas	Framework 2.0	Dissertação	20	-	-	10	8	20	10	50	Framework C++ 1.0 e PI/POO	Até 10x melhor em relacao ao Framework 1.0 e ganhos da OO utilizando cenários com a estrutura PONHASH.
Luciana Wiecheteck	2011	Simulador de Portão Eletrônico (SPE)	Não me recordo	Dissertação	8	4	8	8	3	-	-	-	N/A	Caso de estudo utilizado para validar o método DON proposto.
Glauber Valença	2011	Sistema de controle de vendas	Framework 2.0	Dissertação	20	-	-	10	8	20	10	50	Framework C++ 1.0 e PI/POO	Até 10x melhor em relacao ao Framework 1.0 e ganhos da OO utilizando cenários com a estrutura PONHASH.
Rodrigo Gregori	2012	Meshslicing	Framework C++ 2.0	Disciplina PON	1	1	3	1	3	4	2	1	PI - Linguagem C	A implementação de NOP é de fato menos performativa do que a implementação de IP.
Fernando Muchalski	2012	Cálculo de Produtividade	Framework C++ 2.0	Disciplina PON	1 a 4	1 a 4	1 + (4 * N)	1	1	4	1	1	PI/POO	PON se mostrou menos eficiente que o POO. Porém, a diferença de desempenho apresentou uma tendência de queda a medida que o conjunto de dados processados aumentava.
Danilo Belmonte	2012	Airplane Flight Simulator	Framework C++ 2.0+Threads	Qualificação de Doutorado	5	0	6	4	16	58	70	8	N/A	-
Eduardo Peters	2012	Mira ao Alvo	CoPON	Dissertação	100	100	300	200	3	12	17	3	Framework C++ 2.0	Em média, o coprocessador PON (CoPON), apresentou ganho médio de 45x se considerarmos toda a aplicação PON.
Eduardo Peters	2012	Simulador de Portão Eletrônico (SPE)	CoPON	Dissertação	8	4	8	8	3	-	-	-	Framework C++ 2.0	Framework C++ 2.0
Luiz Viana Melo	2013	Sistema da máquina de lavar	Framework 2.0	Disciplina PON	9	0	6	18	1	2	2	1	Framework Fuzzy (Fabro)	A versão em PON teve desempenho inferior em relação ao sistema convencional.
Clayton Kossoski	2013	Simple Flight Combat PON	Framework Otimizado	Disciplina PON	26	26	27	26	4	22	20	4	N/A	Foi possível fazer funcionar um jogo usando biblioteca Allegro e o framework PON otimizado.
Danilo Belmonte	2013	Semaphore	Framework C++ 2.0+Threads	Experimento para Tese de Doutorado	4	0	5	0	2	6	5	71	Framework C++ 2.0	Framework C++ 2.0+Threads em Servidor com quantidades diferentes de core (núcleos) de processamento apresentou balanceamento de carga entre os diferentes processadores.
Priscila Moraes e Adriano Ronszcka	2013	Mira ao alvo	Compilador PON – Geração de código em C	Disciplina PON / Compiladores	100	100	300	200	3	12	17	3	N/A	Criação do primeiro compilador PON, o qual recebe como entrada um programa em PON e gera um programa em C correspondente.
Adriano Ronska e Cleverson Ferreira	2013	Mira ao alvo	Compilador PON – Geração de código em C++	Disciplina PON / Compiladores	100	100	300	200	3	7	9	3	N/A	Criação do primeiro compilador PON, o qual recebe como entrada um programa em PON e gera um programa em C correspondente.
Rodrigo Gregori	2013	TriMeshSlicing	Framework C++ 2.0	Geometria Computacional	1		3	1	2	4	2	12 a 2016	PI/POO	Tempo de execução médio 450% maior. Melhora da performance, se utilizado o Scheduler KEEPER

Autor(es)	Ano	Nome da Aplicação	Materialização	Contexto	Rules	Sub Conditions	Premises	Instigations	FBE Classes	Attributes	Methods	Instâncias de FBE	Comparado com o que?	Resumo Resultado
Robson Linhares	2013	Semáforos NOCA	NOCA 1.0 em FPGA	Tese de Doutorado	224	96	224	0	-	-	-	-	PI - Linguagem C	O desempenho da versão PI executando em NiOS otimizado é amplamente superior ao apresentado pela aplicação PON.
Robson Linhares	2013	Mira alvo NOCA	NOCA 1.0 em FPGA	Tese de Doutorado	101	100	301	0	-	-	-	-	PI procedimental	NOCA apresenta desempenho melhor que as outras plataformas para um número reduzido de regras ativadas.
Robson Linhares	2013	Sort NOCA	NOCA 1.0 em FPGA	Tese de Doutorado	45	41	62	0	-	-	-	-	PON HD	Algoritmo de ordenação implementado no NOCA apresentou complexidade algorítmica maior do que o esperado pela teoria do PON.
Robson Linhares	2013	Semáforos NOCA	NOCA 1.0 em FPGA	Tese de Doutorado	224	96	224	0	-	-	-	-	PI - Linguagem C	O desempenho da versão PI executando em NiOS otimizado é amplamente superior ao apresentado pela aplicação PON.
Robson Linhares	2013	Mira alvo NOCA	NOCA 1.0 em FPGA	Tese de Doutorado	101	100	301	0	-	-	-	-	PI procedimental	NOCA apresenta desempenho melhor que as outras plataformas para um número reduzido de regras ativadas.
Robson Linhares	2013	Sort NOCA	NOCA 1.0 em FPGA	Tese de Doutorado	45	41	62	0	-	-	-	-	PON HD	Algoritmo de ordenação implementado no NOCA apresentou complexidade algorítmica maior do que o esperado pela teoria do PON
Robson Xavier	2014	Simulador de Transporte Individual I	Framework Otimizado/LingPON 1.0	Dissertação de Mestrado	6	0	8	6	3	3	6	3	POE (Dispatcher)	PON é diferente de POE. Dispatcher mais rápido que PON. PON menos verboso que POE.
Robson Xavier	2014	Simulador de Transporte Individual II	Framework Otimizado	Dissertação de Mestrado	14	0	14	15	4	7	14	4	POE (State)	PON é diferente de POE. Tempo de execução comparável entre implementações de acordo com o número de eventos. PON é menos verboso que POE.
Robson Xavier	2014	Simulador de Transporte Individual III	Framework Otimizado	Dissertação de Mestrado	22	0	18	24	4	7	14	5	POE (Observer)	PON é diferente de POE. Tempo de execução comparável entre implementações de acordo com o número de eventos. Desempenho que se adapta em PON. PON é menos verboso que POE.
Leonardo Pordeus	2015	CTA	LingPON 1.0	Disciplina PON	6	0	12	7	1	2	7	200	PI/POO	Apresentou resultados muito próximos ao da implementação em PI/POO (C++).
Leonardo Pordeus	2015	CTA (Intensidade de Tráfego)	LingPON 1.0	Disciplina PON	18	0	48	32	1	4	11	200	PI/POO	Apresentou resultados muito próximos ao da implementação em PI/POO (C++).
Helio Monte-Alto	2015	Pac-Man	LingPON 1.0	Disciplina PON	385	385	1462	494	7	80	116	30	N/A	A aplicação implementada em LingPON permitiu identificar alguns pontos de melhoria em seu compilador.
Cleverson Ferreira	2015	Mira ao Alvo	LingPON 1.0	Dissertação de Mestrado	1	2	4	1	3	4	4	3	PI/POO	Código LingPON compilado para C e C++ apresentaram melhor desemepnho quando comparado ao PI/POO.
Cleverson Ferreira	2015	SalesOrder - LingPON	LingPON 1.0	Dissertação de Mestrado	25	25	27	27	4	19	28	4	PI/POO	Código LingPON compilado para C e C++ apresentaram melhor desemepnho quando comparado ao PI/POO.
Fernando Schutz	2015	RNAMLP para XOR	Framework C++ 2.0	Disciplina PON	4	-	9		2	20	4	5	PI - Linguagem C	Tempo de execução muito mais longo na aplicação PON.
Luiz Viana Melo	2015	Sistema da máquina de lavar	LingPON Fuzzy	Dissertação de Mestrado	9	0	6	18	1	2	2	1	Framework Fuzzy (Fabro)	A versão em PON teve desempenho inferior em relação ao sistema convencional. Porém, o desenvolvimento foi mais fácil.
Vladimir Krachinski	2015	Meshslicing	LingPON 1.0	Disciplina PON	1	1	3	1	2	4	2	1	Framework C++ 2.0	O desempenho do LingPON foi muito melhor do que o Framework 2.0, e também mais simples de implementar, mas menos performática do que o IP.
Hinsching e Robson Linhares	2015	Busca Sequencial	LingPON 1.0	Iniciação Científica	150	150	400	6	3	4	5	52	PI - Linguagem C	Verificou-se que o desempenho da aplicação desenvolvida em Linguagem C é melhor do que a aplicação LingPO N.
Igor Mendonça	2015	WarshipAttackGame	Framework Java	Estudos do PON	13	1	32	22	5	22	19	6	N/A	Elaboração de artigos CBIC e artigo completo submetido à IEEE-LA
Igor Mendonça	2015	EletronicGate	Framework Java	Estudos do PON	6	2	16	7	2	3	8	2	N/A	Exemplo de como usar interface swing com PON
Márcio Batista	2015	SalesOrder	Framework C++ 1.0/2.0	Disciplina Programação Avançada	16	-	16	28	1	18	28	1	FW PON 1.0/2.0 e PI/POO	Apesar do Framework 2.0 apresentar desempenho melhor quando comparado ao Framework 1.0, a aplicação concebida em POO apresentou melhores resultados comparado ao PON de maneira geral.
Leonardo Pordeus	2015	Sort	PON HD	Disciplina Lógica Reconfigurável	N-1	2*(N-1)	N-1	N	-	-	-	N	N/A	Apresetou resultado O(n). O número de ciclos de clock para ordenação no pior caso, era igual ao número de elementos.
Leonardo Pordeus	2015	PWM	PON HD	Disciplina Lógica Reconfigurável	2	0	2	3	-	-	-	-	VHDL	A implementação em PON HD fez uso da mesma quantidade de unidades lógicas do que a implementação puramento em VHDL.
Ricardo Kerschbaumer	2015	Contador Digital	PON-HD + LingPONHD	Disciplina Compiladores	1	-	2	-	-	-	1	-	VHDL Manual	Performance e tamanho do circuito equivalentes
Robson Linhares	2015	Sort PON HD	PON HD Prototipal	Tese de Doutorado	45	41	62	0	-	-	-	-	VHDL e NOCA FPGA	-

Autor(es)	Ano	Nome da Aplicação	Materialização	Contexto	Rules	Sub Conditions	Premises	Instigations	FBE Classes	Attributes	Methods	Instâncias de FBE	Comparado com o que?	Resumo Resultado
Leonardo Pordeus	2015	Sort	PON HD	Disciplina Lógica Reconfigurável	N-1	2*(N-1)	N-1	N	-	-	-	-	N/A	Apresetou resultado O(n). O número de ciclos de clock para ordenação no pior caso, era igual ao número de elementos.
Leonardo Pordeus	2015	PWM	PON HD	Disciplina Lógica Reconfigurável	2	0	2	3	-	-	-	-	VHDL	A implementação em PON HD fez uso da mesma quantidade de unidades lógicas do que a implementação puramento em VHDL.
Ricardo Kerschbaumer	2015	Controlador Robô hexápode	PON-HD + LingPONHD	Disciplina Compiladores	127	-	50	-	-	-	103	-	N/A	Controlou o robô sem problemas
Ricardo Kerschbaumer	2015	Controlador Robô hexápode	PON-HD + LingPON	Disciplina Compiladores	127	-	50	-	-	-	103	-	N/A	Controlou o robô sem problemas
Ricardo Kerschbaumer	2015	Controlador simples de temperatura	PON-HD + LingPON	Disciplina Compiladores	3	-	4	-	-	-	2	-	N/A	Funcionou corretamente
Robson Linhares	2015	Sort PON HD	PON HD Prototipal	Tese de Doutorado	45	41	62	0	-	-	-	-	VHDL e NOCA FPGA	-
Ricardo Kerschbaumer	2015	Ordenador paralelo de dados (odd even sort)	PON-HD	Disciplina PON	1 à 2999	-	de 2 à 5998	-	-	-		-	VHDL Manual	Performance e tamanho do circuito equivalentes
Ricardo Kerschbaumer	2015	Ordenador paralelo de dados	PON-HD	Disciplina PON	1 à 2999	-	de 2 à 5998	-	-	-	2 à 5999	-	VHDL Manual	Performance e tamanho do circuito equivalentes
Ricardo Kerschbaumer	2016	Contador + Driver p/ 3 Displays de 7 segmentos	PON-HD + LingPONHD	Disciplina Lógica Reconfigurável	38	-	35	-	-	-	38	-	N/A	Funcionou corretamente
Marcos Talau	2016	DistributedFire Net Attr	NOP C++ 2.0-PONNetwork	Disciplina PON	2		6	2	6	3	2	6	N/A	A aplicação demonstrou que o PON é naturalmente distribuído.
Marcos Talau	2016	DistributedFire Net Prem	NOP C++ 2.0-PONNetwork	Disciplina PON	2		12	2	6	0	2	6	N/A	A aplicação demonstrou que o PON é naturalmente distribuído.
Marcos Talau	2016	SummerAirCond	NOP C++ 2.0-PONNetwork	Disciplina PON	7		12	7	9	6	4	9	N/A	A aplicação demonstrou que o PON é naturalmente distribuído.
Marcos Talau	2016	StressTest Net	NOP C++ 2.0-PONNetwork	Disciplina PON	n			n	4	n	8	4	N/A	Foi verificado que o uso de atributos via rede é mais eficiente que o uso de premisas via rede.
Fabio Negrini e Leonardo Pordeus	2016	CTA	LingPON 1.0	Disciplina PON	46	46	126	89	3	12	34	100	PI/POO (C++)	A aplicação PON se apresentou mais lenta que versão desenvolvida sob o PI/POO.
Fernando Suyama	2016	RNA_FMS (p/ a base de flores)	Framework C++ 2.0	Disciplina PON	17	0	10	45	3	39	5	9	PI - Linguagem C	PI obteve os menores tempos de execução.
Fernando Suyama	2016	RNA_FMS (p/ a base de câncer de mama)	Framework C++ 2.0	Disciplina PON	23	0	10	58	3	45	5	12	PI - Linguagem C	PI obteve os menores tempos de execução.
Tiago Martins	2016	Simulação de um veículo seguidor de linha utilizando V-REP e LingPon Embarcado	LingPon 1.0	Disciplina PON	10	13	39	10	1	20	10	1	PI - Linguagem C	A aplicação PI apresentou resultados superiores a LingPON.
Eduardo Bilk	2016	BoxePON	LingPON StaticCPP	Disciplina PON	14	14	64	93	-	6	17	3	PI/POO	A aplicação PON não apresentou ganho aparente em termos de consumo energético em placa embarcada.
Fernando Schutz	2016	RNAMLP para IRIS	Framework C++ 2.0	Tese de Doutorado	10	-	19		3	65	10	9	PI - Linguagem C	Tempo de execução muito mais longo na aplicação PON.
Fernando Schutz	2016	RNAMLP para IRIS	LingPON StaticCPP	Tese de Doutorado	9	9	18	9	3	37	9	9	PI - Linguagem C	Tempo de execução da aplicação PON semelhante à aplicação PI.
Luiz Viana Melo	2016	Controlador de rotação no eixo X de Hexacóptero	LingPON Fuzzy	Dissertação de Mestrado	35	0	12	35	1	1	1	1	Framework Fuzzy (Emanoel Koslosky)	A versão em PON teve desempenho inferior em relação ao sistema convencional. Porém, o desenvolvimento foi mais fácil.
Frederico Miranda	2016	C#Pon1.1Hash	Framework C# 1.0	Disciplina PON	4224	0	51	4224	1	4	1	1	PI/POO	Tempo de execução da aplicação PON próximo ao da aplicação PI/POO.
Wagner Barreto	2016	Portão Eletrônico Distribuído	Framework Java	Disciplina PON	6	0	7	4	1	2	3	1	RPC	Para N apertos de botão implementação RPC troca N mensagens, implementação PON troca 2N mensagens.
Wagner Barreto	2016	2 Phase Commit	Framework Java	Disciplina PON	4	2	19	7	2	12	7	2	RPC	Convencional 3N mensagens. PON N+3 mensagens.
Douglas Krug	2016	Torre de Hanói	LingPON 1.0	Disciplina PON	8	16	41	53	3	11	50	7	PI - Linguagem C	Aplicação LingPON se mostrou mais legível e com melhor desempenho quando comparada à solução PI.
Leonardo Santos	2016	Robocup - Framework C++ 2.0	Framework C++ 2.0	Dissertação de Mestrado	74	74	53	30	1	20	30	6	PI/POO	A aplicação apresentou melhores métricas de complexidade de código (linhas e tokens) e melhor nível de manutenibilidade.
Leonardo Santos	2016	Robocup - LingPON 1.0	LingPON 1.0	Dissertação de Mestrado	444	444	318	180	6	120	180	6	PI/POO e Framework C++ 2.0	Aplicação apresentou piores métricas de complexidade de código (linhas e tokens), devido principalmente à redundância de código da linguagem.

Autor(es)	Ano	Nome da Aplicação	Materialização	Contexto	Rules	Sub Conditions	Premises	Instigations	FBE Classes	Attributes	Methods	Instâncias de FBE	Comparado com o que?	Resumo Resultado
Leonardo Santos	2016	Robocup - LingPON 1.2	LingPON 1.2	Dissertação de Mestrado	74	74	53	30	1	20	30	6	PI/POO, Framework C++ 2.0 e LingPON 1.0	A aplicação apresentou métricas de complexidade de código similares ao código desenvolvido em Framework C++ 2.0 e melhores que o LingPON 1.0 e PI/POO.
Rodrigo Oliveira	2016	PON.IoT	Framework PON.IoT	Dissertação de Mestrado	14	59	600	7	11	11	17	11	N/A	Verificou-se a utilização do PON em sistemas sencientes IoT com Raspberry Pi
Leonardo Pordeus	2016	Mira Alvo	Ling PON (2.0) para NOCA	Dissertação de Mestrado	101	0	201	301	101	201	301	100	NOCASim	Análise em desenvolvimento.
Leonardo Pordeus	2016	Mira Alvo	Ling PON (2.0) para NOCA	Dissertação	101	0	201	301	-	-	-	-	NOCASim	Análise em desenvolvimento.
Leonardo Pordeus	2017	Semáfaros	Ling PON (2.0) para NOCA	Dissertação de Mestrado	-	-	-	-	-	-	-	-	NOCASim	Aplicação em desenvolvimento.
Leonardo Pordeus	2017	Sort	Ling PON (2.0) para NOCA	Dissertação de Mestrado	-	-	-	-	-	-	-	-	NOCASim	Aplicação em desenvolvimento.
Leonardo Pordeus	2017	Semáfaros	Ling PON (2.0) para NOCA	Dissertação	-	-	-	-	-	-	-	-	NOCASim	Aplicação em desenvolvimento.
Leonardo Pordeus	2017	Sort	Ling PON (2.0) para NOCA	Dissertação	-	-	-	-	-	-	-	-	NOCASim	Aplicação em desenvolvimento.

E.1 MIRA AO ALVO

A aplicação Mira Ao Alvo foi desenvolvida por Banaszewski [Banaszewski 2009] sob o *Framework* PON 1.0 sendo comparada com PI onde em suma os resultados de desempenho foram favoráveis ao Framework PON 1.0 dado inclusive a natureza da aplicação. Depois esta aplicação foi redesenvolvida no âmbito do trabalho de [Valença 2013] em *Framework* PON 2.0 e comparadas com *Framework* 1.0, sendo os resultados favoráveis ao *Framework* PON 2.0. Posteriormente, a mesma aplicação foi desenvolvida por Ferreira [Ferreira 2016] utilizando a LingPON 1.0, sendo comparada com Framework PON 2.0 obtendo resultados de desempenho melhores que este.

A aplicação intitulada Mira ao Alvo consiste em um ambiente onde as entidades do tipo mira interagem ativamente com as entidades do tipo alvo, conforme apresentado na Figura 78. 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.

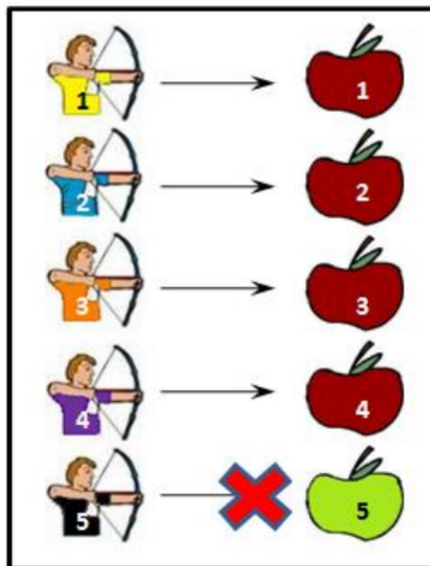

Figura 78: Representação da aplicação Mira ao Alvo [Banaszewski 2009]

.

Para fins de comparação, a presente aplicação foi desenvolvidas com algumas pequenas variações, afim de tornar a interação entre miras e alvos mais complexa 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 a inserção de novas entidades ao ambiente. Isto fez-se necessário para permitir melhor comparação, em termos de desempenho, entre diferentes paradigmas de programação, incluindo o PON e PI.

De um modo geral, as entidades miras e as entidades alvos são representadas,

respectivamente, por arqueiros e maçãs. Cada arqueiro e cada maça é identificado por um número, sendo que o arqueiro somente pode flechar uma maça que apresente o identificador numérico correspondente ao seu. Além desse atributo, cada maçã possui um atributo que representa seu estado atual, outro que explícita se a mesma já foi perfurada por uma flecha e, por fim, um atributo que se refere a sua coloração.

Neste contexto, cada arqueiro somente pode interagir com a respectiva maçã após a constatação de três condicionantes: (a) se a cor da maçã que está posicionada diretamente à sua frente é vermelho, (b) se a maçã que está posicionada diretamente à sua frente está pronta para ser atingida, (c) se ela é identificada pelo seu número correspondente (d) e se o início for autorizado (representado pelo tiro de uma arma) . Apenas se estas três condições forem satisfeitas, o arqueiro está liberado para atingir a respectiva maçã com a projeção de sua flecha, conforme apresentado na Figura 79.

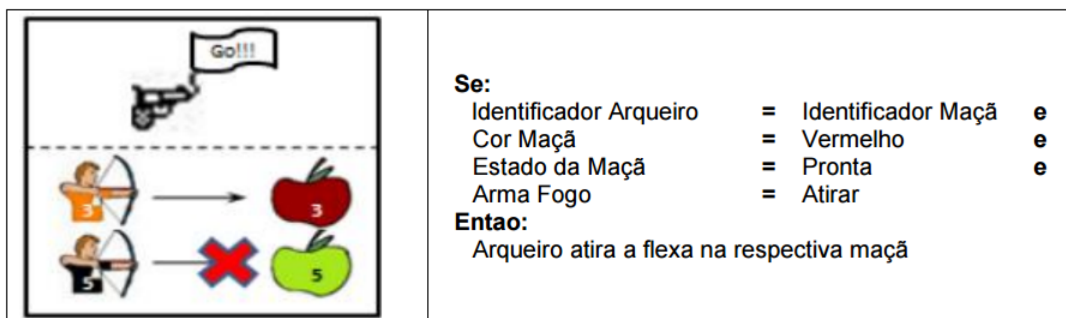

Figura 79: Exemplo de uma *Rule* presente na aplicação Mira ao Alvo

.

Isto dito, a aplicação foi desenvolvida primeiramente pela definição de *FBEAcher* e *FBEApple* com *Attributes* e *Methods* pertinentes. Ainda, um terceiro *FBE* nomeado *Controller* foi criado a fim de representar a arma que define o início da interação entre arqueiros e maças. Neste contexto, diferentes versões desta aplicação, com diferentes quantidades de *FBEs* e *Rules*, foram desenvolvidas com o objetivo de testar a performance das materializações PON. Em alguns casos, foram criadas até 100 *Rules*, sendo que cada *Rule* foi associada a uma das 100 instâncias criadas do *FBEArcher* e 100 instâncias do *FBEApple*. Entretanto, a criação destas *Rules* foi feita através de laços de repetição, fato este que não agrega complexidade real ao desenvolvimento da aplicação, pois trata-se apenas de repetição de *Rules*.

Através de comparações em termos de desempenho com uma solução funcionalmente equivalente desenvolvida no PI, foi possível demonstrar que em cenários em que os estados de atributos variam com baixa ou mediana frequência, a reatividade e pontualidade das notificações faz o PON prevalecer sobre o PI [Banaszewski 2009].

E.2 SISTEMA DE CONDICIONAMENTO DE AR

A aplicação de Sistemas de Condicionamento de Ar foi desenvolvida por Banaszewski [Banaszewski 2009], utilizando o *Framework* PON 1.0. Ainda que não tenha sido redesenvolvida ainda em *Framework* PON 2.0 ou LingPON 1.0, ela apresenta complexidade suficiente para merecer relato.

Isto dito, o Sistema de Condicionamento de Ar é hipoteticamente integrado a um edifício de 16 andares. Cada andar possui uma bomba de calor, uma entrada ajustável de ar e um sensor de temperatura (i.e termômetro), os quais tem os seus estados controlados por um componente centralizado nomeado Aplicação de Controle, conforme representado pela Figura 80.

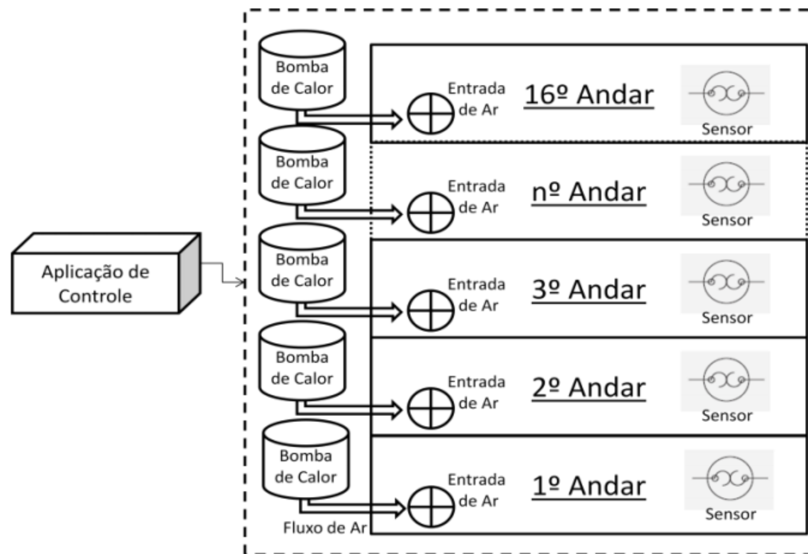

Figura 80: Representação do Sistema de Condicionamento de Ar [Banaszewski 2009].

Neste ambiente, uma bomba de ar tem a função de aquecer ou resfriar o ar em um determinado andar do edifício. Assim, em um determinado momento uma bomba de calor pode se encontrar no estado de *HEATING* (aquecendo), *COOLING* (resfriando) ou *OFF* (desligada). O ar enviado por uma bomba de ar entra no respectivo andar pela entrada de ar ajustável, a qual pode assumir o estado *OPENED* (aberta) ou *CLOSED* (fechada).

Por sua vez, um sensor de temperatura pode apresentar os seguintes estados, os quais representam a temperatura de um andar do edifício:

- **TARGET**: temperatura ideal pré-configurada.
- **UPPER e LOWER GUARD**: correspondem às temperaturas na faixa aceitável com variação de 2° C para mais ou para menos da temperatura ideal.

- **HOT e COLD**: correspondem repectivamente às temperaturas que variam 6° C para mais ou para menos da temperatura ideal.
- **TOO HOT e TOO COLD**: correspondem às temperaturas acima do estado HOT e abaixo do estado COLD.

Neste âmbito, a Aplicação de Controle controla os estados da bomba de calor e das entradas de ar afim de atingir e manter a temperatura ideal para cada andar do edifício. Deste modo, para cada estado de temperatura recebido, a aplicação analisa um conjunto de relações causais e decide sobre as alterações de estados pertinentes.

Para a criação da aplicação PON, foram criados tipos de **FBEs** com *Attributes* e *Methods* pertinentes, os quais foram apropriadamente instanciados. Os FBEs criados foram *HeatPump*, *Vent* e *Sensor*. Subsequentemente, de forma a tratar a regras de funcionamento da aplicação em questão, foram criadas 16 *Rules* para cada andar do edifícios. Como no ambiente simulado o edifício possui 16 andares, um total de 256 *Rules* foram criadas para o sistema de controle.

Afim de comparar o desempenho do PON com o PI em sistemas embarcados, esta aplicação foi desenvolvida sobre a plataforma embarcada relativa à placa de *hardware* *eSysTech* eAT55.

Os experimentos realizados com o Sistema de Condicionamento de Ar confirmaram a superioridade do PON, em termos de desempenho, em relação ao PI, mesmo quando executado sobre plataforma com recursos computacionais limitados. Isto se deve principalmente à redundâncias temporais e estruturais inerente ao código desenvolvido utilizando o PI [Banaszewski 2009].

E.3 SIMULADOR DE JOGO (*PACMAN*)

Desenvolvido sob o *Framework* PON 1.0 e 2.0 por Ronszcka [Ronszcka et al. 2011], esta aplicação consiste em um sistema de controle aplicado à um simulador com características do clássico jogo *Pacman*[1]. Assim como o jogo de inspiração, o ambiente simulado desenvolvido possui corredores que formam um labirinto, limitando as ações de movimento dos personagens no cenário, nomeadamente o *Pacman* e seus inimigos, os Fantasmas. Ainda, os personagens do simulador apresentam comportamento autônomo

[1] Os créditos do jogo e seus direitos autorais pertencem ao indivíduo que o produziu ou a empresa que o publicou. A utilização neste trabalho visa apenas o estudo acadêmico, sem fins lucrativos.

e predeterminado, ou seja, não são controlados por um usuário. A Figura 81 ilustra o ambiente criado pelo simulador.

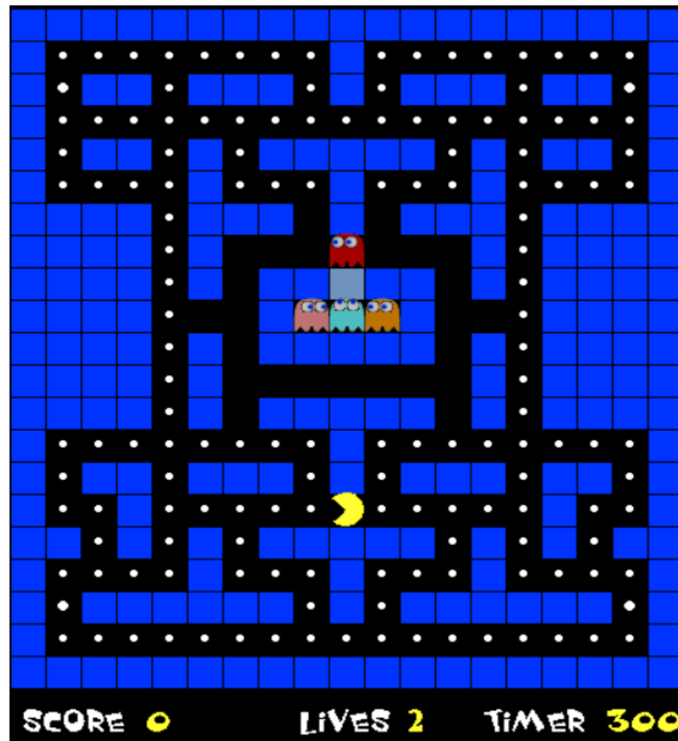

Figura 81: Ambiente gerado pelo simulador [Ronszcka et al. 2011].

Para o *Pacman*, o principal objetivo do jogo é maximizar os pontos ganhos. Ele acumula pontos ao percorrer os corredores do labirinto em busca de pastilhas, que são encontradas por todo o labirinto, dispondo de apenas 300 passos para tal. Enquanto busca maximizar o número de pontos, o *Pacman* deve ser capaz de fugir dos fantasmas que o perseguem.

O simulador foi desenvolvido puramente em PI/POO e apresenta particularidades que beneficiam a definição de regras para a movimentação dos personagens nos corredores do labirinto. Dentre tais particularidades, destaca-se a classificação de esquinas em categorias. As esquinas representam o encontro ou cruzamento de dois ou mais corredores que compõem o labirinto, cada qual com seu formato distinto. O labirinto utilizado no simulador é formado por 9 diferentes formatos de esquinas, com o objetivo de minimizar a quantidade de regras a serem criadas, uma vez que o tratamento das ações dos personagens é baseado nesta classificação.

A aplicação de controle deve ser capaz de controlar as ações do *Pacman* de forma a maximizar sua quantidade de pontos ao fim da execução. Para fins comparativos, a aplicação de controle foi desenvolvida em PON (*Framework* PON 1.0 e 2.0) e PI/POO.

No total, foram criadas 99 estruturas causais (*if-then-else* no PI/POO e *Rules* no PON) e 11 tipos de classes de *FBEs*, nomeados *Pacman*, *Ghost*, *Wall*, *Dot*, *EnergizerDot*, *Timer*, *Maze*, *Score*, *GameState*, *VisualField* e *Corner*.

Ademais, uma nova versão desta aplicação foi desenvolvida por Monte-Alto [Monte-Alto 2015] utilizando a LingPON 1.0. Entretanto, esta aplicação não seguiu a mesma modelagem da aplicação anterior, principalmente, segundo o autor, por limitações técnicas da LingPON 1.0. Nesta nova versão, a aplicação não possui interface gráfica. Ademais, foram definidas para esta nova aplicação 385 *Rules* e 7 *FBEs*, nomeados *Pacman*, *Ghost*, *Wall*, *Dot*, *EnergizerDot*, *TickerEvent* e *MazeLimits*.

Nos experimentos realizados com a aplicação desenvolvida sob o *Framework* PON 2.0, o PI/POO apresentou melhor desempenho quando comparado ao PON. Segundo analises do autor, o fato de o *Framework* PON estar materializado sobre a linguagem C++ afetou o desempenho da aplicação PON. Conclui-se então que uma linguagem de programação e um compilador para o PON deveriam ser desenvolvidos para que o PON alcançasse todo o seu potencial em termos de desempenho.

E.4 SIMULADOR DE TRANSPORTE INDIVIDUAL

Desenvolvido por Xavier [Xavier 2014] utilizando o *Framework* PON 2.0 e LingPON, essa aplicação tem como objetivo simular o controle de um possível exoesqueleto, conforme o apresentado na Figura 82. A origem do desenvolvimento desta aplicação surgiu de uma disciplina de mestrado em que se desenvolveu um simulador de um jogo utilizando simples gráficos 2D com essa temática, a partir do qual se isolou apenas a parte de tratamento de eventos para o desenvolvimento desta aplicação.

De maneira geral, esta aplicação caracteriza-se por um *software* simulador que recebe eventos de dispositivos externos (i.e *joystick* e respectivos botões). Por simplicidade, o autor desta aplicação decidiu por simular também os eventos provenientes dos dispositivos externos, por meio da criação de um *software* “gerador de eventos”.

Os principais requisitos do sistema são:

- O *software* simulador deverá receber eventos de um *joystick* com dois botões (um vermelho e um azul) e de uma chave liga e desliga.
- A chave liga e desliga o exoesqueleto como um todo.
- Os botões ligam e desligam um respectivo braço mecânico (direito ou esquerdo). O

Figura 82: Figura conceitual de um exoesqueleto do projeto Hardiman I da General Electric.

botão vermelho comanda o braço esquerdo, enquanto o botão azul comanda o braço direito.

- O *joystick* movimenta ora o próprio transporte (o exosqueleto), ora um único braço mecânico (separadamente esquerdo ou direito), ou mesmo ambos os braços simultaneamente. O movimento do braço somente ocorre caso o próprio esteja ligado (ou caso ambos os braços estejam ligados). Como requisito, caso ambos os braços mecânicos sejam ligados, ambos são movimentados em movimentação conjunta e na mesma direção. Dessa forma, o *joystick* movimenta o exosqueleto nos eixos X e Y, bem como controla os braços mecânicos (separadamente ou em conjunto) nos eixos X, Y e Z.

A solução técnica PON desenvolvida para esta aplicação apresenta um conjunto de vinte e duas *Rules* e quatro *FBEs*, a saber: *Exoskeleton*, *Arm*, *Event* e *Simulation.*

A aplicação desenvolvida foi utilizada para experimentos de comparação entre o PON e o Paradigma Orientado a Eventos (POE). Os experimentos mostraram que na aplicação POE, as decisões sobre os comportamentos da aplicação ficam dispersas pelo *software*. Em contrapartida, no caso do PON, a tomada de decisão fica expressa de forma centralizada ao utilizar *Rules* [Xavier 2014].

E.5 SISTEMA DE VENDAS

A aplicação Sistema de Vendas foi desenvolvida por Ronszcka utilizando o *Framework* PON 2.0 [Ronszcka et al. 2011]. Posteriormente, esta aplicação foi desenvolvida também a partir da LingPON 1.0 por Ferreira [Ferreira 2016].

De modo geral, esta aplicação consiste na implementação de um sistema de pedido de vendas usual, isto é, uma tradicional aplicação CRUD (acrônimo de *Create*, *Retrieve*, *Update* e *Delete*). O escopo dessa aplicação é resumido pelo diagrama de casos de uso ilustrado na Figura 83.

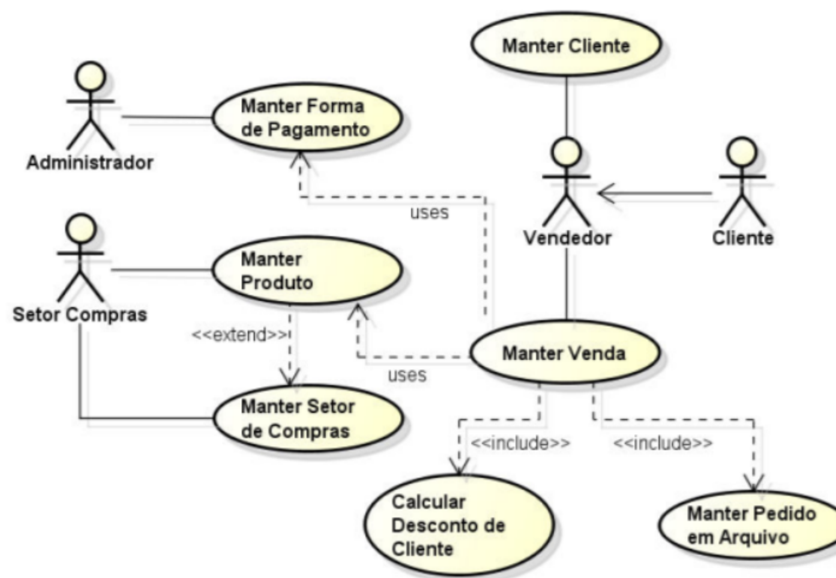

Figura 83: Casos de uso do Sistema de Vendas.

Conforme ilustra a Figura 83, o ator Administrador é responsável por manter as informações do cadastro de formas de pagamentos. O ator Setor de Compras, por sua vez, é responsável por cadastrar e atualizar as informações de produtos e do próprio setor de compras. Ainda, o ator Cliente, solicita uma venda a um respectivo ator Vendedor. Este, por sua vez, cadastra o cliente e efetua a venda propriamente dita.

Inicialmente o cliente (denotado pelo ator cliente) solicita a compra para um respectivo vendedor (denotado pelo ator vendedor). Assim, o vendedor informará o respectivo cliente que realizará o pedido. Uma vez escolhido e aprovado a venda para determinado cliente, deve ser informado o produto que irá compor o pedido. O sistema possui validações quanto à existência de produtos e clientes. Ademais, verifica-se o estoque disponível de tais produtos [Ferreira et al. 2013].

Após todo o ciclo de informe de produtos, a venda poderá ser finalizada após a

inserção da forma de pagamento. Na implementação desse sistema, existem apenas duas formas de pagamento possíveis, a Vista ou a Prazo. O cliente, em seu cadastro, possui uma informação sobre seu limite de crédito. Caso a forma de pagamento escolhida tenha sido a Prazo, o sistema verifica se o cliente tem permissão para efetuar a compra, confrontando o valor total do pedido com seu limite de crédito [Ferreira et al. 2013].

Ademais, no cadastro do cliente há uma informação que lhe concede um tipo de classificação. Utiliza-se tal classificação para a concessão de descontos especiais durante a finalização da venda. Para tanto, existe um total de 20 tipos de 106 classificação de clientes que dispõem de descontos que variam de uma faixa de 0% as 95% [Ferreira et al. 2013].

A título de exemplificação, a Figura 84 demonstra a composição da Rule responsável por finalizar uma venda. Nela estão relacionadas às Premises que deverão ser satisfeitas para que a finalização da venda ocorra. Assim, a primeira Premise verificaria se a forma de pagamento selecionada foi a prazo. Neste caso é necessário validar o limite de crédito disponível para o cliente, o qual faria parte da segunda Premise da Rule em questão. A terceira e última Premise validaria o tipo de desconto concedido para o cliente em questão (dentro os 20 possíveis tipos de descontos).

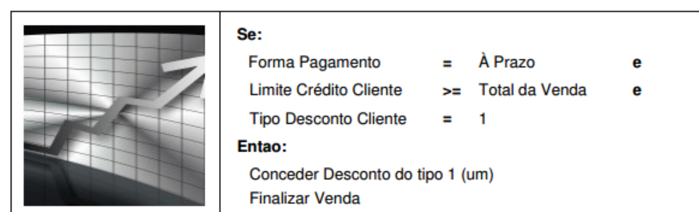

Figura 84: *Rule* responsável por finalizar a venda [Ferreira et al. 2013].

Para a implementação da aplicação foram criados 4 *FBEs*, nomeados *Product*, *Client*, *SalesOrder* e *SalesOrderItem*. Ademais, para compor a base de regras da aplicação foram criadas 20 *Rules* para cada instância do FBE *Product*.

A aplicação desenvolvida sob o *Framework* PON foi utilizada para verificar os benefícios da aplicação de padrões de projeto no desenvolvimento de aplicações PON. Concluiu-se que a utilização de padrões de projetos proporciona maior legibilidade e manutenibilidade à sistemas desenvolvidos utilizando o Framework PON.

A aplicação desenvolvida a partir da LingPON foi utilizada para fins de comparação de desempenho do código gerado pelo compilador com uma aplicação funcionalmente equivalente desenvolvida sob o viés do PI/POO C++. A versão em PI/POO C++ demonstrou desempenho pior quando comparada a aplicação desenvolvida em LingPON.

E.6 PORTÃO ELETRÔNICO

Esta aplicação foi inicialmente desenvolvida por Wiecheteck [Wiecheteck et al. 2011] e aprimorada por Batista [Batista 2013] e Xavier [Xavier 2014]. Trata-se de software exemplo também para tratamento de eventos, tendo como conceito um sistema de automação que recebe o evento de acionamento de um controle remoto para executar operações elementares como abrir e fechar automaticamente um portão. Na verdade, existem versões outras desta aplicação por ser usada como exercício na disciplina que trata de PON, disciplina esta lecionada junto ao CPGEI/UTFPR e PPGCA/UTFPR.

A aplicação é composta por quatro FBEs: *Gate*, *Timer*, *Event* e *Light*. *Gate* representa o portão eletrônico, *Timer* representa o contador de tempo, *Light* representa a lâmpada e *Event* representa os eventos recebidos. Para cumprir os requisitos desta aplicação foram implementadas 10 Rules.

Tanto em [Wiecheteck et al. 2011] como em [Batista 2013], esta aplicação foi utilizada para demonstração e validação de métodos e técnicas para o projeto de aplicações PON. Em [Xavier 2014], a aplicação foi desenvolvida a partir do *Framework* PON e LingPON. Estas aplicações foram comparadas com soluções semelhantes desenvolvidas a partir do POE. Constatou-se que a programação segundo o PON é mais fácil, principalmente pela forma como o conhecimento lógico-causal da aplicação é expresso em *Rules* [Xavier 2014].

E.7 CONTROLE DA ILUMINAÇÃO EM UMA CIDADE VIRTUAL 3D

Esta aplicação, desenvolvida utilizando o *Framework* PON C# 1.0 por Miranda [Miranda 2016] como trabalho final da disciplina de PON ofertada em 2016 na UTFPR, propõe a modelagem de um sistema para controlar a iluminação do ambiente em uma cidade virtual 3D. O relatório apresentado por Miranda nesta disciplina está presente no Anexo C deste trabalho

Neste contexto, cidade virtual foi considerada como sendo um ambiente tridimensional que reconstrói uma estrutura espacial complexa de uma cidade no computador e que pode ser utilizada para fins de planejamento e simulações (DÖLLNER et al., 2005 apud MIRANDA, 2016).

A aplicação PON desenvolvida neste contexto trata-se de um módulo desenvolvido para a cidade virtual 3D. Esta cidade virtual 3D é uma aplicação já existente, na qual é

possível simular diferentes situações presentes em uma cidade real, tais como variação no tráfego de automóveis e diferentes condições climáticas. Desse modo , aplicação PON foi desenvolvida de forma a interagir com esta cidade virtual e controlar o funcionamento do sistema de iluminação ambiente.

Para controlar a iluminação na cidade virtual foi criado apenas um *FBE*, o qual possui três atributos: clima (16 possibilidade), horário (24 possibilidades) e visibilidade (11 possibilidades). A partir destes três atributos, 4224 *Rules* foram criadas. Apesar da grande quantidade de entidades *Rules* presentes na aplicação, fato este que demonstraria um alto nível de complexidade, o sistema de controle de iluminação ainda é uma aplicação simples. Isto porque as *Rules* presentes na aplicação foram geradas a partir da simples combinação linear entre os possíveis valores do três Attributes presentes (clima, horário e visibilidade).

Ademais, esta aplicação é apenas um protótipo, o qual recebe um arquivo de texto com os parâmetros a serem processados, processa essas informações e disponibiliza o resultado através da linha de comando. Neste sentido, até o presente momento a aplicação PON não foi integrada com a cidade virtual, fato este que aumentaria o nível de complexidade da aplicação.

E.8 WARSHIPATTACK GAME

O software desenvolvido é um jogo de combate aéreo, sendo o equivalente a uma implementação simplificada do jogo clássico "*River Raid*". Esse jogo foi escolhido por possuir complexidade moderada, razoável para um estudo prospectivo.

Esse software possui uma interface gráfica em duas dimensões que apresenta o movimento de um avião controlado pelo jogador. O avião deve enfrentar inimigos que aparecem em posições aleatórias. O objetivo do jogador com sua aeronave é terminar uma fase vencendo o maior número de inimigos.

Para a implementação do software, apresentada em [Kossoski 2013], foi utilizado o *Framework* PON e uma biblioteca gráfica externa, a saber *Allegro* [Liballeg 2004]. A aplicação é composta por 6 entidades FBEs e 26 Rules.

Esta aplicação não foi comparada com aplicações desenvolvidas em outras materializações ou paradigmas de programação. O software foi utilizado para a apresentação de uma proposta de um método de teste para processos de desenvolvimento de aplicações PON.

E.9 CTA SIMULATOR

Desenvolvida por Leonardo Pordeus [Pordeus 2015] utilizando a LingPON, esta aplicação tem como objetivos desenvolver estratégias de controle de semáforos, simular regiões de tráfego em uma área urbana e comparar o desempenho de diferentes estratégias de controle. Para tal, esta aplicação deve ser capaz de simular elementos do mundo real, tais como veículos, ruas, semáforos e cruzamentos.

O *software* foi dividido em dois módulos: um módulo de simulação e outro de controle de estratégias. O módulo referente ao simulador foi desenvolvido utilizando a linguagem de programação C++, enquanto o módulo de controle de estratégias foi desenvolvido tanto em PI (C++) quanto em PON (LingPON), para fins de comparação de desempenho.

Para modelar o sistema em entidades PON, foi criado apenas uma classe de *FBE* para representar o semáforo. A quantidade de *Rules* depende da estratégia de controle adotada, sendo que a estratégia com maior número de *Rules* apresenta 18 *Rules*. Em um dos cenários de teste que Pordeus utilizou, foram criados ao menos 100 pares de semáforos e um total de 1800 *Rules*. Entretanto, isto não altera o nível de complexidade da aplicação, uma vez que trata-se apenas de 100 repetições do mesmo conjunto de 18 *Rules*.

Ao comparar o módulo de controle de estratégia desenvolvido em PON com a solução desenvolvida em PI/POO, concluiu-se que a aplicacação desenvolvida segundo o PON apresenta desempenho muito próximo ao desempenho apresentado pela aplicação PI/POO.

E.10 TORRE DE HANÓI

A Torre de Hanói é um problema clássico da matemática, fortemente utilizado como um método lúdico para desenvolver o raciocínio, e também muito utilizado no ensino de lógica e linguagens de programação como um Código que utiliza recursividade [Krug 2016].

Criado em 1883 pelo matemático Édouard Lucas, este problema consiste em um jogo com três hastes e um determinado número de discos posicionados inicialmente na haste da esquerda em ordem decrescente de tamanho, conforme ilustrado na Figura 85.

O objetivo do jogo é mover todos os discos para uma das hastes auxiliares. Embora o objetivo do jogo seja simples, duas regras devem ser respeitadas:

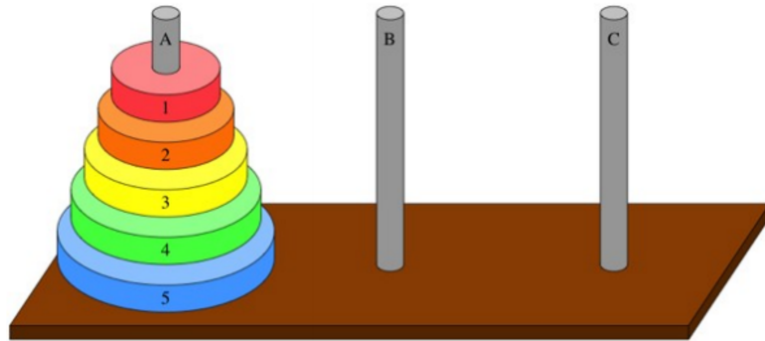

Figura 85: Configuração Inicial da Torre de Hanói [Krug 2016].

- Apenas um disco pode ser movido por vez.
- Um disco maior nunca pode ser colocado sobre um disco menor.

Para solucionar esse problema, Krug desenvolveu uma aplicação que soluciona o problema da Torre de Hanói utilizando a LingPON 1.0. Esta aplicação foi apresentada como trabalho final da disciplina de PON ofertada em 2016 na UTFPR. O relatório apresentado por Krug nesta disciplina está presente no Anexo C deste trabalho

No código-fonte LingPON foram criadas três entidades *FBEs*: Haste, Disco e Controle. Devido a dificuldades encontradas na atual versão da LingPON, a implementação ficou restrita a apenas 3 discos e 8 *Rules* [Krug 2016].

A solução desenvolvida em LingPON teve seu desempenho e facilidade de programação comparado com uma solução equivalente desenvolvida utilizando o Paradigma Procedimental (PP) através da linguagem de programação C. Constatou-se que o desenvolvimento da solução em LingPON foi mais fácil quando comparado ao PP. Isto porque o código em LingPON é mais claro e de mais fácil leitura do que o código em C. Em termos de desempenho, a aplicação LingPON apresentou resultados melhores do que a solução PP. Em um cenário de teste no qual a aplicação deveria resolver 1000 vezes o mesmo problema, a aplicação PP demorou 29.29 *s*., enquanto a solução desenvolvida em LingPON 1.0 e compilada para linguagem C demorou apenas 5 *s*. [Krug 2016].

E.11 ALGORITMO TRIANGULAR MESH SLICING

Uma técnica comum ao processo de manufatura de peças industriais é a construção de um objeto sólido tridimensional (3D) a partir de sucessivas camadas planas de material. Os dados para a construção do objeto vêm de um modelo geométrico gerado por sistemas

computacionais, tais como Computer-Aided Design (CAD) ou obtido a partir de Tomografia Computadorizada. O modelo é então fatiado, a fim de obter superfícies bidimensionais, representando cada camada em que o material de fabricação será adicionado. Uma vez processado, esses dados são enviados para uma máquina que "imprime" cada camada até que o objeto seja construído [Volpato 2007].

Para fins de comparação, Krachinski [Krachinski et al. 2015] implementou o algoritmo de corte tradicional utilizado no processo de manufatura de peças 3D, com base na aplicação desenvolvida por Gregori [Gregori et al. 2012], utilizando o Framework PON e LingPON, e os comparou com uma solução equivalente desenvolvida sob o PI/POO.

O algoritmo de corte tradicional consiste em interceptar uma malha de triângulos por planos 2D imaginários, variando a coordenada em relação ao eixo Z. A estratégia utilizada é simples e até mesmo lenta, mas útil para fins de comparação de desempenho.

A implementação em PON desta aplicação consiste em três classes de *FBE*, a saber *NOPTriangle*, *NOPTraditionalMesh* e *NOPOrderedMesh*, e uma única *Rule*. O *FBE* *NOPTriangle* possui dois *Attributes*, *atZMin* e *atZMax*, os quais têm seus estados avaliados pela única *Rule* presente na aplicação.

Após comparações de desempenho em diferente cenários, concluiu-se que as aplicações PON apresentaram resultados piores do que o a solução desenvolvida sob o PI.

E.12 REFLEXÃO

Conforme apresentado neste Apêndice, uma série de aplicações PON foram desenvolvidas utilizando o *Framework* PON e a LingPON. Cada uma das aplicações apresentadas tinha um objetivo específico, como comparações de desempenho ou facilidade de programação.

À luz do levantamento sobre as aplicações PON realizado neste trabalho, é constatado que seria interessante para a consolidação do PON um experimento que envolvesse o desenvolvimento de uma aplicação de maior envergadura que as previamente desenvolvidas. A aplicação desenvolvida poderia ser utilizada para comparações entre as duas materializações em *software* mais estáveis do PON, nomeadamente *Framework* PON 2.0 e LingPON. Ademais, a comparação das aplicações PON com uma solução funcionalmente equivalente desenvolvida por outrem, externo ao grupo de pesquisa PON, em PI/POO seria de grande valor para a consolidação do PON frente ao demais paradigmas de programação.

Conforme é apresentado na seção 2.4, o *software* de controle de futebol de robôs apresenta-se como uma excelente aplicação a ser desenvolvida sob o PON. Isto porque esta é reconhecida como uma aplicação complexa perante a sociedade científica, a qual é comumente utilizada para validação de novos conceitos e técnicas relacionados à computação [Yoon 2015]. Portanto, a apresentação desta aplicação sob o viés do PON poderá ser de considerável valor, ou ao menos pertinente, para futuras publicações a respeito do PON.

APÊNDICE F – PROGRAMAÇÃO ORIENTADA A AGENTES

Um sistema baseado em agentes é aquele construído sobre um determinado nível de abstração, chamado agente [Wooldridgey e Ciancarini 2001]. Sistemas baseado em agentes podem ser compostos por apenas um agente, entretanto seu maior potencial reside na aplicação de sistemas compostos por multi agentes. Por agente, entende-se uma entidade computacional que apresenta as seguintes propriedades:

- Autonomia: agentes possuem informações sobre seu estado atual e são capazes de tomar decisões sobre o que fazer sem intervenção direta humana ou de outros agentes. Tais informação são mantidas encapsuladas pelo agente, de modo que não seja acessível a nenhum outro agente que compõe o sistema.
- Reatividade: todo agente está inserido em ambiente e deve ser capaz de, percebendo qualquer alteração no meio em que está inserido, responder a tais impulsos.
- Proatividade: agentes não simplesmente agem em resposta ao ambiente em que estão inseridos, eles devem ser capaz de, por iniciativa própria, tomar decisões que os aproximem de seus objetivos.
- Habilidade Social: agentes se comunicam uns com outros através de uma determinada linguagem de comunicação [Genesereth e Ketchpel 1994] e, normalmente, possuem a habilidade de cooperar um com os outros de forma a alcançar objetivos comuns.

Ao projetar um sistema baseado em agentes, é importante determinar o quão sofisticado será o raciocínio dos agentes. Entende-se por raciocínio o processo de escolha de qual ação será executada. Agentes deliberativos ou cognitivos possuem um modelo explícito e simbólico do ambiente em que está inserido e, portanto, são capazes de raciocinar, planificar e negociar com outros agentes a fim de coordenar suas ações [Wooldridge et al. 1995, Nwana 1996]. Por outro lado, agentes reativos são aqueles que não possuem modelos internos de seus ambientes, ao invés disso, agem apenas de forma a responder à estímulos externos [Nwana 1996].

Agentes reativos apresentam certa semelhança com as entidades computacionais que compõem o PON. Ao ter seu estado alterado, por exemplo, um *Attribute* envia uma notificação às entidades *Condition* associadas. O *Attribute* não executa nenhum tipo de deliberação para decidir se deve ou não enviar a notificação, ele apenas a envia como resposta à um estímulo (mudança de seu estado).

Normalmente, programadores familiarizados com abordagens orientada a objetos não conseguem perceber nenhuma novidade na abordagem orientada a agentes. Isto porque objetos são definidos como entidades computacionais que encapsulam algum estado, são capazes de executar ações ou métodos neste estado e podem comunicar-se através de mensagens. Portanto, há suposta relação entre objetos e agentes [Wooldridgey e Ciancarini 2001]. Entretanto, apesar de existirem evidentes semelhanças, existem diferenças significativas entre agentes e objetos que justificam sua diferenciação como nível de abstração.

A primeira diferença a ser destacada é referente ao grau de autonomia. Em linguagens de programação como *Java* e C++, pode-se declarar atributos e métodos com escopo privado, ou seja, são acessíveis somente de dentro do próprio objeto, respeitando assim o princípio de encapsulamento da orientação a objetos. Sendo assim, um objeto pode ser idealizado como exibindo autonomia sobre o seu próprio estado. Porém, um objeto não apresenta, por definição, controle sobre seu comportamento. Ademais, caso um dado objeto tenha um método *"m"* de escopo público, qualquer outro objeto que compõe o sistema poderá invocar tal método, alterando seu estado interno. Nesse caso, o objeto não terá controle sobre quando o método será invocado [Wooldridgey e Ciancarini 2001].

Em um sistema multi agentes, entretanto, não existe a ideia de um agente invocando métodos uns dos outros, mas sim solicitando que uma determinada ação seja executada. Quando um agente recebe tal solicitação, ele pode decidir por executar ou descartar a ação. Portanto, o controle sobre qual ação será executada é diferente para sistemas orientado a agente e objetos. No caso orientado a objetos, a decisão encontra-se com o objeto que chama o método. Já no caso orientado a agentes, a decisão recai sobre o agente que recebe a solicitação [Wooldridgey e Ciancarini 2001].

Outra importante diferença diz respeito ao controle de execução. Por definição, cada agente tem sua própria *thread* de execução, a qual o mantém em um ciclo infinito de observação do ambiente, atualizando seu estado interno e selecionando quais ações serão executadas, gerando assim descentralização de processamento [Resnick 1997].

ANEXO A – BNF DA LINGPON 1.0

Neste anexo é apresentado a especificação da linguagem de programação da LingPON 1.0 segundo Backus-Naur Form (BNF) extraída de [Ferreira 2016].

Código 1: Especificação da linguagem PON LingPON 1.0 segundo Backus-Naur Form (BNF) [Ferreira 2016].

```
1   PROGRAM                       : fbes inst strategy rules main
2                                 | fbes inst strategy rules
3                                 ;
4
5   inst                          : INST declarations END_INST
6                                 ;
7
8   strategy                      : STRATEGY estrategy_declaration END_STRATEGY
9                                 ;
10
11  estrategy_declaration         : NO_ONE
12                                | BREADTH
13                                | DEPTH
14                                ;
15
16
17  declarations                  : declaration
18                                | declaration declarations
19                                ;
20
21  declaration                   : type ids
22                                ;
23
24                                ;
25  ids                           : id
26                                | id COMMA ids
27                                ;
28
29  rules                         : rule
30                                | rule rules
31                                ;
32
33  rule                          : RULE rule_body END_RULE
34                                | RULE id rule_body END_RULE
35                                | RULE depends id rule_body END_RULE
36                                | RULE id depends id rule_body END_RULE
37                                ;
38
39  depends                       : DEPENDS
40                                ;
41
42  rule_body                     : decl_condition decl_action
43                                | decl_properties decl_condition decl_action
44                                ;
45
46  decl_properties               : PROPERTIES properties_body END_PROPERTIES
```

```
47                                ;
48
49  properties_body               : properties_type value
50                                | properties_body properties_type value
51                                ;
52
53  properties_type               : PRIORITY
54                                | KEEPER
55
56  decl_condition                : CONDITION condition_body END_CONDITION
57                                | CONDITION id condition_body END_CONDITION
58                                ;
59
60  condition_body                : subcondition operator condition_body
61                                | subcondition
62                                ;
63
64  operator                      : AND
65                                | OR
66                                ;
67
68  subcondition                  : SUBCONDITION id subcondition_body END_SUBCONDITION
69                                ;
70
71  subcondition_body             : premise AND subcondition_body
72                                | premise
73                                ;
74
75  premise                       : PREMISE exp
76                                | PREMISE id exp
77                                | PREMISE IMP exp
78                                | PREMISE IMP id exp
79                                ;
80
81  exp                           : fator comp fator
82
83  comp                          : EQ
84                                | NE
85                                | LT
86                                | GT
87                                | LE
88                                | GE
89                                ;
90
91  fator                         : id
92                                | NUMBER
93                                | boolean
94                                | FLOATVALUE
95                                | STRINGVALUE
96                                | CHARVALUE
97                                ;
98
99  boolean                       : TRUE
100                               | FALSE
101                               ;
102
103 decl_action                   : ACTION action_body END_ACTION
104                               | ACTION id action_body END_ACTION
105                               ;
106
107 action_body                   : action_elements action_body
108                               | action_elements
109                               ;
110
111 action_elements               : instigation
112                               | method_use
113                               | exp SEMICOLON
114                               ;
115
116 instigation                   : INSTIGATION method_use
117                               | INSTIGATION id method_use
118                               ;
119
```

```
120  method_use                   : id LP RP SEMICOLON
121                               ;
122
123  id                           : ID
124                               | ID POINT ID
125                               ;
126
127  fbes                         : fbe
128                               | fbe fbes
129                               ;
130
131  fbe                          : FBE fbe_body END_FBE
132                               | FBE id fbe_body END_FBE
133                               ;
134
135  fbe_body                     : decl_attributes decl_methods
136                               | decl_attributes
137                               ;
138
139  decl_attributes              : ATTRIBUTES attributes END_ATTRIBUTES
140                               ;
141
142  attributes                   : attributes_body
143                               | attributes_body attributes
144                               ;
145
146  attributes_body              : type id value
147                               | type id SEMICOLON
148                               ;
149
150  type                         : BOOLEAN
151                               | INTEGER
152                               | PFLOAT
153                               | STRING
154                               | CHAR
155                               | id
156                               ;
157
158  value                        : NUMBER
159                               | boolean
160                               | id
161                               | FLOATVALUE
162                               | STRINGVALUE
163                               | CHARVALUE
164                               ;
165
166  decl_methods                 : METHODS methods END_METHODS
167                               ;
168
169  methods                      : method_body
170                               | method_body methods
171                               ;
172
173  method_body                  : METHOD id LP id ASSIGN id method_operator value RP
174                               | METHOD id LP id ASSIGN id method_operator id RP
175                               | METHOD id LP id ASSIGN value RP
176                               | METHOD id LP id ASSIGN id RP
177                               | METHOD id LP RP INNER_CODE_METHOD
178                               ;
179
180  method_operator              : PLUS
181                               | MINUS
182                               | MULT
183                               | DIV
184                               ;
185
186  main                         : MAIN INNER_CODE_MAIN
```

ANEXO B – BNF DA LINGPON 1.2

Neste anexo é apresentado a especificação da linguagem de programação da LingPON 1.2 segundo Backus-Naur Form (BNF). Em vermelho encontra-se destacado as diferenças entre a BNF da LingPON 1.0 e LingPON 1.2.

Código 1: Especificação da linguagem PON LingPON 1.2 segundo Backus-Naur Form (BNF).

```
1   PROGRAM                         : fbes inst strategy rules main
2                                   | fbes inst strategy rules
3                                   ;
4
5   inst                            : INST declarations END_INST
6                                   ;
7
8   strategy                        : STRATEGY estrategy_declaration END_STRATEGY
9                                   ;
10
11  estrategy_declaration           : NO_ONE
12                                  | BREADTH
13                                  | DEPTH
14                                  ;
15
16
17  declarations                    : declaration
18                                  | declaration declarations
19                                  ;
20
21  declaration                     : type ids
22                                  ;
23
24                                  ;
25  ids                             : id
26                                  | id COMMA ids
27                                  ;
28
29  rules                           : rule
30                                  | rule rules
31                                  | formRule
32                                  | formRule rules
33                                  ;
34
35  rule                            : RULE rule_body END_RULE
36                                  | RULE id rule_body END_RULE
37                                  | RULE depends id rule_body END_RULE
38                                  | RULE id depends id rule_body END_RULE
39                                  ;
40
41  formRule                        : FORM_RULE rule_body END_FORM_RULE
42                                  | FORM_RULE id rule_body END_FORM_RULE
43                                  | FORM_RULE depends id rule_body END_FORM_RULE
44                                  | FORM_RULE id depends id rule_body END_FORM_RULE
```

```
45                           ;
46
47  fbeRules                 : fbeRule
48                           | fbeRule fbeRules
49                           ;
50
51  fbeRule                  : FBE_RULE rule_body END_FBE_RULE
52                           | FBE_RULE id rule_body END_FBE_RULE
53                           | FBE_RULE depends id rule_body END_FBE_RULE
54                           | FBE_RULE id depends id rule_body END_FBE_RULE
55                           ;
56
57  rule_body                : decl_condition decl_action
58                           | decl_properties decl_condition decl_action
59                           ;
60
61  decl_properties          : PROPERTIES properties_body END_PROPERTIES
62                           ;
63
64  properties_body          : properties_type value
65                           | properties_body properties_type value
66                           ;
67
68  properties_type          : PRIORITY
69                           | KEEPER
70
71  decl_condition           : CONDITION condition_body END_CONDITION
72                           | CONDITION id condition_body END_CONDITION
73                           ;
74
75  condition_body           : subcondition operator condition_body
76                           | subcondition
77                           ;
78
79  operator                 : AND
80                           | OR
81                           ;
82
83  subcondition             : SUBCONDITION id subcondition_body END_SUBCONDITION
84                           ;
85
86  subcondition_body        : premise AND subcondition_body
87                           | premise
88                           ;
89
90  premise                  : PREMISE exp
91                           | PREMISE id exp
92                           | PREMISE IMP exp
93                           | PREMISE IMP id exp
94                           ;
95
96  exp                      : fator comp fator
97
98  comp                     : EQ
99                           | NE
100                          | LT
101                          | GT
102                          | LE
103                          | GE
104                          ;
105
106 fator                    : id
107                          | NUMBER
108                          | boolean
109                          | FLOATVALUE
110                          | STRINGVALUE
111                          | CHARVALUE
112                          ;
113
114 boolean                  : TRUE
115                          | FALSE
116                          ;
117
```

```
118  decl_action                    : ACTION action_body END_ACTION
119                                 | ACTION id action_body END_ACTION
120                                 ;
121
122  action_body                    : action_elements action_body
123                                 | action_elements
124                                 ;
125
126  action_elements                : instigation
127                                 | method_use
128                                 | exp SEMICOLON
129                                 ;
130
131  instigation                    : INSTIGATION method_use
132                                 | INSTIGATION id method_use
133                                 ;
134
135  method_use                     : id LP RP SEMICOLON
136                                 ;
137
138  id                             : ID
139                                 | ID POINT id
140                                 ;
141
142  fbes                           : fbe
143                                 | fbe fbes
144                                 ;
145
146  fbe                            : FBE fbe_body END_FBE
147                                 | FBE id fbe_body END_FBE
148                                 ;
149
150  fbe_body                       : decl_attributes decl_methods fbeRules
151                                 | decl_attributes decl_methods
152                                 | decl_attributes
153                                 ;
154
155  decl_attributes                : ATTRIBUTES attributes END_ATTRIBUTES
156                                 ;
157
158  attributes                     : attributes_body
159                                 | attributes_body attributes
160                                 ;
161
162  attributes_body                : type id value
163                                 | type id SEMICOLON
164                                 ;
165
166  type                           : BOOLEAN
167                                 | INTEGER
168                                 | PFLOAT
169                                 | STRING
170                                 | CHAR
171                                 | id
172                                 ;
173
174  value                          : NUMBER
175                                 | boolean
176                                 | id
177                                 | FLOATVALUE
178                                 | STRINGVALUE
179                                 | CHARVALUE
180                                 ;
181
182  decl_methods                   : METHODS methods END_METHODS
183                                 ;
184
185  methods                        : method_body
186                                 | method_body methods
187                                 ;
188
189  method_body                    : METHOD id LP id ASSIGN id method_operator value RP
190                                 | METHOD id LP id ASSIGN id method_operator id RP
```

```
191                                          | METHOD id LP id ASSIGN value RP
192                                          | METHOD id LP id ASSIGN id RP
193
194                                          | METHOD id LP RP INNER_CODE_METHOD
195                                          ;
196
197      method_operator                     : PLUS
198                                          | MINUS
199                                          | MULT
200                                          | DIV
201                                          ;
202
203      main                                : MAIN INNER_CODE_MAIN
```

ANEXO C – RELATÓRIOS AINDA NÃO PUBLICADOS SOBRE APLICAÇÕES PON

Neste anexo se encontram dois relatórios no formato de artigos. Esses relatórios foram o resultado da disciplina “Tópicos Especiais Em Engenharia da Computação: Paradigma Orientado A Notificações” ofertada pelo Curso de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI/UTFPR), sob a supervisão do prof. Dr. Jean Marcelo Simão. Esses relatórios foram referenciados na Subseção E deste trabalho. Por fim, os relatórios anexados tiveram o acordo dos autores e do professor da disciplina.

Torre de Hanói com LingPON – Paradigma Orientado a Notificações

Douglas Lusa Krug
Instituto Federal do Paraná - IFPR
União da Vitória, PR
douglas.krug@ifpr.edu.br

***Resumo*- Este artigo relata a experiência da implementação de um problema clássico de lógica de programação, a Torre de Hanói, utilizando o Paradigma Orientado a Notificações – PON, utilizando a Linguagem PON – LingPON. Através dele é possível entender um pouco mais sobre este novo paradigma, observando algumas comparações e sugestões de melhoria para o seu desenvolvimento.**

***Abstract*- This article reports the experience of implementing a classical problem in programming logic, Hanoi Tower, using the NOP - Notification Oriented Paradigm, using the NOP language - *LingPON*. Through it is possible understand a little more about this new paradigm, noting some comparisons and improvement suggestions for its development.**

***Palavras chave*- Paradigma Orientado a Notificações; Lógica de Programação; Torre de Hanói.**

***Keywords*- Notification Oriented Paradigm; Programming Logic; Hanoi Tower.**

I. Introdução

O presente artigo relata o desenvolvimento de um problema clássico no ensino de lógica de programação, Torre de Hanói, utilizando o Paradigma Orientado a Notificações – PON, materializado em Linguagem PON – LingPON.

O Paradigma Orientado a Notificações – PON, vem sendo desenvolvido por um grupo de pesquisadores da Universidade Tecnológica Federal do Paraná – UTFPR, apresentando-se como uma alternativa para o desenvolvimento de aplicações em plataforma de *software* e *hardware*, ele se propõe a resolver certos problemas existentes nos paradigmas usuais de programação, como o Paradigma Imperativo – PI, e o Paradigma Declarativo – PD [1][2].

Desde sua concepção inicial, o PON vem passando por evoluções e sendo validado em diversas aplicações, tanto em *software* como em *hardware*.

Em termos de desenvolvimento de *software*, sua primeira aplicação foi através de um *framework*, desenvolvido em C++, em sua evolução como paradigma, nasceu a Linguagem PON, denominada LingPON.

Para validar a facilidade de utilização da LingPON foi escolhido um problema clássico no estudo de lógica de programação, a Torre de Hanói, com o intuito de validar a facilidade de programação da linguagem, e a versatilidade, pois este problema é usualmente resolvido utilizando recursividade.

Inicialmente será abordado um pouco sobre o PON e sua materialização em LingPON, na sequência um breve histórico sobre a Torre de Hanói e sua explicação.

Será também explicado a forma de desenvolvimento utilizando a LingPON e uma comparação de desempenho entre o desenvolvimento em PON e o desenvolvimento no Paradigma Procedimental - PP.

Para finalizar serão apresentadas algumas dificuldades encontradas, acompanhada de sugestões de melhoria para a LingPON.

II. Paradigma Orientado a Notificações

O Paradigma Orientado a Notificações – PON apresenta melhorias em comparação aos paradigmas vigentes, que por tempo vivem uma inércia de evolução, corrigindo certas deficiências apresentadas nestes, mas também aproveitando-se de pontos fortes que consagraram estes paradigmas.

O PON encontra inspirações no PI, como a flexibidade algorítimica e a abstração de classes/objetos da Programação Orientada a Objetos – POO, assim como aproveita conceitos próprios do PD, como a facilidade de programação em alto nível e a representação do conhecimento em regras, dos Sistemas Baseados em Regras – SBR [3].

Os principais elementos do PON são as *Fact Base Elements* – FBE e as *Rules*, as FBEs podem ser associadas a objetos do mundo real, e as rules podem ser associadas a regras de relação lógico causal.

O modelo e a lógica de funcionamento do PON podem ser descritos da seguinte forma: As *Rules* são compostas por *Conditions* e *Actions*, as *Conditions* podem se relacionar com uma ou mais *Premisses*, que por sua vez, são responsáveis por verificar os *Attributes* de uma FBE. Cada *Premisse* é composta por uma referência a um *Attribute*, por um operador lógico e um valor. Esta referência de um *Attribute* utilizada na *Premisse* é quem notifica a mudança de seu estado.

Uma *Rule* pode ser composta de uma ou mais *Premisses*, a partir do momento em que todas as *Premisses* são aprovadas, a *Rule* é aprovada e notifica uma *Action*. A *Action* referencia uma ou mais *Instigations*, as quais são associadas a *Methods* da FBE. Sempre que o valor de um *Attribute* é alterado ele mesmo notifica as *Premisses* que são relacionadas a ele. As *Premises*, por sua vez, são reavaliadas e, através de uma operação lógica é comparada ao novo valor do *Attribute* com uma constante ou um valor notificado por outro *Attribute*. Caso o resultado lógico da reavaliação da entidade *Premises* seja alterado, a *Premise* notifica um conjunto de entidades

Conditions relacionadas a ela. Em seguida, as *Conditions* também têm seus estados lógicos reavaliados de acordo com os resultados lógicos das *Premises*. Assim, quando todas as entidades *Premises* que compõem uma entidade *Condition* apresentam seus valores lógicos verdadeiros, a entidade *Condition* também é satisfeita, aprovando a execução da sua respectiva *Rule*. Com isso, a entidade *Action* agregada a esta *Rule* é executada, invocando os *Methods* necessários através das entidades *Instigations* [4].

A partir deste mecanismo de notificações é possível desenvolver programas com melhor desempenho, menor número de redundâncias estruturais e temporais. Estes programas são mais apropriados para paralelismo e distribuição do que os sistemas computacionais desenvolvidos por meio das soluções baseadas em paradigmas atuais [5].

Uma das formas de materialização do PON para *software* é utilizando a LingPON e o compilador construído para ela.

De modo geral, o código fonte da LingPON segue um padrão de declarações. Primeiramente, o desenvolvedor precisa definir os FBEs de seu programa. Em seguida, o desenvolvedor precisa declarar as instâncias de tais FBEs, bem como definir a estratégia de escalonamento das *Rules*. Subsequentemente, é necessário definir as *Rules* para fins de avaliação lógico causal dos estados do FBEs por meio de notificações. Por fim, é possível adicionar código específico da linguagem alvo escolhida no processo de compilação (e.g. C ou C++) com a utilização do bloco de código *main* [6].

III. Torre de Hanói

A Torre de Hanói é um problema clássico da matemática, fortemente utilizado como um método lúdico para desenvolver o raciocínio, e também muito utilizado no ensino de lógica e linguagens de programação como um exemplo de algoritmo que utiliza recursividade.

Este problema foi criado em 1883 pelo matemático Francês Édouard Lucas e consiste em um jogo com três hastes, e um determinado número de discos, postos inicialmente na haste da esquerda, em ordem decrescente de tamanho, conforme ilustrado na Figura 1.

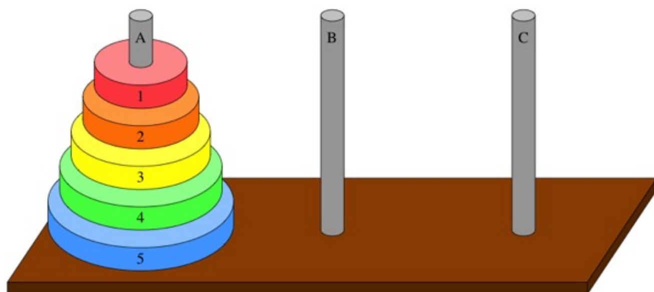

Figura 1. Torre de Hanói – Configuração Inicial [7]

O objetivo do jogo é mover todos os discos para uma das hastes auxiliares, ficando conforme ilustrado na Figura 2.

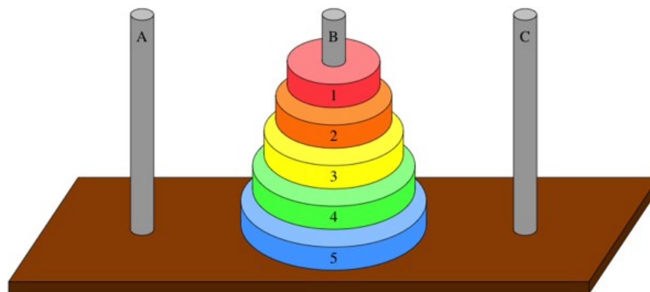

Figura 2. Torre de Hanói – Final [7]

Embora o objetivo do jogo seja simples, duas regras devem ser seguidas:

1. Apenas um disco pode ser movido por vez;
2. Um disco maior nunca pode ser colocado em cima de um disco menor.

É também esperado que o objetivo seja cumprido com um número mínimo de movimentos, representado pela fórmula matemática (2^n - 1), onde *n* é igual ao número de discos.

Desta forma, quando temos apenas 3 discos o número mínimo de movimentos necessário é 7, para 5 discos é 31 e assim por diante.

O objetivo de escolha deste problema é validar o paradigma e a LingPON quanto a facilidade de programação e sua versatilidade, pois usualmente este problema é resolvido utilizando recursividade.

IV. Desenvolvimento em LingPON

A materialização escolhida para o desenvolvimento do trabalho foi LingPON para atender ao objetivo que foi proposto, e compilado para C, C++ 1.0 e C++ 1.1.

Inicialmente foram definidas as FBEs necessárias para solucionar o problema, sendo elas Haste, Disco e Controle.

A FBE Haste conta com atributos que controlam o último disco que está em cada haste, para poder validar se o disco que está sendo movimentado para a haste é menor do que o que já está presente na haste.

Na FBE Disco contém o atributo que identifica em qual haste o disco está posicionado, seja haste A, B ou C.

Para a FBE Controle foram criados quatro atributos, sendo um atributo para controlar o último disco movimentado, um atributo para contar a quantidade de movimentos, um atributo para controlar se um movimento está sendo realizado, evitando paralelismo de movimentos, e um último atributo para armazenar o último movimento realizado, evitando repetição de movimentos.

Foram criadas 3 instâncias da FBE Haste, sendo uma instância para cada haste que compõe o problema. Para a FBE Disco foram criadas 3 instâncias também, sendo uma para cada disco, neste caso, o número de instâncias deve ser ampliado conforme o número de discos é ampliado. Caso seja necessário aumentar o número de discos, também é necessário criar novas regras para controlar os movimentos.

Já para a FBE Controle apenas uma instância foi criada, pois ela serve para controlar os movimentos de todos os discos entre todas as hastes.

Devido as dificuldades e sugestões que serão relatadas nos tópicos “Dificuldades Encontradas” e “Sugestões de Melhorias” a implementação em LingPON ficou restrita a apenas 3 discos, a ideia inicial era deixar as regras flexíveis assim como na implementação em PP.

Foram criadas 8 regras, sendo 7 para controlar os movimentos e uma para indicar a finalização do processo. As 7 regras que controlam o movimento baseiam-se nas regras do jogo, onde não é possível movimentar um disco maior para cima de um disco menor, além disso elas controlam os movimentos realizados, evitando movimentos desnecessários. Também para evitar o paralelismo de movimentos um atributo da FBE Controle foi utilizado para indicar o momento em que o movimento está concluído.

V. Comparação de Desempenho

Além de desenvolver um programa em LingPON para melhorar o entendimento do paradigma, a ideia também é desenvolver um algoritmo para solucionar o mesmo problema em um paradigma vigente, para poder realizar algumas comparações.

Para comparação deste trabalho os seguintes critérios foram selecionados: facilidade de desenvolvimento; legibilidade do código; número mínimo de movimentos; e desempenho.

O desenvolvimento do algoritmo de comparação para solucionar a Torre de Hanói foi desenvolvido em C, utilizando o Paradigma Procedimental – PP, de forma não recursiva.

Quanto à facilidade de desenvolvimento, é possível afirmar que a LingPON foi mais fácil, a partir do momento em que entende-se o funcionamento da mesma, a programação fica intuitiva.

A legibilidade do código gerado em LingPON é mais claro e de mais fácil leitura do que o gerado em C.

Quanto à quantidade de linhas geradas, em C foram geradas 180 linhas de código, já em LingPON foram geradas 286 linhas de código.

Ambos os algoritmos atingiram o objetivo de realizar a tarefa com o número mínimo de movimentos necessário para isso. Embora a flexibilidade e número de discos apenas foi possível utilizando a implementação no paradigma vigente.

Quanto ao desempenho, pode-se afirmar, sem sombra de dúvidas que o código gerado utilizando o PON é melhor do que o gerado em PP.

Como não foi possível flexibilizar o número de discos em PON, para realizar a comparação de desempenho entre as implementações, foi optado por executar várias vezes o mesmo algoritmo, dessa forma permitindo um elevado número de movimentos para avaliar o desempenho.

Para comparação o código em LingPON foi compilado em C, C++ 1.0 e C++ 1.1. Era desejável gerar o código também para C++ 2.0 porém não foi possível devido a erros no compilador que serão detalhados no tópico “Dificuldades Encontradas”.

A Figura 3 demonstra uma tabela de comparação entre os códigos compilados.

Repetições	C	PON – C 1.0	PON – C++ 1.0	PON – C++ 1.1
5	0,02		0,02	0,02
10	0,04		0,02	0,02
20	0,07		0,03	0,03
50	0,15		0,06	0,08
100	0,29		0,12	0,14
500	1,45		0,66	0,64
1000	2,84	1,00	1,28	1,37
10000	29,29	5,00	12,63	13,25
100000	289,44	56,00	127,69	129,10
1000000	2920,78	565,00	1250,59	1277,90

Figura 3. Tabela de Comparação de Desempenho

A tabela demonstra a execução dos programas compilados e C (PP), C (PON), C++ 1.0 (PON) e C++ 1.1 (PON) para o número de repetições que consta na coluna Repetições, o tempo relatado é sem segundos.

Analisando os dados é possível observar que o código gerado em C (PON) teve um desempenho melhor do que os demais programas.

Também é possível observar que existe uma pequena diferença de desempenho entre os programas compilados em C++ 1.0 e C++ 1.1, ambos PON.

O desempenho do programa desenvolvido em C, utilizando o PP, perdeu para todos os programas utilizando o PON, independente da compilação.

Esta comparação pode ser observada melhor nas Figuras 4, 5, 6 e 7.

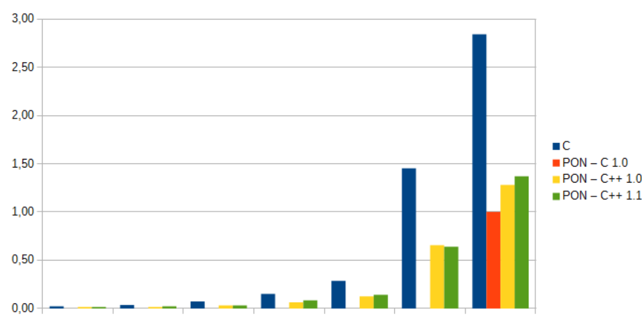

Figura 4. Gráfico em Coluna – Até 1000

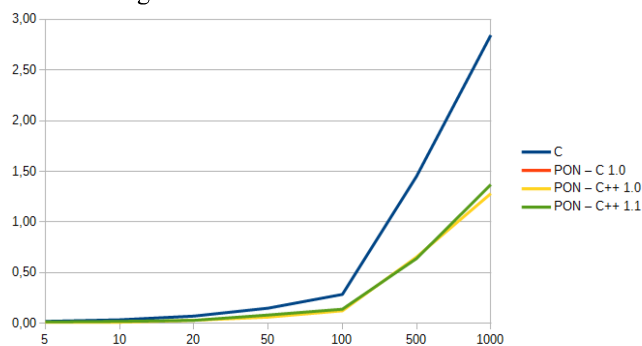

Figura 5. Gráfico em Linhas – Até 1000

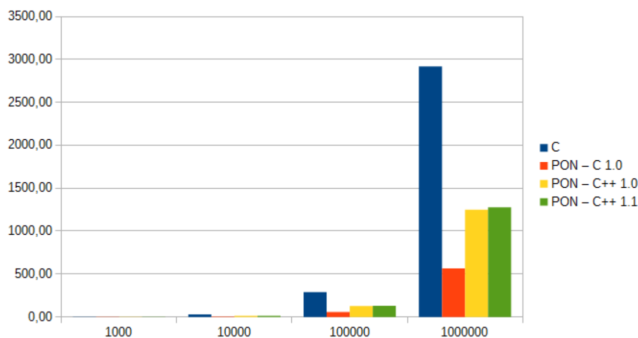

Figura 6. Gráfico em Coluna – A partir de 1000

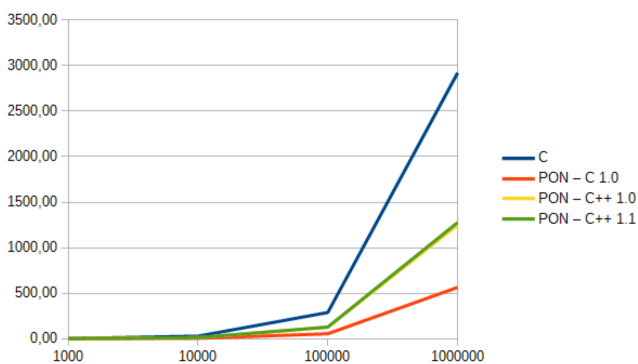

Figura 7. Gráfico em Linhas – A partir de 1000

VI. Dificuldades Encontradas

Como é de se esperar em qualquer tecnologia que está em desenvolvimento, algumas dificuldades de implementação do algoritmo proposto foram encontradas, estas serão relatadas neste tópico.

Uma das dificuldades encontradas foi a instabilidade do compilador, compreensível devido à constante evolução que o mesmo vem passando no decorrer do tempo. Devido a esta instabilidade não foi possível gerar o código alvo para C++ 2.0, pois o mesmo estava convertendo o tipo de dados de todos os atributos para *boolean*, independente do tipo definido em LingPON.

Algo semelhante ocorreu no código gerado para C, porém como a estrutura é diferente, este problema foi contornado manualmente no código alvo gerado.

Um ponto que gerou certo retrabalho para criar uma solução de contorno, é o fato de que quando mais de uma instância de uma FBE está sendo utilizada, a notificação de regras está sendo realizada indevidamente, não observando a instância que está sendo utilizada nas premissas. Para contornar este problema foi necessário criar um atributo isolado para cada instância dentro da FBE, multiplicando o número de atributos pelo número de instâncias.

O algoritmo escolhido é melhor solucionado quando é possível utilizar vetores, atualmente a LingPON não possibilita a utilização de vetores de forma direta, e a utilização de um vetor construído não foi possível devido ao problema comentado no parágrafo anterior.

Na medida da evolução do algoritmo em LingPON, ocorreu a tentativa de flexibilização do número de discos, esta flexibilização foi descartada devido a um problema encontrado, que é a impossibilidade de comparar atributos entre si nas premissas, sendo possível apenas comparar um atributo com um valor constante.

VII. Sugestões de Melhorias

Este trabalho além de auxiliar no entendimento do PON também foi útil para encontrar algumas dificuldades de implementação, e com base nisso sugerir melhorias, como é esperado deste tipo de experimento.

Com base nas dificuldades encontradas, entendo que a LingPON poderia ser aprimorada com as seguintes melhorias:

1. **Utilização de vetores**: Embora seja possível utilizar vetores construídos, entendo que um vetor clássico poderia auxiliar o desenvolvimento de vários algoritmos em LingPON, como é o do caso apresentado;
2. **Comparação entre atributos nas premissas**: Atualmente foi possível construir as premissas comparando um atributo a uma constante, entendo que é necessário tornar esta comparação dinâmica, sendo necessária a comparação de valores entre atributos de diferentes instâncias;
3. **Alterar valor de atributos de diferentes instâncias**: Atualmente um método apenas consegue alterar o valor de atributos da própria instância em LingPON, entendo que o fato de permitir a alteração de atributos de diferentes instâncias através dos métodos das FBEs tornará o a LingPON mais robusta e dinâmica.

VIII. Conclusões

A partir do estudo realizado sobre o PON e suas materializações através de exposição em sala de aula, e principalmente da prática realizada, foi possível conhecer um pouco mais deste novo paradigma.

Com base no que foi relatado neste artigo, é possível identificar uma grande facilidade de programação em LingPON, fácil entendimento do código gerado e também um grande ganho de desempenho em comparação com o paradigma imperativo.

Como é esperado de uma tecnologia em desenvolvimento, problemas ainda são encontrados, e através deles surgem sugestões de melhorias.

Entendo que este trabalho pôde contribuir para o desenvolvimento do paradigma, principalmente no que tange sua materialização em LingPON.

References

[1] Robson Ribeiro Linhares, Jean Marcelo Simão e Paulo Cézar Stadzisz. Arquitetura de Computador Orientada a Notificações – ARQPON. Pedido de Patente. INPI, 2014.

[2] Jean Marcelo Simão e Paulo Cézar Stadzisz. Paradigma Orientado a Notificações (PON) – Uma Técnica de Composição e Execução de Software Orientado a Notificações. Pedido de Patente. INPI, 2008.

[3] Robson Duarte Xavier. Paradigmas de Desenvolvimento de Software: Comparação entre Abordagens Orientada a Eventos e Orientada a Notificações. Dissertação de Mestrado, PPGCA/UTFPR. Curitiba, 2014. Disponível em: http://repositorio.utfpr.edu.br/jspui/bitstream/1/1006/1/CT_PPGCA_M_Xavier%2c%20Robson%20Duarte_2014.pdf.

[4] Roni Fabio Banaszewski. Paradigma Orientado a Notificações: Avanços e Comparações. Dissertação de Mestrado, CPGEI/UTFPR. Curitiba, 2009. Disponível em: http://files.dirppg.ct.utfpr.edu.br/cpgei/Ano_2009/dissertacoes/Dissertacao_500_2009.pdf.

[5] Leonardo Faix Pordeus. Notification Oriented Paradigm (NOP): CTA Simulator. Curitiba, 2015.

[6] Cleverson Avelino Ferreira. Linguagem e Compilador para o Paradigma Orientado a Notificações (PON): Avanços e Comparações. Dissertação de Mestrado, PPGCA/UTFPR. Curitiba, 2015. Disponível em: http://repositorio.utfpr.edu.br/jspui/bitstream/1/1414/1/CT_PPGCA_M_Ferreira%2c%20Cleverson%20Avelino_2015.pdf.

[7] Torres de Hanói. Disponível em: https://pt.khanacademy.org/computing/computer-science/algorithms/towers-of-hanoi/a/towers-of-hanoi Acessado em Maio de 2016.

Um estudo comparativo entre o paradigma orientado a notificações (PON) e o paradigma orientado a objetos (OO) aplicado em um problema de uma cidade virtual

Frederico Severo Miranda
CPGEI / UTFPR
Avenida Sete de Setembro, 3165
Curitiba-PR - CEP 80.230-910
E-mail: fdr.miranda@gmail.com

***Resumo*—Qual paradigma utilizar para processar informações de um Web Service especialista em condições climáticas e então reproduzir a iluminação do ambiente de uma cidade real em uma cidade virtual visando o menor tempo de processamento?**

No âmbito de responder a esta pergunta, uma solução é implementar algoritmos levando em conta o paradigma vigente (OO) e um novo paradigma conhecido como paradigma orientado a notificações (PON). Desta forma, é possível compará-los para descobrir qual paradigma é o mais eficiente.

Como resultado, seis algoritmos foram implementados bem como uma bateria de testes foram realizadas com o objetivo de avaliar o desempenho dos mesmos.

Com o desempenho dos algoritmos avaliados, conclui-se que o algoritmo implementado com o conceito PON apresentou um desempenho extremamente satisfatório sendo superado apenas por um algoritmo dentre os seis implementados. O PON surge como um novo conceito que obriga o desenvolvedor a mudar sua maneira de pensar em como implementar softwares e que se tiver esforços concentrados poderá constituir em um futuro próximo um novo patamar.

***Keywords*—Cidades Virtuais, Cidades Inteligentes, Cidades Digitais, TIC, Programação Orientada a Notificações (PON).**

I. Introdução

A Tecnologia da Informação e Comunicação (TIC) está presente na vida diária da sociedade. Ela tem sido utilizada no trabalho, nos relacionamentos, serviços públicos, entretenimento e lazer com o objetivo de superar a exclusão social, melhorar o desempenho econômico, criar oportunidades de emprego, melhorar a qualidade de vida e promover a participação social [1].

A TIC possibilitou a criação de várias aplicações inovadoras que tornam a vida mais fácil [2]. Mas atualmente não foram encontradas plataformas computacionais com o objetivo de centralizar e integrar aplicações voltadas à população envolvendo aspectos de caráter social, governamental, comercial, educacional, científico e entretenimento que são associadas ao ambiente físico de vivência (i.e., a cidade) das pessoas. Este problema é observado na condição atual na qual são oferecidas aplicações em diferentes formatos e propósitos para atender a diversidade das demandas da sociedade, como as cidades digitais, cidades inteligentes, software de entretenimento, mídias sociais, lojas eletrônicas e sites de busca, mas que praticamente não se relacionam entre si e não têm vínculo regional (não se associam com o ambiente de vivência das pessoas). Diante deste contexto, vislumbra-se uma cidade virtual que deve reproduzir alguns aspectos do mundo real, como por exemplo, as condições climáticas e iluminação ambiente.

Com a utilização de Web Services especialistas é possível obter informações sobre as condições climáticas e então reproduzir a iluminação ambiente da cidade real. O problema que existe neste cenário trata-se em como processar estas informações da maneira mais rápida possível.

Diante deste fato, uma possível solução é implementar algoritmos levando em conta o paradigma vigente (OO) e um novo paradigma conhecido como programação orientada a notificações (PON) e assim compará-los levando em consideração o tempo de processamento.

Como resultado, são apresentados seis algoritmos para resolver este problema de processamento das informações advindas do Web Service. Além dos seis algoritmos e devido a busca por otimização, foi aprimorado o framework *NOP C# 1.0* gerando assim o novo framework *NOP C# 1.1*.

O algoritmo implementado com o conceito do PON mostrou-se eficiente para resolver o problema sobre condições climáticas perdendo em tempo de processamento apenas para um algoritmo dentre os seis implementados.

II. Revisão da literatura

Esta seção irá abordar dois assuntos distintos para uma melhor compreensão da problemática envolvida. O primeiro irá abordar a temática referente a cidades virtuais, cidades digitais e cidades inteligentes e o segundo irá abordar conceitos sobre o paradigma orientado a notificações.

A. Cidades virtuais, cidades inteligentes e cidades digitais

Uma das criações mais importante e poderosa da humanidade foi a Internet. Pessoas, sistemas e “coisas” estão conectados entre si com a possibilidade de compartilhar informações.

Algumas estimativas sugerem que em 2020, 50 bilhões de recursos estarão conectados na internet [3]. O Fórum Econômico Mundial acredita que esta hiperconectividade irá trazer um alto impacto no futuro e considera que estamos vivendo em um “mundo hiperconectado” [4]. Esta conectividade possibilita a criação de novos produtos e serviços para melhorar a vida das pessoas [5]. A conectividade possibilitou o avanço em diversas áreas de pesquisa, como por exemplo, o desenvolvimento das cidades virtuais, cidades inteligentes e cidades digitais.

1) Cidades virtuais: Tecnologias tridimensionais (3D) têm sido recentemente introduzidas em diferentes aplicações e áreas tais como ambientes virtuais (mundo virtual/cidade virtual), realidade aumentada, computação desktop, sistemas para área de saúde, jogos 3D, tele-operação e computação ubíqua [6]. Cidades virtuais possuem várias definições, mas neste artigo será considerada como um ambiente tridimensional que reconstrói uma estrutura espacial complexa de uma cidade no computador e que pode ser utilizada para diversos fins: planejamento urbano, gerenciamento de redes wireless, turismo virtual, simulação de tráfego, simulação de ambiente [7]. As cidades virtuais para propósitos sérios estão se tornando cada vez mais difundidas, isso ocorre pelo fato de suportarem uma ampla gama de atividades que podem ser executadas. Estas atividades são oriundas de diversas áreas como educação, entretenimento e socialização [8]. Apesar das advertências sobre o aumento da dependência dos seres humanos com a tecnologia, pesquisadores ainda preveem que em 2018 as cidades virtuais serão consideradas como a principal plataforma de negócios e oportunidades [9]. A figura 1 mostra um exemplo de uma cidade virtual.

Figura 1. Exemplo de uma cidade virtual.

2) Cidades inteligentes: Foi publicado pelas Nações Unidas um relatório sobre a migração de pessoas das áreas rurais para as áreas urbanas. Este relatório mostra que em 1950 30% da população mundial viviam em áreas urbanas, em 2014 este número sobe para 54% e em 2050 existe a previsão de que 66% da população mundial estará vivendo nas cidades [10]. As cidades constituem o principal local da realização de atividade humana e econômica, que fornecem aos seus habitantes uma grande oportunidade de desenvolvimento. Entretanto, à medida que crescem em tamanho e complexidade, surge uma diversidade de problemas que podem ser difíceis de serem solucionados. O gerenciamento das áreas urbanas se faz necessário para suportar a competitividade econômica, simultaneamente deve reforçar a coesão social e ambiental e neste sentido aumentar a qualidade de vida da sua população [11]. Com o avanço das TICs, surge o conceito de “Cidades Inteligentes” como uma solução para resolver problemas das áreas urbanas [12]. Este conceito surgiu pela primeira vez em 1990 e muitos pesquisadores têm enfatizado a tecnologia, inovação e globalização no processo de urbanização [13]. As cidades inteligentes possuem uma definição abrangente que considera diversos fatores, mas sempre é associada com a interconexão entre fatores humano-sociais com a TIC [14]. Também pode ser definida como a integração entre a infraestrutura tecnológica e física para melhorar o desenvolvimento econômico, político, social e urbano [15]. Assim sendo, a tecnologia, comunicação e as rotinas diárias das pessoas coexistem em um mesmo ambiente [16]. A Figura 2 aborda as 6 áreas de atuação de uma cidade inteligente.

Figura 2. As principais áreas de atuação de uma cidade inteligente. Adaptado de [17].

Com a indústria de TIC, é possível incorporar sistemas e sensores wireless que coletam informações e colaboram entre si para ajudar os usuários em um objetivo específico [18]. Como exemplo, podemos considerar uma aplicação no tráfego urbano, onde os carros coletam dados através de GPS, câmeras e outros sensores e enviam estas informações a outros carros que estão em quadras próximas, informando assim um possível congestionamento [12].

3) Cidades digitais: Os constantes desafios sociais, políticos e financeiros enfrentados pelas cidades têm exigido dos seus gestores uma busca por soluções inovadoras para melhorar a qualidade de vidas da população [19]. Os avanços na indústria de TIC tem alterado o modo como gestores públicos e a população interagem entre si, possibilitando a criação de portais públicos [20]. Estes portais WEB fazem parte de um termo que envolve uma série de definições e

características conhecido como "Cidades Digitais". Cidade Digital é a união entre aplicações WEB que melhoram a qualidade de vida dos cidadãos e uma infraestrutura física que suporta estas aplicações [21].

A Figura 3 mostra a composição de uma cidade digital.

Figura 3. Composição de uma cidade digital.

Estas aplicações podem ser vistas como serviços WEB [22] que conectam departamentos públicos, empresas, escolas, comunidades, organizações [23] com o objetivo de oferecer serviços públicos de qualidade para a população (serviços fiscais, serviços de saúde, serviços de polícia), serviços públicos para empresas, serviços de informação (condições climáticas, informação geográfica, tráfego), serviços de entretenimento (jogos, turismo) e serviços relacionados a recursos humanos [24]. A cidade virtual também deve possuir uma infraestrutura física para permitir que a inclusão digital e interações sejam possíveis [25], como por exemplo, os telecentros que disponibilizam Internet para uso gratuito e a construção de redes de fibra ótica para interligar órgãos públicos [26].

B. *Paradigma orientado a notificações - PON*

Em linhas gerais, o Paradigma Orientado a Notificações (PON) se propõe a resolver certos problemas existentes nos paradigmas usuais de programação, nomeadamente o Paradigma Declarativo (PD) e o Paradigma Imperativo (PI). Na verdade, o PON unifica as principais características e as vantagens do PD (e.g. representação do conhecimento em regras) e do PI (e.g. flexibilidade de expressão e nível apropriado de abstração). Ademais, o PON supostamente resolve, já em termos de modelo, várias das deficiências e inconvenientes de PI e PD em aplicações de software e mesmo de hardware, possivelmente desde ambientes monoprocessados a completamente multiprocessados - copiado de [27].

De fato, o PON permite desacoplar expressões causais do código-fonte, ao considerar cada uma destas e seus fatos relacionados como entidades computacionais, as quais são objetos nas atuais implementações em software e módulo de circuito nas atuais implementações em hardware. Estas entidades são notificantes, permitindo assim uma escalabilidade de desempenho em ambientes de processamento paralelo ou não, bem como melhor aproveitamento de recursos em ambiente distribuído, tanto em implementações de software quanto de hardware - copiado de [27].

Particularmente no tocante a software, isto é diferente dos programas usuais do PI (salientando os Orientados a Objetos - OO) e do PD (salientado os chamados Sistemas Baseados em Regras - SBR). Nestes, as expressões causais são passivas e acopladas (senão fortemente acopladas) a outras partes do código, além de haver algum ou mesmo muito desperdício de processamento, conforme o caso - copiado de [27].

Acredita-se que o PON ainda pode evoluir, a luz do seu cálculo assintótico, no tocante a questões de desempenho. Ainda, vislumbra-se também a evolução do PON no tocante à facilidade de programação. Com a linguagem PON (denominada doravante "LingPon") e seu compilador (denominado doravante "compilador PON"), torna-se possível desenvolver aplicações específicas em uma linguagem conformada ao PON e gerar resultados, em termos de código, sem a adição de estruturas de dados caras - copiado de [27].

III. MEDOTOLOGIA

Para resolver o problema da iluminação na cidade virtual, foram adotados três atributos que são obtidos através do Web Service de condições climáticas:

- Weather: CLEAN, RAIN, MOSTLY CLOUD etc. (São 16 possíveis valores).
- Hour: valores inteiros entre 0 e 23.
- Visibility: valores inteiros entre 0 e 10.

Em seguida foram mapeadas todas as combinações possíveis conforme figura 4. Assim, são criados 4224 (16*24*11) cenários/regras e cada um associado com um resultado específico.

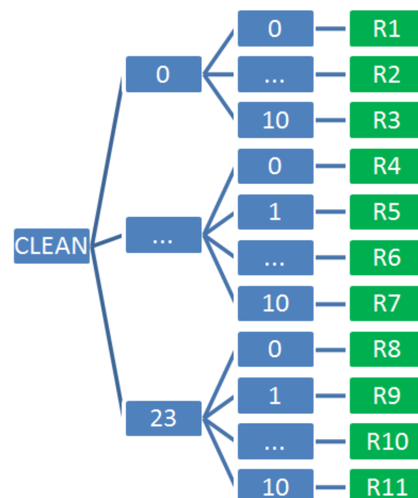

Figura 4. Combinações entre os atributos Weather (1ª coluna), Hour (2ª coluna) e Visibility (3ª coluna). Neste exemplo está sendo mostrado apenas o Weather CLEAN, mas devem ser consideradas todas as 16 possibilidades. A 4ª coluna (cor verde) representa o resultado específico para cada regra.

Para o estudo comparativo proposto foram implementados seis algoritmos nomeados da seguinte forma *C#*, *C#Otim*, *C#OtimV2*, *C#Pon*, *C#Pon1.1* e *C#Pon1.1Hash*.

A. C#

O algoritmo representado pela figura 5 foi implementado sem utilizar as melhores práticas em programação, pois o objetivo principal era codificar da forma mais rápida possível. Neste contexto temos um algoritmo que sempre avalia todas as regras para qualquer entrada avaliada.

```
if (item.getWeather() == "SKY_CLEAR" && item.getHour() == 0 && item.getVisibility() == 0)
{
    value = 0.1f;
}
if (item.getWeather() == "SKY_CLEAR" && item.getHour() == 0 && item.getVisibility() == 1)
{
    value = 0.1f;
}
if (item.getWeather() == "SKY_CLEAR" && item.getHour() == 0 && item.getVisibility() == 2)
{
    value = 0.1f;
}
if (item.getWeather() == "SKY_CLEAR" && item.getHour() == 0 && item.getVisibility() == 3)
{
    value = 0.1f;
}
```

Figura 5. Este algoritmo SEMPRE irá avaliar todas as regras. OBS: apenas algumas linhas de códigos são exibidas, mas devem ser consideradas as 4224 regras

B. C#Otim

O algoritmo descrito na figura 6 foi implementado otimizando o algoritmo C#. Desta forma, um melhor tratamento de *string´s* foi realizado e quando uma determinada regra é encontrada, o algoritmo interrompe sua execução.

```
if (weather.Equals(SKY_CLEAR))
{
    if (hour == 0)
    {
        if (visibility == 0)
        {
            value = 0.1f;
        }
        else if (visibility == 1)
        {
            value = 0.1f;
        }
        else if (visibility == 2)
        {
            value = 0.1f;
        }
```

Figura 6. Este algoritmo finaliza sua execução quando uma regra é encontrada. OBS: apenas algumas linhas de códigos são exibidas, mas devem ser consideradas as 4224 regras.

C. C#OtimV2

O algoritmo descrito na figura 7 foi implementado utilizando o conceito de SWITCH CASE. Desta forma, independente do cenário a ser avaliado, o mesmo será encontrado pontualmente.

D. C#PON

O algoritmo visualizado na figura 8 foi implementado utilizando o paradigma orientado a notificações (PON) com o uso do framework NOP C# 1.0.

```
switch (weather)
{
    case SKY_CLEAR:
        {
            switch (hour)
            {
                case 0:
                    {
                        switch (visibility)
                        {
                            case 0: { value = 0.5f; break; }
                            case 1: { value = 0.5f; break; }
                            case 2: { value = 0.5f; break; }
                            case 3: { value = 0.5f; break; }
                            case 4: { value = 0.5f; break; }
                            case 5: { value = 0.5f; break; }
                            case 6: { value = 0.5f; break; }
                            case 7: { value = 0.5f; break; }
                            case 8: { value = 0.5f; break; }
                            case 9: { value = 0.5f; break; }
                            case 10: { value = 0.5f; break; }
                        }
                        break;
                    }
```

Figura 7. Este algoritmo encontra pontualmente o cenário avaliado, maximizando assim seu tempo de processamento. OBS: apenas algumas linhas de códigos são exibidas, mas devem ser consideradas as 4224 regras.

```
Condition cond1 = new Condition(Condition.CONJUNCTION);
cond1.addPremise(new Premise(wetaherController.atWeather, SKY_CLEAR, Premise.EQUAL, false));
cond1.addPremise(new Premise(wetaherController.atHour, 0, Premise.EQUAL, false));
cond1.addPremise(new Premise(wetaherController.atVisibility, 0, Premise.EQUAL, false));
Action action1 = new Action();
new Rule("", scheduler, cond1, action1, false);
action1.addInstigation(new Instigation(wetaherController.atValue, 0.1));
```

Figura 8. Algoritmo implementado com o conceito PON. OBS: Apenas algumas linhas de códigos são exibidas, mas devem ser consideradas as 4224 regras.

E. C#PON1.1

O algoritmo representado pela figura 9 foi implementado utilizando o paradigma orientado a notificações (PON) com algumas alterações em sua modelagem. Neste contexto, todas as premissas foram criadas uma única vez evitando assim redundância estrutural.

```
Premise prClear = new Premise(wetaherController.atWeather, SKY_CLEAR, Premise.EQUAL, false);
Premise prScatteredClouds = new Premise(wetaherController.atWeather, SKY_SCATTERED_CLOUDS, Premise.EQUAL, false);

Premise prHour0 = new Premise(wetaherController.atHour, 0, Premise.EQUAL, false);
Premise prHour1 = new Premise(wetaherController.atHour, 1, Premise.EQUAL, false);

Premise prVisib0 = new Premise(wetaherController.atVisibility, 0, Premise.EQUAL, false);
Premise prVisib1 = new Premise(wetaherController.atVisibility, 1, Premise.EQUAL, false);

Condition cond1 = new Condition(Condition.CONJUNCTION);
cond1.addPremise(prClear);
cond1.addPremise(prHour0);
cond1.addPremise(prVisib0);
Action action1 = new Action();
new Rule("", scheduler, cond1, action1, false);
action1.addInstigation(new Instigation(wetaherController.atValue, 0.5));
```

Figura 9. Devem ser consideradas a criação de 16 premissas Weather, 24 premissas do tipo Hour e por fim 11 premissas do tipo Visibility. OBS: apenas algumas linhas de códigos são exibidas, mas devem ser consideradas as 4224 regras.

Para este algoritmo, algumas alterações também foram realizadas no framework NOP C# 1.0: melhor comparação de *string's*, alterações de comandos *foreach* e alteração de comandos *if...else*. Todas estas alterações pontuais resultaram em uma melhor performance do algoritmo levando em conta seu tempo de execução.

- Comparação de *string´s*: alterado conforme figura 10.

- *Foreach*: alterado alguns comandos *foreach* para o tradicional comando *for*.
- *if...else*: alterado conforme figura 11.

```
if (string.Compare(myString, value) != 0) //se for diferente
  {
      myString = value;
      notifyPremises();
  }

Alterado para:
if (!value.Equals(myString))//se for diferente
  {
      myString = value;
      notifyPremises();
  }
```

Figura 10. Melhor desempenho na forma de comparar *string´s*.

```
if (amountElements == amountElementsTrue)
 {
     return true;
 }
     return false;

Alterado para:
return amountElements == amountElementsTrue;
```

Figura 11. Melhor desempenho para um cenário específico do comando *if*.

F. C#PON1.1Hash

Este algoritmo utiliza o mesmo código do algoritmo C#PON1.1. A diferença é que sua estrutura de dados é baseada em HASH. Desta forma as alterações nos valores dos atributos notificam exatamente as premissas que interessam.

Todos os seis algoritmos foram submetidos a uma bateria de testes e executados (modo segurança) no sistema operacional Windows 7 Home Premium 64-bit (AMD 4300 Quad-Core 3.8Ghz, 8 GB RAN). A bateria de testes é composta pelos seguintes cenários: Test1, Test100, Test200, Test300, Test500, Test600, Test700, Test800, Test900, Test1k, Test5k, Test6k, Test7k, Test8k, Test10K.

Como exemplo, a leitura de cada teste se faz da seguinte maneira (o mesmo raciocínio deve ser aplicado a todos os outros testes):

- Test1: possui 5 arquivos com registros aleatórios contendo 1 registro.
- Test100: possui 5 arquivos com registros aleatórios contendo 100 registros.
- Test5k: possui 5 arquivos com registros aleatórios contendo 5000 registros.
- TestX: possui 5 arquivos com registros aleatórios contendo X registros.

Um exemplo de arquivo utilizado é mostrado na figura 12.

Cada um dos seis algoritmos (um por vez) foi executado levando em conta como entrada os arquivos da bateria de teste e então o tempo de processamento foi anotado. Como exemplo, é descrito a seguir o teste realizado com a bateria **Test100**. Conforme dito anteriormente, todas as baterias possuem cinco arquivos com registros aleatórios, neste sentido, os algoritmos C#, C#Otim, C#OtimV2, C#Pon, C#Pon1.1 e C#Pon1.1Hash processam o primeiro arquivo que pertence ao

```
1  FOG 2 2
2  SCATTERED_CLOUDS 23 0
3  SCATTERED_CLOUDS 13 4
4  HEAVY_RAIN 17 8
5  MOSTLY_CLOUDY 5 9
6  HEAVY_RAIN 8 3
7  OVERCAST 23 10
8  RAIN 15 2
9  MOSTLY_CLOUDY 6 7
10 HEAVY_RAIN 19 5
11 HEAVY_RAIN 23 2
12 HEAVY_THUNDERSTORMS_AND_RAIN 7 9
13 MOSTLY_CLOUDY 2 10
14 HEAVY_RAIN 7 10
```

Figura 12. Exemplo de arquivo utilizado.

Test100 e o tempo de execução é anotado. Em seguida, os mesmos processam o segundo arquivo e o tempo de execução é anotado. Este processo continua até o quinto arquivo. O mesmo raciocínio é aplicado a todas as baterias de testes.

IV. Resultados

Este artigo apresenta como resultado um estudo comparativo entre seis algoritmos, sendo que três (C#Pon, C#Pon1.1 e C#Pon1.1Hash) utilizam o paradigma orientado a notificações (PON) e os outros três (C#, C#Otim e C#OtimV2) utilizam o paradigma orientado a objetos (OO).

A média do tempo de execução de cada algoritmo são evidenciados conforme as tabelas I e II.

	Test1	Test100	Test200	Test300	Test500	Test600	Test700
C#	4,546	4,8384	4,986	5,462	6,1106	6,1792	6,3172
C#Otim	0,6076	0,7092	0,8424	0,9708	1,0032	1,0488	1,0504
C#Pon	0,003	0,1946	0,4096	0,6322	0,8792	0,9272	1,0536
C#Pon1.1	0,001	0,109	0,234	0,374	0,405	0,421	0,452
C#Pon1.1H	0,001	0,090	0,215	0,303	0,351	0,381	0,4
C#OtimV2	0,001	0,171	0,201	0,343	0,358	0,372	0,374

Tabela I
MÉDIA DO TEMPO DE EXECUÇÃO (EM SEGUNDOS) DOS ALGORITMOS.

	Test800	Test900	Test1k	Test5k	Test6k	Test7k	Teste8k	Test10k
C#	6,5018	6,5472	6,6952	7,2144	7,62	7,7956	8,1644	9,9768
C#Otim	1,0618	1,0754	1,0894	1,5882	1,7926	1,959	2,0506	2,5668
C#Pon	1,086	1,199	1,306	5,7492	6,6762	8,7298	9,8746	11,8732
C#Pon1.1	0,468	0,483	0,499	1,107	1,401	1,695	1,802	2,207
C#Pon1.1H	0,421	0,458	0,470	0,987	1,124	1,4	1,421	1,723
C#OtimV2	0,399	0,405	0,442	0,967	1,151	1,242	1,351	1,7

Tabela II
MÉDIA DO TEMPO DE EXECUÇÃO (EM SEGUNDOS) DOS ALGORITMOS.

A representação do comportamento dos valores das tabelas I e II podem ser visualizados na figura 13. Para uma melhor visualização deste comportamento apenas os melhores resultados serão visualizados na figura 14.

Além do estudo comparativo, otimizações pontuais foram realizadas no framework NOP C# 1.0. Estas otimizações se referem a uma melhor comparação entre *string´s*, substituição de alguns comandos *foreach* por *for* e um aprimoramento em alguns blocos de comandos *if...else*. O framework NOP C# 1.0 foi evoluído para o framework NOP C# 1.1 de tal forma que agora é possível utilizar a estrutura de dados Hash para notificar as premissas do PON.

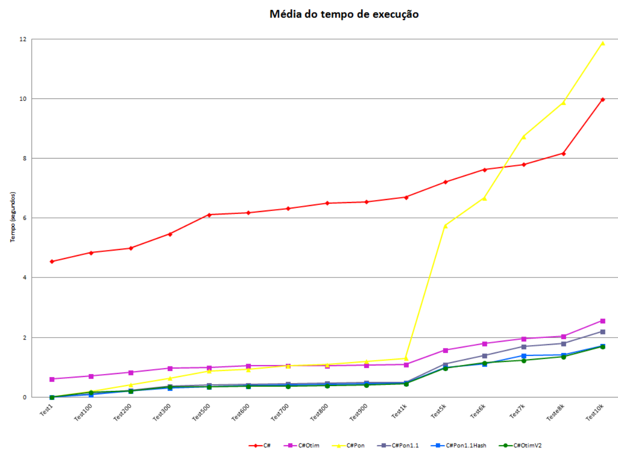

Figura 13. Comportamento do tempo de execução dos algoritmos.

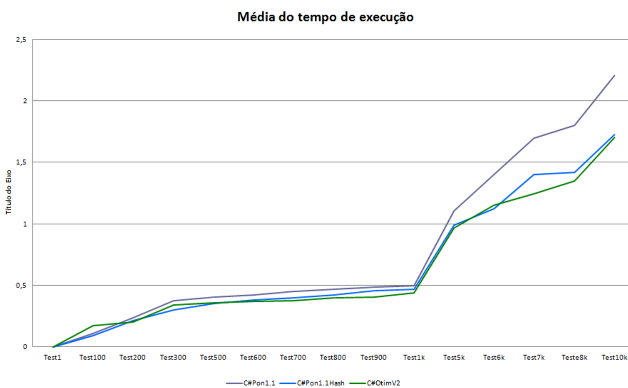

Figura 14. Comportamento do tempo de execução dos algoritmos.

V. Conclusão

Uma das soluções possíveis para reproduzir a iluminação ambiente de uma cidade real em uma cidade virtual é a utilização de Web Services especialistas. Um problema que ocorre seria o tempo de processamento destas informações.

Diante deste fato, é proposto a implementação de 6 algoritmos para que os mesmos pudessem ser comparados levando em conta o tempo de execução.

Seis algoritmos são apresentados como resultado para resolver o problema de processamento das informações sobre condições climáticas.

Comparando o desempenho dos seis algoritmos conclui-se que o paradigma C#Pon1.1 apresenta um desempenho satisfatório sendo superado apenas por um algoritmo dentre os seis implementados. A utilização do PON em relação a codificação é simples, bastando apenas importar uma Dynamic-link library (DLL) e então utilizar seus recursos disponíveis. A ideia por trás da programação orientada a notificações (PON) obriga o desenvolvedor a mudar sua maneira de pensar em como codificar e que se tiver esforços concentrados neste paradigma pode-se tornar uma nova forma de desenvolver softwares.

Uma das desvantagens na utilização do PON é quantidade de linhas de código que são geradas em relação a outros paradigmas, mas esta desvantagem não sobrepõe os benefícios disponibilizados pelo PON.

VI. Trabalhos futuros

Implementar um wizard para que seja possível diminuir o esforço na geração das regras do PON e buscar novas soluções para processar as informações sobre condições climáticas e então comparar seu desempenho com os algoritmos implementados.

Referências

[1] A. Grguric, S. Desic, M. Mosmondor, I. Benc, J. Krizanic, and P. Lazarevski, "Proof-of-concept applications for validation of ICT services for elderly care," *MIPRO, 2010 Proceedings of the 33rd International Convention*, pp. 355–359, 2010.

[2] K. S. Nwizege, F. Chukwunonso, C. Kpabeb, and S. Mmeah, "The impact of ICT on computer applications," *Proceedings - UKSim 5th European Modelling Symposium on Computer Modelling and Simulation, EMS 2011*, pp. 435–439, 2011.

[3] D. Evans, "The Internet of Things - How the Next Evolution of the Internet is Changing Everything," *CISCO white paper*, no. April, pp. 1–11, 2011.

[4] W. E. Forum, G. A. Council, and C. Systems, "Perspectives on a Hyperconnected World," no. January, 2013.

[5] I. Thomas, L. Fedon, A. Jara, and Y. Bocchi, "Towards a Human Centric Intelligent Society: Using Cloud and the Web of Everything to Facilitate New Social Infrastructures," *9th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pp. 319–324, 2015.

[6] E. Lombardo, *Virtual, Augmented and Mixed Reality: Designing and Developing Augmented and Virtual Environments*, S. E. Shumaker, Randall, Lackey, Ed. Springer, 2014.

[7] J. Döllner, H. Buchholz, M. Nienhaus, and F. Kirsch, "Illustrative Visualization of 3D City Models," *Visualization and Data Analysis*, vol. 5669, no. c, pp. 42–51, 2005.

[8] M. Fominykh, E. Prasolova-Forland, M. Morozov, A. Gerasimov, F. Bellotti, R. Berta, S. Cardona, and A. De Gloria, "Universcity: Towards a holistic approach to educational virtual city design," *2010 16th International Conference on Virtual Systems and Multimedia, VSMM 2010*, pp. 371–374, 2010.

[9] A. Hardin, J. Nicholson, A. Krishen, and D. Nicholson, "Virtual world entrepreneurship," *Proceedings of the Annual Hawaii International Conference on System Sciences*, pp. 4315–4322, 2013.

[10] U. Nations, *World Urbanization Prospects: The 2014 Revision, Highlights (ST/ESA/SER.A/352)*, 2014.

[11] A. Monzon, "Smart Cities and Green ICT Systems (SMARTGREENS), 2015 International Conference on," *Smart Cities and Green ICT Systems (SMARTGREENS), 2015 International Conference on*, pp. 1–11, 2015.

[12] Y. Chuantao, X. Zhang, C. Hui, W. Jingyuan, C. Daven, and D. Bertrand, "A literature survey on smart cities," *Sci China Inf Sci*, vol. 58, no. 5818, pp. 1–18, 2015.

[13] R. W. SMILOR, *THE TECHNOPOLIS PHENOMENON*, R. &. LITTLEFIELD, Ed. Rowman & Littlefield Publishers, 1992.

[14] A. Caragliu, C. D Bo, K. Kourtit, and P. Nijkamp, *Smart Cities*, second edi ed. Elsevier, 2015, vol. 22.

[15] K. Layne and J. Lee, "Developing fully functional E-government: A four stage model," *Government Information Quarterly*, vol. 18, no. 2, pp. 122–136, 2001.

[16] G. Piro, I. Cianci, L. A. Grieco, G. Boggia, and P. Camarda, "Information centric services in Smart Cities," *Journal of Systems and Software*, vol. 88, no. 1, pp. 169–188, 2014.

[17] R. Giffinger, C. Fertner, H. Kramar, R. Kalasek, N. Pichler, and E. Meijers, "Smart cities: Ranking of European medium-sized cities," Tech. Rep. October, 2007.

[18] B. Mattoni, F. Gugliermetti, and F. Bisegna, "A multilevel method to assess and design the renovation and integration of Smart Cities," *Sustainable Cities and Society*, vol. 15, pp. 105–119, 2015.

[19] D. A. Rezende, G. D. S. Madeira, L. D. S. Mendes, G. D. Breda, B. B. Zarpelão, and F. D. C. Figueiredo, "Information and telecommunications project for a digital City: A Brazilian case study," *Telematics and Informatics*, vol. 31, no. 1, pp. 98–114, 2014.
[20] B. W. Wirtz and O. T. Kurtz, "Local e-government and user satisfaction with city portals - the citizens service preference perspective," *International Review on Public and Nonprofit Marketing*, 2016.
[21] G. S. Yovanof and G. N. Hazapis, "An architectural framework and enabling wireless technologies for digital cities & Intelligent urban environments," *Wireless Personal Communications*, vol. 49, no. 3, pp. 445–463, 2009.
[22] L. S. Mendes, M. L. Bottoli, and G. D. Breda, "Digital cities and open MANs: a new communications paradigm," *2009 IEEE Latin-American Conference on Communications, LATINCOM '09 - Conference Proceedings*, pp. 1–8, 2009.
[23] E. P. GUERREIRO, *Cidade digital - Infoinclusão social e tecnologia em rede*, 1st ed., Senac São Paulo, Ed. Senac São Paulo, 2006.
[24] D. Zhu, Y. Li, J. Shi, Y. Xu, and W. Shen, "A service-oriented city portal framework and collaborative development platform," *Information Sciences*, vol. 179, no. 15, pp. 2606–2617, 2009.
[25] F. Duarte, F. de Carvalho Figueiredo, L. Leite, and D. Alcides Rezende, "A Conceptual Framework for Assessing Digital Cities and the Brazilian Index of Digital Cities: Analysis of Curitiba, the First-Ranked City," *Journal of Urban Technology*, vol. 21, no. 3, pp. 37–48, 2014.
[26] Ministério das Comunicações, "Cidades Digitais," 2014. [Online]. Available: http://www.mc.gov.br/cidades-digitais
[27] C. A. Ferreira, "LINGUAGEM E COMPILADOR PARA O PARADIGMA ORIENTADO A NOTIFICAÇÕES (PON): AVANÇOS E COMPARAÇÕES ," p. 245, 2015.

ANEXO D – RELATÓRIO DA DISCIPLINA LINGUAGENS/COMPILADORES - 2015

Este anexo apresenta o relatório apresentado pelos alunos Leonardo Pordeus, Fernando Schutz, Leonardo Santos e Ricardo Kerschbaumer como trabalho final da disciplina “Linguagens e Compiladores” ofertada pela UTFPR em 2015 e ministrada por Prof. Dr. João Alberto Fabro e Prof. Dr. Jean Marcelo Simão.

Introdução

O Paradigma Orientado a Notificações (PON) é uma nova abordagem para o desenvolvimento de sistemas computacionais de maneira mais eficiente quando comparado a sistemas baseados em paradigmas tradicionais, como a Programação Procedimental e a Programação Orientada a Objetos (POO) do Paradigma Imperativo (PI), assim como os Sistemas baseados em Regras (SBR) do paradigma declarativo (PD) (SIMÃO; STADZISZ, 2008) (BANASZEWSKI, 2009).

O PON propõe uma solução para os problemas destes paradigmas, que apresentam deficiências com relação a redundâncias estruturais, temporais e forte acoplamento entre suas entidades, diminuindo o desempenho e gerando maior dificuldade de paralelização e distribuição. Tais vantagens são constituídas por uma maior facilidade na concepção de sistemas que apresentem paralelismo ou distribuição, além da redução ou eliminação de alguns dos problemas clássicos de software PI e PD, tais como redundâncias de execução e acoplamento excessivo entre entidades computacionais (SIMÃO; STADZISZ, 2008) (BANASZEWSKI, 2009).

Para o desenvolvimento de softwares fazendo uso do PON, primeiramente foram realizadas pesquisas com framework C++ (BANASZEWSKI, 2009) e uma segunda versão otimizada do framework C++ (RONSZCKA, 2012), permitindo a criação de softwares PON sob abordagem de POO. Porém nessas abordagens há um elevado uso de estruturas de dados, como *lists, vectors, hashs*, entre outras, que degradam o desempenho das aplicações construídas sobre o conceito do PON. Assim, observou-se a necessidade de desenvolvimento de um compilador e de uma linguagem específica para o PON, denominada LingPon.

Outras pesquisas também exploraram a implementação do PON em *hardware* com uso de lógica reconfigurável (SIMÃO *et al*, 2012) seguindo os conceitos do PON. Peters (2012) propôs a implementação em lógica

reconfigurável de um co-processador PON (CoPON), uma solução híbrida, na qual a parte da aplicação responsável pelo processamento factual é executada em um núcleo von Neumann e a parte da aplicação responsável pelo cálculo lógico-causal e propagação de notificações é executada por meio de um co-processador baseado nos princípios do PON. Outrossim, uma arquitetura de processador foi desenvolvida de acordo com o modelo do PON, sendo denominada Notification-Oriented Computer Architecture (NOCA) (LINHARES,2015).

O trabalho da disciplina de linguagens e compiladores teve como objetivo dar continuidade a linguagem LingPon. A primeira versão do LingPon foi materializada na disciplina Linguagens e Compiladores no ano de 2014 e aprimorada por (FERREIRA, 2015) em sua dissertação de mestrado. Em sua primeira versão foi concebida uma gramática própria, definida por uma *BNF* (*Backus Normal Form*) e com as ferramentas flex e bison. A Figura apresenta um diagrama de blocos da estrutura da primeira versão do compilador. Na qual, é composto por um código escrito na linguagem LingPon, que representa a aplicação, o compilador e três possíveis códigos intermediários (C, C++ e Framework).

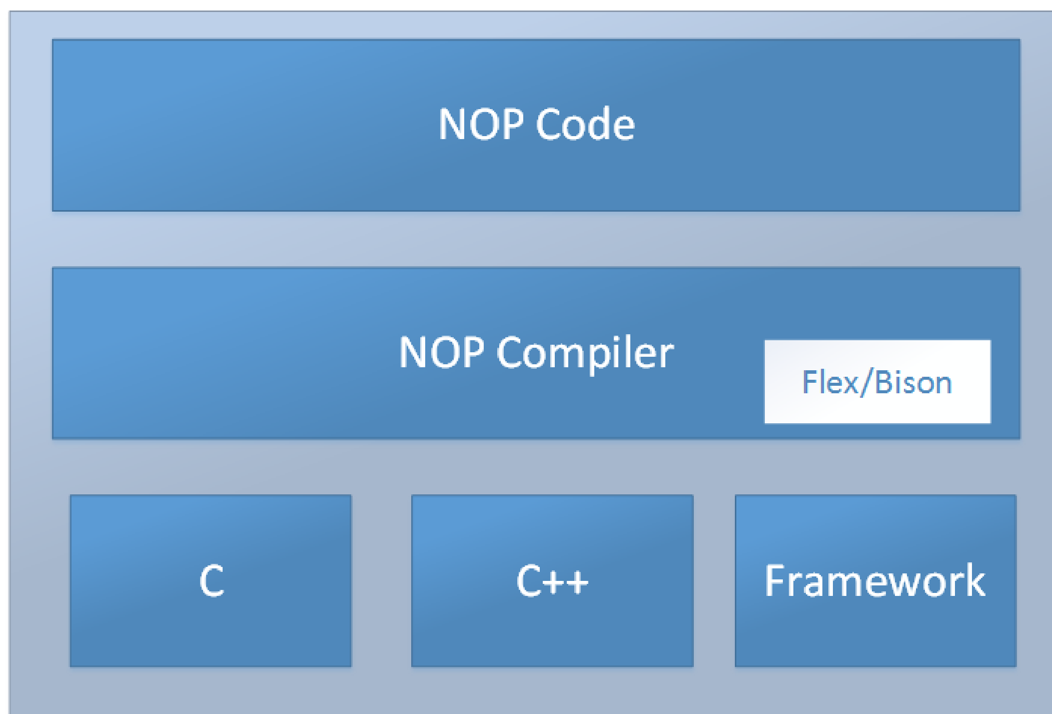

Figura – Diagrama de Blocos LingPon

Para a disciplina Linguagens e Compiladores de 2015 foram acrescentadas novas funcionalidades à linguagem, como regras de formação, compilação para *Notification Oriented Computer Architecture* (NOCA) (LINHARES, 2015), VHDL e uma nova versão C++, na qual é gerado um código seguindo os conceitos do PON de forma estática.

Linguagem Atual

a) Pré Requisitos e Instalação

Para o desenvolvimento do compilador é necessário instalar previamente os analisadores léxicos e semânticos, que são as ferramentas Flex/Lex e Bison. Para a sua compilação também é necessário configurar o compilador g++. No *Windows* o compilador pode ser configurado através da ferramenta *cygwin* (https://cygwin.com/install.html) (Tutorial de instalação do *cygwin*:
http://cs.calvin.edu/curriculum/cs/112/resources/installingEclipse/cygwin/
Marcar a opção de instalação do Flex e Bison).

Não existe no momento um instalador do compilador. Para sua utilização é necessário primeiramente, realizar o *check-out* do projeto no servidor *svn* disponível no seguinte endereço (https://200.134.17.9/svn/NOP/NOP_Compilador/NOP_v2015/). Após realizar o check-out do projeto é necessário seguir o processo abaixo (no *Windows*, através do *cygwin*) para realizar a sua compilação.

```
flex lex_pon.l
bison -d bison_pon.y
g++ lex.yy.c bison_pon.tab.c *.cpp -o pon
```

b) Linguagem PON

A linguagem PON pode ser dividida em cinco partes, sendo:

- Declaração de *FBE´s*;
- Instâncias das *FBE´s*;
- Estratégias de escalonamento;
- Definição das *Rules*;
- Definição da função *Main* do programa.

O Algoritmo a seguir apresenta o código referente a estrutura básica de um programa escrito na linguagem PON.

Algoritmo 1: Estrutura da linguagem PON

```
1   fbe Apple
2      . . .
3   end_fbe
4
5   fbe Archer
6      . . .
7   end_fbe
8
9   ----------------
10
11  inst
12     Apple apple1, apple2
13     Archer archer1, archer2
14  end_inst
15
16  ----------------
17
18  strategy
19     . . .
20  end_strategy
21
22  ----------------
23
24  rule RlTurnOn1
25     . . .
26  end_rule
27
28  rule RlTurnOn2
29     . . .
30  end_rule
31
32  ----------------
33
34  main {
35     . . .
36  }
```

O Algoritmo a seguir apresenta o exemplo de código para criação de uma FBE.

Algoritmo 2: Exemplo de criação de *FBEs*

```
1   fbe Archer
2       Attributes
3           boolean atHasFired false
4       end_Attributes
5
6       Methods
7           method mtFire(atHasFired = true)
8           method mtInnerCode() begin_method cout << "" << endl; end_method
9       end_Methods
    end_fbe
```

O Algoritmo a seguir demonstra como declarar instâncias das FBE's.

Algoritmo 3: Exemplo de instanciações de *FBEs*

```
1    inst
2       Apple apple1, apple2
3       Archer archer1, archer2
4    end_inst
```

O Algoritmo a seguir apresenta como declarar o tipo de estratégia de escalonamento. Sendo as possibilidades: no_one, breath e depth.

Algoritmo 4: Exemplo de definição de estratégia de escalonamento

```
1    strategy
2    no_one
3    end_strategy
```

O Algoritmo a seguir apresenta como declarar uma *Rule.*

Algoritmo 5: Exemplo de criação de *Rules*

```
1     rule rlTurnOn
2
3
4        properties
5            keeper true
6        end_properties
7
8        condition
9           subcondition A
10             premise imp prIsCrossed apple1.atIsCrossed == false
11                and
12             premise imp prHasFired archer1.atHasFired == false
13                and
14             premise prReadyToFire controller1.atHasFired == true
15          end_subcondition
16       end_condition
17
18
19       action
20          instigation inArcherFire1 archer1.mtFire
21          instigation inAppleCrossed1 apple1.mtExplode
22       end_action

      end_rule
```

O Algoritmo a seguir apresenta como declar*a*r a função *Main.*

Algoritmo 6: Exemplo do código Main

```
1    main {
2            apple->setisCrossed(false);
3            archer->sethasFired(false);
4            controller->setfire(true);
5    }
```

c) Tipos de atributos do PON

Os tipos de atributos permitidos na linguagem PON são:

- boolean
- integer
- string
- char
- pfloat

Além desses tipos atributos, é possível realizar a composição de uma FBE. O algoritmo a seguir apresenta como realizar a composição em uma FBE.

d) Compilando

Para compilar e executar um programa utilizando o compilador PON é necessário executar os seguintes passos:

1. Crie o programa PON conforme exemplificado anteriormente e salve em um arquivo de código fonte preferencialmente com a extensão .pon;
2. Execute o processo de criação do compilador PON (pon em Linux e pon.exe em Windows). Veja etapa de geração do compilador.
3. Execute o comando abaixo:

   ./pon 1 < nome_do_programa.pon (Linux)

   ./pon.exe 1 < nome_do_programa.pon (Windows)

O comando descrito no passo três é composto pela chamada de execução do compilador PON criado com o nome pon. São necessários dois argumentos para executar o processo de compilação. O primeiro argumento é a opção de geração de código intermediário que varia de 1 a 7.

1. C;
2. C++;
3. Framework;
4. Pré-Compilador;
5. NOCA;
6. Static C++;
7. VHDL;

Já o segundo argumento atribui o código fonte inserido no arquivo nome_do_programa.pon e irá criar o código intermediário PON conforme opção selecionada.

Obs: vale ressaltar que o código intermediário gerado será criado na pasta compiladosc para o código intermediário em C, cppcompilados para o código intermediário em C++ ou Static C++, compilados para código intermediário em Framework, nopcompilados para a opção do pré-compilador, nocacompilados para código NOCA e VHDLcompilados para código intermediário VHDL.

Regras de Formação

O conceito de Regras de Formação ou Formation Rules foi proposto por (SIMÃO, 2001; SIMÃO, STADZISZ e KÜNZLE,2003) para permitir a criação de Rules específicas, a partir da representação genérica de uma Rule. Este conceito é bastante útil quando o conhecimento causal de uma Rule é comum para diferentes conjuntos de instâncias de FBEs, ou seja, um conjunto de Rules específicas se diferencia apenas nas instâncias referenciadas.

Um exemplo do uso das Regras de Formação é em cenário de simulação de um conjunto de semáforos. Cada semáforo da simulação possui o mesmo conjunto de regras, se diferenciando apenas na instância declarada. Caso ocorra uma simulação com um número elevado de semáforos, seria necessário replicar as regras manualmente para cada instancia declarada. Tornando o processo de desenvolvimento muito trabalhoso. Com o uso de regras de formação, o conhecimento da regra é genérico para todas as instancias de semáforos. Assim, para cada instancia declarada, esta regra genérica é replicada para as instancias específicas.

No trabalho realizado na disciplina, foi adicionado a *BNF* os tokens *formRule* e *end_formRule* para diferenciar de uma *Rule* comum. O Algoritmo abaixo apresenta a estrutura sintática de uma regra de formação na linguagem PON. Quando o conhecimento é genérico em uma regra, é feita referencia ao nome da FBE, ao invés do nome de uma instância específica.

Algoritmo 8: Exemplo do código FormRule

```
formRule rlTurnOn


   condition
      subcondition A
         premise prIsCrossed Apple.atIsCrossed == false
            and
         premise prHasFired Archer.atHasFired == false
            and
         premise prReadyToFire Controller.atHasFired == true
      end_subcondition
   end_condition

   action
      instigation inArcherFire1 Archer.mtFire
      instigation inAppleCrossed1 Apple.mtExplode
   end_action

end_formRule
```

O processo de regras de formação se encontra na opção quatro, ou seja, pré-compilação. Na qual cada regra de formação é replicada na forma de combinação das FBEs que a compõem, na forma de regras tradicionais.

Regras de FBE

O conceito de Regras de FBE ou *FBE Rules* foi criado de forma a facilitar o desenvolvimento de aplicações PON que necessitam aplicar um

conjunto de regras à todas as instâncias de um determinado *FBE.* Utilizando *FBE Rules*, torna-se possível a criação de *FBEs* como entidades computacionais autônomas, ou seja, que apresentam *Rules* intrínsecas à sua instanciação. Trata-se de um caso particular de Regra de Formação, na qual a *Rule* está relacionada apenas a um determinado tipo de *FBE.*

No trabalho realizado na disciplina, foi adicionado a *BNF* os tokens *fbeRule* e *end_fbeRule* para diferenciar de uma *Form Rule* e de uma *Rule* comum. O Algoritmo abaixo apresenta a estrutura sintática de uma *FBE Rule* na linguagem PON.

Algoritmo 9: Exemplo do código FbeRule

```
fbeRule ruleRobot
   condition
     subcondition conditionNumber1
       premise PrHasFired2 Robot.hasKicked == true
     end_subcondition
    end_condition
    action
       instigation instigation1 Robot.mtDebug();
    end_action
end_fbeRule
```

O processo de compilação de *FBE Rules* se encontra na opção quatro, ou seja, pré-compilação. Cada *FBE Rule* é replicada para cada uma das instâncias da FBE que as compõem, na forma de regras tradicionais.

Importante: Para que a regra seja aplicada a toda nova instância de uma determina *FBE* é necessário que as *Premises* e *Instigations* façam referência ao nome do FBE e não à sua instância. No exemplo acima, a *Premise* “PrHasFired2” possui como primeiro parâmetro “*Robot.hasKicked”*. Nesse caso, “Robot” é o nome de um *FBE* e, portanto, essa regra será aplicada a todas as instâncias de “Robot” que forem criadas, independentemente de quantas sejam.

Ponteiros

Para permitir a utilização inicial de ponteiros na LingPON, uma série de alterações foram feitas no código do compilador para C++.

Primeiramente, foram criados dois novos tokens no interpretador léxico (Lex): NEW e PTR. O token NEW reconhece a sequência “new”, enquanto PTR

reconhece “^”, que é o caractere especial que designa ponteiros nessa proposta. Dessa forma, sequências desse tipo podem ser reconhecidas:

```
Archer^ archer = new Archer
```

A próxima alteração foi feita no parser Bison. Foi criada uma estrutura do tipo list da Standard Library para armazenar todos os IDs de ponteiros que são reconhecidos, e essa estrutura será utilizada posteriormente na geração de código C++. A alteração feita no Bison permite reconhecer sequências do tipo em declarações de *Insts*:

```
Archer^ archer
```

```
Archer^ acher1, ^archer2, ^archer3
```

Também permite a utilização dos ponteiros dentro de métodos, em construções do tipo:

```
method mtNewArcher(archer1 = new Archer)
```

A próxima alteração foi feita na parte de geração de código C++. Para que não fosse necessário fazer uma alteração muito grande nesse trecho de código, foi criado um novo tipo de entidade dentro da classe Attribute, chamado A_PTR, juntamente com os tipos antigos A_BOOLEAN, A_INTEGER e assim por diante. O atributo do tipo A_PTR é utilizado dentro da classe CPPCompiler para fazer a geração de código do ponteiro. Para isso, um método auxiliar foi criado dentro da classe Compiler, que serve de base para CPPCompiler. O método createInstantiationPtr varre a lista de identificadores e cria atributos do tipo A_PTR, que posteriormente são lidos para imprimir código em C++ utilizando a sintaxe de ponteiros.

NOCA

O NOCA é uma arquitetura de computador alternativa às arquiteturas de computadores tradicionais, tais como von Neumann e fluxo de dados. Essa arquitetura foi desenvolvida de acordo com o modelo de execução PON, permitindo a execução de aplicações desenvolvidas segundo este paradigma.

Para isso, Linhares (2015) listou os requisitos abaixo:

A NOCA deve ser capaz de executar software composto unicamente de elementos do PON e, opcionalmente, também de funções sequenciais de acordo com o modelo von Neumann.

- A NOCA deve ser genérica, no sentido de que qualquer alteração na aplicação PON sendo executada, dependa somente de alterações de software, portanto não requerendo qualquer reconfiguração de hardware.
- A NOCA deve definir uma arquitetura de conjunto de instruções (Instruction Set Architecture, ou ISA) que implemente as funcionalidades dos elementos da cadeia de notificações do PON.
- A NOCA deve definir unidades de processamento que sejam capazes de executar as instruções da ISA e o fluxo de notificações de forma paralela.
- A NOCA deve ser capaz de executar uma aplicação PON mesmo que esta seja composta de mais elementos notificantes do que o número de unidades de processamento disponíveis para sua execução. Isto viabiliza a escalabilidade, no sentido de que o tamanho de uma aplicação PON a ser executada é limitado somente pela quantidade de memória disponível para armazenamento do respectivo software.

Com base nesses requisitos, alguns elementos do metamodelo do PON, são mapeados para a ISA do NOCA (Attribute, Premise, Condition, Method e Method von Neumann). Do ponto de vista da linguagem PON, a compilação usando a opção seis, gera um código intermediário que contém a aplicação desenvolvida, formada apenas pelas instruções da ISA.

Compilando para VHDL

É possível compilar as aplicações escritas em PON diretamente para VHDL, de forma que o código gerado possa ser compilado e executado em uma FPGA. Como a cadeia de notificações do PON é executada de forma diferenciada no hardware da FPGA nem todos os elementos do PON tem seu paralelo no código VHDL. Os principais elementos são: O *attribute* que

armazena os dados, as *premises* que realizam as avaliações sobre os dados e os *methods* que alteram os dados dos *attributes*. As conditions são apenas operações AND sobre as saídas das *premises*, as *instigations* são apenas ligações e assim por diante. A seguir serão detalhados os procedimentos necessários para que se possa executar a aplicação PON em uma FPGA.

Para que uma aplicação PON possa ser compilada para uma FPGA são necessários além do arquivo VHDL gerado na compilação, os seguintes arquivos:

- data_type_pkg.vhd
- NOP_attribute.vhd
- NOP_method.vhd
- NOP_premise.vhd

A segui será feita uma breve descrição de cada um deles.

O arquivo data_type_pkg.vhd contém a declaração genérica dos dados a serem armazenados no atributo, assim qualquer número de bits de qualquer tipo de dado pode ser armazenado em um atributo. Além disso, é possível concatenar um número variável de entradas em um atributo, resolvendo os conflitos por ordem de precedência.

O arquivo NOP_attribute.vhd contém a declaração dos registradores que armazenam de forma genérica os dados dos *attributes* notificando as *premises* quando alguma alteração ocorre.

O arquivo NOP_method.vhd contém o código das operações realizadas sobre os *attributes* e a lógica para fazer a atualização dos mesmos.

O arquivo NOP_premise.vhd contém o código que realiza os testes sobre os valores dos *attributes* determinando assim se a condição é verdadeira ou não.

A atual implementação do compilador PON para VHDL possui algumas limitações. A principal delas é que apenas os tipos de dados *boolean* e *integer* estão implementados. Os *methods* são capazes de realizar apenas as quatro operações básicas, soma, subtração, multiplicação e divisão, além da

atribuição de valores. Os dados do tipo *integer* são interpretados como inteiros com sinal de 32 bits e o resultado da multiplicação é truncado em 32 bits.

A resolução de conflitos é automática no código gerado, assim se dois ou mais *methods* tentarem atualizar o valor de um *attribute* ao mesmo tempo o *attribute* vai receber o valor do *method* que aparecer primeiro no código.

Como não existe a estrutura “main” no VHDL, o bloco main do programa PON foi utilizado para determinar as entradas, as saídas, o nome da entidade e o nome do arquivo para o VHDL gerado. Para declarar um *attribute* como entrada é necessário adicionar a seguinte linha ao bloco main no PON.

in <nome_fbe>.<nome_attribute>

ex: in controlador.liga

Para declarar um *attribute* como saída é necessário adicionar a seguinte linha ao bloco main no PON.

out <nome_fbe>.<nome_attribute>

ex: out controlador.contador

Para determinar o nome da entidade e consequentemente o nome do arquivo VHDL gerado é necessário adicionar a seguinte linha ao bloco main no PON.

entity <nome_entidade>

ex: entity controlaRobo

Assim a entidade no código gerado vai receber o nome <nome_entidade> e o arquivo VHDL gerado vai receber o nome <nome_entidade>.vhd.

Se o compilados não encontrar a palavra “entity” no bloco main o nome adotado é “NOPHD”.

Como o bloco main não é interpretado pelo flex/bison não é possível adicionar comentários ou realizar a verificação de erros, assim deve-se tomar cuidado com a sintaxe.

Para que se possa realizar a compilação do código gerado devem ser realizadas algumas configurações no compilador VHDL. Inicialmente é necessário adicionar todos os arquivos ao projeto, o código gerado e os arquivos apresentados anteriormente. A seguir é necessário configurar a ferramenta para interpretar o VHDL como VHDL 2008. A versão 2008 não é padrão e é necessária para compilar os tipos de dados utilizados.

Uma característica importante do código VHDL gerado é que os elementos do PON são componentes independentes, assim é possível utilizar o visualizador RTL para encontrar erros nas aplicações.

Integração

Cada funcionalidade nova na linguagem foi implementada de maneira individual, sendo necessário realizar a integração de cada uma das partes. Para isso foram realizadas reuniões no laboratório LSIP, a fim de resolver eventuais conflitos em arquivos fontes comuns do compilador. Como resultado, foram acrescentadas novas funcionalidades a linguagem e compiladores, com o objetivo de desenvolver o estado da técnica do compilador. A mostra o diagrama de blocos do compilador PON, após a realização das integrações dos trabalhos individuais da disciplina.

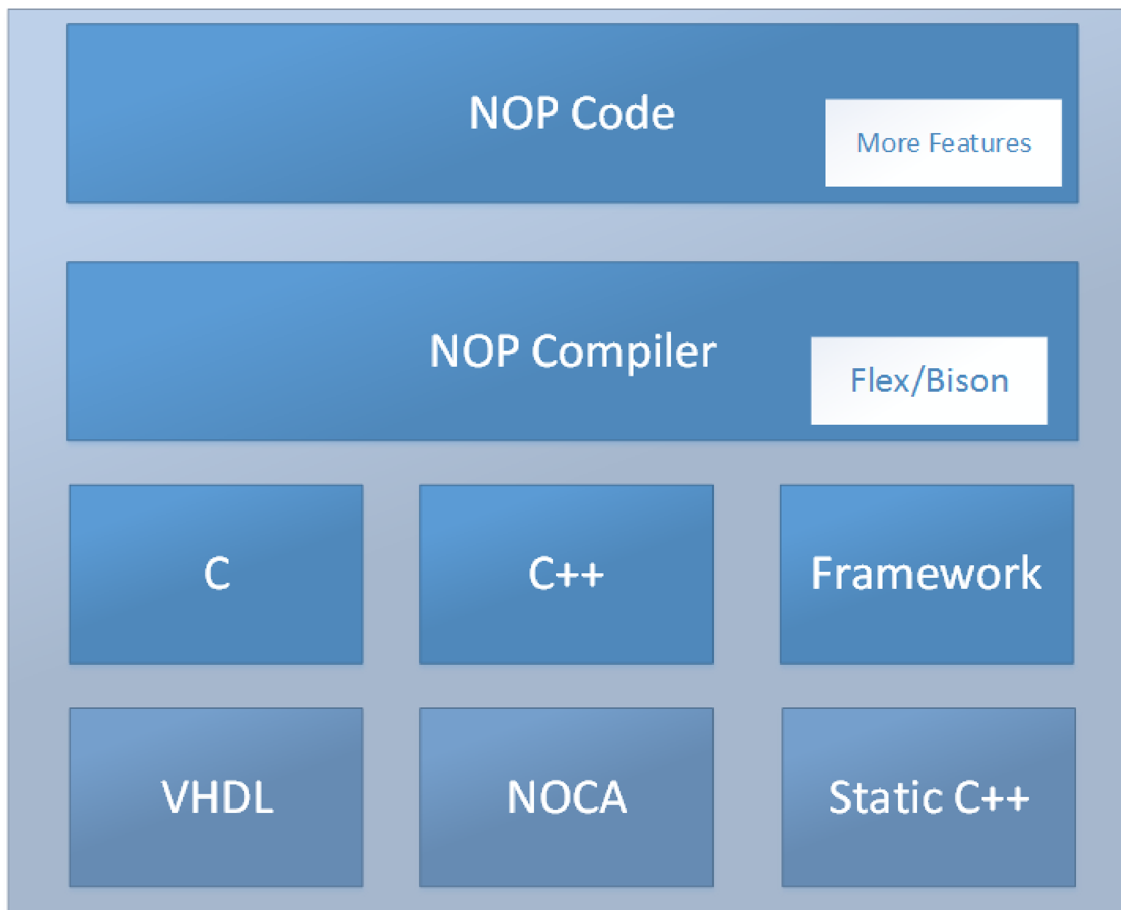

Figura – Diagrama de Blocos após Integração.

Referências

BANASZEWSKI, R. F. **Paradigma Orientado a Notificações : Avanços e Comparações**. Dissertação de Mestrado. CPGEI, UTFPR. Curitiba, Brasil, 2009.

FERREIRA, C. A. **Linguagem e compilador para o paradigma orientado a notificações (PON): Avanços e comparações.** 2015. Dissertação de Mestrado, PPGCA, UTFPR. Curitiba, Brasil, 2015.

LINHARES, R. R., **Contribuição para o desenvolvimento de uma arquitetura de computação própria ao paradigma orientado a notificações**. Tese de Doutorado, CPGEI, UTFPR. Brasil, 2015.

PETERS, E. **Coprocessador para Aceleração de Aplicações Desenvolvidas Utilizando Paradigma Orientado a Notificações**. 2012. Dissertação de Mestrado, CPGEI, UTFPR. Curitiba, Brasil, 2012.

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., 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.

SIMÃO, J. M. ; STADZISZ, P. C. **Paradigma Orientado a Notificações (PON) - Uma Técnica de Composição e Execução de Software Orientada a Notificações**. 2008, Brasil. Patente: Privilégio de Inovação. Número do registro: PI08055181, data de depósito: 26/11/2008, título: "PEDIDO DE PATENTE: Paradigma Orientado a Notificações (PON) Uma Técnica de Composição e Execução de Software Orientada a Notificações." , Instituição de registro:INPI - Instituto Nacional da Propriedade Industrial. Instituição(ões) financiadora(s): Universidade Tecnológica Federal do Paraná.

SIMÃO, J. M.; LINHARES, R. R. ; WITT, F. A. ; LIMA, C. R. E. ; STADZISZ, P. C. **Paradigma Orientado a Notificações em *Hardware* Digital**. 2012, Brasil. Patente: Privilégio de Inovação. Número do registro: BR102012026429, data de depósito: 16/10/2012, título: "PEDIDO DE PATENTE: Paradigma Orientado a Notificações em Hardware Digital" , Instituição de registro:INPI - Instituto Nacional da Propriedade Industrial. Instituição(ões) financiadora(s): UTFPR, 2012b.

RONSZCKA, A. F. **Contribuição Para a Concepção de Aplicações no Paradigma Orientado a Notificacões (PON) Sob o Viés de Padrões**. 2012. Dissertação de Mestrado, CPGEI, UTFPR. Curitiba, Brasil, 2012. Disponível em http://files.dirppg.ct.utfpr.edu.br/cpgei/Ano_2012/dissertacoes/CPGEI_Dissertacao_608_2012.pdf.

ANEXO E – RELATÓRIO DA DISCIPLINA LINGUAGENS/COMPILADORES - 2016

Este anexo apresenta o relatório apresentado pelos alunos Eduardo Bilk de Athayde e Fabio Negrini como trabalho final da disciplina "Linguagens e Compiladores" ofertada pela UTFPR em 2016 e ministrada por Prof. Dr. João Alberto Fabro e Prof. Dr. Jean Marcelo Simão.

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ (UTFPR)
PROGRAMA DE PÓS-GRADUAÇÃO EM
COMPUTAÇÃO APLICADA (PPGCA)

EDUARDO BILK DE ATHAYDE - DISCENTE PPGCA/UTFPR

FABIO NEGRINI – DISCENTE CPGEI/UTFPR

IMPLEMENTAÇÃO DE COMPILAÇÃO PARA C++ NAMESPACES PARA A LINGPON E OTIMIZAÇÕES NO TRATAMENTO DE PREMISSAS

RELATÓRIO DE TRABALHO FINAL

DISCIPLINA: TÓPICOS AVANÇADOS EM ENGENHARIA DE SOFTWARE

CÓDIGO: CAES101 - TURMA: PGCA – ANO/PERÍODO 2016/03

TEMA: LINGUAGENS E COMPILADORES.

OBJETO DE ESTUDO: LINGPON – LINGUAGEM E COMPILADOR PARA PARADIGMA ORIENTADO A NOTIFICAÇÕES (PON).

PROFESSORES:

PROF. DR. JOÃO ALBERTO FABRO.

PROF. DR. JEAN MARCELO SIMÃO.

CURITIBA – PR, BRASIL
DISCIPLINA TERCEIRO TRIMESTRE DE 2016
RELATÓRIO MARÇO DE 2017

1 INTRODUÇÃO

O Paradigma Orientado a Notificações (PON) teve sua origem em 2005 quando foi apresentada uma proposta de abordagem holônica para sistemas de manufatura (HMS) como um meta-modelo [1]. Uma vez visto que este meta-modelo possuía potencial além desta abordagem, esforços foram direcionados para a definição deste em forma de paradigma [2].

Em tempo, no PON há dois conjuntos de entidades principais, nomeadamente as *Fact Base Elements* (*FBEs*) e as *Rules*. As *FBEs* tratam o conhecimento factual e execucional no âmbito do PON por meio de sub-entidades chamadas *Attributes* e *Methods*. Por sua vez, as entidades *Rules* tratam do conhecimento lógico causal do PON por meio de sub-entidades *Premises*-*Conditions* e *Actions*-*Instigations*. Ainda, todo o processo de inferência de dá por notificações que partem dos *FBE*-*Attributes* para Rules, por meio de *Premises*-*Conditions*, e depois evoluem para os *Methods*-*FBE*, por meio de *Instigations*-*Actions*. Este processo de inferência é detalhado em [1].

Em teoria, este processo de Inferência Orientado a Notificações (ION) do PON, traz benefícios como o evitar de redundâncias o que permitiriam bom desempenho, bem como o desacoplamento implícito de entidades o que facilitaria distribuição, dentre outros, Como uma consequência natural, materializações seria então a próxima etapa do avanço do PON para fins de verificar suas propriedades. Isto dito, o PON foi inicialmente materializado como um framework para a linguagem C++ [3] e, posteriormente, uma linguagem de programação própria e respectivo compilador, chamado de LingPON [4]. Desde então, os esforços estão sendo direcionados para a evolução da LingPON, seja para otimização de suas materializações, seja para melhora na linguagem tanto no aspecto de facilidade quanto no aspecto de novas ferramentas [4].

Primeiramente, a LingPON permitia gerar código em Framework C++ PON 2.0, linguagem C segundo o PON e linguagem C++ segundo o PON [5]. Neste âmbito, subsequentemente o LingPON teve uma proposta de materialização em C++ estático, a qual apresentou considerável melhora em sua execução em relação a primeira versão da LingPON que ainda não estava a contento. Entretanto, o chamado LingPON estático ainda apresenta consideráveis limitações em sua usabilidade como

linguagem de programação, como a integração com código não estático, algo resolvido por definição na primeira versão do LingPON.

Isto considerado, este documento apresenta duas melhorias no tocante ao LingPON. Essas melhorias são fruto da disciplina de Tópicos Avançados em Engenharia de Software ministrado na fase 3 do ano de 2016. O tema da disciplina foi Linguagens e Compiladores e objeto de estudo foi justamente a LingPON.

A primeira parte deste documento propõe uma mudança na materialização de C++ estático para C++ orientado a *namespaces* que permitiu a melhor integração com bibliotecas externas em C++ e códigos não estáticos em geral, mantendo equivalente desempenho. A segunda parte é uma proposta para importação de codificação C++ a ser integrada à linguagem, integração esta que só foi possível com a mudança para *namespaces*. A terceira e última parte deste documento propõe várias melhorias nos controles das chamadas premissas ou *premises* da LingPON afim de otimizar e facilitar a sua utilização.

1.1 PROBLEMÁTICA

A implementação de compilação de LingPON para C++ com classes estáticas (*StaticCPP*), *i.e.* LingPON Estático, trouxe melhorias significativas na linguagem no tocante a desempenho [5]. Contudo, ela trouxe um ônus para a extensibilidade do código PON. De fato, uma vez que todas as classes criadas são estáticas, não há possibilidade de incluir bibliotecas externas já que classes estáticas não podem conter elementos não estáticos. Em suma, ainda não há, em LingPON Estático, uma maneira razoável de incluir bibliotecas para uso em *Methods* de *FBEs* nem bibliotecas externas customizadas.

Nesse contexto, o trabalho relatado por este documento teve como objetivo implementar a geração de código em LingPON para C++ com o uso de *namespaces* ao invés de classes estáticas, afim de sanar a problemática da extensibilidade do código PON e ao mesmo tempo manter o desempenho alcançado pela abordagem de implementação sem uso de sobrecargas de C++ OO em código gerado. Adicionalmente foi implementado um método para importação de código externo em PON, bem como para qualquer inclusão necessária para métodos PON.

Isto considerado, *namespaces* em C++ podem se referir a variáveis, funções, estruturas, enumerações, classes e membros de classes e/ou estruturas. Ademais, namespaces podem ajudar a tratar ambiguidade em nomes de entidades como variáveis, funções etc. De fato, a medida que uma aplicação cresce a necessidade de gerenciar ambiguidade aumenta, sendo os escopos proporcionados por namespaces uteis para tal.

Neste âmbito, a Figura 1 mostra um exemplo de aumento de complexidade na gestão de escopos de uma aplicação.

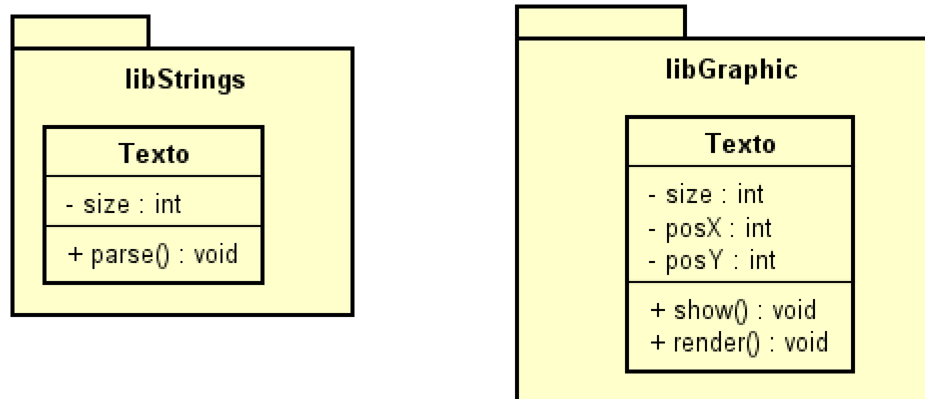

Figura 1 – Exemplo de bibliotecas com mesmo nome de classe.

O alto uso de bibliotecas aumenta consideravelmente o risco de haver nome repetidos em um mesmo escopo. A biblioteca gráfica *libGraphic* mostrada na Figura 1 contém uma classe para manipulação de elementos gráfico “Texto”, que representa uma caixa de texto. Por sua vez, a biblioteca *libStrings* representa uma biblioteca para manipulação de *strings* que possui uma classe “Texto” que nesse contexto manipula *strings* de grande tamanho.

Isso posto, caso uma aplicação que deseje utilizar ambas classes mostradas na Figura 1 deve fazer gestão de seu escopo, afim de sanar a ambiguidade das classes associadas. Justamente, *Namespaces* são formas de manter código vinculado a um escopo definido pelo programador. Então, nesse caso, seria possível associar cada classe representada na Figura 1 a um namespace e, com isso, sanar a incompatibilidade. Isso promove a mesma versatilidade e velocidade que havia na implementação *StaticCPP* porém não impossibilita a utilização de código não estático nos métodos das *FBEs*.

2 IMPLEMENTAÇÃO NAMESPACES

Para a criação da geração de código em *namespace* foram considerados os esforços do *StaticCPP* quanto a lógica de notificações e materialização de entidades PON demonstrados na Figura 2. Tais esforços serviram de base para a criação da geração de código em *Namespaces*. Em *StaticCPP*, cada uma dessas entidades do PON é materializada com uma classe estática. Entretanto, isso não interfere no tocante a dinâmica de notificações, cuja lógica é reaproveitável na abordagem por *namespaces*.

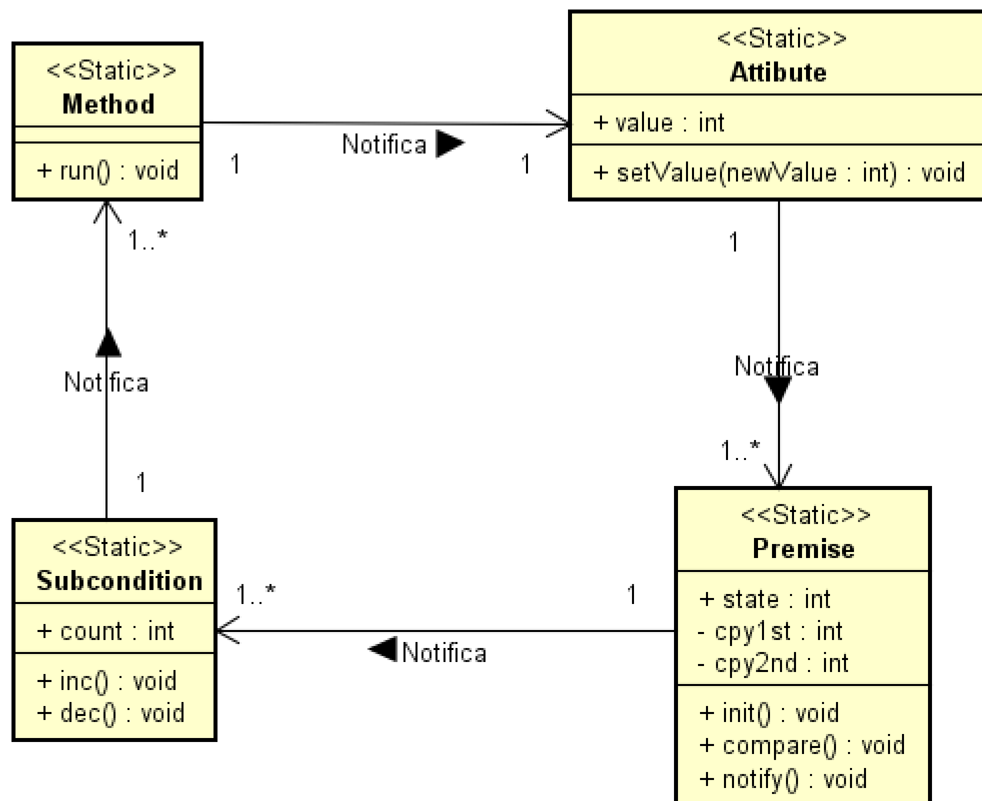

Figura 2 – Diagrama de materialização de notificações e entidades do *StaticCPP*.

A Figura 2 mostra a estrutura da cadeia de notificação entre as entidades PON: *Method*, Attribute, *Premise* e *Subcondition*. Esta é uma subdivisão organizacional de *Condition* de *Rule*, em suma. Isto dito, a execução do procedimento *run()* de um *Method* altera o valor de um *Attribute*, que por sua vez notifica as *Premise(s)* interessadas. Caso haja alteração em seu estado, esta notifica a(s) *Subcondition(s)* pertinente(s), então, caso a *Subcondition* seja aprovada ela dispara a execução do(s) *Method(s)* associados.

A abordagem para geração de código em *namespaces* seguiu a mesma lógica, contudo ao invés de gerar um arquivo para cada entidade agrupou-se todas as *Premises* em um arquivo "*Premises*", todas as *Subconditions* em um arquivo "*Subconditions*" e cada *Method* e *Attribute* relativo a *FBE* foi compilado no arquivo "*Instantiations*". A Figura 3 esboça a mudança.

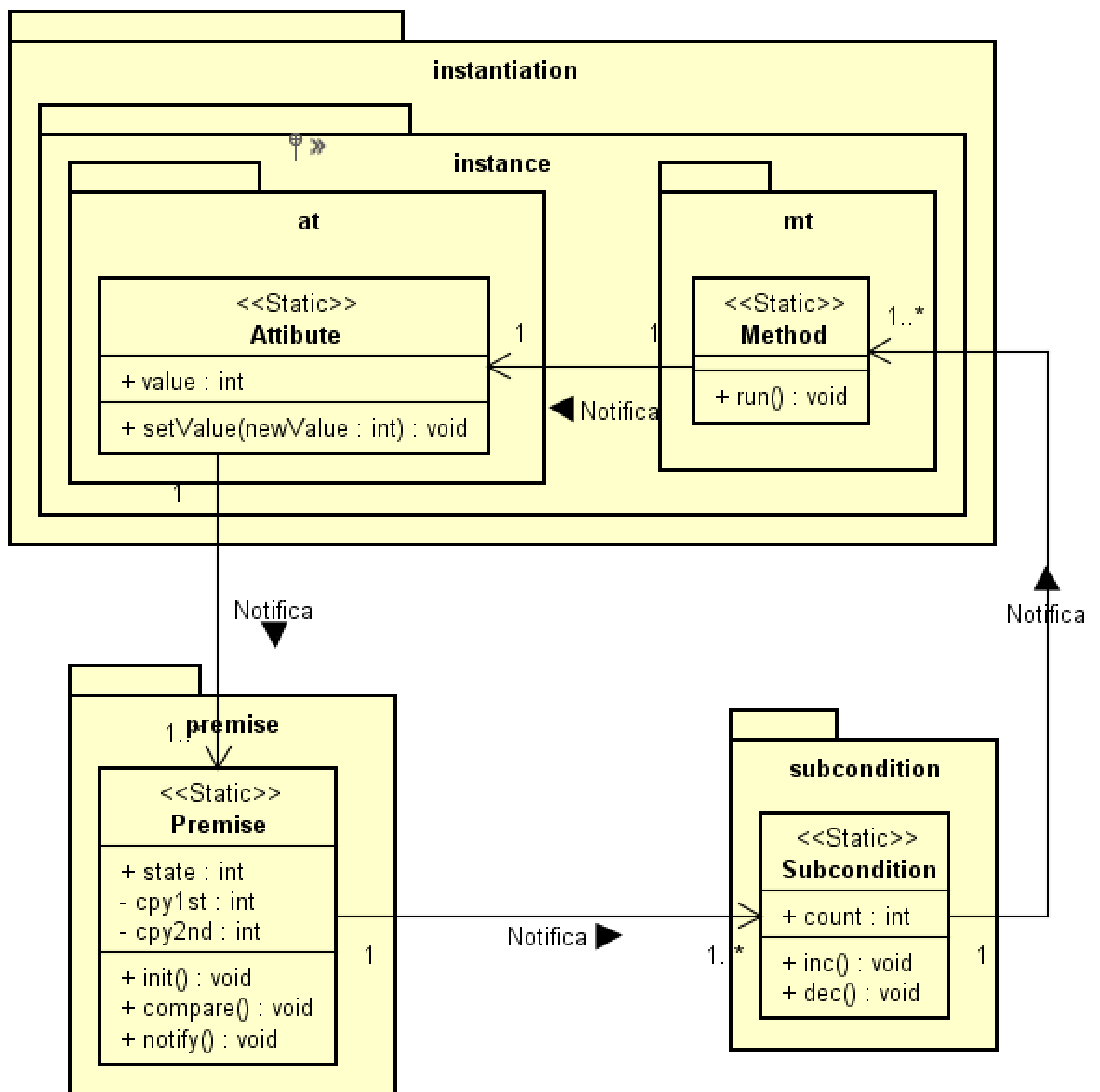

Figura 3 - Modificações de materialização da geração de código em *namespaces* em relação com *StaticCPP*.

A Figura 3 demonstra a inclusão das entidades *Method* e *Atribute* dentro de uma entidade (*namespace)* chamada *instantiation*. O namespace *instantiation* contém todas as instâncias de *FBE* da aplicação. Por sua vez todas as *Premises* e *Subconditions* da aplicação são alocadas nos *namespaces premise* e *subcondition*

respectivamente. Consolidando então as 3 entidades geradas em seus respectivos arquivos, para fins organizacionais, *ie Instantiation, Premise* e *Subcondition*.

Os tópicos seguintes expõem em maior detalhe cada uma das entidades (arquivos) e seu mecanismo interno.

2.1 *INSTANTIATIONS*

O *namespace Instantiations* (.h | .cpp) agrega todas as instâncias de *FBE* declaradas no código LingPON. A Figura 4 mostra o exemplo de código LingPON da aplicação chamada portão eletrônico (*electronicgate.pon*) [REF].

```
1  fbe Gate
2      attributes
3        integer gateState 0
4      end_attributes
5      methods
6        method opened(gateState) begin_method exProvider.getOpen(); end_method
7        method closed(gateState) begin_method exProvider.getClose(); end_method
8      end_methods
9  end_fbe
10
11 fbe Event
12     attributes
13       integer eventState 0
14     end_attributes
15     methods
16       method reset(eventState = 0)
17     end_methods
18 end_fbe
19
20 inst
21   Gate gate
22   Event event
23 end_inst
```

Figura 4 - Declaração de *FBEs* e Instâncias do código LingPON *electronicgate.*

Na Figura 4 as linhas 1-9 e 11-18 declaram, respectivamente as *FBEs Gate* e *Event*. Nas linhas 20-23 são declaradas as instâncias destas *FBEs* (*gate* e *event*, respectivamente). O código gerado da compilação do arquivo em questão pode ser visto na Figura 5.

```
#pragma once
namespace instantiation{
  namespace event{
    namespace at{
      namespace eventState{
        extern bool value;
        extern void setValue(bool newValue);
      }
    }
    namespace mt{
      extern bool reset();
    }
  }
  namespace gate{
    namespace at{
      namespace gateState{
        extern bool value;
        extern void setValue(bool newValue);
      }
    }
    namespace mt{
      extern bool opened();
      extern bool closed();
    }
  }
}
```

Figura 5 - Arquivo *instantiations.h* gerado pela compilação para *namespaces.*

Cada instância está dentro do *namespace “instantiation”* e cria um próprio *namespace* com métodos e atributos correlatos a declaração da *FBE*, cada qual em seu respectivo *namespace* também.

Dessa forma uma chamada de método pode ser feita acessando cada namespace até o método. Por exemplo, a execução do método *“opened*” da instância “*gate*” pode ser invocada pela chamada *instantiation::gate::mt::opened().*

2.2 *PREMISES*

Correlato com a lógica implementada em *StaticCPP* a geração de código em *namespace* também atribui os mesmos métodos (*init, compare* e *notify*) para cada *Premise* gerada em LingPON e as agrupa no arquivo chamado *premises* (.h | .cpp).

Cada *Premise* encontra-se dentro no *namespace* “*premise”* e detém as variáveis *state, cpy1st* e *cpy2nd.* Sua mecânica segue a mesma já implementada no *StaticCPP* na qual é inicializada com o procedimento *init* e quando notificada

(execução do método *notify*) executa a comparação para verificar se a Premise foi aprovada (método *compare*).

A Figura 6 ilustra as premissas geradas para o caso do programa *electronicgate.pon*.

```
1  #pragma once
2  namespace premise{
3      namespace prGateIsClosed{
4          extern bool state;
5          extern int cpy1st, cpy2nd;
6          extern void init();
7          extern void compare();
8          extern void notify_gate_gateState(int newValue);
9      }
10     namespace prGateIsOpened{
11         extern bool state;
12         extern int cpy1st, cpy2nd;
13         extern void init();
14         extern void compare();
15         extern void notify_gate_gateState(int newValue);
16     }
17     namespace prRemoteControlOn{
18         extern bool state;
19         extern int cpy1st, cpy2nd;
20         extern void init();
21         extern void compare();
22         extern void notify_event_eventState(int newValue);
23     }
24 }
```

Figura 6 - Implementação de premissas em *namespaces*.

2.3 *SUBCONDITIONS*

As *Subconditions* detêm os procedimentos *inc* e *dec* e uma variável *count*. Os procedimentos são executados à medida que os estados das *Premises* são alterados e no momento em que o contador aponta que todas as *Subcondition* de uma determinada *Rule* estão aprovadas a respectiva *Action* é acionada.

A lógica é inspirada no mesmo estilo daquela implementada no *StaticCPP*, sendo que agora com o agrupamento de todas as *subconditions* em um *namespace* chamado *subcondition.*

A Figura 7 mostra a geração do arquivo de cabeçalho das subcondition quanto que a Figura 8 mostra sua implementação.

```
#pragma once
namespace subCondition{
    namespace a2{
        extern int count;
        extern void inc();
        extern void dec();
    }
    namespace a1{
        extern int count;
        extern void inc();
        extern void dec();
    }
}
```

Figura 7 - Geração das *subconditions* em *namespaces.*

```
#include "subconditions.h"
#include "instantiations.h"

namespace subCondition{
    namespace a2{
        int count = 0;
        void inc(){
            count++;
            if (count == 2){
                instantiation::event::mt::reset();
                instantiation::gate::mt::closed();
            }
        }
        void dec(){
            count--;
        }
    }
    namespace a1{
        int count = 0;
        void inc(){
            count++;
            if (count == 2){
                instantiation::event::mt::reset();
                instantiation::gate::mt::opened();
            }
        }
        void dec(){
            count--;
        }
    }
}
```

Figura 8 - Implementação de *subconditions* para *namespaces.*

Na Figura 8 as linhas 10-11 e 23-24 mostram as *Instigations* de *Methods* referentes ao documentado no código LingPON (Figura 9).

```
29 rule rlOpeningGate
30   condition
31   subcondition a1
32     premise prRemoteControlOn event.eventState == 1 and
33     premise prGateIsClosed gate.gateState == 0
34   end_subcondition
35   end_condition
36     action
37       instigation inNone1 event.reset();
38       instigation inOpening1 gate.opened();
39     end_action
40 end_rule
41
42 rule rlClosingGate
43   condition
44     subcondition a2
45       premise prRemoteControlOn event.eventState == 1 and
46       premise prGateIsOpened gate.gateState == 1
47     end_subcondition
48   end_condition
49   action
50     instigation inNone2 event.reset();
51     instigation inClosing1 gate.closed();
52   end_action
53 end_rule
```

Figura 9 - Código de Rules para LingPON.

As linhas 37-38 e 50-51 mostradas na Figura 9 correspondem respectivamente ao código gerado na Figura 8 linhas 23-24 e 10-11.

3 IMPORT.PON

Com a finalização da implementação em *namespace* o problema para importar bibliotecas externas com classes não estáticas estava sanado. Contudo ainda não havia um método para realizar a inclusão de bibliotecas nos métodos PON. Para isso criou-se um arquivo padrão chamado *import.pon* que quando presente na compilação do código PON será interpretado e conterá as informações de inclusões necessárias segundo o programador.

A Figura 10 mostra a estrutura padrão para o arquivo de importação, bem como um exemplo para obtenção do código de abertura e fechamento do portão eletrônico (*electronicgate.pon*) por uma classe externa ao PON.

```
1  instantiations
2    #include "codExterno.h"
3    ExternalProvider* exProvider = new ExternalProvider();
4  end_instantiations
5
6  main
7
8  end_main
```

Figura 10 - Estruturação do arquivo import.pon.

Na Figura 10 a linha 1 e 4 delimitam a declaração que será inserida no arquivo das instâncias, enquanto as linhas 6 e 8 delimitam inclusões para o arquivo *main.cpp*.

As linhas 2-3 mostram respectivamente a inclusão da biblioteca externa e a instanciação da classe que será usada no método da FBE.o identificador *"exProvider"* foi usado no método PON, como demonstrado na linha 6-7 da Figura 4.

A linha 6-7 da Figura 4 também mostra uma nova sintaxe inserida no compilador LingPON (incluído na sua BNF) na qual o atributo que recebe o valor de um método externo (chamado entre as palavras chave *begin_* e *end_method*) é colocado como parâmetro da chamada de método.

Ou seja, na declaração *method opened (gateState) begin_method exProvider.getOpen(); end_method* o atributo *gateState* recebe o retorno da função *exProvider.getOpen()*.

Para tanto foi necessário alterar a BNF (a partir do arquivo de projeto do compilador chamado *bison_pon.y*) para adicionar a opção de compilação nº 8 e a nova sintaxe do método, que não interfere nos compiladores anteriores.

4 TESTES COMPARATIVOS

A fim de investigar os impactos relativos a desempenho da implementação da geração de código para *namespace*, buscou-se a comparação com o o código gerado em *StaticCPP*.

Para parametrização foi usado o código do *electronicgate.pon*, mostrado nas Figura 4 e Figura 9. Considerou-se o tempo em segundos de um milhão de acionamentos de portão (aprovação de Rule) como uma amostra. Dessa forma

coletou-se oito mil amostras compiladas com a otimização “-O3” do compilador GNU *g++* em uma máquina Linux (no caso, em ambiente virtual).

Uma vez que o ambiente estava suscetível a interferências do Sistema Operacional decidiu-se eliminar os mil valores mais altos da distribuição afim de reduzir o desvio padrão das amostras.

Isso posto os resultados obtidos (Figura 11) mostraram-se satisfatórios uma vez que não reduziram o desempenho em relação ao programa gerado pela compilação para *StaticCPP*:

- Valor mínimo amostrado em código *StaticCPP*: **39,91s**
- Valor mínimo amostrado em código *Namespaces*: **39,97s**
- Valor médio amostrado em código *StaticCPP*: **44,57s**
- Valor médio amostrado em código *Namespaces*: **44,10s**
- Valor mediano amostrado em código *StaticCPP*: **44,54s**
- Valor mediano amostrado em código *Namespaces*: **43,85s**
- Valor máximo amostrado em código *StaticCPP*: **48,27s**
- Valor máximo amostrado em código *Namespaces*: **48,76s**

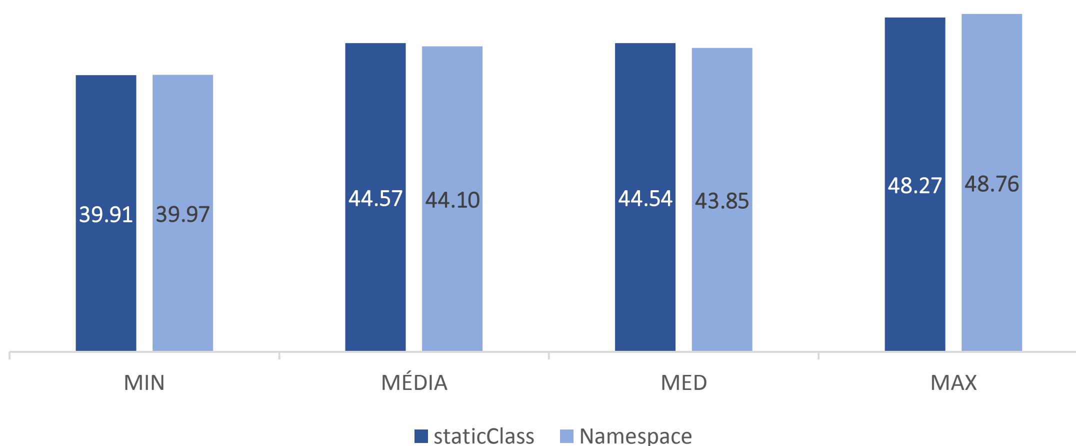

Figura 11 – Gráfico comparativo entre *Namespace* e *StaticCPP*

Nesse contexto a implementação de geração de código em *namespace* mostra-se de grande valia uma vez que sana a deficiência do código gerado em *staticCPP* sem penalizar o desempenho.

5 OTIMIZAÇÕES NO TRATAMENTO DE PREMISE

5.1 VINCULAR PREMISE NOMINAIS DE VÁRIAS RULES EM UMA ÚNICA PREMISE

Motivação

Sempre que uma *Premise* em PON é criada, ela é identificada pelas variáveis envolvidas para que sejam notificadas em caso de mudança de seu valor. Uma mesma *Premise* utilizada em várias *Rules* em PON acaba então sendo uma duplicidade desnecessária e que acaba impactando na performance, visto que dois elementos podiam ser convertidos em apenas um. Ademais, isso confrontava a própria teoria do PON sobre o evitar implícitos de redundâncias.

O objetivo desta melhoria é converter em um mesmo elemento *Premise* as *Premises* atualmente redundantes em LingPON de forma a otimizar o número de elementos na tabela de compilação e, consequentemente, reduzir o número de notificações durante sua execução.

Realização

Tome-se como base o exemplo abaixo, onde as premissas PON serão identificas em um mesmo elemento:

```
rule rlOpeningGate
   condition
         subcondition a1
               premise prRemoteControlOn  event.atEventState == 1 and
               premise prGateIsClosed gate.atGateState == 0
         end_subcondition
   end_condition
   ...
end_rule

rule rlClosingGate
   condition
         subcondition a2
               premise prRemoteControlOn  event.atEventState ==1 and
              premise prGateIsOpened gate.atGateState == 1
         end_subcondition
   end_condition
   ...
end_rule
```

De fato, já havia uma codificação anterior onde havia uma vinculação das *Premises* de mesmo nome em um único elemento mas não era utilizada. Foi feita uma revisão do código para melhor verificação de seu funcionamento depurando para garantir o vínculo a um mesmo elemento.

```
entityFound = semanticAnalyser.getEntity(internalID);

if (entityFound == 0) {

    premise = new Premise(internalID);
    premise->ref = leftId;
    premise->ref1 = rightId;
    premise->imp = imp;
```

Resultados

Como demonstração de resultado, o compilador foi modificado para gerar *log* de saída e apresentar o resultado da compilação e demonstrar que as duas chamadas da mesma *Premises* foram identificadas e unificadas em um mesmo elemento.

```
fbe Gate
    attributes
        integer atGateState 0
    end_attributes
    methods
         method mtOpened(atGateState = 1)
         method mtClosed(atGateState = 0)
    end_methods
end_fbe

fbe Event
    attributes
        integer atEventState 0
    end_attributes
    methods
        method mtReset(atEventState = 0)
    end_methods
end_fbe

inst
    Gate gate
    Event event
end_inst

strategy
    no_one
end_strategy
```

```
rule rlOpeningGate
    condition
        subcondition a1
                premise prRemoteControlOn event.atEventState == 1 and
                premise prGateIsClosed gate.atGateState == 0
        end_subcondition
    end_condition
    action
                instigation inNone1 event.mtReset();
        instigation inOpening1 gate.mtOpened();
    end_action
end_rule

rule rlClosingGate
    condition
        subcondition a2
                premise prRemoteControlOn event.atEventState == 1 and
                premise prGateIsOpened gate.atGateState == 1
        end_subcondition
    end_condition
    action
                instigation inNone2 event.mtReset();
        instigation inClosing1 gate.mtClosed();
    end_action
end_rule

main {

        //configura o portão como fechado
        gate->setatGateState(0);

        //abre o portão
        event->setatEventState(1);

        //fecha portão
        event->setatEventState(1);

}
```

sony@sony-VPCSE15FB: ~/workspace/PON/Debug

```
sony@sony-VPCSE15FB:~/workspace/PON/Debug$ ./PON 6 < 01_premisse_reuse_id.pon
Compilado para C++ Estatico, veja os resultados na pasta cppcompilados
Premissa prRemoteControlOn inicializada
Premissa prGateIsClosed inicializada
Premissa prRemoteControlOn reaproveitada
Premissa prGateIsOpened inicializada
sony@sony-VPCSE15FB:~/workspace/PON/Debug$
```

5.2 PERMITIR INFORMAR APENAS O IDENTIFICADOR DA PREMISE QUANDO ESTA FOR PREVIAMENTE DECLARADA

Motivação

Na LingPON, a declaração de uma premissa possui os seguintes argumentos: nome, variável FBE, operador, operando (que pode ser uma variávelFBE ou uma constante). A figura abaixo demonstra com detalhes todos os elementos:

Em casos onde uma *Premise* em PON possa ser reaproveitada a redefinição dos argumentos que seguem o nome passam a ser redundantes, visto que a *Premise* em PON já foi anteriormente definida. Manutenções no código passam a ser oneradas, pois a mudança de uma premissa obriga e revisão em todos os pontos onde a mesma premissa é utilizada.

O objetivo desta melhoria é permitir que uma Premise, uma vez declarada e nominada, possa ser reutilizada apenas declarando seu identificador. Esta nova sintaxe possui a vantagem de ser mais limpa e, caso a *Premise* seja redefinida, não será necessário alterar todas as instâncias desta premissa, bastando alterar apenas sua primeira definição.

Realização

O exemplo abaixo, a *Premise* prRemoteControlOn declarada abaixo:

```
rule rlOpeningGate
   condition
         subcondition a1
               premise prRemoteControlOn event.atEventState == 1 and
               premise prGateIsClosed gate.atGateState == 0
```

```
        end_subcondition
    end_condition
    …
end_rule
```

É novamente utilizada em outra condição:

```
rule rlClosingGate
    condition
        subcondition a2
            premise prRemoteControlOn event.atEventState == 1 and
            premise prGateIsOpened gate.atGateState == 1
        end_subcondition
    end_condition
    …
end_rule
```

Com a melhoria proposta, a chamada à premissa pode ser substituída pela seguinte sintaxe:

```
rule rlClosingGate
    condition
        subcondition a2
            premise prRemoteControlOn and
            premise prGateIsOpened gate.atGateState == 1
        end_subcondition
    end_condition
    …
end_rule
```

No arquivo *bison_pon.y*, nas declarações de premissas, foi adicionada uma redefinição de premissa:

```
premise            : PREMISE exp {$$ = compiler->createPremise("", ((PremiseCompType*)$2)->leftId, ((PremiseCompType*)$2)->op, ((PremiseCompType*)$2)->rightId,0);}
                   | PREMISE id {$$ = compiler->linkPremise($2);}
                   | PREMISE id exp {$$ = compiler->createPremise($2, ((PremiseCompType*)$3)->leftId, ((PremiseCompType*)$3)->op, ((PremiseCompType*)$3)->rightId,0);}
                   | PREMISE IMP exp {$$ = compiler->createPremise("", ((PremiseCompType*)$3)->leftId, ((PremiseCompType*)$3)->op, ((PremiseCompType*)$3)->rightId,1);}

                   | PREMISE IMP id exp {$$ = compiler->createPremise($3, ((PremiseCompType*)$4)->leftId, ((PremiseCompType*)$4)->op, ((PremiseCompType*)$4)->rightId,1);}
                   ;
```

Adicionalmente o método *Compiler::linkPremisse* foi definido conforme abaixo:

```
Entity * Compiler::linkPremise(std::string userEntityId){

        Entity *entityFound = semanticAnalyser.getEntity(userEntityId);
        Premise *premise;

        if (entityFound==0){
                semanticAnalyser.addError("Não foi possível localizar definição da premissa " + userEntityId);
                return NULL;
        }else{
                return (Entity *)premise;
        }
}
```

Resultado

Como demonstração de resultado, o compilador foi modificado para gerar *log* de saída e apresentar o resultado da compilação e demonstrar que as duas chamadas da mesma *Premise* foram identificadas e unificadas em um mesmo elemento, sendo que a segunda declaração da *Premise* foi informado apenas o primeiro argumento –nome:

```
fbe Gate
    attributes
        integer atGateState 0
    end_attributes
    methods
                method mtOpened(atGateState = 1)
          method mtClosed(atGateState = 0)
    end_methods
end_fbe

fbe Event
    attributes
        integer atEventState 0
    end_attributes
    methods
        method mtReset(atEventState = 0)
    end_methods
end_fbe

inst
    Gate gate
    Event event
end_inst

strategy
    no_one
end_strategy

rule rlOpeningGate
    condition
        subcondition a1
                premise prRemoteControlOn event.atEventState == 1 and
```

```
                premise prGateIsClosed gate.atGateState == 0
        end_subcondition
    end_condition
    action
        instigation inNone1 event.mtReset();
        instigation inOpening1 gate.mtOpened();
    end_action
end_rule

rule rlClosingGate
    condition
        subcondition a2
                premise prRemoteControlOn and
                premise prGateIsOpened gate.atGateState == 1
        end_subcondition
    end_condition
    action
        instigation inNone2 event.mtReset();
        instigation inClosing1 gate.mtClosed();
    end_action
end_rule

main {

        //configura o portão como fechado
        gate->setatGateState(0);

        //abre o portão
        event->setatEventState(1);

        //fecha portão
        event->setatEventState(1);

}
```

```
sony@sony-VPCSE15FB: ~/workspace/PON/Debug
sony@sony-VPCSE15FB:~/workspace/PON/Debug$ ./PON 6 < 02_premisse_declare.pon
Compilado para C++ Estatico, veja os resultados na pasta cppcompilados
Premissa prRemoteControlOn inicializada
Premissa prGateIsClosed inicializada
Premissa prRemoteControlOn aproveitada somente com declaração do identificador
Premissa prGateIsOpened inicializada
sony@sony-VPCSE15FB:~/workspace/PON/Debug$
```