ROBSON DUARTE XAVIER

PARADIGMAS DE DESENVOLVIMENTO DE *SOFTWARE*: COMPARAÇÃO ENTRE ABORDAGENS ORIENTADA A EVENTOS E ORIENTADA A NOTIFICAÇÕES

Dissertação submetida ao Programa de Pós-Graduação em Computação Aplicada da Universidade Tecnológica Federal do Paraná como requisito parcial para a obtenção do título de Mestre em Computação Aplicada.

Curitiba PR
Setembro de 2014

ROBSON DUARTE XAVIER

PARADIGMAS DE DESENVOLVIMENTO DE *SOFTWARE*: COMPARAÇÃO ENTRE ABORDAGENS ORIENTADA A EVENTOS E ORIENTADA A NOTIFICAÇÕES

Dissertação submetida ao Programa de Pós-Graduação em Computação Aplicada da Universidade Tecnológica Federal do Paraná como requisito parcial para a obtenção do título de Mestre em Computação Aplicada.

Área de concentração: *Engenharia de Sistemas Computacionais*

Orientador: Prof. Dr. João Alberto Fabro
Co-orientador: Prof. Dr. Jean Marcelo Simão

Curitiba PR
Setembro de 2014

Dados Internacionais de Catalogação na Publicação

X3p 2014

Xavier, Robson Duarte
Paradigmas de desenvolvimento de software : comparação entre abordagens orientada a eventos e orientada a notificações / Robson Duarte Xavier.-- 2014.
240 p.: il.; 30 cm

Texto em português, com resumo em inglês.
Dissertação (Mestrado) - Universidade Tecnológica Federal do Paraná. Programa de Pós-Graduação em Computação Aplicada, Curitiba, 2014.
Bibliografia: p. 179-190.

1. Software - Desenvolvimento. 2. Processamento de eventos (Informática). 3. Paradigma orientado a notificações. 4. Programação (Computadores). 5. Métodos de simulação. 6. Engenharia de sistemas. 7. Computação - Dissertações. I. Fabro, João Alberto, orient. II. Simão, Jean Marcelo, coorient. III. Universidade Tecnológica Federal do Paraná – Programa de Pós-graduação em Computação Aplicada. IV. Título.

CDD 22 -- 621.39

Biblioteca Central da UTFPR, Câmpus Curitiba

ATA DE DEFESA DE DISSERTAÇÃO DE MESTRADO Nº 25

Aos 29 dias do mês de setembro de 2014 realizou-se na sala C-301 a sessão pública de Defesa da Dissertação de Mestrado intitulada "Paradigmas de desenvolvimento de software: comparação entre abordagens orientada a eventos e orientada a notificações", apresentada pelo aluno **Robson Duarte Xavier** como requisito parcial para a obtenção do título de Mestre em Computação Aplicada, na área de concentração “Engenharia de Sistemas Computacionais”, linha de pesquisa “Sistemas Embarcados”.

Constituição da Banca Examinadora:

Prof. Dr. João Alberto Fabro, UTFPR - CT (Presidente) _____

Prof. Dr. Marcos Antonio Quináia, UNICENTRO _____

Prof. Dr. Paulo Cézar Stadzisz, UTFPR - CT _____

Prof. Dr. Robson Ribeiro Linhares, UTFPR – CT _____

Prof. Dr. Jean Marcelo Simão, UTFPR – CT _____

Prof. Dr. Andrey Ricardo Pimentel, UFPR – CT _____

Em conformidade com os regulamentos do Programa de Pós-Graduação em Computação aplicada e da Universidade Tecnológica Federal do Paraná, o trabalho apresentado foi considerado _____ (aprovado/reprovado) pela banca examinadora. No caso de aprovação, a mesma está condicionada ao cumprimento integral das exigências da banca examinadora, registradas no verso desta ata, da entrega da versão final da dissertação em conformidade com as normas da UTFPR e da entrega da documentação necessária à elaboração do diploma, em até _____ dias desta data.

Ciente (assinatura do aluno): _____

(para uso da coordenação)

A Coordenação do PPGCA/UTFPR declara que foram cumpridos todos os requisitos exigidos pelo programa para a obtenção do título de Mestre.

Curitiba PR, _____/_____/_____ _____

"A Ata de Defesa original está arquivada na Secretaria do PPGCA".

*Aos queridos tios Virgílio Pinheiro (in memoriam) *17/03/1942 †25/04/2012, João Pinheiro (in memoriam) *19/12/1939 †25/04/2013, José Pinheiro (in memoriam) *15/02/1936 †18/10/2013, lavradores, desbravadores e guerreiros dos tempos de outrora.*

*Ao meu Ligamento Cruzado Anterior (LCA) do joelho direito *25/04/1977, rompido em 29/11/2013.*

*À querida avó Geralda Xavier (in memoriam) *18/08/1925 †09/02/2014, lavradora, guerreira, sábia, tranquila, mãe de treze filhos, vaidosa, bem humorada e de coração enorme.*

À Dúvida e à Morte, amantes daqueles que vivem.

Agradecimentos

Agradeço primeiro a você, que está lendo estes agradecimentos, pela sua curiosidade, tempo e coragem. Caso você leia o trabalho inteiro, fugazes palavras não retratariam fielmente meu agradecimento. Agradeço a Deus, meus pais, meus irmãos, e reverencio solenemente meus nobres orientadores Prof. Dr. João Alberto Fabro e Prof. Dr. Jean Marcelo Simão, que demonstraram esperança muito maior que a minha própria esperança, e também pela dedicação, tempo e direcionamentos deste trabalho.

Agradeço a todos aqueles que ajudaram, de forma direta ou indireta, na realização desta pesquisa. Primeiramente, aos professores das bancas de todos os Seminários de Qualificação, Profs. Dr. Carlos Alberto Maziero, Dr. Paulo Cézar Stadzisz e Dr. Robson Ribeiro Linhares, cujos apontamentos engradeceram e tornaram melhor meu trabalho. Também agradeço aos membros externos da banca examinadora, pela atenção e contribuição dedicadas, Profs. Dr. Andrey Pimentel e Dr. Marcos Quináia. Em terceiro lugar, muito obrigado a todos os professores e colaboradores do PPGCA e da UTFPR.

Congratulo, pelos excelentes trabalhos, Dr. Roni Banaszewski , Ma. Luciane Wiecheteck, doutorando Me. Adriano Ronszcka, Me. Marcio Batista, Me. Glauber Valença, dessa forma, a todos do Grupo de Pesquisa do Paradigma Orientado a Notificações (PON), capitaneados pelos ilustres criadores do PON Profs. Simão e Stadzisz. Também agradeço aos caros colegas de mestrado (mestres ou em mui breve mestres) Rodrigo Gregori, Fernando Muchalski, Marcelo Manhães, Paulo Henrique, Cleverson Avelino e Clayton Kossoski.

Aos professores da minha *alma mater* Universidade Estadual de Maringá (UEM) Me. Flávio Arnaldo Braga e Dr. Sérgio Roberto Pereira da Silva (*in memoriam*), pela ajuda e incentivo no início da caminhada de mestrado. E aos meus amigos e amigas como André, Arthur, Beto, Cefernan, Clara, Daniel, Douglas, Fernando, Gustavo, Elen, Iatskiu, João, Jorge, Ju, Lázaro, Leandro, Luciano, Luna, Marcel, Marcelo, Marcos, Maycol, Parpinelli, Rafa, Rafael, Rebeca, Sabrina, Simão, Thais e Vinícius... Pela companhia durante a caminhada.

E talvez os agradecimentos mais importantes: a todos que me ajudaram nessa imensa, dificultosa e turbulenta caminhada (ou seria maratona?), que respeitaram meu espaço e tempo, que me ouviram, incentivaram e esperaram, que partilharam do seu tempo, que disseram o que tinha que ser dito, por sua honestidade e auxílio nos meus passos. Meu eterno obrigado!

“More is not better (or worse) than less, just different.” — The paradigm paradox. (Peter Van Roy, 2009)

“Mais não é melhor (ou pior) do que menos, apenas diferente” — O Paradoxo do Paradigma. (Peter Van Roy, 2009)

"It's a dangerous business, Frodo, going out your door. You step onto the road, and if you don't keep your feet, there's no knowing where you might be swept off to." (J.R.R. Tolkien, The Lord of the Rings, 1954).

"É perigoso sair porta afora, Frodo", ele costumava dizer. "Você pisa na Estrada, e, se não controlar seus pés, não há como saber até onde você pode ser levado. …" — Frodo, citando Bilbo (J.R.R. Tolkien, O Senhor dos Anéis, 1954).

“*All we have to decide is what to do with the time that is given us.*” — Gandalf. (J.R.R. Tolkien, The Fellowship of the Ring, 1954).

“Tudo o que temos de decidir é o que fazer com o tempo que nos é dado.” — Gandalf. (J.R.R. Tolkien, A Irmandade do Anel, 1954).

Resumo

O objetivo deste trabalho é comparar dois paradigmas distintos de desenvolvimento de *software*, o emergente Paradigma Orientado a Notificações (PON) e o Paradigma Orientado a Eventos (POE). Objetivos, métodos e ferramentas são apresentados, e são descritos dois casos de estudo (o primeiro em três cenários), respectivas reflexões, experimentos e dados. A comparação é teórico-prática, correlacionando características estruturantes em PON e POE conforme uma taxonomia comum, os mensurando em complexidade de código-fonte (números de linhas de código, escopos e *tokens*) e os comparando em medições durante execução (tempo de reposta e tempo total de execução), por meio da construção e experimentação dos casos de estudo em ambos os paradigmas. Como resultados, identifica-se que PON, apesar de ter inspiração em eventos, utilizando notificações em seu ciclo de execução, apresenta diferenças conceituais em relação a POE. Além disso, no atual estado da técnica, utilizando o *Framework* PON, apresenta tempo de resposta durante execução comparável ao POE, enfatizando desempenho que se adapta ao contexto do *software* (tempos de resposta menores quando os eventos devem ser desprezados e tempos de resposta maiores quando da execução de mais uma ação por evento). Já quando se utiliza uma linguagem e compilador específico para PON (LingPON e respectivo código compilado) os tempos de resposta foram lineares e comparáveis ao POE.

Palavras-chave: Paradigmas de Programação, Paradigma Orientado a Eventos, Paradigma Orientado a Notificações, Comparação de Paradigmas de Programação.

Abstract

The objective of this work was comparing two distinct *software* development paradigms, namely, the emerging Notification Oriented Paradigm (NOP) and Event-driven Paradigm (EDP). Objectives, methods and tools are presented and two case studies are described (the first with three scenarios) and their respective reflexions, experiments and data. The comparison is theoretical and practical, correlating structural characteristics in NOP and EDP in a common taxonomy, therefore comparing them in code complexity (number of lines of code, closures and tokens) and comparing them with respect to performance (response time and total execution time), by building and instrumenting the case studies in both paradigms. As results, was identified that NOP, despite being inspired by events, using notifications on its execution model, shows conceptual differences from EDP. Moreover, in the present state of technology, NOP has response time during execution comparable to EDP implementation, and NOP has performance that adapts to *software* context (faster response times when events should be ignored and higher response times when executing more actions per event. With specific NOP language and compiler (LangNOP) data shows linear response times comparable to the EDP.

Keywords: Programming Paradigms, Event-driven Paradigm, Notification Oriented Paradigm, Comparing Programming Paradigms.

SUMÁRIO

Resumo.....xiii

Abstract.....xv

Lista de Figuras.....xxi

Lista de Tabelas.....xxv

Lista de Algoritmos.....xxvii

Lista de Abreviações.....xxix

Capítulo 1 - Introdução.....1

1.1 O Problema de Pesquisa: Comparação do Paradigma Orientado a Notificações (PON) com o Paradigma de Programação Orientado a Eventos (POE).....2

1.2 Justificativa.....3

1.3 Objetivos.....5

1.4 Métodos e Ferramentas.....5

1.4.1 Revisão Bibliográfica.....6

1.4.2 Projeto de Dissertação de Mestrado.....6

1.4.3 Casos de Estudo.....7

1.4.4 Comparações.....9

1.4.5 Fase de Conclusão.....10

1.5 Organização do trabalho.....11

Capítulo 2 - Revisão do Estado da Arte.....13

2.1 Paradigmas de Programação.....13

2.2 Paradigma Orientado a Notificações.....20

2.3 Paradigma Orientado a Eventos.....26

2.4 Técnicas do Paradigma Orientado a Eventos.....36

2.4.1 Técnica Handler-Dispatcher....37
2.4.2 Técnica State Pattern....39
2.4.3 Técnica Observer Pattern....41
2.5 Reflexões sobre a Revisão do Estado da Arte....43

Capítulo 3 - Casos de estudo....47
3.1 Caso de Estudo 1: Simulador de Transporte Individual....48
3.1.1 Primeiro Cenário....50
(I) Solução PON para o Primeiro Cenário....53
(II) Solução POE via despachante – Dispatcher Handler (event loop)....57
3.1.2 Segundo Cenário....62
(I) Solução PON para o Segundo Cenário....68
(II) Solução POE para o Segundo Cenário – Handler-Dispatcher e State Pattern....72
3.1.3 Terceiro Cenário....78
(I) Solução PON para o Terceiro Cenário....82
(II) Solução POE para o Terceiro Cenário – Observer Pattern....89
3.1.3 Reflexões do Primeiro Caso de Estudo....97
3.2 Caso de Estudo 2: Portão Eletrônico....98
(I) Solução PON para o Portão Eletrônico....102
(II) Primeira Solução POE (Handler-Dispatcher) para o Portão Eletrônico....110
(III) Segunda Solução POE (State Pattern) para o Portão Eletrônico....113
(IV) Terceira Solução POE (Observer Pattern) para o Portão Eletrônico....117
3.3 Reflexões dos casos de estudo....120

Capítulo 4 - Comparações entre os Paradigmas Orientado a Eventos e Orientado a Notificações....125
4.1 Comparação de características estruturantes....126
4.2 Comparações em complexidade de código-fonte de software em PON e POE....138
4.3 Comparações em medidas de execução de software em PON e POE....147
4.3.1 Primeiro experimento: PON versus Handler-Dispatcher....149
4.3.2 Segundo experimento: PON versus StateMachineHandler....152
4.3.3 Terceiro experimento: PON versus ObserverHandler....154
4.3.4 Quarto experimento: PON versus ObserverHandler....156

4.3.5 Quinto experimento: Caso de Estudo Portão Eletrônico PON versus POE (Dispatcher, State e Observer).....161
4.3.6 Sexto experimento: comportamento irregular a partir de 32 mil eventos recebidos no software em Framework PON.....165
4.4 Resultados/Reflexões das comparações.....168

Capítulo 5 - Conclusões e Trabalhos Futuros.....171
5.1 Conclusão.....171
5.2 Contribuições principais deste trabalho.....173
5.3 Contribuições secundárias deste trabalho.....174
5.4 Trabalhos futuros.....175
5.4.1 Comparações com outros paradigmas.....175
5.4.2 Facilidade de programação.....175
5.4.3 PON em inovações para Sistemas Auto Adaptativos.....176
5.4.4 PON versus Máquinas de Estado.....177

Referências Bibliográficas.....179

Apêndice A - Gráficos de desempenho.....191

Apêndice B - Dados da instrumentação básica da plataforma dos experimentos.....201

Apêndice C - Investigação executada para identificação da anomalia/irregularidade. .203

Apêndice D - Ferramentas Utilizadas.....205

Anexo A - Especificação Formal da Linguagem PON.....207

Lista de Figuras

Figura 1: Taxonomia de paradigmas de programação [Van Roy, 2009].....16
Figura 2: Entidades PON [Simão et al., 2012a].....21
Figura 3: Exemplo de Rule [adaptado de Simão e Stadzisz, 2008, 2009].....22
Figura 4: Inferência por notificações [Simão at al., 2014].....23
Figura 5: Método para Projeto de Software em PON - Desenvolvimento Orientado a Notificações (DON) [Wiecheteck, 2011].....26
Figura 6: Evento, condição detectada, notificação [Faison, 2006].....27
Figura 7: Paradigma Orientado a Eventos adaptado de [Hansen e Fossum, 2010].....28
Figura 8: Classificação de Linguagens para Programação Reativa adaptado de [Bainomugisha et al., 2013].....30
Figura 9: Um diagrama conceitual do componente Dispatcher [Ferg, 2006].....38
Figura 10: Sistema tradicional orientado a eventos com ciclo de eventos segundo Samek [Samek, 2008].....39
Figura 11: Diagrama de Classes original do padrão de projeto State [Gamma et al., 1995]....40
Figura 12: Diagrama de Classes original padrão de projeto Observer [Gamma et al., 1995].. 42
Figura 13: Figura conceitual de um exosqueleto adaptada do projeto Hardiman I da General Electric [Fick e Makinson, 1971].....49
Figura 14: Chave ON/OFF, Joystick (sem botões), e movimentos do Joystick para mover o transporte no primeiro cenário.....51
Figura 15: Funções do exosqueleto no ambiente de simulação do primeiro cenário.....51
Figura 16: Progressão da simulação em atividades do primeiro cenário.....52
Figura 17: Diagrama de Classes do software em PON do primeiro cenário.....53
Figura 18: Diagrama de Sequência relacionando dispositivo gerador de eventos, software e Transport.....58
Figura 19: Diagrama de Classe Solução Dispatcher Handler em POE para o Primeiro Cenário. O componente Dispatcher propicia uma relação indireta.....59
Figura 20: Diagrama de Sequência Solução Dispatcher Handler em POE para o primeiro

cenário. 60
Figura 21: Chave ON/OFF e Joystick com um único botão para mover o exosqueleto (ou o braço mecânico) no segundo cenário.....63
Figura 22: Movimentos do joystick para mover o exosqueleto (esquerda) ou o braço mecânico (direita) no segundo cenário.....63
Figura 23: Funções do exosqueleto e braço mecânico para o segundo cenário.....64
Figura 24: Progressão da simulação em atividades (visão resumida).....65
Figura 25: Progressão da simulação em atividades (exclusivamente um braço mecânico).....66
Figura 26: Progressão da simulação em atividades (visão completa).....67
Figura 27: Classes do software em PON para o segundo cenário.....69
Figura 28: Excerto Diagrama de Classes em POE – State Pattern.....73
Figura 29: Excerto do diagrama de Classes em POE do segundo cenário.....74
Figura 30: Diagrama de Sequência da solução State Pattern em POE para o segundo cenário. 75
Figura 31: Chave ON/OFF e Joystick com dois botões para mover o exosqueleto ou o(s) braço(s) mecânico(s) no terceiro cenário.....78
Figura 32: Movimentos do joystick para mover o exosqueleto (esquerda) ou os braços mecânicos (direita) no terceiro Cenário (análogo ao segundo cenário).....79
Figura 33: Funções do exosqueleto e dos braços mecânicos para o terceiro cenário.....80
Figura 34: Progressão da simulação em estados do terceiro cenário (visão resumida).....82
Figura 35: Classes do software em PON para o terceiro cenário.....84
Figura 36: Excerto do diagrama de Classes em POE do terceiro cenário (Observer Pattern)..91
Figura 37: Diagrama de Sequência da solução Observer Pattern em POE para o terceiro cenário.....93
Figura 38: Desenho conceitual de um controle remoto para um portão eletrônico.....98
Figura 39: Funções do Simulador de Portão Eletrônico.....100
Figura 40: Diagrama de estados do Simulador Portão Eletrônico.....101
Figura 41: Diagrama de Classes do software Simulador de Portão Eletrônico em PON.....103
Figura 42: Diagrama de Classes do software Simulador de Portão Eletrônico em POE - Dispatcher.....111
Figura 43: Diagrama de Sequência de Handler-Dispatcher em POE para o Simulador de Portão Eletrônico.....112

Figura 44: Excerto do Diagrama de Classes do software Simulador de Portão Eletrônico em POE – State Pattern.....114
Figura 45: Excerto do Diagrama de Classes do software Simulador de Portão Eletrônico em POE.....115
Figura 46: Diagrama de Classes do software Simulador de Portão Eletrônico em POE - Observer.....118
Figura 47: Taxonomia de paradigmas de programação incluindo PON em visão aproximada (excerto) [expandido a partir de Van Roy, 2009].....136
Figura 48: Taxonomia de paradigmas de programação incluindo o PON – em destaque - visão completa da taxonomia [expandido a partir de Van Roy, 2009].....137
Figura 49: Gráfico linhas de código-fonte PON e POE do primeiro caso de estudo Exoskeleton.....139
Figura 50: Gráfico de diferenças entre linhas de código-fonte modificadas, acrescentadas e removidas entre cenários, em PON e POE, do primeiro caso de estudo – Exoskeleton.....140
Figura 51: Gráfico de linhas de código-fonte PON e POE do segundo caso de estudo Portão Eletrônico.....141
Figura 52: Gráfico número de closures PON e POE do primeiro caso de estudo Exoskeleton em todos os cenários.....142
Figura 53: Gráfico número de closures PON e POE do segundo caso de estudo Portão Eletrônico.....144
Figura 54: Gráfico número de tokens PON e POE do primeiro caso de estudo Exosqueleto. 145
Figura 55: Gráfico número de tokens PON e POE do segundo caso de estudo Portão Eletrônico.....146
Figura 56: Gráfico com tempos de execução para cada solução, na simulação do primeiro cenário.....150
Figura 57: Gráfico com tempos de resposta (ms) por tipo de evento POE, PON e PON-Compilador durante a execução na simulação no primeiro cenário para 10.000 n-steps. 151
Figura 58: Gráfico com tempos de resposta (ms) por tipo de evento POE x PON durante a execução na simulação no primeiro cenário para 1.000.000 n-steps.....152
Figura 59: Gráfico com tempos de execução para cada solução, na simulação do primeiro

cenário.....154
Figura 60: Gráfico com tempos de execução para cada solução, na simulação do primeiro cenário.....156
Figura 61: Desenho do roteiro do quarto experimento PON e Observer, terceiro cenário do primeiro caso de estudo.....159
Figura 62: Gráfico com tempos de execução para cada solução, nesta segunda simulação do terceiro cenário do primeiro caso de estudo.....160
Figura 63: Gráfico de tempo de execução do caso de estudo Portão Eletrônico com tempo de referência 1ms.....164
Figura 64: Gráfico com tempos de resposta (ms) por tipo de evento POE x PON durante a execução na simulação no primeiro cenário para 1.000 n-steps.....192
Figura 65: Gráfico com tempos de resposta (ms) por tipo de evento POE x PON durante a execução na simulação no segundo cenário para 1.000 n-steps.....193
Figura 66: Gráfico com tempos de resposta (ms) por tipo de evento POE x PON durante a execução na simulação no segundo cenário para 10.000 n-steps.....194
Figura 67: Gráfico com tempos de resposta (ms) por tipo de evento POE x PON durante a execução na simulação no segundo cenário para 1.000.000 n-steps.....195
Figura 68: Gráfico com tempos de resposta (ms) por tipo de evento POE x PON durante a execução na simulação no terceiro cenário para 1.000 n-steps.....196
Figura 69: Gráfico com tempos de resposta (ms) por tipo de evento POE x PON durante a execução na simulação no terceiro cenário para 10.000 n-steps.....197
Figura 70: Gráfico com tempos de resposta (ms) por tipo de evento POE x PON durante a execução na simulação no terceiro cenário para 1.000.000 n-steps.....198
Figura 71: Gráfico de tempo de execução do caso de estudo Portão Eletrônico com tempo de referência 1,000ms.....199
Figura 72: Gráfico de tempo de execução do caso de estudo Portão Eletrônico com tempo de referência 100ms.....199
Figura 73: Gráfico de tempo de execução do caso de estudo Portão Eletrônico com tempo de referência 10ms.....200

Lista de Tabelas

Tabela 1: Paradigmas/conceitos [adaptado de VAN ROY, 2009] (S)IM/(N)ÃO.....19
Tabela 2: Rules, Condition e suas Premises, Methods instigados do software do primeiro cenário exosqueleto em PON.....55
Tabela 3: Rules, Condition e suas Premises e Methods instigados do software do segundo cenário exosqueleto e braço mecânico em PON.....70
Tabela 4: Rules, Condition e suas Premises e Methods instigados do software do terceiro cenário exosqueleto e dois braços mecânicos (esquerdo e direito) em PON.....87
Tabela 5: Rules, Condition e suas Premises e Methods instigados do software Simulador de Portão Eletrônico em PON.....107
Tabela 6: Unidades computacionais e seus respectivos estados em PON [Banaszewski, 2009].
.....129
Tabela 7: PON incluso na Taxonomia [adaptado de VAN ROY, 2009] (S)IM/(N)ÃO.....131
Tabela 8: Características de PON segundo a taxonomia expandida [adaptado de VAN ROY, 2009] (S)IM/(N)ÃO.....135
Tabela 9: Tempos de execução em milissegundos (ms) para cada solução, na simulação do primeiro cenário.....149
Tabela 10: Tempos de execução em milissegundos (ms) para cada solução, na simulação no segundo cenário.....153
Tabela 11: Tempos de execução em milissegundos (ms) para cada solução, na primeira simulação no terceiro cenário.....155
Tabela 12: Tempos de execução em milissegundos (ms) para cada solução, na segunda simulação no terceiro cenário.....160
Tabela 13: Tempos totais incluindo tempo ideal de referência dos experimentos do segundo caso de estudo.....164
Tabela 14: Dados de instrumentação Framework PON, PON-Compilador e Observer no sexto experimento.....166

Lista de Algoritmos

Algoritmo 1: Pseudocódigo do Paradigma Imperativo.....14
Algoritmo 2: Pseudocódigo elementar de não-determinismo observável adaptado de [Van Roy e Haridi, 2004].....18
Algoritmo 3: Inicialização de Premissas Compartilhadas em SimpleEventHandlerPON.cpp. 56
Algoritmo 4: Código de três regras em SimpleEventHandlerPON.cpp.....56
Algoritmo 5: Código-fonte em LingPON em exoskeleton.pon.....57
Algoritmo 6: Primeira versão do código para execução de ações ou retransmissão de mensagem via objeto Despachante em Dispatcher.cpp.....61
Algoritmo 7: Versão definitiva do código para execução de ações ou retransmissão de mensagem via objeto Despachante em Dispatcher.cpp utilizando SWITCH-CASE.....62
Algoritmo 8: Código de três regras PON em ExoskeletonEventHandlerSystemPON.cpp.....72
Algoritmo 9: Código do método dispatch em Dispatcher.cpp.....76
Algoritmo 10: Código do método moveForward() em Exoskeleton.cpp (State Pattern).....77
Algoritmo 11: Código de declaração do método moveForward() em ExoskeletonState.h (State Pattern).....77
Algoritmo 12: Código do método moveForward em ExoskeletonONArmON.cpp (State Pattern).....77
Algoritmo 13: Código de quatro regras PON em ExoskeletonTwoArmEventHandlerSystemPON.cpp.....89
Algoritmo 14: Código do método dispatch() e notify() em Dispatcher.cpp.....94
Algoritmo 15: Código do método update() em Exoskeleton.cpp.....95
Algoritmo 16: Código do método update() em Arm.cpp.....96
Algoritmo 17: Código de quatro Rules em ElectronicGatePON.cpp.....108
Algoritmo 18: FBEs Gate e Event, e quatro Rules em código LingPON em electronicgate.pon 109
Algoritmo 19: Código para execução de ações via objeto Despachante em Dispatcher.cpp utilizando SWITCH-CASE do Simulador de Portão Eletrônico.....113

Algoritmo 20: Código do método update() em GateClosed.cpp (State Pattern).....116
Algoritmo 21: Código do método update() em GateClosing.cpp (State Pattern).....116
Algoritmo 22: Código do método update() em GateOpened.cpp (State Pattern).....116
Algoritmo 23: Código do método update() em GateOpening.cpp (State Pattern).....116
Algoritmo 24: Código do método update() em GateStopClosing.cpp (State Pattern).....116
Algoritmo 25: Código do método update() em GateStopOpening.cpp (State Pattern).....117
Algoritmo 26: Excerto de código para execução de ações ou retransmissão de mensagem via objeto Despachante em Gate.cpp (um ConcreteObserver) utilizando SWITCH-CASE do Simulador de Portão Eletrônico.....119
Algoritmo 27: Código em LingPON para demonstrar closures Methods, Subcondition e Premise (destacadas em negrito) em exoskeleton.pon.....143

Lista de Abreviações

	Original	Tradução
DSL	*Domain Specific Language*	Linguagem Específica de Domínio
EDP	*Event-driven Programming*	Programação Orientada a Eventos
FBE	*Fact Base Element*	Elemento da Base de Fatos
GUI	*Graphical User Interface*	Interface Gráfica de Usuário
IDE	*Integrated Development Environment*	Ambiente de Desenvolvimento Integrado
ISR	*Interrupt Service Routines*	Rotinas de Interrupção do Serviço
NOP	*Notification Oriented Paradigm*	Paradigma Orientado a Notificações
OO	Orientado a Objetos	*Object Oriented*
PD	Paradigma Declarativo	*Declarative Paradigm*
PFR	Paradigma Funcional Reativo	*Functional Reactive Programming*
PF	Paradigma Funcional	*Functional Paradigm*
PI	Paradigma Imperativo	*Imperative Paradigm*
POE	Paradigma Orientado a Eventos	*Event-driven Paradigm*
PON	Paradigma Orientado a Notificações	*Notification Oriented Paradigm*
POO	Programação Orientada a Objetos	*Object Oriented Programming*
TR	Tempo de Referência	*Reference Time*
TTR	Tempo Total de Referência	*Total Reference Time*
RAD	*Rapid Application Development*	Desenvolvimento Rápido de Aplicações
SBR	Sistema Baseado em Regras	*Rule-based System*

Capítulo 1 - Introdução

As técnicas de programação baseadas no estado da arte contemplam, entre outros e particularmente, o Paradigma Orientado a Objetos (POO) e os Sistemas Baseados em Regras (SBRs). Estes paradigmas podem ser genericamente classificados como Paradigma Imperativo (PI) e Paradigma Declarativo (PD) que englobam respectivamente o POO e os SBR [Banaszewski, 2009][Linhares *et al.*, 2011].

Ao seu turno, POE é o Paradigma Orientado a Eventos, que dita um modelo de construção de *software* que tem como base o tratamento de eventos. No contexto de *software*, um evento produzido instiga uma ação em uma unidade computacional tratadora de eventos, em momento de ocorrência imprevisível e definido somente em tempo de execução [Harel e Pnueli, 1985][Eugster *et al.*, 2003][Faison, 2006][Hansen e Fossum, 2010][Etzion e Niblett, 2011].

Outrossim, o Paradigma Orientado a Notificações, acrônimo PON, incorpora conceitos próprios de PI e PD, assim englobando características como a flexibilidade algorítmica e possibilidade de abstração da POO, bem como representação do conhecimento em regras dos SBRs. Além disso, sua organização estrutural é composta de unidades computacionais de pequeno porte, reativas, coesas e desacopladas, que se comunicam em um ciclo de notificações [Banaszewski, 2009][Simão *et al.*, 2012a, 2012b, 2012c].

Isto posto, trabalhos anteriores realizaram comparações, em termos qualitativos e de desempenho, entre a atual materialização do PON (implementado em um *Framework* otimizado, desenvolvido em linguagem C++ [Valença *et al.*, 2011]), com materializações puras de POO (Paradigma Orientado a Objetos) e Sistemas Baseados em Regras (SBR). Tais trabalhos apontaram evidências que as materializações do PON tem melhor desempenho que SBRs com avançadas e (industrialmente) impactantes máquinas de inferência (isso ocorre em determinadas condições). Também são especificamente melhores que POO quando há muitas relações causais e variáveis com baixa ou média variação de estados [Banaszewski, 2009] [Linhares *et al.*, 2011][Batista *et al.*, 2011][Ronszcka *et al.*, 2011][Valença *et al.*, 2011][Simão *et al.*, 2012a, 2012b, 2012c].

Por exemplo, quanto mais aprimorada se dá a materialização do PON (*e.g.* em termos

de otimização de estruturas de dados) melhores são os resultados [Valença *et al.*, 2011][Simão *et al.* 2012b]. Atualmente, há pesquisas em andamento visando o desenvolvimento de uma linguagem específica, e seu respectivo compilador, para o PON [Ronszcka, *et al.*, 2013]. Existem também iniciativas objetivando execução direta de programas PON em arquiteturas específicas de *hardware* [Peters, 2012], arquiteturas paralelas com distribuição de carga de *software* [Belmonte *et al.*, 2012] e até um processador especificamente projetado para executar de forma otimizada programas em PON [Linhares *et al.*, 2014].

Não obstante estes avanços, ainda não há registros de comparações elaboradas entre POE e PON. Justamente, este trabalho se propõe a comparar e diferenciar o Paradigma Orientado a Eventos (POE) e o Paradigma Orientado a Notificações (PON), considerando aspectos qualitativos e quantitativos.

Portanto, este trabalho pretende responder a seguinte pergunta de pesquisa: quais são as diferenças e semelhanças entre o Paradigma Orientado a Eventos (POE) e o Paradigma Orientado a Notificações (PON), considerando suas bases teóricas, características estruturantes, medidas de complexidade de código-fonte e medições durante execução de *software*?

As próximas seções deste capítulo introdutório são: Seção 1.1 que apresenta o problema de pesquisa relacionando o respectivo contexto; Seção 1.2 que detalha a motivação e justificativas para este estudo; Seção 1.3 que elenca o objetivo geral e objetivos específicos pretendidos com este trabalho; Seção 1.4 que descreve os métodos e ferramentas utilizadas; e por fim, a Seção 1.5 apresenta a organização dos capítulos subsequentes deste trabalho.

1.1 O Problema de Pesquisa: Comparação do Paradigma Orientado a Notificações (PON) com o Paradigma de Programação Orientado a Eventos (POE)

Pesquisas anteriores sobre o PON referenciaram POE [Banaszewski, 2009] [Ronszcka, 2012] [Valença, 2012] [Simão *et al.*, 2012a], porém não confrontaram diretamente POE e PON para determinar as diferenças, semelhanças ou relações conceituais entre ambos os paradigmas.

Da mesma maneira, pesquisas anteriores também conceberam *softwares* tratadores de eventos em PON, a saber, Portão Eletrônico [Wiecheteck, 2011][Wiecheteck, Stadzisz e

Simão, 2011] e Terminal Telefônico [Linhares *et al.*, 2011][Valença, 2012]. Isto é, podem (por aderência) também ser concebidos em técnicas inatas e clássicas de POE como *Handler-Dispatcher*, *State* e *Observer* (essas técnicas serão detalhadas no Capítulo 3). No entanto, a abordagem de análise (qualitativa e quantitativa) desses *softwares* não foi substancialmente realizada sob o ponto de vista de eventos em *software*. O objetivo deste projeto de pesquisa, portanto, é comparar programação de *software* tanto em PON quanto em POE, utilizando como ponto de partida *software* anteriormente concebido em PON, ou criando *software* inédito tanto em PON quanto em POE.

O principal intuito da pesquisa aqui proposta é discutir POE e PON em conjunto. Como fundamento, ambos podem ter inspirações semelhantes para sua criação como facilitar construção de *software*. No entanto, eles são díspares em suas características, como na forma de pensar e resolver problemas ao programar *software*. Além disso, PON intenciona resolver problemas de desenvolvimento como alto acoplamento, redundância de avaliações causais e respectivo desperdício em desempenho [Banaszewski, 2009][Ronszcka, 2012].

Critérios quantitativos para avaliação de desempenho (e redundâncias de avaliações) e qualitativos de comparação foram utilizados anteriormente, contudo, mesmo com todas as pesquisas e inovações do grupo de pesquisa em PON, pouca informação estruturada e evidências estão disponíveis para relacionar conceitualmente PON com outros paradigmas de programação (*e.g.* equivocadamente já se argumentou que PON é idêntico ao POE e até mesmo ao *Observer Pattern*). Esse problema se deve à falta de uma forma de avaliação conceitual, objetiva e de alto nível, para formular uma base estruturada comum para comparação, generalização e análise de conceitos de paradigmas de programação.

1.2 Justificativa

Aplicações profissionais têm crescido em número e complexidade e suas regras de negócio nos códigos-fonte podem ser perdidas, mal interpretadas e são difíceis de manter. Neste domínio de *software* profissional, a parte que trata eventos é tida como uma das principais culpadas pela quantidade de defeitos e problemas de desempenho [Samek, 2008] [Edwards, 2009][Bainomugisha *et al.*, 2013], reforçando essas difíceis características de organizar as regras de negócio implementadas no *software*.

Especificamente, na literatura se enfatiza que programar sistemas interativos

utilizando técnicas clássicas de tratamento de eventos, como o padrão de projetos (*design pattern*) *Observer* [Gamma *et al.*, 1995], é difícil e propenso a erros [Edwards, 2009][Maier e Odersky, 2012]. Contudo esta ainda é uma forma de implementação muito utilizada em ambientes de desenvolvimento empresariais. Por esses motivos (dificuldades em organizar regras em *software* e técnicas pouco satisfatórias), alternativas em POE são uma área de pesquisa ativa atualmente em programação [Salvaneschi e Mezini, 2014][Salvaneschi *et al.*, 2014b]. Além disto, também há outras iniciativas de disseminar o uso da orientação a eventos na programação, tanto em ambiente acadêmico quanto profissional, por meio da comunidade "Computação Baseada em Eventos"[1], pelo recentemente apresentado "Manifesto Reativo"[2], e no curso *on-line* sobre "Princípios da Programação Reativa"[3], disponível no ambiente de educação à distância *Coursera*.

Por outro lado, as escolhas de paradigma de programação (e respectivas linguagens) pelos programadores, projetistas e engenheiros de *software* são normalmente realizadas de forma *adhoc*, influenciadas por inércia cultural, tecnologias evangelizadas por fabricantes e forças de mercado. Nesse caso, informações estruturadas de comparação são convenientes, pois podem ser proveitosas como referência para tomadas de decisão em futuros projetos de programação profissionais. Também é intento deste trabalho permitir que a comparação aqui realizada auxilie futuras comparações com outros paradigmas e linguagens, do próprio grupo de pesquisa do PON.

Dessa forma, esta dissertação de mestrado se propõe a responder as seguintes perguntas:

- Como comparar PON com outros paradigmas de programação como o POE?
- Quais as diferenças e semelhanças entre PON e POE?
- PON é uma alternativa para tratamento de eventos em *software*?

1 Organização acadêmica e profissional da Computação Baseada em Eventos: http://event-based.org/

2 Manifesto Reativo disponível em: http://www.reactivemanifesto.org/

3 Primeira edição do curso (2013) disponível em: https://www.coursera.org/course/reactive

1.3 Objetivos

Este trabalho possui como objetivo principal:

- Comparar os paradigmas POE e PON segundo os critérios de suas características estruturais, medições de complexidade de código e medições durante execução, buscando detectar as vantagens e desvantagens relativas de cada paradigma em relação ao outro. Os propósitos são demonstrar o PON como uma alternativa para tratamento de eventos e propiciar uma maneira de comparar PON com POE.

Para atingir esse objetivo, esta pesquisa abrange os seguintes objetivos específicos:

- Classificar e articular as características estruturantes dos dois paradigmas de programação (POE e PON) conforme uma taxonomia comum;
- Comparar os programas gerados (*software)* para resolver problemas idênticos em ambos os paradigmas, utilizando critérios de complexidade de código-fonte como número de linhas, número de escopos ou *closures* (recipientes de escopo léxico) e número de *tokens* necessários (ver seção 1.4 – Métodos e Ferramentas – para maiores detalhes sobre estes critérios);
- Comparar medidas de execução – desempenho em termos de tempo de execução e tempo de resposta – em ambos os paradigmas resolvendo problemas idênticos.

1.4 Métodos e Ferramentas

A estratégia desta pesquisa foi teórico-prática, com comparações qualitativas e quantitativas. Dessa forma, os métodos utilizados no desenvolvimento deste trabalho formaram o seguinte conjunto realizado em cinco fases: Revisão Bibliográfica, Projeto de Dissertação de Mestrado, Casos de Estudo, Comparações e Redação. A seguir, essas fases são apresentadas.

1.4.1 Revisão Bibliográfica

Primeiramente, foram angariadas as referências bibliográficas, realizadas análises de trabalhos, publicações, pesquisas (*e.g.* dissertações, teses e artigos) e *softwares* feitos anteriormente nos dois paradigmas, e também realizada programação (*software*) para utilizar as técnicas e soluções de ambos os paradigmas (de maneira preliminar). Dessa forma, essa primeira fase consistiu em uma revisão da literatura com objetivo de compor um referencial teórico consistente no Paradigma Orientado a Notificações bem como no Paradigma Orientado a Eventos. Adicionalmente, foram descobertas as técnicas mais relevantes em POE (*Dispatcher*, *State Pattern* e *Observer*) e também foi realizado o primeiro Seminário de Qualificação "Revisão do Estado da Arte".

1.4.2 Projeto de Dissertação de Mestrado

A segunda fase, "Projeto de Dissertação de Mestrado", teve como objetivos ampliar a revisão bibliográfica (aprofundamento esse que incluiu a Programação Reativa), identificar trabalhos que compararam Paradigmas de Programação e/ou que compararam *software* programado em diferentes paradigmas, e, por fim, compor propriamente o projeto de pesquisa incluindo (em resumo) a proposta, objetivos, métodos, casos de estudo e comparações. Nesta fase foi possível analisar e definir a abordagem teórico-prática para realizar a comparação entre os paradigmas PON e POE neste trabalho de pesquisa, sendo composto basicamente dos seguintes alicerces: construção de casos de estudo, comparação estruturada (conceitual), comparação sob a ótica de programação de *software* (complexidade de código) e comparação de desempenho de *software* em execução (tempo total de execução e tempo de resposta). Por fim, também foi realizado o segundo Seminário de Qualificação "Projeto de Dissertação de Mestrado".

1.4.3 Casos de Estudo

A terceira fase consistiu na criação de casos de estudo para posterior uso e avaliação das comparações escolhidas (estruturante, complexidade de código e desempenho). Os casos de estudo desenvolvidos são referentes a *softwares* que tratam eventos simulados (denominados *Simulador de Transporte Individual* e *Simulador de Portão Eletrônico*). Nesta fase foram modelados esses casos de estudo e realizadas as respectivas implementações em *Framework* PON, LingPON e PON-Compilador e, por fim, POE implementado em C++ em três diferentes técnicas.

Convém explicar a tática de escolha dos experimentos que tratam eventos (e respectivos casos de estudo). A saber, em resumo, eventos podem ser externos e internos (em relação ao *software*); discretos (ocorrem em pontos separados no tempo) ou contínuos (ação ininterrupta no sistema, também chamados de comportamentos); de recebimento direto (origem diretamente conectada com o destino) ou recebimento indireto (*e.g.* sistema de *middleware* que trata a distribuição de eventos, *peer-to-peer*); temporais (marcação discreta ou tempo transcorrendo *per se*) [Bainomugisha *et al.*, 2013]; de emissão ou recebimento em paralelo ou concomitante; síncronos ou assíncronos (os primeiros bloqueiam o emissor entre emissão e respectiva ação no recebedor, os seguintes não); de *hardware*, *software*, ou especializações de Sistemas de Informação; de dados (*e.g.* inserção, exclusão, atualização); para distribuição; para relacionamento entre outros eventos (membro e categorização, generalização e especialização) [Faison, 2006]; para identificação de padrões, filtragem e descarte; para transformação; para confirmação de recebimento de outros eventos; para responder o término de uma execução; para tratamento em barramento compartilhado entre sistemas; para disposição em filas ou *buffers;* complexos (*i.e.* organização complexa para regras de tratamento de múltiplos eventos esparsos no tempo como exemplo clássico uma "Praça de Pedágio"); eventos ocorrendo em "tempestade", "rajadas" e com "sazonalidade" [Etzion e Niblet, 2011]. Pode-se perceber que há uma grande variedade de classificações e respectivos modelos de estudo de eventos, entretanto para este trabalho científico se teve como premissas isolar ao máximo os experimentos de fatores externos, simplificar os tipos de eventos que podem ocorrer, de forma a poder comparar as técnicas pontualmente, e buscar uma visão na atividade de programação. Desta forma, os objetivos ao compor e escolher os experimentos foram, em uma escala gradativa: avaliar o tratamento de eventos discretos para tomada de decisão em *software*, avaliar o tratamento de mesmos tipos de eventos recebidos

em estados diferentes de um *software*, avaliar o tratamento de eventos que executa mais de uma ação ao mesmo tempo, e avaliar *software* que trate eventos externos e internos (*i.e.* sendo o evento interno a marcação discreta de tempo). Poderiam ser implementados pelo menos um *software* em cada técnica de tratamento de eventos, e igualmente pelo menos um *software* para cada tipificação de evento encontrada, porém o trabalho seria impraticável. Dessa forma, por convenção, os casos de estudo escolhidos assim o foram por cumprir esses citados requisitos de avaliação de tratamento de eventos.

Uma complicação adicional aos objetivos deste trabalho foi o fato de que a programação no Paradigma Orientado a Eventos, mesmo se tratando de um único paradigma, possui diferentes maneiras de materialização, com diferentes técnicas para sua programação (conforme discussão da seção 2.5 Reflexões sobre a Revisão do Estado da Arte). Assim, as técnicas de programação em POE que foram avaliadas são: *Handler-Dispatcher* (*event loop*), Máquina de Estados (*i.e.* em implementação via *State Pattern*) e *Observer Pattern* (*callback programming*) [Gamma *et al.*, 1995][Ferg, 2006][Faison, 2006][Samek. 2008][Hansen e Fossum, 2010][McKellar, 2012]. Estas técnicas são apresentadas em maiores detalhes na seção 2.4 Técnicas do Paradigma Orientado a Eventos do capítulo 2. Em PON, a codificação foi desenvolvida utilizando a metodologia DON – Desenvolvimento Orientado a Notificações [Wiecheteck, 2011][Wiecheteck, Stadzisz e Simão, 2011], descrita brevemente na seção 2.2 Paradigma Orientado a Notificações, e posteriormente implementada em duas materializações: no *Framework* C++ otimizado, como proposto por Valença [Valença *et al.*, 2011] e posteriormente Ronszcka [Ronszcka, 2012], e em um compilador específico para o PON – denominado neste trabalho preliminarmente de PON-Compilador, proposto por [Ferreira *et al.*, 2013].

Portanto, o método utilizado neste trabalho se apoia e é composto pela implementação de dois casos de estudo utilizando três variações de técnicas de programação em POE, e em PON (duas diferentes materializações *Framework* e LingPON/PON-Compilador), seguida de avaliações sobre as implementações realizadas (fase seguinte do método deste trabalho). Também, a realização dos *softwares,* experimentos e medições, utilizando ao máximo técnicas isoladamente, almejou diminuir influências e ruídos (*bias*), e respectivos problemas de medição, que geralmente ocorrem em casos de estudo ao misturar muitas técnicas.

No primeiro caso de estudo, o foco foi comparar de maneira pontual as técnicas de construção de *software* tanto em POE quanto em PON. Esse caso de estudo engloba um *software* tratador de eventos, utilizando como conceito exemplo um simulador de movimentação de um exosqueleto e seu respectivo braço mecânico (sendo, portanto, um *toy*

problem, pois é implementada apenas a parte de tratamento de eventos do *software*, e não o *hardware* e *software* completo que executaria em conjunto com esta parte de *software*). Esta implementação foi dividida em três cenários: o primeiro para utilizar a técnica POE *Handler-Dispatcher* (*event loop*), o segundo para utilizar a técnica Máquina de Estados (*State*), e o terceiro e último cenário para uso de *Observer Pattern*.

No segundo caso de estudo foi revisitado um caso estudado em pesquisas anteriores em PON (Portão Eletrônico) [Wiecheteck, 2011][Wiecheteck, Stadzisz e Simão, 2011] [Batista, 2013]. Trata-se de um exemplo claro de um *software* que trata eventos mudando de estado, sendo um programa que já foi concebido originalmente em PON. O objetivo foi justamente confrontar um *software* em POE com um *software* cuja formulação do problema, modelagem e construção foram natos em PON e a principal diferença em relação ao primeiro caso de estudo refere-se justamente à mudança de estado do próprio *software*.

1.4.4 Comparações

Na quarta fase, foi possível analisar e definir, com base nas fases anteriores e principalmente em relação aos casos de estudo realizados na fase anterior, os elementos necessários para a comparação entre ambos os paradigmas de programação PON e POE. Assim, foram realizadas primordialmente três comparações: estruturante utilizando uma taxonomia comum (abordagem conceitual), complexidade de código (abordagem de medições dos artefatos de código-fonte da programação de *software*) e medições de *software* em execução (abordagem desempenho de *software*). Por convenção, a escolha desses tipos de comparação utilizou principalmente o critério da comparação de linguagens e programação de *software* em paradigmas diferentes.

Primeiramente, em abordagem coexistente (realizada praticamente durante todo este trabalho de pesquisa de maneira incremental), foram realizadas as comparações das características estruturantes inerentes aos dois paradigmas (comparação qualitativa, empírica). Foram utilizadas as definições de registro, *closure* (recipientes de escopo léxico), independência, *named state* (estado nomeado), não-determinismo observável e conceitos singulares, como proposto por [Van Roy, 2009], redefinindo a taxonomia de paradigmas para incluir o PON. A apresentação detalhada destes conceitos é realizada na seção 2.1 Paradigmas de Programação e os resultados são apresentados na seção 4.1 Comparação de características estruturantes.

Sob a ótica de programação de *software* (segundo tipo de comparação), essa pesquisa utilizou os seguintes critérios quantitativos de complexidade de código: medidas de número de linhas de código (*LOC-Lines of Code*), número de recipientes com escopo léxico (*closures*) e número de *tokens*. Um recipiente de escopo léxico (*closure*) é associado com cada diferente escopo utilizado em uma linguagem de programação, como os blocos de programação (por exemplo, o bloco de comandos de uma estrutura de repetição, ou o escopo de variáveis locais de um método ou procedimento). Já um *token* é uma unidade sintática de uma linguagem de programação, reconhecida pelo compilador, que pode ser um identificador de variáveis ou nomes de funções ou procedimentos, uma palavra reservada (*for*, *if*, *else*, em linguagem C, por exemplo), ou outros elementos que compõe a gramática da linguagem. A seleção destes critérios de comparação foi baseada em outros projetos que também abordam práticas análogas para comparação de código entre linguagens e paradigmas diferentes [Dunkels *et al.*, 2006][Sant'Anna *et al.*, 2013][Salvaneschi *et al.*, 2014b].

Na terceira e última abordagem de comparação com foco em desempenho (medição durante execução de *software*), para obter dados que permitissem obter devidas conclusões para esta pesquisa, foram realizados experimentos simulando a geração de eventos (externos e internos) para cada caso de estudo de *software*, objetivando obter comparações quantitativas relacionadas a tempo de execução (tempo de resposta e tempo total de execução). A principal motivação foi obter dados que permitissem a comparação de desempenho de *software* que trate eventos, tanto em PON quanto em POE.

Para ambos os paradigmas e todas as soluções construídas, as medições quantitativas dos experimentos foram captadas durante a execução de *software* como tempo total de execução [Brennan *et al.*, 2010] (ambos os casos de estudo), tempo de resposta (para o primeiro caso de estudo) e executar completamente e em conformidade um roteiro de testes (para o segundo caso de estudo).

Por fim, também foi realizado o terceiro e último Seminário de Qualificação "Resultados Preliminares" e também publicado artigo científico em periódico com Resultados Preliminares em [Xavier, 2014].

1.4.5 Fase de Conclusão

Findando este trabalho de pesquisa, a quinta fase denominada Conclusão, se realizou a redação do documento de dissertação, respectiva defesa do presente trabalho e posteriormente

as devidas correções textuais e gráficas neste documento finalizado que se apresenta ao leitor.

1.5 Organização do trabalho

Este trabalho possui cinco capítulos. Após esta "Introdução" (Capítulo 1), o Capítulo 2 ("Revisão do Estado da Arte") serve para apresentar os conceitos básicos sobre POE e PON no contexto desse trabalho. Logo em seguida, o Capítulo 3 ("Casos de Estudo") apresenta três cenários de um caso de estudo denominado "Simulação de Transporte Individual", desenvolvido especificamente para este trabalho, e outro caso de estudo denominado "Portão Eletrônico", no qual uma implementação em PON já estava disponível, tendo sido projetadas e implementadas as três versões do código POE; a seguir, no Capítulo 4 (Comparações PON versus POE), são articuladas as características conceituais de ambos os paradigmas em uma taxonomia comum, sendo também enumerados dados sobre complexidade de código-fonte e mostrados dados obtidos em medições durante execução (*e.g.* tempo total de execução e tempo de resposta). É também apresentada neste capítulo uma discussão dos resultados de comparação. Por fim a "Conclusão" (Capítulo 5) encerra este trabalho, elencando suas contribuições, e indicando possibilidades de trabalhos futuros.

Capítulo 2 - Revisão do Estado da Arte

Neste capítulo são apresentadas as bases teóricas sobre as quais se desenvolve o restante deste trabalho. Em primeiro lugar, relacionam-se genericamente os paradigmas de programação com algumas de suas vertentes (*e.g.* Paradigma Imperativo e Paradigma Declarativo). Em seguida, o Paradigma Orientado a Notificações é descrito com suas particularidades e pesquisas em andamento. Após, o Paradigma Orientado a Eventos é exposto com seus atributos, respectivo estado da arte e suas principais técnicas. Findando o capítulo, são apresentadas algumas reflexões sobre o estado da arte desta pesquisa.

2.1 Paradigmas de Programação

A utilização de técnicas de PI (programação imperativa), particularmente a POO (programação orientada a objetos), costuma atrair os desenvolvedores devido a questões como inércia cultural, riqueza de abstração e flexibilidades algorítmicas. Em PI, em suma, concebem-se programas como sequências de instruções utilizando-se buscas sobre entidades passivas (dados e comandos) organizadas segundo uma lógica de execução que envolve, inclusive, a avaliação de expressões causais (*e.g.* se-então) [Banaszewski *et al.*, 2007] [Brookshear, 2012][Gabbrielli e Martini, 2010][Linhares *et al.*, 2011][Simão *et al.* 2012a].

Particularmente, estas expressões são frequentemente avaliadas repetidamente sem necessidade, degradando o desempenho computacional. Isto pode ser exemplificado com um conjunto de expressões se-então (*if-then*) que avaliam os estados de objetos em um laço de repetição ‘infinito’. Cada expressão condicional avalia estados de atributos de objetos e, se aprovada, chama alguns métodos destes, que podem mudar os estados dos atributos. [Brookshear, 2012][Gabbrielli e Martini, 2010][Simão e Stadzisz, 2008, 2009][Simão *et al.* 2012a, 2012b, 2012c]. Esta situação é apresentada no Algoritmo 1 [Linhares *et al.*, 2011].

Algoritmo 1: Pseudocódigo do Paradigma Imperativo

```
1 enquanto ( verdade ) faça
2   se ( ( objeto1.atributo1 = 1 ) e ( objeto2.atributo1 = 1 ) ) então
3   objeto1.método1(); objeto2.método2();
4   fim-se
5   . . .
6   se ( ( objeto1.atributoN = N ) e ( objeto2.atributoN = N ) ) então
7   objeto1.métodoN(); objeto2.métodoN();
8   fim-se
9 fim-enquanto
```

Observa-se no exemplo que o laço de repetição força sequencialmente a avaliação (inferência) de todas as condições. Entretanto, muitas destas são desnecessárias porque somente alguns objetos têm o valor de atributo modificado. Isto pode ser considerado pouco importante neste exemplo simples e pedagógico, sobretudo se o número N de expressões causais for pequeno. Contudo, se considerado um sistema complexo, com várias partes como aquela, pode-se ter grande diferença de desempenho [Simão e Stadzisz, 2008][Linhares *et al.*, 2011][Simão *et al.*, 2012a].

Por sua vez, no PD (Paradigma Declarativo), enfatizando os SBR (Sistemas Baseados em Regras), existe a vantagem da programação em alto nível. Primeiro se define uma Base de Fatos composta por entidades como objetos com atributos e métodos. Depois se define a 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, que compara regras e fatos (*e.g.* estados de atributos) gerando novos fatos e, portanto, um ciclo de inferência. Não obstante a organização e algoritmos eficientes de inferência, a programação em PD normalmente é computacionalmente cara em termos de estruturas de dados processadas [Scott, 2008][Simão e Stadzisz, 2008, 2009][Linhares *et al.*, 2011][Simão *et al.* 2012a, 2012b, 2012c].

Entretanto, em uma análise mais profunda, PI e PD são similares no tocante à inferência que normalmente se dá por entidades monolíticas baseadas em pesquisas sobre entidades (quase) passivas que conduzem a programas com passos de execução interdependentes. Essas características contribuem para a existência de sobreprocessamento e forte acoplamento entre expressões causais e estrutura de fatos/dados, o que dificulta a execução dos programas de maneira otimizada, bem como paralela ou distribuída [Gabbrielli e Martini, 2010][Simão e Stadzisz, 2008, 2009][Linhares *et al.*, 2011][Simão *et al.* 2012a, 2012b, 2012c].

Ainda, outras abordagens como Programação Orientada a Eventos (POE) e

Programação Funcional (PF) não resolvem esses problemas, mesmo reduzindo alguns deles como evitar certas redundâncias [Scott, 2008][Brookshear, 2012][Simão *et al.* 2012a, 2012b]. Isto ocorre visto que o algoritmo em cada programa, módulo, função ou procedimento é codificado utilizando PD ou PI, deste modo implicando nas deficiências outrora delineadas como redundância de código e acoplamento, mesmo considerando que elas são atenuadas por eventos ou chamadas de funções [Simão *et al.* 2012a, 2012b]. A abordagem POE será discutida detalhadamente no final desta seção.

O Paradigma Funcional, por vezes considerado um subtipo do PD, é assim chamado visto que sua operação fundamental se utiliza de funções como argumentos. Um programa é construído como uma função principal, que recebe argumentos de entrada (*e.g.* funções) e resulta em uma saída. A função principal é tipicamente definida em termos de outras funções, que também são definidas em termos de outras funções, dessa forma até que as funções elementares sejam as primitivas da linguagem de programação. Recursão é uma característica elementar (e recorrente) no PF. [Hughes, 1989][Van Roy e Harefidi, 2004][Scott, 2008] [Brookshear, 2012][Simão *et al.*, 2012a].

Isto posto, para definir paradigma de programação são citados David Watt [Watt, 2004] e Peter Van Roy [Van Roy e Haridi, 2004][Van Roy, 2009]. O primeiro explica que um paradigma de programação consiste na seleção de um conjunto de conceitos chave da programação, como tipos de dados, variáveis, escopo, abstração, concorrência e controle, utilizados em sinergia para formar um estilo de programação [Watt, 2004][Banaszewski, 2009].

O segundo, Peter Van Roy, define que um paradigma de programação é um sistema formal que define como a programação é realizada. Ademais, cada paradigma tem o seu próprio conjunto de técnicas (e conceitos) para programação, e assim possui sua respectiva forma de estruturar o pensamento na construção dos programas [Van Roy e Haridi, 2004][Van Roy, 2009][Banaszewski, 2009][Hansen e Fossum, 2010].

Peter Van Roy, a partir do seu trabalho em [Van Roy e Haridi, 2004], desenvolveu uma taxonomia para paradigmas de programação em [Van Roy, 2009] (Figura 1: Taxonomia de paradigmas de programação [Van Roy, 2009]). O autor estabelece quatro principais conceitos das linguagens de programação, a saber: registros (*record*), recipientes com escopo léxico (*closures*), independência (concorrência) e estado nomeado (*named state*).

Em resumo, registro (*record*) é uma estrutura de dados, isto é, “um grupo de referências para itens de dados com acesso indexado para cada item”. Esse conjunto de referências é utilizado para compor estruturas de informações. Exemplificando, uma *struct* em

linguagem C fornece acesso a determinado item utilizando o identificador da estrutura, mais um operador (.), e o identificador de item (*e.g. Pessoa.Nome* e *Exoskeleton.atExoskeletonState*). Estruturas de dados importantes são derivadas de registros, como exemplo *arrays*, listas, *strings*, árvores e tabelas *hash* [Van Roy, 2009].

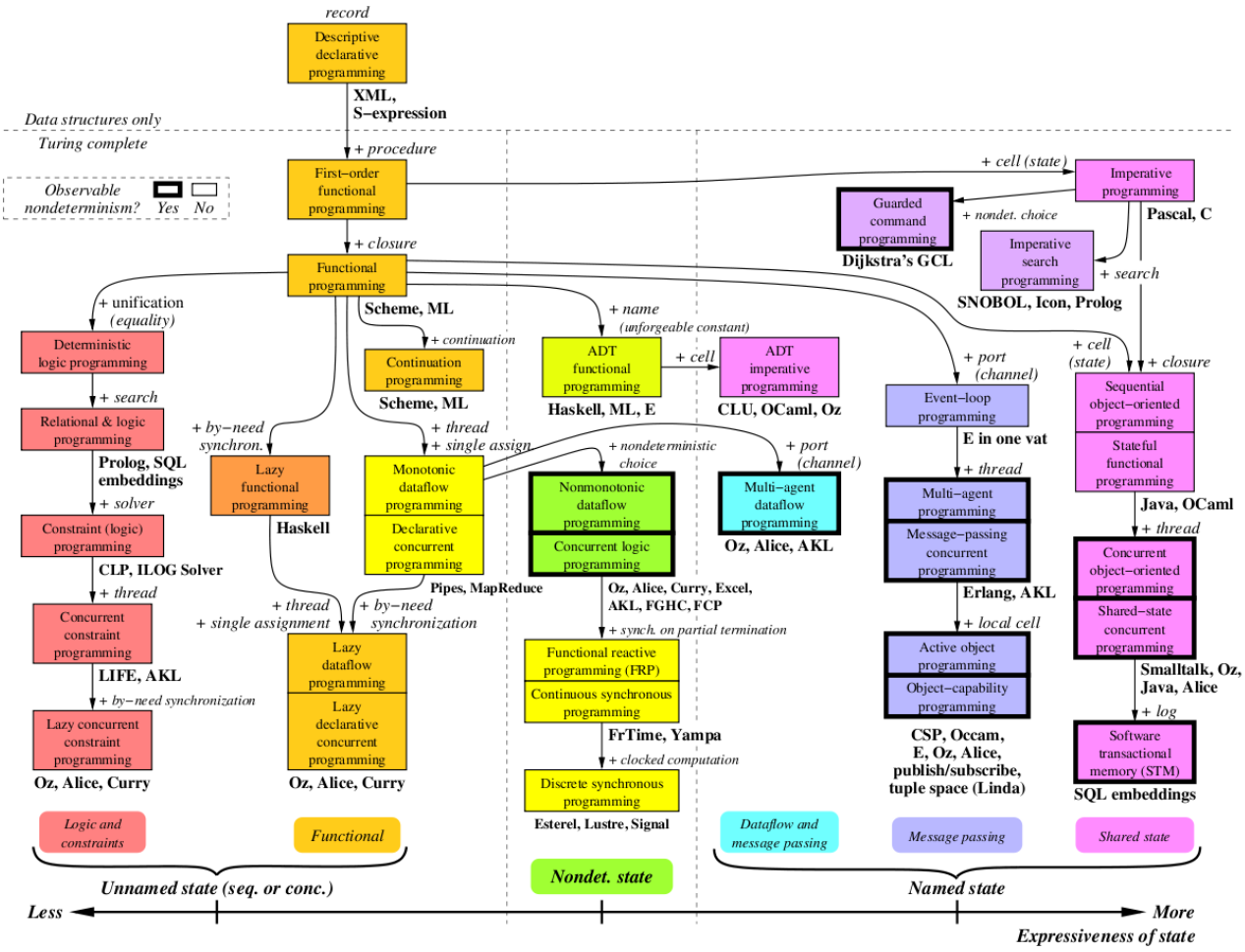

Figura 1: Taxonomia de paradigmas de programação [Van Roy, 2009]

Recipientes com escopo léxico (*closures*) são análogos à soma do corpo do procedimento (*procedure*) com a declaração do procedimento (*i.e.* procedimentos, objetos, funções, classes, componentes de *software* são *closures*). Igualmente, a separação entre a declaração e a execução de uma instrução viabiliza criar blocos e estruturas de controle como SE (*if*), e ciclos ENQUANTO (*while*). Em suma, na programação, *closures* separa os escopos internos e externos de seus respectivos recipientes [Van Roy, 2009].

Já independência é a capacidade de construir *software* em partes independentes, dando suporte à concorrência (*e.g.* execução simultânea). Quando existe uma ordem especificada de execução entre duas partes, elas são sequenciais. Partes concorrentes, que executam

simultaneamente ou alternadamente sem ordem específica, podem ser implementadas para conter interação (comunicação) [Van Roy, 2009].

O quarto conceito é estado nomeado (*named state*), que representa a capacidade de armazenar informação em um componente computacional com nome (*i.e.* com uma denominação desse componente), por exemplo, para guardar uma sequência de valores durante a execução (*i.e.* transcorrer do tempo). Ou seja, uma variável, de nome único, que registra e mantém diferentes dados durante um período de execução. Esta entidade é implementada usando uma memória interna nos componentes (*cell*). Este conceito estabelece, de forma ampla, nomes como variáveis de programas, endereços de memória, tabelas ou registros em banco de dados, arquivos em sistemas de arquivos. Em *software*, por meio do conceito de estado nomeado, se introduz uma abstração de noção de tempo nos programas. Segundo o autor, é possível criar uma entidade com uma identidade (nome) cujo comportamento muda conforme a execução do programa. Sem estado nomeado (*e.g.* variáveis) é difícil modelar problemas que precisam manter estado. Logo, um nome estável durante o tempo, que pode ser utilizado para contar eventos ou acumular observações durante esse mesmo tempo, é o estado nomeado.

Esse conceito contrasta com o Paradigma Funcional, pois quando uma função é chamada com os mesmos argumentos, a resposta é sempre a mesma, ou seja, "funções não mudam" [Hughes, 1989][Van Roy, 2009]. Um exemplo de paradigma com menor expressividade de estado é o próprio Paradigma Funcional (*i.e.* não há estado nomeado). Exemplos de paradigmas com maior expressividade de estado nomeado (*i.e.* estado compartilhado) são os Paradigmas Imperativo (*e.g.* variável global) e Orientado a Objetos (*e.g.* variável de classe) [Van Roy, 2009].

O eixo horizontal da taxonomia (segundo a Figura 1) gradua os paradigmas conforme a expressividade do estado nomeado, iniciando na esquerda indicando menor expressão, e quanto mais à direita se visualiza a figura, maior é a expressividade [Van Roy, 2009]. O eixo vertical da taxonomia gradua os paradigmas de cima para baixo, conforme suas características, iniciando com as características básicas (*record* e *closure*), e incluindo mais características específicas de cada paradigma.

Ainda na Figura 1, destacam-se graficamente os paradigmas que possuem completeza de Turing (*i.e.* abaixo da linha pontilhada), em contraste com o modelo elementar de estruturas de dados, diagramado acima da mesma linha (*e.g. records* como XML, *S-expression*) [Van Roy, 2009]. Conceitualmente, um programa construído em um paradigma Turing-completo equivale em capacidade computacional à máquina de Turing (*i.e.* resolve

qualquer problema computável) [Van Roy e Haridi, 2004]. A máquina de Turing é um modelo formal de computação (*i.e.* matemático), capaz de responder um problema computável havendo tempo e espaço (memória) suficientes [Turing, 1936].

Além disso, não-determinismo observável também está diagramado na representação dos paradigmas. Nesse caso, indicado por uma borda espessa caso não-determinismo observável seja possível ou borda fina caso inexistente. Segundo David Watt [Watt, 2004] e Van Roy [Van Roy e Haridi, 2004] primordialmente não-determinismo caracteriza-se pela escolha do que executar em seguida. Contrapõe-se ao determinismo, o qual sempre elenca caminhos definidos de execução. A característica do não-determinismo observável ocorre naturalmente quando há concorrência já que atividades concorrentes são independentes (*e.g. threads*), tanto que a especificação do programa não determina qual atividade executar em primeiro lugar. Isto é apresentado no Algoritmo 2.

Algoritmo 2: Pseudocódigo elementar de não-determinismo observável adaptado de [Van Roy e Haridi, 2004].

```
1 numero := 0
2   thread T1 faça
3   numero := 1
4   fim
5   thread T2 faça
6   numero := 2
7   fim
8 imprima numero
```

Não-determinismo observável ocorre quando o resultado da execução não-determinística do programa produz resultados diferentes em diferentes execuções. O tratamento do não-determinismo observável pelo programador é complexo, pois pode ser necessário verificar a garantia de exclusão mútua ou sincronização de acesso a regiões críticas, por exemplo, para acesso a periféricos ou arquivos. Não-determinismo observável também pode ser denominado condição de concorrência (*i.e.* condição de corrida – *race condition*) [Van Roy, 2009].

A Tabela 1 apresenta alguns paradigmas de programação segundo a taxonomia de Van Roy, identificando os principais conceitos associados a cada paradigma, a saber: (I) registro (*record*); (II) recipiente com escopo léxico (*closure*); (III) independência (concorrência), (IV) estado nomeado (*named state*), (V) não-determinismo observável. Para cada paradigma, são

também apresentados exemplos de linguagens de programação que o implementam.

Tabela 1: Paradigmas/conceitos [adaptado de VAN ROY, 2009] (S)IM/(N)ÃO.

Conceito **Paradigma**	**I**	**II**	**III**	**IV**	**V**	**Exemplo de linguagens de programação**	**Características (+)**
Lógico e Relacional	S	S	S	N	N	Prolog, SQL embeddings	*+unification(equality)* *+search*
Funcional	S	S	N	N	N	Scheme, ML	
Funcional Reativo	S	S	S	N	N	Fran, FrTime	*+thread* *+single assign* *+nondeterministic choice* *+sync. on partial termination*
Síncrono Discreto	S	S	S	N	N	Esterel, Lustre, Signal	*+thread* *+single assign* *+nondeterministic choice* *+sync. on partial termination* *+clocked computation*
Ciclo de eventos (*event loop*)	S	S	N	S	N	E in one vat	*+port(channel)*
Objeto ativo / Objeto apto	S	S	S	S	S	*Publish-Subcribe,* E	*+port(channel)* *+thread* *+local cell*
OO – Sequencial	S	S	N	S	N	Java, Ocaml	*+cell(state)*
OO – Concorrente	S	S	S	S	S	Smalltalk, Java	*+cell(state)* *+thread*

Neste âmbito, alguns dos paradigmas já citados são listados a seguir e classificados segundo a taxonomia de Van Roy. De menor expressão de estado, Programação Lógica e Relacional (coluna de cor vermelha na Figura 1) é “Turing completa”, apresenta ausência de não-determinismo observável, contendo os conceitos de registro (*record*), recipientes com escopo léxico (*closure*), igualdade (*unification*) e busca (*search*). Como exemplo, as linguagens de programação seriam Prolog e SQL embeddings.

Programação Funcional (coluna cor laranja na Figura 1) é “Turing completa”, apresenta ausência de não-determinismo observável, contendo os conceitos de registro (*record*) e recipientes com escopo léxico (*closure*). Exemplos de linguagem de programação seriam Scheme e ML.

O paradigma “Event-loop Programming” é apresentado na cor roxa com borda fina (Figura 1), sendo “Turing completo” e apresentando ausência de não-determinismo observável. Contém os conceitos de registro (*record*), recipientes com escopo léxico (*closure*) e porta (*port* ou *channel*). Em programação, porta é uma referência de conexão lógica. Porta e canal são conceitos semelhantes, que permitem a troca de mensagens, sendo que canal é análogo a barramento e porta é análogo à entrada. [Van Roy, 2009]. A taxonomia elenca

também um exemplo de linguagem em "*Event-loop Programming*" - E[4] in one vat (i. e. linguagem E em um único processo) [Miller *et al.*, 2005].

De maior expressão de estado, Programação Orientada a Objetos Concorrente (coluna cor rosa na Figura 1) é "Turing completa", apresenta não-determinismo observável e os conceitos de registro (*record*), recipientes com escopo léxico (*closure*), célula (*cell* – estado nomeado compartilhado) e linhas de execução concorrente (*threads*). Exemplos seriam as linguagens Java e Smalltalk.

2.2 Paradigma Orientado a Notificações.

O PON encontra inspirações no PI, como a flexibilidade algorítmica e a abstração em forma de classes/objetos da POO. O PON também aproveita conceitos próprios do PD, como facilidade de programação em alto nível e a representação do conhecimento em regras dos SBR. Assim, o PON permite o uso (de parte) de ambos os estilos de programação em seu modelo, ainda que os evolua e mesmo os revolucione no tocante ao processo de inferência ou cálculo lógico-causal [Simão e Stadzisz, 2008, 2009][Banaszewski, 2009][Linhares *et al.*, 2011][Simão *et al.* 2012a].

O PON apresenta resposta a problemas destes paradigmas, como 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. Justamente, o PON apresenta outra forma de realizar tais avaliações ou inferências via entidades computacionais de pequeno porte, ativas e desacopladas que colaboram por meio de notificações pontuais e são criadas a partir do 'conhecimento' de regras [Linhares *et al.*, 2011][Simão *et al.* 2012a, 2012b, 2012c].

Mais precisamente, o PON possui objetos que tratam dos elementos da base de fatos, que são genericamente modelados pela classe *FBE* (*Fact Base Element*), conforme a Figura 2. Cada *FBE* trata de seus atributos por meio de objetos da classe *Attribute* e seus serviços por meio de objetos da classe *Method*. Os objetos *FBE*, por meio de seus *Attributes* e *Methods*, são passíveis de correlação causal por meio de *Rules*, as quais se constituem em elementos fundamentais do PON [Simão e Stadzisz, 2008][Linhares *et al.*, 2011][Simão *et al.* 2012a, 2012b, 2012c].

---

4 Linguagem de programação *E* disponível em http://erights.org/

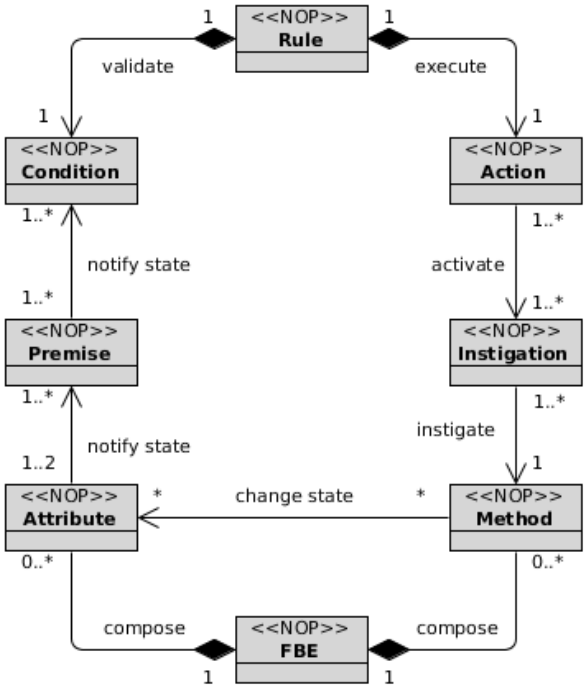

Figura 2: Entidades PON [Simão *et al.,* 2012a].

A Figura 3 apresenta um exemplo de Rule, justamente na forma de uma regra causal. Na verdade, a *Rule* é uma entidade computacional composta por outras entidades, conforme a Figura 2, as quais podem ser vistas como objetos. Por exemplo, a *Rule* apresentada é composta por um objeto *Condition* e um objeto *Action*. A *Condition* trata da decisão da *Rule*, enquanto a *Action* trata da execução das ações da *Rule*. Assim sendo, *Condition* e *Action* trabalham juntas para realizar o conhecimento lógico e causal da *Rule* [Simão e Stadzisz, 2008. 2009][Linhares *et al.*, 2011][Simão *et al.* 2012a, 2012b, 2012c].

A *Rule rlMoveForwardExoskleton* apresentada na Figura 3 faz parte do caso de estudo detalhado na seção 3.1 Caso de Estudo 1: Simulador de Transporte Individual. A *Condition* da *Rule* refere-se à decisão sobre a interação entre um *FBE* '*Event*' e um *FBE* '*Exoskeleton*'. Esta *Condition* tem dois objetos *Premises*. Estes realizam as seguintes verificações no tocante aos *FBEs*: a) o exosqueleto está ligado? b) o evento é *Joystick* para cima? Assim, conclui-se (em geral) que os estados de *Attributes* de *FBEs* compõem os fatos avaliados pelas *Premises* [Simão e Stadzisz, 2008][Xavier *et al.*, 2014].

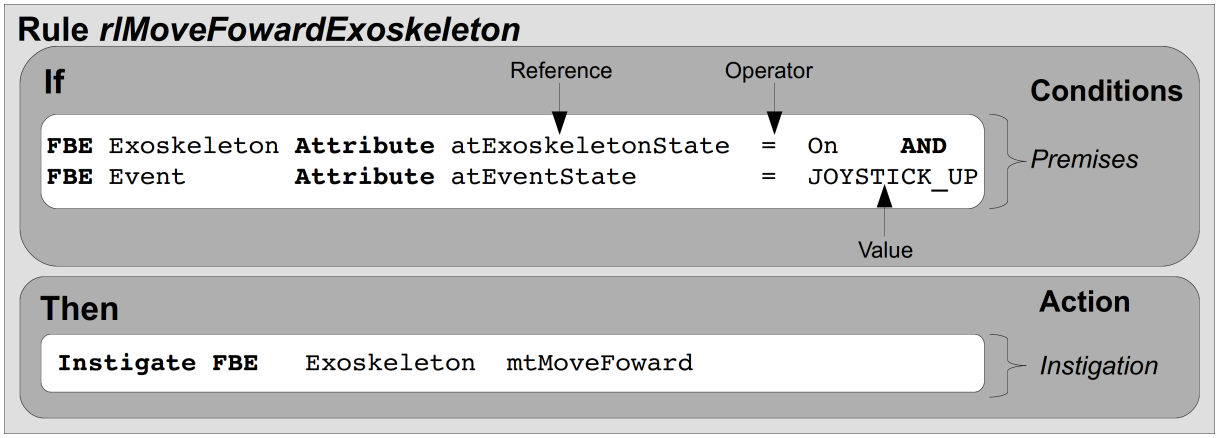

Figura 3: Exemplo de *Rule* [adaptado de Simão e Stadzisz, 2008, 2009].

Cada *Premise* avalia o estado de um ou dois *Attributes* de *FBE*, sendo que para cada mudança de estado de *Attribute* da *FBE*, ocorrem automaticamente avaliações (lógicas) somente nas *Premises* relacionadas com eventuais mudanças nos seus estados. Igualmente, a partir da mudança de estado das *Premises*, ocorrem automaticamente avaliações (causais) somente nas *Conditions* relacionadas com eventuais mudanças de seus estados. Isto se dá por uma cadeia de notificações entre objetos inteligentes, o que se constitui no fundamento do PON. [Simão e Stadzisz, 2008][Linhares *et al.*, 2011][Simão *et al.* 2012a].

Cada *Attribute* notifica somente as *Premises* relevantes sobre seus estados apenas quando necessário. Cada *Premise* similarmente notifica as *Conditions* relevantes dos seus estados. Baseado nestes estados notificados cada *Condition* pode ser aprovada. Se sim, a respectiva *Rule* pode ativar sua *Action*. Ainda, o conhecimento de quem se deve notificar se dá na composição das *Rules* [Banaszewski, 2009][Simão e Stadzisz, 2008, 2009][Linhares *et al.*, 2011][Simão *et al.* 2012a].

Por sua vez, uma *Action* também é um objeto que se conecta a outros objetos, as *Instigations*. No exemplo, a *Action* contém uma Instigation que instiga o exosqueleto a andar para frente [Xavier *et al.*, 2014]. Cada *Instigation* instiga um ou mais métodos para realizarem serviços de um *FBE*. Ainda, cada método de *FBE* é tratado por um *Method* cuja execução geralmente muda o estado de um ou mais *Attributes*. Estes conceitos de *Attribute* e *Method* seriam uma evolução dos conceitos similares da OO. A diferença é o desacoplamento explícito da classe proprietária e a colaboração pontual para com *Premises* e *Instigations* [Simão e Stadzisz, 2008][Linhares *et al.*, 2011][Simão *et al.* 2012a]. Graficamente, o

funcionamento se dá conforme modelado no Ciclo de Notificações na Figura 4[5] [Simão *et al.*, 2014]

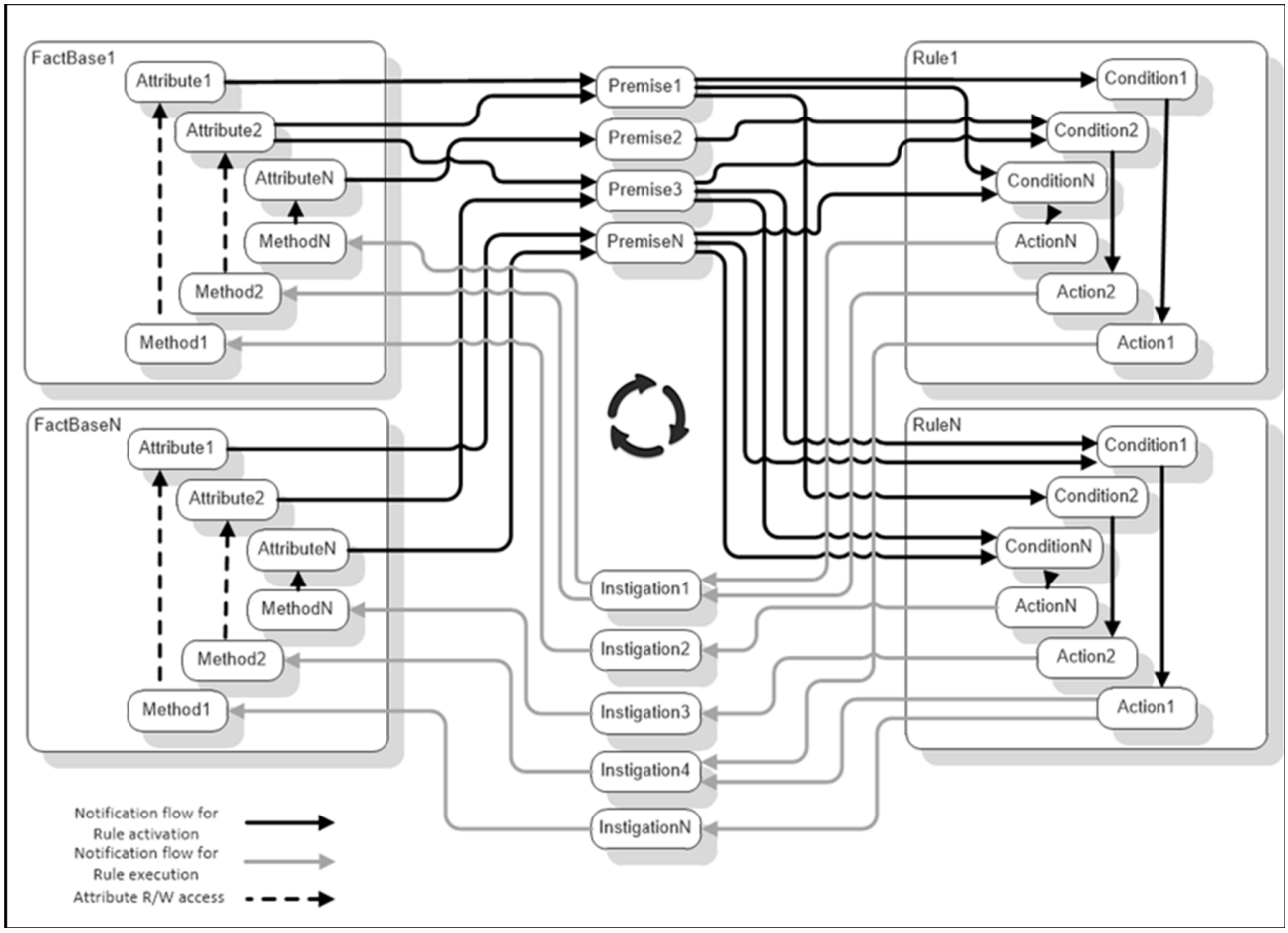

Figura 4: Inferência por notificações [Simão at al., 2014]

Isto considerado, nota-se que a essência da computação no PON está organizada e distribuída entre entidades autônomas e reativas que colaboram por meio de notificações pontuais. Este arranjo forma o mecanismo de notificações, o qual determina o fluxo de execução das aplicações. Por meio deste mecanismo, as responsabilidades de um programa são divididas entre os objetos do modelo, o que permite execução otimizada e 'desacoplada' (*i.e.* minimamente acoplada) 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.* 2012a, 2012b, 2012c].

A natureza do PON leva a uma nova maneira de compor *software*, na qual os fluxos de

5 Existe uma ressalva na Figura 4 – Inferência por notificações: no atual estado da arte em PON: as *Rules* possuem somente uma *Condition* e uma *Action*. Não obstante, no momento da manufatura deste trabalho, esta é a figura mais atualizada do grupo de pesquisa tendo sido publicada em [Simão at al., 2014].

execução são distribuídos e colaborativos nas entidades. Ainda, muito embora o PON permita compor *software* em alto nível na forma de regras, sem o conhecimento de sua essência (*i.e.* inferência por notificações), este conhecimento é importante. Por exemplo, é importante saber dos impactos de desempenho e das estratégias de distribuição, como o agrupamento de elementos de maior fluxo de notificações juntos. Assim sendo, o PON permite uma nova maneira de estruturar, executar e conceber os artefatos de *software* [Simão e Stadzisz, 2008] [Linhares *et al.*, 2011][Simão *et al.* 2012a, 2012b, 2012c].

Atualmente, o PON está materializado na forma de um *Framework* implementado na linguagem de programação C++ enquanto uma linguagem-compilador PON está em desenvolvimento [Ferreira *et al.*, 2013]. Um paradigma pode se materializar em outro, e isto é natural em paradigmas emergentes (*e.g.* um programa POO feito em linguagem procedimental, como a primeira implementação da linguagem C++, que era um tradutor desta linguagem em código fonte em linguagem C [Stroustrup, 1996]). Todavia, de fato, seria mais conveniente e apropriado um ambiente efetivamente voltado para o PON [Banaszewski, 2009][Valença *et al.*, 2011][Simão *et al.* 2012b, 2012c].

Neste ínterim, a materialização do PON dita primária foi comparada em termos de desempenho com implementações PI/POO e PD/SBR. No caso de PI/POO houve comparações com programas C++/OO usuais [Banaszewski, 2009][Simão *et al.* 2012a]. No caso de PD/SBR, houve comparações com dois *shells*, CLIPS e RuleWorks, que usam o eficiente motor de inferência RETE [Forgy, 1982]. Estas comparações apresentaram resultados a favor do PON, ainda que sobre *toy problems* [Banaszewski, 2009]. Também houve comparações qualitativas e assintóticas favoráveis ao PON em relação a motores de inferência como RETE, TREAT, LEAPS e HAL [Banaszewski, 2009].

Todavia, outros testes sobre aplicações reais se fizeram necessários para verificar a eficiência e eficácia do PON, em termos de desempenho, particularmente em relação ao paradigma dominante POO. Tais testes mostram que a materialização do PON é melhor que POO quando há muitas relações causais e variáveis com baixa ou média variação de estados [Banaszewski, 2009][Linhares *et al.*, 2011][Batista *et al.*, 2011][Ronszcka *et al.*, 2011] [Valença *et al.*, 2011][Simão *et al.* 2012a, 2012b].

Ademais, quanto melhor se constituir a materialização do PON (*e.g.* em termos de otimização de estruturas de dados), melhores são os resultados de desempenho do PON. Esta afirmação advém da comparação entre o *Framework* PON Prototipal e o *Framework* Primário em [Banaszewski, 2009], e do mesmo modo da comparação entre o *Framework* Primário e a nova versão *Framework* PON Otimizado [Valença *et al.*, 2011][Simão *et al.* 2012b, 2012c].

Isto dito, a partir dos experimentos elaborados em trabalhos anteriores como [Banaszewski, 2009], esta pesquisa evolui para casos de estudo a partir do avanço obtido com a elaboração do *Framework* PON dito Otimizado [Batista *et al.*, 2011][Ronszcka *et al.*, 2011] [Valença *et al.*, 2011] e, indo além, também com o Compilador PON em desenvolvimento [Ferreira *et al.*, 2013]. Isto permitirá definir a curva evolutiva entre cada materialização ou *Framework* PON, também relacionado ao atual estado da técnica.

Igualmente, no estado da técnica e mesmo da arte em PON, há também pesquisas em desenvolvimento como linguagem nativa de programação – LingPON – e respectivo compilador [Ferreira *et al.*, 2013], teste funcional em *software* PON [Kossoski et al., 2014], arquitetura de hardware específica coprocessador PON [Peters, 2012], arquitetura paralela com distribuição de carga de *software* (*i.e. multicore*) [Belmonte *et al.*, 2012], processador nativo em PON ARQPON [Linhares *et al.*, 2014] e aplicações na computação sensível ao contexto [Simão *et al.*, 2014]. Ainda, acerca da máquina de inferência que realiza o cálculo lógico-causal do PON há estudos avaliando a sua extensão para utilização de lógica *Fuzzy* [Melo, 2013]. E em termos de processo de engenharia de *software*, o estado da arte específico para o PON é o DON – Desenvolvimento Orientado a Notificações e seu inerente Perfil para UML [Wiecheteck, 2011][Wiecheteck, Stadzisz e Simão, 2011].

Basicamente, o perfil UML denominado Perfil PON define os principais conceitos do PON por meio da utilização de mecanismos de extensão da UML (*e.g.* estereótipos) [Wiecheteck, Stadzisz e Simão, 2011]. Já o método DON, que faz uso do Perfil PON, consiste das fases de *software* requisitos e projeto, com o objetivo de modelar *software* para construção em PON. DON é organizado em oito passos: “Capturar requisitos”, “Criar Modelo de Caso de Uso”; “Criar Modelo de Classes”, “Criar Modelo de Atividades de Alto Nível”, “Criar Modelo de Componentes”, “Criar Modelo de Sequência”, “Criar Modelo de Comunicação”, “Criar Modelo de Redes de Petri”. Os dois primeiros passos fazem parte da fase de requisitos, enquanto os outros seis passos seguem a fase de projeto de *software* conforme a Figura 5 [Wiecheteck, 2011].

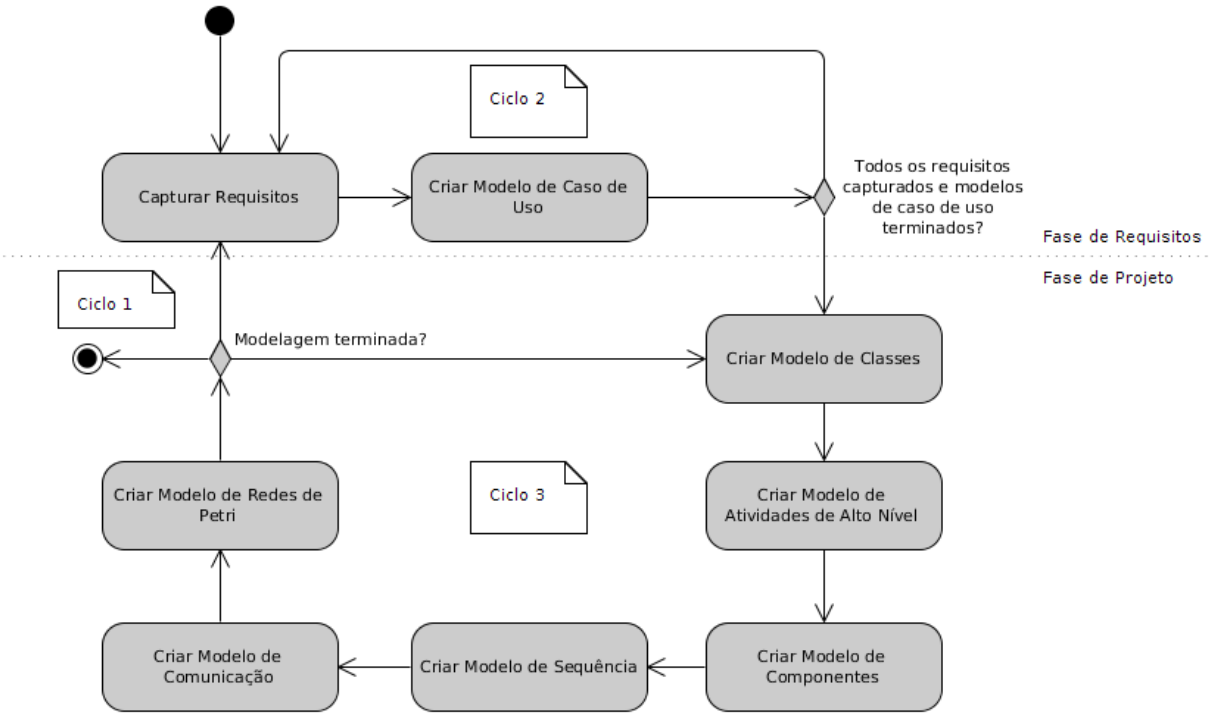

Figura 5: Método para Projeto de *Software* em PON - Desenvolvimento Orientado a Notificações (DON) [Wiecheteck, 2011].

Após esta breve descrição do PON, a próxima seção apresenta o POE, e algumas de suas diferentes técnicas de programação, as quais serão utilizadas para o posterior desenvolvimento e comparação entre os paradigmas.

2.3 Paradigma Orientado a Eventos

Para compreender o Paradigma Orientado a Eventos é necessário articular o conceito de evento em relação a *software*. Faison, que é um dos autores da área de estudo do POE, define evento e notificação [Faison 2006] (conforme a Figura 6):

"*Um evento é uma condição detectada que pode disparar uma notificação.*

Uma notificação é um sinal provocado por um evento, enviada para um recebedor definido em tempo de execução."

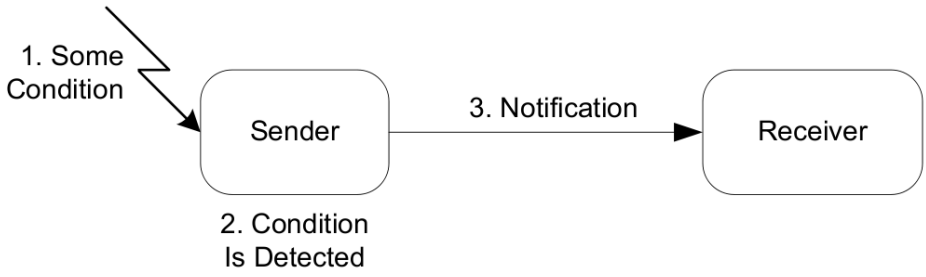

Figura 6: Evento, condição detectada, notificação [Faison, 2006].

Gero Mühl também define notificação como sendo um dado que concretiza um evento, ou seja, contém informação descrevendo o evento. A notificação é criada por um observador do evento e pode indicar sua simples ocorrência, todavia comumente carrega informações que descrevem suas circunstâncias [Mühl *et al.*, 2006].

Na mesma linha, Etzion e Nibblet definem evento como uma ocorrência dentro de um sistema ou domínio particular [Etzion e Niblett, 2011]. Os autores instituem outro significado da palavra evento, traduzindo-a como uma entidade evento de programa (*e.g.* objeto do POO) que representa uma dada ocorrência em um sistema computacional.

Ademais, [Etzion e Niblett, 2011] definem situação, análogo ao conceito de notificação [Faison 2006]. Apesar da diferença em terminologia (notificação e situação), conceitualmente são ideias complementares:

"*Situação é uma ocorrência de evento que pode requerer uma reação.*"

Alinhado a essa ideia, [Rumbaught *et al.*, 1999] elenca uma definição de evento no contexto da UML (*Unified Modeling Language*):

"*Cada evento corresponde à especificação de uma ocorrência significante, com tempo e espaço determinados.*"

Em outra definição, [Eugster *et al.*, 2003] relata que eventos são encontrados em duas formas: mensagens e invocações. No primeiro caso, eventos são entregues a um assinante (do inglês *subscriber*) por meio de uma operação simples, por exemplo, *notify()* – notificação. No segundo caso, *i.e.* invocação, eventos iniciam a execução de operações específicas no *subscriber*.

Findando a conceituação de evento, [Mühl *et al.*, 2006] enuncia que qualquer acontecimento de interesse que pode ser observado pela perspectiva de um sistema é considerado um evento. Assim, [Mühl *et al.*, 2006] e [Le *et al.*, 2013] listam três tipos de eventos em *software*:

• Evento de tempo, que expressa transcorrer do tempo (*timer*);

• Evento físico, como o aparecimento de uma pessoa detectada por câmeras;

• Mudança detectável de estado do sistema (*e.g.* uma mudança de valor de uma variável durante a execução). Este é o tipo mais relevante, pois os dois primeiros tratam da detecção de eventos relacionados ao ambiente externo ao programa [Mühl *et al.*, 2006].

Seguindo adiante, conforme as definições anteriores de paradigmas de programação, o Paradigma Orientado a Eventos é primordialmente o modelo de construção de *software* que trata eventos. Um evento pode ser um botão pressionado, uma interrupção de *hardware* ou uma mensagem recebida, oriundo de um sujeito, produtor, publicador, usuário, interface, componente, sensor ou sistema. Esse evento instiga uma determinada ação (*i.e.* função, método, processo ou comportamento). A ação comumente está contida em um tipo determinado de módulo, como um bloco, objeto, observador, consumidor ou até mesmo agente (conforme Figura 7) [Eugster *et al.*, 2003][Faison, 2006][Ferg, 2006][Samek, 2008] [Hansen e Fossum, 2010][Simão *et al.*, 2012a][Le *et al.*, 2013].

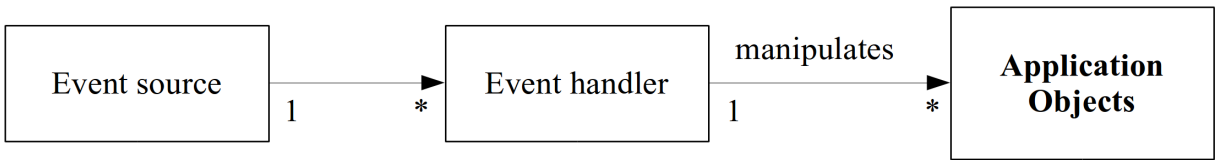

Figura 7: Paradigma Orientado a Eventos adaptado de [Hansen e Fossum, 2010].

Ainda, eventos ocorrem em sequência imprevisível e definida somente em tempo de execução [Harel e Pnueli, 1985]. Harel definiu os sistemas reativos, que se relacionam dinamicamente com o ambiente externo, dessa forma respondendo continuamente a estímulos de ordem desconhecida oriundos do ambiente em que está inserido. Segundo Harel, a classificação de sistemas reativos se opõe aos sistemas transformacionais, *i.e.* sistemas que computam saídas a partir de um conjunto de entradas, executando transformações, como por exemplo um compilador. O autor concebeu uma dicotomia: sistemas transformacionais versus sistemas reativos. Uma maneira de modelar os sistemas reativos, definida pelo próprio autor, é a modelagem de estados (*i.e. StateCharts*) [Harel, 1987][Harel e Politi, 1998].

Gérard Berry complementarmente define os sistemas interativos que interagem conforme sua própria velocidade com o usuário ou outros programas [Berry, 1989]. Os sistemas de tempo compartilhado (*timesharing*), por exemplo, são sistemas interativos, pois gerenciam fatias de tempo das interações com o ambiente externo. Dessa maneira, esses

programas se diferenciam dos programas transformacionais, dos programas reativos em que a interação é determinada pelo ambiente externo e não pelo programa, e também dos programas de tempo real. Os programas de tempo real, ao seu turno, se distinguem visto que tem como requisito atender limites impostos de tempo, além da interação contínua com o ambiente [Berry, 1989].

Os exemplos de programas reativos seriam *drivers* de sistemas operacionais, emissores e receptores dos protocolos de comunicação, gerenciador de interface de teclado e dispositivo apontador - *i.e. mouse* (*e.g.* barra de menu ou barra de rolagem) [Berry e Gonthier, 1992]. De acordo com os autores, os programas reativos são compostos de três camadas:

1. Uma camada de interface com o ambiente, responsável por recepção de estímulos físicos e produção de saídas. Gerencia interrupções, recebe dados de sensores, opera atuadores, transformando estímulos físicos externos em sinais lógicos internos e vice-versa;
2. Um *kernel* reativo que contém a lógica do sistema. Gerencia e opera as entradas e saídas lógicas. É o componente que decide quais computações e saídas devem ser geradas em reação às entradas;
3. Uma camada de manipulação de dados que executa computações usuais requisitada pelo *kernel* reativo.

Assim, nas palavras dos autores, observa-se a característica de "orientação à entrada" dos programas reativos (*i.e. input-driven*) [Berry e Gonthier, 1992]. Dessa forma, a programação reativa se faz análoga e em convergência ao Paradigma Orientado a Eventos: manter interação contínua com o ambiente, processar eventos e executar tarefas correspondentes como atualizar o estado da aplicação e informar dados [Pucella 1998] [Bainomugisha *et al.*, 2013].

Conforme Bainomugisha, na literatura das linguagens de programação reativas, suas duas principais características são comportamentos e eventos [Bainomugisha *et al.*, 2013]. Comportamento é o termo utilizado para valores que variam no tempo, continuamente, e são "cidadãos de primeira classe" nessas linguagens (*i.e.* do inglês *first class citizen*). Um exemplo tradicional seria o tempo em si [Elliott and Hudak 1997]. Eventos se referem a fluxos de alterações de valor (potencialmente infinitos) e ao contrário de comportamentos, que mudam continuamente ao longo do tempo, os eventos ocorrem em pontos discretos no tempo (por exemplo botão do teclado pressionado, mudança de localização) [Cooper, 2008].

Ainda, a programação reativa pode se realizar por meio dos Paradigmas Funcional

Reativo (*e.g.* linguagem Fran[6] [Elliott e Hudak, 1997]), Síncrono, de Fluxo de Dados, Fluxo de Dados/Síncrono, e também dos parentes da programação reativa (*i.e.* do inglês *cousins*, geralmente extensões embarcadas em outras linguagens como *Rx* para .Net[7]) conforme a Figura 8 [Bainomugisha *et al.*, 2013].

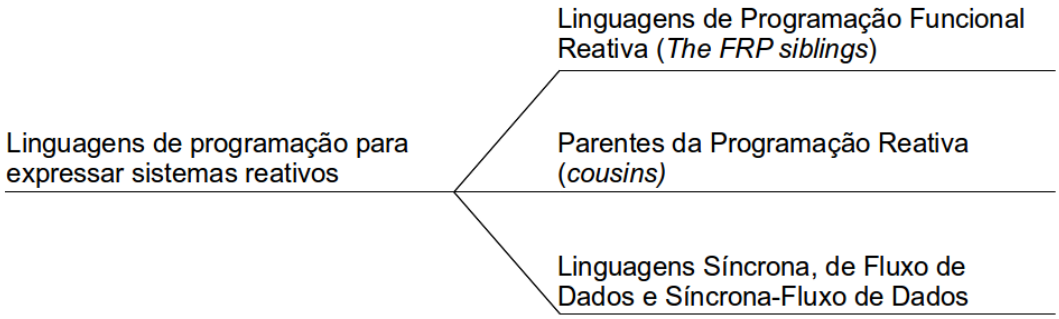

Figura 8: Classificação de Linguagens para Programação Reativa adaptado de [Bainomugisha *et al.*, 2013].

Como desdobramento do PF, o Paradigma Funcional Reativo (PFR) é uma forma de programação para trabalhar com valores mutáveis (*i.e.* fluxo contínuo). Dessa maneira, PFR é fundamentado acerca da noção de valores que variam continuamente durante o tempo (*i.e.* comportamento). A propagação de mudanças desses valores é importante pois são automaticamente executadas por meio de um modelo de execução computacional implícito. Um exemplo elementar desse conceito seria o uso de uma planilha e a dependência entre células com dados e fórmulas que as relacionam [Elliott e Hudak, 1997][Van Roy, 2009] [Bainomugisha *et al.*, 2013].

A programação síncrona é o paradigma mais antigo proposto para desenvolver sistemas reativos com restrições de tempo real [Bainomugisha *et al.*, 2013]. Como exemplos, dos autores Gérard Berry e Laurent Cosserat, a linguagem Esterel para programar *kernels* reativos [Berry e Cosserat, 1984] (Esterel pertence ao Paradigma Síncrono Determinístico segundo Van Roy [Van Roy, 2009]) e do autor Simão Toscani a linguagem reativa síncrona – RS – implementada e embarcada no Prolog [Toscani, 1993][Librelotto, 2001].

Linguagens de programação síncrona são baseadas na hipótese do sincronismo, ou seja, as reações são atômicas, em relação ao ambiente externo, duram tempo zero (atribui-se que são instantâneas), e consideram que o ambiente externo permanece imutável durante a

6 Linguagem disponível em http://conal.net/fran/

7 *The Reactive Extensions Rx*.Net disponível em http://msdn.microsoft.com/en-us/data/gg577609.aspx

execução da reação. Essa hipótese propicia programas simples e compiláveis em eficientes autômatos de estado finito [Berry e Gonthier, 1992][Elliott e Hudak, 1997][Librelotto, 2001] [Bainomugisha *et al.*, 2013].

Outra abordagem para programas reativos é a programação de fluxo de dados (*dataflow*). Um programa em *dataflow* se expressa em um grafo dirigido, com nós representando operações e arcos representando dependências entre as computações. Exemplos desse paradigma seriam a linguagem *LabVIEW*[8] [Kalkman, 1995] e as plataformas *Simulink*[9], *Meemo*[10] e *Ankhor FlowSheet*[11].

Já as linguagens de fluxo de dados síncronas (*synchronous dataflow*), somam as características da hipótese do sincronismo e de fluxo de dados. Especificamente, a estrutura do grafo é conhecida em tempo de compilação, logo pode ser estaticamente prevista e convertida em um programa sequencial, nesse caso não necessitando dinamicidade em tempo de execução [Bainomugisha *et al.*, 2013]. Exemplos de linguagem de fluxo de dados síncronas seriam Lustre [Halbwachs *et al.*, 1991] e Signal [Amagbégnon *et al.*, 1995].

Mais além, outros autores definem eventos, sistemas de *software* orientados a eventos e POE, acrescentando e listando técnicas, conceitos e exemplos de uso. Exemplificando, Michael L. Scott caracteriza que "*um evento é algo que um programa em execução (um processo) precisa responder, que ocorre fora do programa, em tempo imprevisto.*" Os exemplos de eventos mais comuns seriam de entrada em uma interface de usuário (*GUI – Graphical User Interface*) como teclar, mover apontador (*mouse*) e cliques de botão [Scott, 2008].

Da mesma forma, eventos podem ser operações de rede ou atividades de I/O assíncronas: a chegada de uma mensagem ou o término de uma operação de disco requisitada anteriormente. Nas linguagens modernas de programação e sistemas de tempo compartilhado de execução (*run-time systems*), eventos geralmente são tratados por linhas separadas de controle de execução – *threads* [Dabek *et al.*, 2002] [Scott, 2008][McKellar, 2012].

J. Glenn Brookshear conceitua os Sistemas de *Software* Dirigidos a Eventos, que seriam sistemas de *software* cujos procedimentos são ativados implicitamente por eventos ao invés de requisições explícitas. Um exemplo seria um procedimento ativado pelo resultado de um botão pressionado, ao invés de ser chamado diretamente por outra unidade do programa.

8 Linguagem LabVIEW disponível em http://www.ni.com/labview

9 Plataforma Simulink disponível em http://www.mathworks.com/products/simulink/index.html

10 Plataforma Meemo disponível em http://meemoo.org/

11 Plataforma Ankhor disponível em http://www.ankhor.com

Seriam diferentes dos sistemas de *software* cujos procedimentos são explicitamente chamados por comandos em outro local no próprio programa [Brookshear, 2012]. Conforme este autor, um sistema de *software* dirigido a eventos "*consiste de vários procedimentos que descrevem o que deve ocorrer como resultado de vários eventos.*" Quando o sistema é executado estes procedimentos ficam inativos até que um dado evento relacionado ocorra. Eles então se tornam ativos, executam sua respectiva tarefa e retornando em seguida à inatividade [Brookshear, 2012].

Entretanto, Robert Sebesta define que tratamento de eventos (*Event Handling*) é similar a tratamento de exceções (*Exception Handling*), embora de menor complexidade. Um tratador de eventos (*event handler*) é um segmento de código que é executado em resposta ao aparecimento de um evento. *Event handlers* permitem ao programa ser responsivo às ações de usuário. Eventos são criados por ações externas, como interações de usuário ao utilizar uma interface de *software*. Implementar reações para interações de usuário via componentes de interface é a forma mais comum de tratamento de eventos [Sebesta, 2012].

Jessica McKellar, Stuart Hansen e Stephen Ferg relacionam técnicas do POE. McKellar caracteriza que Programação Orientada a Eventos é um paradigma de programação no qual o fluxo de execução é determinado por eventos externos. É definido por um ciclo de eventos (*event loop*) e do uso de *callbacks* para realizar ações quando eventos acontecem [McKellar, 2012].

Ferg, por sua vez, complementa as técnicas de POE elencando-as como tratadoras de eventos (*e.g.* fila de eventos, *event handler*, *dispatcher*, *biding*, *registered-event handler*) [Ferg, 2006], apresentando a utilização da Programação Orientada a Objetos Dirigida a Eventos (*Object-Oriented Event-Driven Programming*), ou seja, POO para tratamento de eventos. São ressaltadas duas técnicas, a primeira baseada no padrão de projeto *Observer*[12] [Gamma *et al.*, 1995], que pode ser considerada sinônimo das técnicas de *callback* ou tratamento registrado de eventos (*registered-event handler*). A outra técnica refere-se à Máquina de Estados – padrão de projetos *State* [Faison, 1993][Gamma *et al.*, 1995][Schmidt *et al.*, 2000][Ferg, 2006][Hansen e Fossum, 2010], em convergência ao modelo *StateCharts* de Harel [Harel, 1987][Harel e Politi, 1998]. Todas as técnicas também figuram em [Faison, 2006], [Samek, 2008] e [Etzion e Niblett, 2011].

Além de *Observer* e *State*, [Gamma *et al.*, 1995] relaciona outras soluções POO para tratamento de eventos como *Mediator* e *Chain of Responsibility*. Os seguintes padrões de

12 Um objeto-sujeito (*subject*) mantém uma lista de objetos-observadores (*observers*) previamente cadastrados e notifica-os automaticamente em uma mudança de seu estado.

projeto, inspirados a partir de *Observer*, também tratam de eventos no contexto de Sistemas Distribuídos: *Publisher/Subscriber* [Buschmann *et al.*, 1996][Eugster *et al.*, 2003], *Proactor* e *Reactor* [Schmidt *et al.*, 2000].

Logo, soluções POE são construídas por meio de técnicas inerentes aos paradigmas elementares, a saber, do Paradigma Imperativo (tanto Procedimental quanto Orientado a Objetos). Tanto que POE é visto com um subtipo do Paradigma Procedimental [Brookshear, 2012] ou complementar ao POO [Ferg, 2006][Faison, 2006][Hansen e Fossum, 2010]. Além disso, programação funcional e orientada a eventos usadas em conjunto seria algo comum (*e.g.* ponteiros para funções, C++ *functors* – funções como objetos que sobrecarregam *operator()* –, e programação reativa) [Van Roy, 2009][Hansen e Fossum, 2010][Simão *et al.*, 2012a][Valença, 2012][Bainomugisha *et al.*, 2013][Salvaneschi *et al.*, 2014].

Mais além, o ciclo de eventos (*event loop*) e POE são largamente utilizados na indústria. Esta técnica é a base para a concepção, por exemplo, do tratamento de eventos de interface ao usuário em sistemas operacionais (*e.g. widgets*, *GUIs*, controle de dispositivos de entrada como teclado, dispositivo apontador – *mouse*, *touchpad*, *joystick*). Referências seriam *Windows MFC*[13], *Microsoft .NET/C#*[14], *X.Org XWindowProtocol*[15], *Digia QT*[16], *Java AWT*[17], *TwistedMatrixLabs Twisted*[18], *Microsoft VisualBasic*[19], *Embarcadero Delphi/Object Pascal*[20] [Samek, 2008]. Ademais, os ambientes citados como exemplo *VisualBasic*, *C#*, *Java AWT* e *Delphi* historicamente transformaram o desenvolvimento orientado a eventos em relação à interface com o usuário (ambientes também utilizados para prototipagem rápida, comumente relacionados ao processo de engenharia de *software Rapid Application Development* – RAD).

Ademais, diversos autores advogam vantagens na utilização de POE como [Dabek *et al.*, 2002][Faison, 2006][Samek, 2008][Bainomugisha *et al.*, 2013]. Um modelo orientado a eventos para *software* geralmente é uma boa escolha quando o programa realiza muitas tarefas que são largamente independentes (desta forma não existe intercomunicação ou espera entre elas) e algumas dessas tarefas bloqueiam enquanto esperam por eventos. Aplicações de rede

13 Biblioteca MFC disponível em http://msdn.microsoft.com/en-us/library/0x0cx6b1.aspx

14 Linguagem C#/.Net disponível em http://msdn.microsoft.com/en-us/library/aa645739(v=vs.71).aspx

15 Especificação do sistema X Window disponível em http://www.x.org

16 Plataforma e linguagem Qt disponível em http://qt-project.org/

17 Especificação da biblioteca gráfica Java disponível em http://docs.oracle.com/javase/7/docs/api/java/awt/package-summary.html

18 *Engine* de rede orientado-a-eventos disponível em http://twistedmatrix.com

19 Plataforma/linguagem Visual Basic disponível em http://msdn.microsoft.com/en-us/library/ms172877.aspx

20 Plataforma e linguagens Delphi disponível em http://www.embarcadero.com/products/delphi

geralmente contemplam essas propriedades sendo candidatas ao POE [McKellar, 2012].

Os autores [Dabek *et al.*, 2002] e [Le *et al.*, 2013] acrescentam que utilizar POE requer menor esforço de desenvolvimento e pode levar a *software* mais robusto. Da mesma forma, um sistema baseado em eventos reduz a sua complexidade como um todo [Faison, 2006]. Seriam características intrínsecas de POE o baixo acoplamento, o controle baseado em estado e a capacidade de computação concorrente e distribuída [Hansen e Fossum, 2004].

Em contrapartida, a complexidade aumenta para entender todo um conjunto de elementos computacionais necessários para resolver um determinado problema [Faison, 2006]. De um lado, partes menores que interagem em um *software* baseado em eventos geralmente são mais simples sob o ponto de vista estrito de código, neste caso, propiciando componentes menores, razoavelmente coesos e desacoplados. Por outro lado, aumenta-se a dispersão da lógica e os relacionamentos no *software*, sendo necessário percorrer e compreender todo o conjunto de unidades de uma solução [Faison, 2006][Edwards, 2009] [Bainomugisha *et al.*, 2013][Salvaneschi e Mezini, 2014].

Sob o ponto de vista de processos de engenharia de *software* exclusivos de sistemas orientados a eventos amplamente aceitos pela comunidade, foram encontradas somente ferramentas como o Diagrama de Estados (*e.g.* StateCharts) [Harel, 1987], extensões da *UML* para a modelagem de sistemas orientados a eventos [Rumbaught *et al.*, 1999]. Nesse âmbito, Faison também elenca uma série de ferramentas para modelagem como *UML*, *Lollipop*, *SDL*, *Espresso*, *Catalysis*, *Signal Wiring*, sendo essa última de sua própria autoria. [Faison, 2006].

No estado da arte em POE, existem diversas linguagens de programação baseadas em evento, incluindo linguagens reativas, multiparadigmas e específicas de domínio (DSL – *Domain Specific Language*). Neste contexto, POE é utilizado em muitos tipos de aplicações como robótica, sistemas distribuídos, aplicações sensíveis ao contexto, sistemas auto adaptativos, sistemas de interação homem-máquina, sistemas de tempo real, *drivers* de dispositivos e interação multimodal (*i.e.* interação com sistemas ou dispositivos computacionais por meio de mais de um tipo de estímulo de entrada ou saída, *e.g.* utilização de diferentes dispositivos, gestos, voz, tato, texto, movimentos oculares, reconhecimento facial, reconhecimento de imagens).

Em robótica, cita-se a linguagem INI [Le *et al.*, 2012][Le et al 2013]. A principal característica da INI é que ela combina os estilos de programação baseada em evento e baseada em regras. Eventos ou regras podem ser definidos independentemente ou em conjunto. Eventos em INI executam em paralelo, tanto de maneira síncrona quanto assíncrona, conforme restrições. A linguagem INI é executada pela JVM (Java Virtual

Machine) e está disponível em [Le, 2012]. Em sistemas embarcados, alguns dos exemplos são *Protothreads* (*threads* para programação de *Event Driven Programs*) [Dunkels *et al.*, 2006] e *Quantum Platform*[21] (máquinas de estado hierárquicas que implementam o Paradigma de Objetos Ativos – *actors*) de [Samek, 2008].

Em processamento de eventos, *EventScript* [Cohen e Kalleberg, 2008] é uma linguagem simples para programação de processos reativos. Um fluxo de eventos produzido é combinado às expressões regulares, e ações são executadas a partir equivalência de padrões de eventos. As ações incluem atribuição de valores computados bem como a emissão de eventos de saída. Cabe mencionar outras linguagens como *EventJava* (*i.e.* multiparadigma, aspectos, OO e eventos) [Holzer *et al.*, 2011] para computação pervasiva (*i.e.* conceito análogo e complementar à computação ubíqua, que objetiva realizar as interações homem-máquina o mais transparente possível tornando a percepção da máquina invisível [Weiser 1991]) e *Microsoft P* [Desai *et al.* 2012], que é uma linguagem específica de domínio, utilizada na construção do *driver* USB do Windows 8. Segundo os autores, "*P permite ao programador especificar o sistema como uma coleção de máquinas de estado que interagem"*.

Vem a propósito referenciar linguagens especificamente em programação reativa. Como exemplo pode-se citar a linguagem Céu[22] [Sant'Anna *et al.*, 2012], inspirada em Esterel, que unifica os estilos de fluxo de dados - *dataflow* - e imperativo das linguagens síncronas reativas, com objetivo de propiciar uma alternativa de alto nível aos modelos orientados a eventos e *multithreaded* para sistemas embarcados [Sant'Anna *et al.*, 2013]. A linguagem Elm[23] é uma linguagem funcional reativa concorrente para *software* Web, cuja compilação gera HTML, CSS e Javascript [Czaplicki, 2012][Czaplicki e Chong 2013]. Por fim, *EScala*[24] e *REScala*[25], extensões da linguagem de programação *Scala*, que provê suporte para eventos como atributos de objetos, integrando os estilos de programação orientada a eventos, orientada a aspectos e funcional reativo, somando o encapsulamento da OO [Gasiunas *et al.*, 2011][Salvaneschi *et al.*, 2014].

---

21 Plataforma QP disponível em http://www.state-machine.com/

22 Linguagem Céu disponível em http://www.ceu-lang.org/

23 Linguagem Elm disponível em http://elm-lang.org/

24 Linguagem EScala disponível em http://www.stg.tu-darmstadt.de/research/escala/downloads_1/index.en.jsp

25 Linguagem REScala disponível em http://www.stg.tu-darmstadt.de/research/rescala_menu/overview_rescala.en.jsp

2.4 Técnicas do Paradigma Orientado a Eventos

Inicialmente, nesta seção, se apresentam os motivos para a seleção das técnicas POE *Handler-Dispatcher*, *State Pattern* e *Observer Pattern*. Neste sentido, é oportuno enfatizar que questionamentos em relação aos trabalhos anteriores do grupo de pesquisa do PON tentaram equiparar PON ao POE, argumentando serem (genericamente) sobrepostos (principalmente quanto ao *Observer Pattern*). O presente trabalho apresenta evidências que inferem, justamente, as inspirações e diferenças de PON em relação à POE.

Primeiramente, as técnicas escolhidas são onipresentes na computação. Soluções de mercado, técnicas e estilos de programação profissional (*e.g.* jogos), *frameworks*, bibliotecas, linguagens, IDEs de desenvolvimento e pesquisas científicas comumente se utilizam das técnicas *Handler-Dispatcher* (ciclo de eventos), *State* e *Observer* (e também suas variações, composições e evoluções). Ainda, em análise de estado da técnica (sob ponto de vista industrial) e da arte (sob ponto de vista da literatura pesquisada neste trabalho), as técnicas percebidas como mais citadas, comparadas, utilizadas, alicerces de referência, tanto criticadas bem como apontadas como melhores alternativas em pesquisas e aplicações que tratam eventos são *Dispatcher*, *State* e *Observer*.

Em suma, "*o estado atual desses assuntos é: desenvolvedores implementam aplicações reativas no mundo confortável dos objetos*" [Salvaneschi e Mezini, 2014]. Por outro lado, as alternativas oferecem uma solução atraente (*e.g.* FRP, linguagens síncronas), muito embora não obtenha sucesso por conta de não oferecerem a flexibilidade necessária, não se integrarem com OO bem como não trabalham de forma transparente com objetos mutáveis [Salvaneschi e Mezini, 2014][Salvaneschi *et al.*, 2014b]. Ainda, David Watt defende que "*linguagens funcionais e lógicas são prejudicadas por sua incapacidade de modelar estado de forma natural, a não ser por mais ou menos extensões adhoc que destroem a clareza conceitual dessas linguagens.*" [Watt, 2004]

Ou seja, uma pergunta oportuna no estado atual que também perpassa este trabalho é justamente "PON resolve eventos sendo uma alternativa viável, flexível e integrável?" Para tanto também são necessárias comparações com essas técnicas basilares de POE.

Mais além, a escolha de técnica em POE mais próxima ao PON se dá em relação ao *Observer Pattern*. O Paradigma Orientado a Notificações tem inspirações em *Observer*, como demonstrado no trabalho de [Ronszcka, 2012], onde o próprio PON foi analisado pelo viés de padrões incluindo o padrão *Observer* e no trabalho que compôs o PON em Java (*i.e.* controle

holônico) [Izidoro e Quináia, 2014]. Entretanto, ainda não existe um trabalho no qual se compara *software* construído utilizando *Observer* (exclusivo e ortodoxo) e *software* construído em PON.

Em seguida, nesta seção, as três técnicas mais utilizadas para o desenvolvimento de *software* no paradigma POE são efetivamente apresentadas: *Handler-Dispatcher*, *State Pattern* e *Observer Pattern*.

2.4.1 Técnica *Handler-Dispatcher*

A técnica *Handler-Dispatcher*, ou "componente Despachante e componente Tratador", compõe-se basicamente das seguintes funcionalidades: receber cada evento do *software*, analisar o evento para determinar o seu tipo, e então redirecionar cada evento para o tratador (*Handler*) que tem como capacidade tratar eventos de um tipo especificado [Ferg, 2006] [Samek, 2008].

O *Dispatcher* deve processar um fluxo de eventos de entrada (geralmente o despachante mais simples é serial e síncrono). Para tanto sua lógica inclui algum tipo de reconhecimento de eventos (comumente um ciclo com um teste ou condição de finalização), no qual recebe o evento, o despacha, e reinicia o ciclo novamente para o próximo evento no fluxo de entrada [Ferg, 2006]. Esta técnica é também conhecida como *polling* [Samek, 2008] [Peters, 2012]. Van Roy também cita exclusiva e explicitamente essa técnica como um paradigma em si, *Event Loop Programming*, conforme a Figura 1 [Van Roy, 2009].

O diagrama da Figura 9 demonstra graficamente a solução *Handler-Dispatcher* e o papel (simplificado) de um *Dispatcher*.

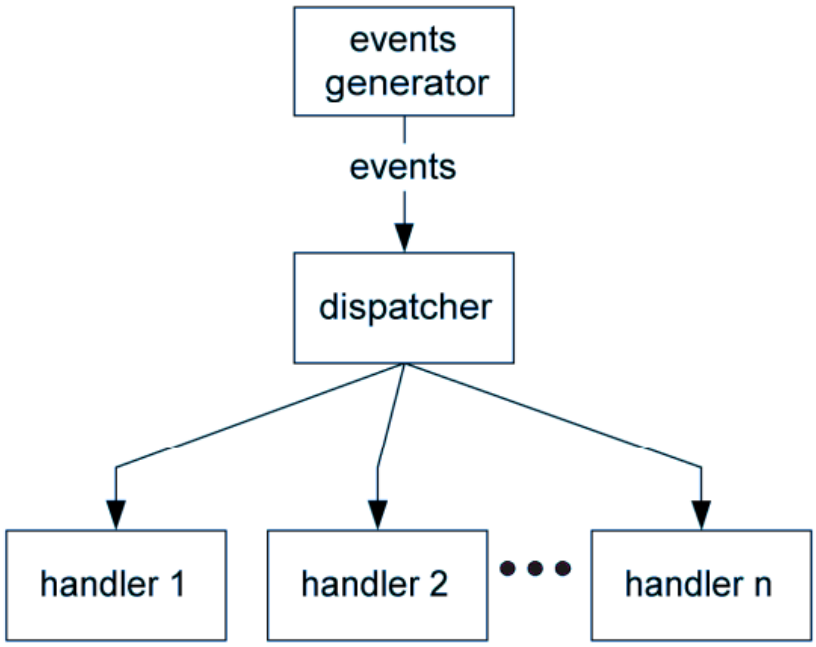

Figura 9: Um diagrama conceitual do componente *Dispatcher* [Ferg, 2006].

No diagrama é possível identificar um bloco com os geradores de eventos (*events generators*), o fluxo de itens de dados recebidos (eventos), o componente *Dispatcher* e um conjunto de tratadores de eventos (*handlers*). Samek também demonstra o uso da técnica *Dispatcher* (o autor também usa como sinônimo componente demultiplexador), *Handler* e ciclo de eventos. Demultiplexador, na sua conceituação clássica, é um componente responsável por distribuir informações de uma única entrada para uma das diversas saídas (o inverso do multiplexador). A Figura 10 ilustra graficamente um sistema tradicional orientado a eventos [Samek, 2008]. Destaca-se no centro da figura o componente *Dispatcher* (denominado como *event dispatcher*). Identificam-se também na Figura 10 componentes avançados em tratamento de eventos como fila de eventos (*event queue*), recebimento de interrupções (*ISRs – Interrupt Service Routines*) e processamento de tempo inativo (*idle processing*).

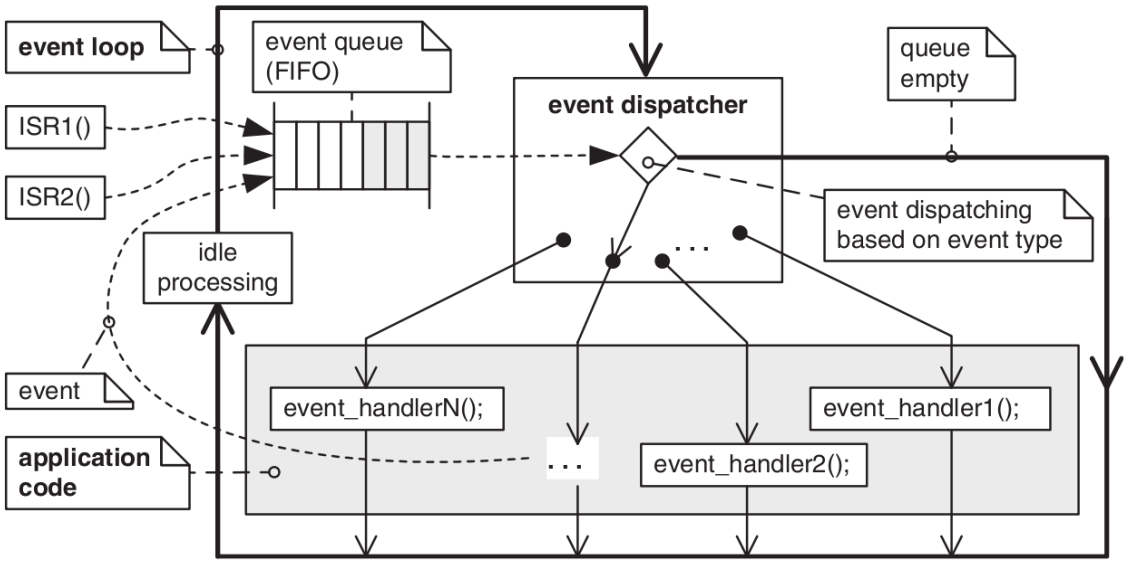

Figura 10: Sistema tradicional orientado a eventos com ciclo de eventos segundo Samek [Samek, 2008].

2.4.2 Técnica *State Pattern*

A técnica *State Pattern* [Gamma *et al.*, 1995] é uma das implementações de uma Máquina de Estados Finito [Samek, 2008]. Segundo Samek, as máquinas de estados são o formalismo mais conhecido para especificar e implementar sistemas orientados a eventos para tratamento de eventos de entrada em tempo hábil. Uma lista elencada de técnicas de Máquina de Estados pelo autor é:

- Instruções <*SWITCH-CASE*> aninhadas;
- Tabela de estados;
- Padrão de projeto OO *State Pattern*;
- Combinações das técnicas.

O padrão comportamental Orientado a Objetos *State* permite que um objeto altere seu comportamento quando seu estado interno muda. O objeto aparentará que mudou de classe [Gamma *et al.*, 1995]. Esse padrão ajuda o desenvolvedor nas situações em que um objeto recebe requisições de outros objetos, e responde diferentemente dependendo de seu estado corrente. Dessa maneira, *State Pattern* descreve como objetos podem exibir comportamentos diferentes em cada estado (seu próprio contexto). Ainda, esse padrão representa um ponto

crucial na Programação Orientada a Eventos [Ferg, 2006][Faison, 2006][Samek, 2008] [Hansen 2010]. A Figura 11 apresenta a estrutura original do padrão de projeto *State*.

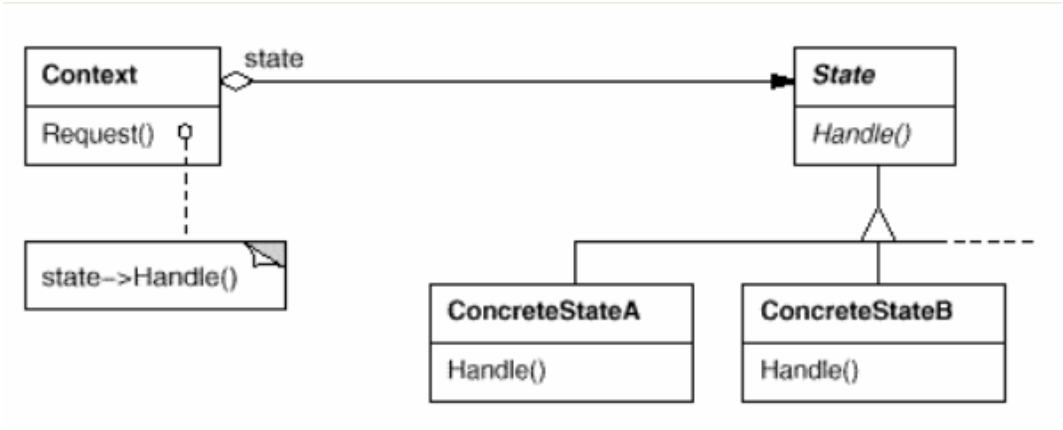

Figura 11: Diagrama de Classes original do padrão de projeto State [Gamma *et al.*, 1995].

Como ilustrado na Figura 11, os participantes desse padrão são [Gamma *et al.*, 1995]:

- *Context* (contexto): define uma interface comum de interesse aos componentes clientes e associa por agregação uma instância de uma subclasse *ConcreteState* que define o estado corrente;
- *State* (classe abstrata estado): define uma interface para encapsular o comportamento associado com um estado particular do contexto – *Context*;

- *ConcreteState* (subclasse estado concreto): cada subclasse implementa um comportamento associado dentro do estado do contexto – *Context*.

Basicamente o funcionamento se dá a partir de uma requisição de tratamento de um cliente (outro objeto ou componente de *software*), ilustrado na Figura 11 pelo método *state->Handle()* que ocorre dentro do método *Request()*. O contexto delega a execução do método *Handle()* para a hierarquia *State* (Estados e suas subclasses), por meio de polimorfismo.

No padrão, "pedaços de comportamento" são chamados de estados (implementados como subclasses de *State*), e as mudanças de comportamento em resposta a qualquer evento corresponde ao tratamento de eventos ou a uma transição de estado (e respectiva mudança de comportamento ao tratar eventos). Por exemplo, em um estado específico, o *software* pode

responder a um subconjunto de eventos permitidos (ignorando outros fora do subconjunto) e produzir somente um subconjunto de respostas possíveis [Samek, 2008].

A utilização desse padrão proporciona vantagens como: encapsula comportamento específico particionando comportamento em diferentes classes que representam diferentes estados; explicita transições entre estados [Gamma *et al.*, 1995]; provê bom desempenho e bom uso de memória [Samek, 2008]. Em contraponto, não explicita o componente responsável que define as transições de estado (*e.g.* a própria classe *Context* ou as subclasses *ConcreteState* podem implementar essas definições); implica em gerenciar criação e destruição de objetos da hierarquia *State*; usa herança dinâmica (polimorfismo) [Gamma *et al.*, 1995]; geralmente compromete o encapsulamento entre *Context* e *State* e o padrão não é hierárquico [Samek, 2008].

2.4.3 Técnica *Observer Pattern*

O padrão comportamental Orientado a Objetos *Observer* permite uma dependência "um-para-muitos" entre objetos, de modo que quando um objeto muda seu estado, todos os seus dependentes são notificados e atualizados automaticamente. Também é conhecido com *Dependents* e *Publish-Subscribe* [Gamma *et al.*, 1995]. Esse padrão auxilia o programador em situações nas quais um *software* possui diversos objetos que colaboram entre si, e é necessário manter a consistência entre esses objetos relacionados. Assim, o padrão estabelece o relacionamento entre objetos, objetivando baixo acoplamento. O conceito chave são os papéis *subject* (sujeito) e *observer* (observador). Esse padrão também representa um ponto essencial na Programação Orientada a Eventos, sendo largamente utilizado [Ferg, 2006][Faison, 2006] [Samek, 2008][Hansen e Fossum, 2010][Salvaneschi e Mezini, 2014]. A Figura 12 apresenta a estrutura original do padrão de projeto *Observer*.

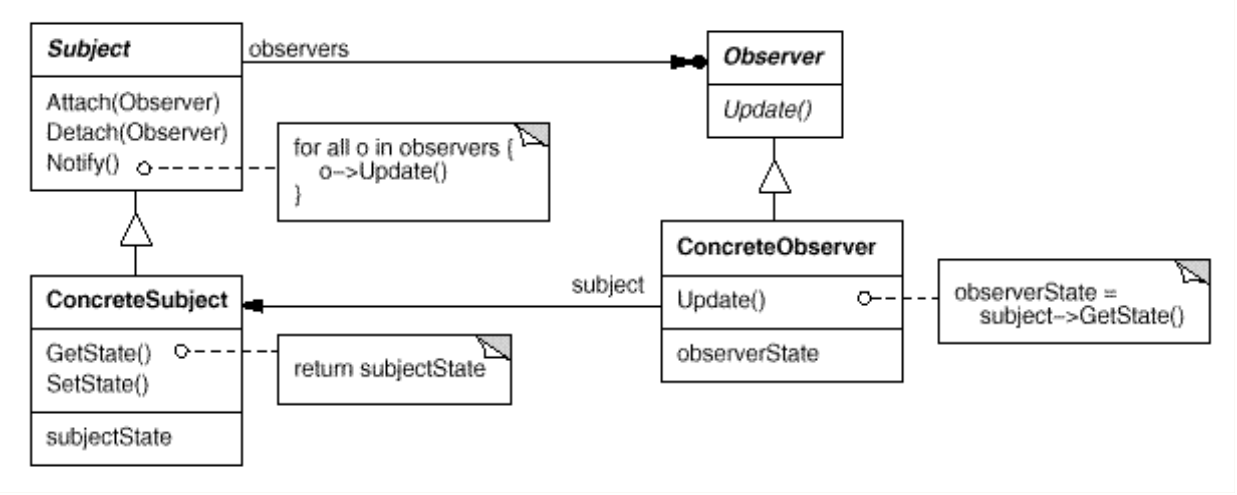

Figura 12: Diagrama de Classes original padrão de projeto *Observer* [Gamma *et al.*, 1995].

Como ilustrado na Figura 12, os participantes desse padrão são [Gamma *et al.*, 1995]:

- *Subject* (sujeito): conhece todos seus observadores, qualquer número de *Observers* pode observar um *Subject*. Define uma interface com a funcionalidade de anexar e desanexar observadores, por meio dos métodos *attach()* e *detach()*;
- *Observer* (classe abstrata observador): define uma interface de atualização para objetos que devem ser notificados sobre mudanças em um *Subject*;
- *ConcreteSubject* (subclasse sujeito concreto): armazena o estado de interesse para as instâncias concretas de subclasse *ConcreteObserver*. Envia notificações para seus observadores – *Observers* – quando seu estado muda;
- *ConcreteObserver* (subclasse observador concreto): mantém uma referência ao objeto *ConcreteSubject*. Armazena o estado que deve ser consistente com o estado do *Subject*. Implementa a interface de atualização *Observer* para manter essa consistência de estado com o *Subject*.

Basicamente, o funcionamento se inicia a partir de uma mudança de estado no *Subject*, que então percorre sua lista de *Observers* notificando-os sobre essa mudança de estado. Essa notificação pode conter os dados da mudança de estado ou o próprio estado (*e.g.* uma mensagem detalhada) caracterizando assim o chamado *push model*. Por outro lado, o *pull model* é o modelo no qual a notificação é mínima não enviando informações, cabendo ao *Observer* requisitar as informações de mudança. Em *pull model* as interações também são

chamadas de *publish-subscribe* (publicação e inscrição). O *Subject* é o publicador das notificações, enviando-as sem necessariamente conhecer os *Observers*. Os *Observers* é que se inscrevem para receber notificações sobre as mudanças de estado dos publicadores - *Subjects*.

A utilização desse padrão proporciona vantagens como: acoplamento abstrato entre *Subject* e *Observer* (objetiva mínimo acoplamento nas subclasses concretas), comunicação em *broadcast*, gerenciar interessados na comunicação dinamicamente em tempo de execução (funções anexar e desanexar) e bom desempenho (somente os interessados recebem as notificações) [Gamma *et al.*, 1995]. No entanto existem vários pontos de atenção: pode ocorrer baixo desempenho por atualizações inesperadas (toda notificação sempre é propagada para todos os observadores); referências perdidas podem ocorrer caso existam *Subjects* destruídos e os *Observers* busquem informações (assim é necessário assegurar a consistência dos *Subjects*); podem ser necessários mais componentes para gerenciar a comunicação (por exemplo em caso de dependências entre *Observers*) [Gamma *et al.*, 1995]; prejudica a legibilidade do programa por conta do registro dinâmico de objetos; efeitos colaterais em chamadas de métodos; estabelece inversão do controle (o controle está no *Observer* que executa conforme sua própria lógica) [Salvaneschi e Mezini, 2014]; e, por fim, aumenta a complexidade do código – conforme literatura este problema é denominado de *callback hell* [Edwards, 2009][Salvaneschi *et al.*, 2014b].

2.5 Reflexões sobre a Revisão do Estado da Arte

Neste capítulo, foram apresentados os conceitos relativos ao estudo de paradigmas de programação e foram detalhados os dois paradigmas que se pretende comparar: PON e POE, e mais três das técnicas primordiais de POE.

Os conceitos de PON objetivam construir *software* mais facilmente. Por meio da programação inspirada na forma declarativa (*e.g.* fatos e regras), no tocante à codificação, PON permite moldar uma solução em forma mais natural, encapsulando complexidades de execução mediante seu modelo de ciclo de notificações. Ademais, é caracterizado por uma pesquisa recente, extensa, sólida e emergente, incluindo pedidos de patente, artigos publicados em conferências e revistas, e um grupo de trabalho e pesquisa estabelecido. No estado da arte, apresenta um método de engenharia de *software* para o PON, o processo Desenvolvimento Orientado a Notificações (DON), evoluindo por meio de linguagem,

compilador e processador nativos, além de distribuição e paralelismo de *software* bem como em avanços da máquina de inferência (*e.g.* Lógica *Fuzzy*).

Por sua vez, POE anuncia vantagens como o menor acoplamento e maior encapsulamento, por meio da separação entre os componentes geradores de eventos e respectivos tratadores de eventos nos sistemas de *software*. É consolidado pela sua imensa literatura e seu vasto uso industrial, bem como por suas técnicas disseminadas mais comuns como *Handler-Dispatcher*, Máquina de Estados (*e.g. State Pattern*) e *Observer Pattern*.

Entretanto, não foram encontrados na literatura pesquisada processos de engenharia de *software* (exclusivamente) de sistemas orientados a eventos amplamente aceitas pela comunidade, apenas ferramentas como o Diagrama de Estados (*e.g. StateCharts*) [Harel, 1987] ou extensões da UML para a modelagem de sistemas orientados a eventos [Rumbaught *et al.*, 1999], ou modelagens como *Lollipop*, *SDL*, *Espresso*, *Catalysis*, *Signal Wiring* [Faison, 2006], e não um processo completo de engenharia de *software* explícito para eventos (em contrapartida ao DON).

Além disso, existe uma superposição de características entre o Paradigmas Orientado a Eventos e a Programação Reativa. Conforme a literatura, além da característica de "eventos", a principal nuança ou diferença entre a Programação Reativa e a Programação Orientada a Eventos, em sua essência, seria a característica inata comportamento (*i.e.* do inglês *behavior*) nos programas reativos (*e.g.* tempo em si). Samek, por exemplo, advoga em suas próprias palavras: "*Sistemas dirigidos a eventos são alternativamente denominados sistemas reativos*" [Samek, 2008] e vai além, chamando atenção ao prevalente paradigma "*evento-ação*" que reconhece unicamente a dependência entre um tipo de evento e uma ação (*i.e.* associação direcional) delegando o contexto para técnicas *adhoc*. Segundo o autor, "*evento-ação*" – do inglês *event-action* – seria uma simplificação de POE no qual cada tratador de eventos mistura conjuntos de estados não organizados [Samek, 2008].

Em uma análise crítica, almejando o conceito estrito, POE poderia ser ortogonal aos outros paradigmas de programação, uma vez que se realiza por meio deles[26]. Ou seja, é resolvido em outros paradigmas e, grosso modo, pelo suporte a eventos em cada linguagem de programação ("linguagens hospedeiras"). A diferenciação seria o modo de estruturar o pensamento – estímulos externos tratados em componentes computacionais – dessa forma modelando um problema, com seu modo de construção de *software*, com suas respectivas técnicas para tal. Como sistemas reativos são análogos aos sistemas orientados a eventos

26 Essa possível ortogonalidade conceitual não é exclusividade de POE, também ocorrendo com o Paradigma Orientado a Aspectos, tipagem em linguagens de programação, metaprogramação e DSL [Van Roy, 2009].

[Samek, 2008], o Paradigma Reativo (ou o Paradigma Orientado a Eventos) poderia ser o sobrenome dos Paradigmas em que se coaduna: Funcional *Reativo*, Síncrono *Reativo*, Lógico *Reativo*, Orientado a Notificações *Reativo*, OO *Reativo*, Imperativo *Reativo*.

Nesse sentido, seria necessário unificar o entendimento de POE ou Paradigma Reativo, o que pode ocorrer de maneira formal em trabalhos futuros – uma revisão sistemática aprofundada sobre o tema específico Paradigma Reativo extrapola o escopo desse trabalho.

Por fim, POE tem como alicerce o tratamento de eventos, que ocorrem em momentos imprevisíveis e definidos somente em tempo de execução. Em contrapartida, PON tem como alicerce construir *software* modelando o conhecimento por meio de regras, utilizando componentes computacionais de pequeno porte (*e.g. FBEs* e *Rules*) que se comunicam em um ciclo de notificações.

Assim, neste trabalho de pesquisa o conceito de POE refere-se ao "*POO Reativo*" e "*Paradigma Imperativo Reativo*" (em suas técnicas mais comuns da literatura, como *Handler-Dispatcher*, Máquina de Estados-*State Pattern* e *Observer Pattern*), que serão comparados ao "*PON Reativo*" (*i.e.* POO, PI e PON resolvendo eventos).

No capítulo seguinte são apresentados os casos de estudo desenvolvidos, tanto utilizando o POE em três de suas formas de implementação (*Handler-Dispatcher* (*event loop*), Máquina de Estados (*State Pattern*) e *Observer Pattern* (*callback programming*)), quanto em PON.

Capítulo 3 - Casos de estudo

Cabe retomar resumidamente os princípios para escolha dos casos de estudo desta dissertação (conforme outrora exposto na subseção 1.4.3 Casos de Estudo da seção 1.4 Métodos e Ferramentas). Inicialmente e conforme convenção definida, são casos clássicos de *softwares* para tratamento de eventos. Dessa forma, *software* que recebe eventos, os identifica e executa ações. Eles possuem as seguintes funcionalidades: tratamento de eventos discretos (eventos que ocorrem em ponto único no tempo) para tomada de decisão em *software*, tratamento de mesmos tipos de eventos recebidos em estados diferentes do *software*, tratamento de eventos que executam mais de uma ação ao mesmo tempo e *software* que trate evento externo e evento interno ao *software*.

Assim sendo, neste trabalho foram concebidos dois casos de estudo para tratamento de eventos em *software*. O primeiro, composto de três partes complementares, é denominado Simulador de Transporte Individual. Trata-se de um *software* exemplo para receber eventos, identificá-los, e executar decisões conforme requisitos (posteriormente nos experimento os eventos foram gerados de forma simulada). Para facilitar a analogia *evento-ação* em *software* (como mero conceito), simula a movimentação de um exosqueleto e seu respectivo braço mecânico (*i.e.* se enfatiza que é um simples tratador de eventos, sem interfaces com *hardware,* outros *softwares* ou componentes complexos).

Tal caso de estudo, no tocante ao POE, apresenta três cenários: o primeiro para utilizar a técnica *Handler-Dispatcher* (*event loop*), o segundo para utilizar a técnica Máquina de Estados (*State Pattern*) e o terceiro e último para utilizar o padrão *Observer Pattern* (*callback programming*). O objetivo é comparar as características de *software* para tratamento de eventos discretos e as respectivas formas de construí-lo em PON, separadamente de cada técnica POE.

Adiante, o segundo caso de estudo simula o tratamento de eventos em um *software* Portão Eletrônico (*i.e.* também como mero conceito, e.g. não há partes mecânicas), composto a partir dos requisitos expostos em trabalhos anteriores do grupo de pesquisa do PON como [Wiecheteck, 2011][Wiecheteck, Stadzisz e Simão, 2011][Batista, 2013]. Além do tratamento de eventos discretos e externos ao *software* recebidos para decisão, em diferentes estados do

software e executando mais de uma ação, se inclui um evento discreto interno do próprio *software* (*i.e.* avisos pontuais de marcação de tempo). Ressalta-se que a mudança de estado ao tratar eventos é característica notória deste segundo caso de estudo.

Assim, foi possível verificar a forma de programar *software* em ambos os paradigmas, ou seja, como se organizam tecnicamente os componentes computacionais para satisfazer esses propósitos determinados em eventos (*i.e.* requisitos). E, posteriormente, todas as comparações e medições realizadas se referem e se utilizaram dos casos de estudo aqui descritos, modelados e construídos.

Neste capítulo, as próximas seções que se seguem são: 3.1.1 Primeiro Cenário com suas duas soluções PON e *Handler-Dispatcher*; 3.1.2 Segundo Cenário com suas duas soluções PON e *State*; 3.1.3 Terceiro Cenário com suas duas soluções PON e *Observer*; e 3.2 Caso de Estudo 2: Portão Eletrônico com suas quatro soluções PON, *Handler-Dispatcher, State* e *Observer*. Por fim, são apresentadas reflexões acerca dos casos de estudo na seção 3.3.

3.1 Caso de Estudo 1: Simulador de Transporte Individual

Esta seção se apresenta o caso de estudo de um *software* simulador em três cenários, de *software* para tratamento de eventos gerados tanto manualmente quanto artificialmente. A avaliação dos resultados dos experimentos é apresentada na seção 4.3. A inspiração para este primeiro caso surgiu de um cenário fictício onde um piloto guia um exosqueleto e seu(s) respectivo(s) braço(s) mecânico(s). Não foi realizada implementação efetiva de nenhum *hardware* real, nem feita qualquer simulação física de tal equipamento. A origem desse caso de estudo 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 este caso de estudo.

Na Figura 13 é apresentado um desenho conceitual de um possível exosqueleto, adaptado do projeto *Hardiman* I (anos 1967-1971), que foi uma das primeiras tentativas de construir um exosqueleto energizado [Fick e Makinson, 1971].

Figura 13: Figura conceitual de um exosqueleto adaptada do projeto Hardiman I da General Electric [Fick e Makinson, 1971].

De maneira geral, este caso de estudo caracteriza-se por ser um *software* simulador que recebe eventos de dispositivos de entrada como uma chave liga e desliga, *joystick* e seus respectivos botões. Por simplicidade, os eventos provenientes destes dispositivos foram também simulados por meio da criação de um *software* "gerador de eventos", que pode ser programado para geração dos respectivos eventos segundo sequências pré-determinadas. Isto permite que haja uma variação da configuração entre cenários (com alterações de requisitos, como por exemplo o uso de dois *joysticks*, ou mais botões). Desta forma, o primeiro cenário foca em uma simples sequência de eventos recebidos para tomada de decisão em *software*, o segundo cenário trata eventos de mesmo tipo recebidos em estados diferentes do *software*, e o terceiro e último cenário trata eventos recebidos que executam mais de uma ação. Os programas que controlam os diversos cenários foram desenvolvidos tanto em POE quanto em PON.

É possível fazer uma relação histórica entre os três cenários e técnicas originais em POE. O primeiro cenário usa somente um *joystick*, movimenta hipoteticamente (característica também dos próximos cenários) somente o transporte, e a técnica POE utilizada é *Dispatcher* (*event loop*) [Ferg. 2006][Samek, 2008][Hansen e Fossum, 2010]. O segundo cenário usa um *joystick* e um botão, move tanto o exosqueleto quanto um único braço mecânico, e a técnica POE é Máquina de Estados [Faison, 1993][Samek, 2008]. O terceiro cenário usa um *joystick* e dois botões, move tanto o exosqueleto quanto ambos os braços mecânicos, e a técnica POE é *Observer* (*callback programming*) [Gamma, 1995][Ferg, 2006][Faison, 2006][Samek, 2008]. A cada cenário é possível inferir o uso de cada técnica para resolver cada um dos requisitos específicos, dessa forma contribuindo também para o entendimento de como se programa em POE, e também em PON.

3.1.1 Primeiro Cenário

Os requisitos para o primeiro cenário do *software* Simulador de Transporte Individual são:

- O *software* simulador deverá receber eventos de um *joystick* e de uma chave liga e desliga.
- A chave liga e desliga o exosqueleto (respectivamente chave fechada e chave aberta).
- O *joystick* movimenta o exosqueleto nos eixos X e Y (conforme relação de movimentos da Figura 14).
- Os eventos recebidos da chave e do *joystick* são tratados e interpretados pelo *software* para tomada de decisão (sair da simulação, ligar e desligar o exosqueleto, mover o exosqueleto).

A Figura 14 mostra graficamente (conceitualmente) uma chave liga-desliga, um *joystick* (somente manche - sem botões) e a relação de movimentos do *joystick* nos eixos X e Y para mover o exosqueleto.

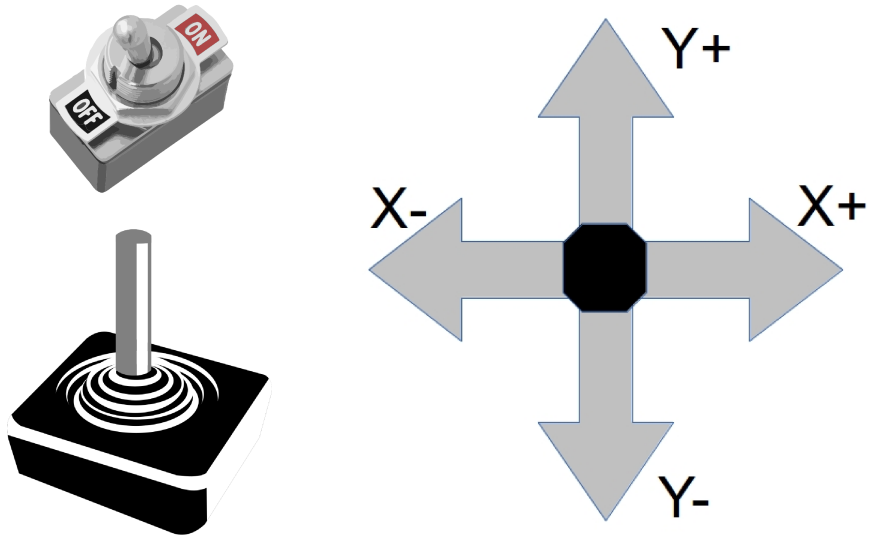

Figura 14: Chave ON/OFF, Joystick (sem botões), e movimentos do Joystick para mover o transporte no primeiro cenário.

A Figura 15 mostra as funções do exosqueleto, neste primeiro cenário, por meio de um diagrama de caso de uso em UML.

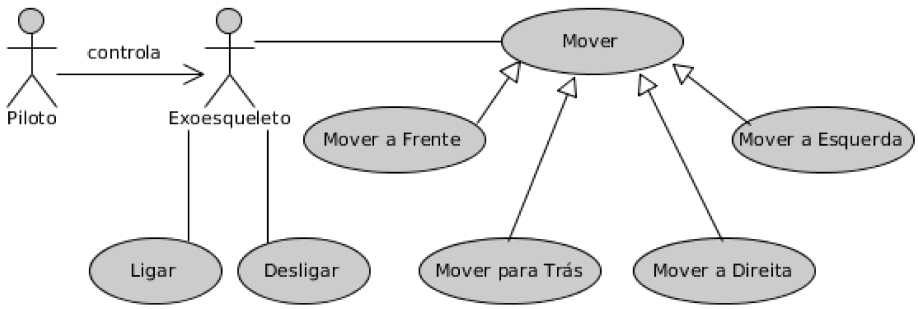

Figura 15: Funções do exosqueleto no ambiente de simulação do primeiro cenário.

A Figura 16 apresenta o diagrama de atividades em UML para este cenário do caso de estudo, contendo os detalhes de eventos necessários para as decisões nos *decision nodes*.

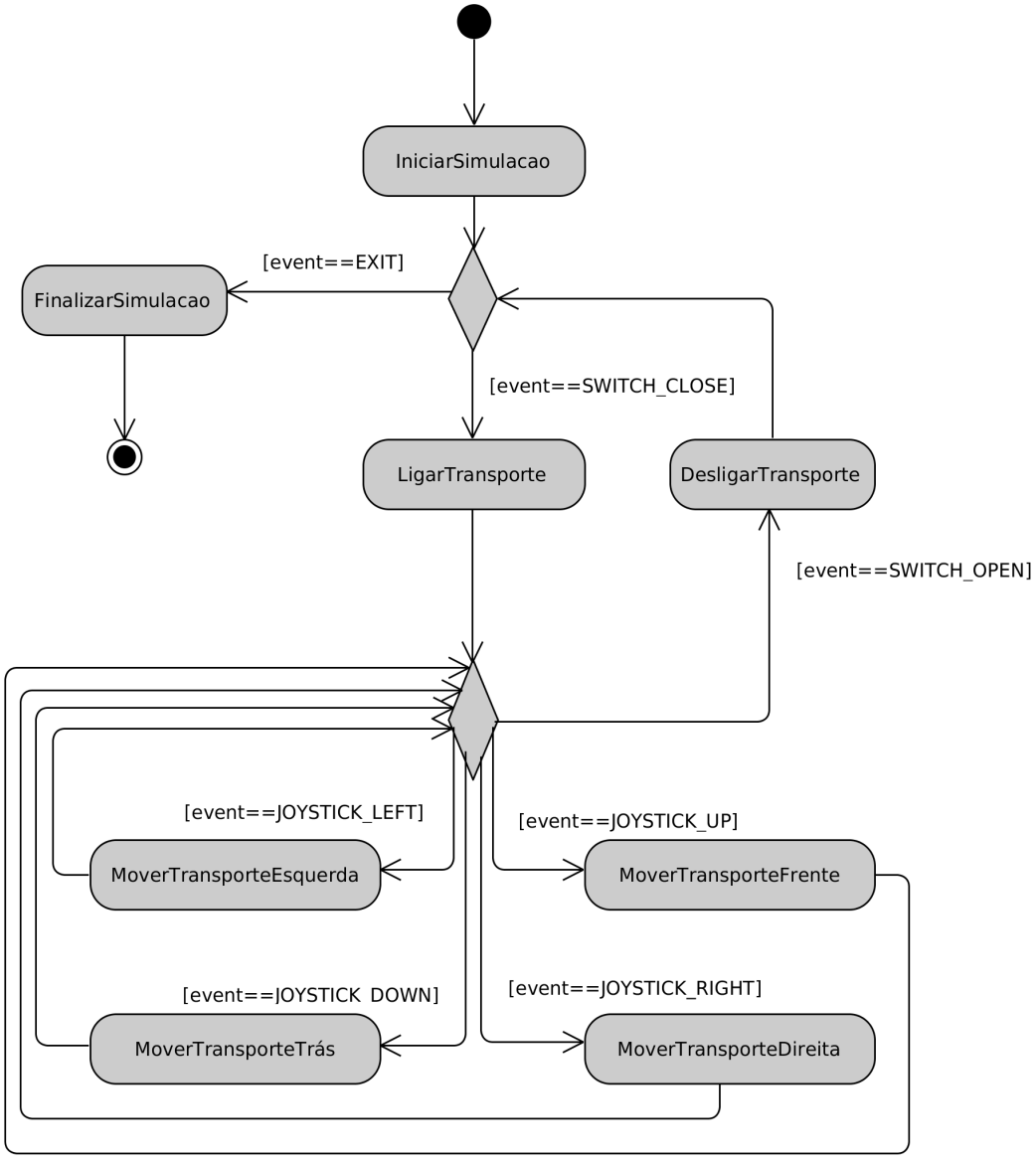

Figura 16: Progressão da simulação em atividades do primeiro cenário.

No diagrama de atividades em UML, o nó de decisão (*decision node*) demonstra graficamente uma tomada de decisão, para escolha de atividades subsequentes (ações), baseada em um conjunto de condições (eventos necessários). Neste sentido, após a tomada de decisão, uma ação é executada no ambiente de simulação. Dessa forma, no *software* em execução, os eventos recebidos da chave e do *joystick* são utilizados para tomada de decisão.

A seguir são apresentadas as soluções implementadas para este primeiro cenário, tanto em PON (Solução I) quanto em POE (Solução *Handler-Dispatcher*).

(I) Solução PON para o Primeiro Cenário

Nesta seção é apresentada a solução PON para o primeiro cenário. Em correlação com trabalhos anteriores como [Simão *et al.*, 2012a], a Figura 17 apresenta o modelo estrutural em PON por meio de um diagrama de classes UML, identificando os elementos que compõe a base de fatos FBE – estereótipo <<NOP_FBE>> – e a aplicação PON (*i.e.* estereótipo <<NOP_Application>>). Ressaltam-se as classes em destaque (cor branca) que tem origem no *Framework* PON Otimizado desenvolvido em C++.

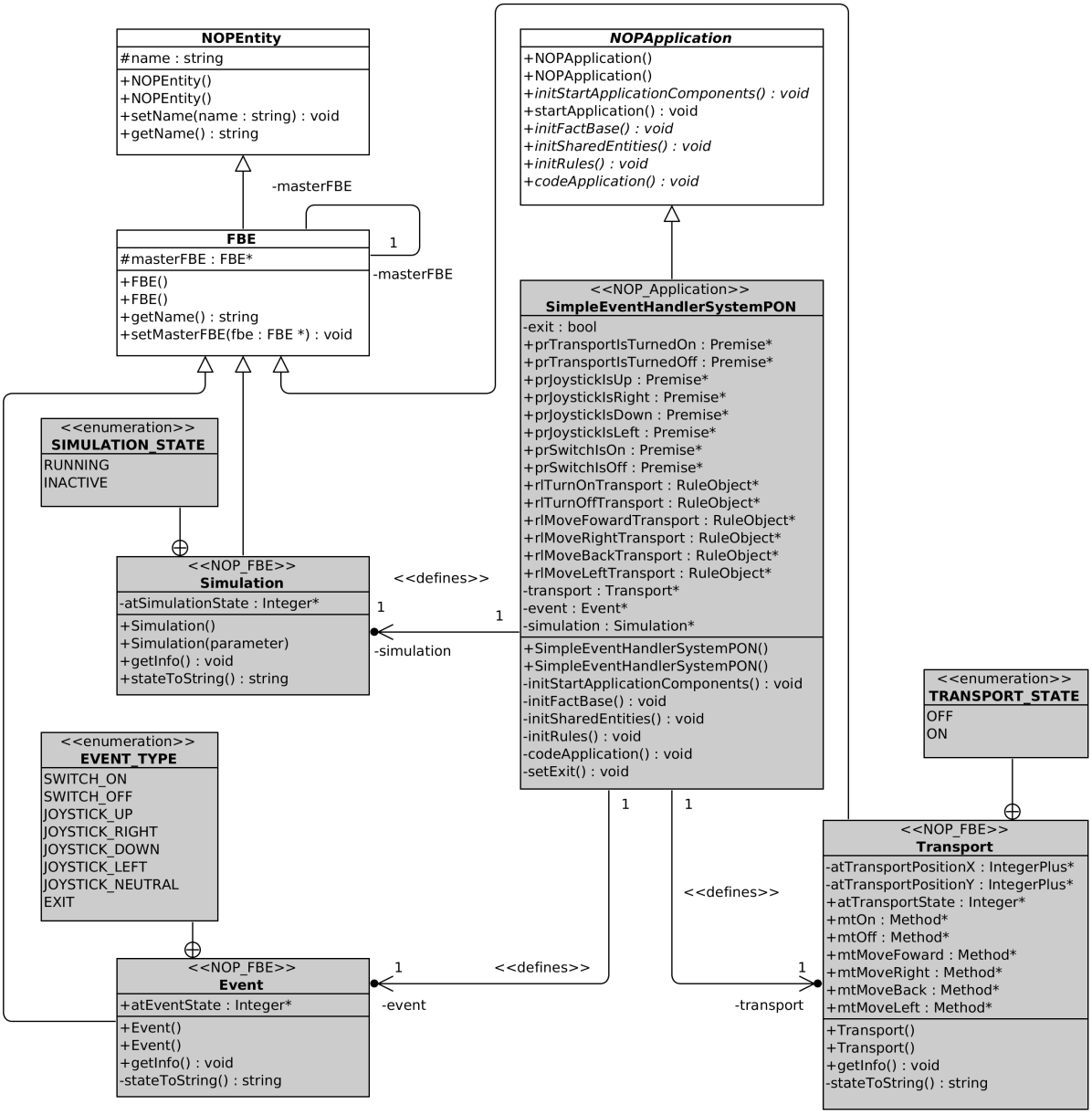

Figura 17: Diagrama de Classes do *software* em PON do primeiro cenário.

Conforme o diagrama exposto na Figura 17, três *FBEs* foram modelados, a saber: *Transport*, *Event* e *Simulation*. *Transport* representa o exosqueleto, *Simulation* representa a simulação do *software* e *Event* representa os eventos recebidos. A classe *SimpleEventHandlerSystemPON* é o componente de *software* responsável por criar, iniciar, associar os outros componentes de *software* da solução (*i.e.* análogo à classe Principal em OO).

Além disso, enumeradores (*i.e.* C++ *enumerations*) foram utilizadas para compor os tipos de estado dos objetos (*TRANSPORT_STATE*, *SIMULATION_STATE* e *EVENT_TYPE*). *Transport* contém o enumerador *TRANSPORT_STATE* que representa seus estados Ligado e Desligado (*ON* e *OFF*), *Simulation* contém o enumerador *SIMULATION_STATE* que representa seus estados Rodando e Inativo (*RUNNING* e *INACTIVE*), e por fim, *Event* utiliza o enumerador *EVENT_TYPE* que enumera os eventos possíveis para tratamento pelo *software* neste cenário (*SWITCH_ON*, *SWITCH_OFF*, *JOYSTICK_UP*, *JOYSTICK_RIGHT*, *JOYSTICK_DOWN*, *JOYSTICK_LEFT*, *JOYSTICK_NEUTRAL*, *EXIT*).

Como demonstrado em [Wiecheteck, 2011] e [Batista, 2013], a partir de uma atividade de análise correlacionando os elementos <<*NOP_FBE*>> e o Diagrama de Atividades, o levantamento e a criação das regras são realizados para o *software* em PON. Perguntas básicas auxiliam neste levantamento:

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

Exemplificando, aqui se enumeram duas regras utilizando essas perguntas neste primeiro cenário do caso de estudo:

1) Regra: Ligar o Transporte

Qual o objetivo da regra? Ligar o Transporte.

O que precisa acontecer para que a regra seja executada? A simulação deve estar RODANDO, o Transporte deve estar DESLIGADO e o evento recebido deve ser CHAVE LIGADA.

O que acontece se a regra for executada? O Transporte muda para o estado LIGADO.

2) Regra: Mover o Transporte para Frente

Qual o objetivo da regra? Mover o Transporte para frente.

O que precisa acontecer para que a regra seja executada? O transporte deve estar LIGADO e o evento recebido deve ser JOYSTICK PARA FRENTE (*JOYSTICK_UP*).

O que acontece se a regra for executada? O Transporte anda para frente (move-se no eixo Y positivo).

Ou seja, as *Rules* são identificadas em correspondência com o Diagrama de Atividades em UML (Figura 16), conforme definido no processo de *software* Desenvolvimento Orientado a Notificações (DON) [Wiecheteck *et al.*, 2011]. Assim, para cumprir os requisitos do caso de estudo, foram implementadas 6 *Rules* listadas na Tabela 2.

Tabela 2: *Rules*, *Condition* e suas *Premises, Methods* instigados do *software* do primeiro cenário exosqueleto em PON.

Rule **Nome**	***Condition* e suas *Premises***	***Method* instigado**
1 rlTurnOnExoskeleton	atSimulationState == RUNNING && atTransportState == OFF && atEventState == SWITCH_ON	Exoskeleton→mtOn
2 rlTurnOffExoskeleton	atSimulationState == RUNNING && atTransportState == ON && atEventState == SWITCH_OFF	Exoskeleton→mtOff
3 rlMoveForwardExoskeleton	atSimulationState == RUNNING && atTransportState == ON && atEventState == JOYSTICK_UP	Exoskeleton→mtForward
4 rlMoveRightExoskeleton	atSimulationState == RUNNING && atTransportState == ON && atEventState == JOYSTICK_RIGHT	Exoskeleton→mtRight
5 rlMoveBackExoskeleton	atSimulationState == RUNNING && atTransportState == ON && atEventState == JOYSTICK_DOWN	Exoskeleton→mtBack
6 rlMoveLeftExoskeleton	atSimulationState == RUNNING && atTransportState == ON && atEventState == JOYSTICK_LEFT	Exoskeleton→mtLeft

Em seguida, no Algoritmo 3, se enfatiza um dos aspectos técnicos em PON: Premissas Compartilhadas (*Shared Premises*). São *Premises* utilizadas por mais de uma *Rule*. Esse código da implementação pertence ao contexto de inicialização do *software* em PON, referente aos estados rodando e inativo de *Simulation* e aos estados ligado ou desligado de *Transport*.

Algoritmo 3: Inicialização de Premissas Compartilhadas em SimpleEventHandlerPON.cpp.

```
41 void SimpleEventHandlerSystemPON::initSharedEntities() {
42
43        PREMISE(prSimulationIsRunning, simulation->atSimulationState,
                  Simulation::RUNNING, Premise::EQUAL, Premise::STANDARD, false);
          PREMISE(prSimulationIsInactive, simulation->atSimulationState,
44                Simulation::INACTIVE, Premise::EQUAL, Premise::STANDARD, false);

          PREMISE(prExoskeletonIsTurnedOff, exoskeleton->atExoskeletonState,
                  Exoskeleton::OFF, Premise::EQUAL, Premise::STANDARD, false);
45        PREMISE(prExoskeletonIsTurnedOn, exoskeleton->atExoskeletonState,
                  Exoskeleton::ON, Premise::EQUAL, Premise::STANDARD, false);
46 ...

47
```

No Algoritmo 4, é possível conferir o código de criação de três *Rules* (de um total de seis regras para a solução do problema) codificadas no *Framework* otimizado. Essas *Rules* referem-se respectivamente ao Ligar (chave fechada), Desligar (chave desligada) e Mover para Frente (*joystick* para frente) o transporte genérico – *Transport*.

Algoritmo 4: Código de três regras em SimpleEventHandlerPON.cpp.

```
68 ...
69 RULE(rlTurnOnExoskeleton, scheduler, Condition::CONJUNCTION);
70        rlTurnOnExoskeleton->addPremise(prSimulationIsRunning);
71        rlTurnOnExoskeleton->addPremise(prExoskeletonIsTurnedOff);
72        rlTurnOnExoskeleton->addPremise(prKeyIsClosed);
73        rlTurnOnExoskeleton->addInstigation(INSTIGATION(exoskeleton->mtOn));
74
75 RULE(rlTurnOffExoskeleton, scheduler, Condition::CONJUNCTION);
76        rlTurnOffExoskeleton->addPremise(prSimulationIsRunning);
77        rlTurnOffExoskeleton->addPremise(prExoskeletonIsTurnedOn);
78        rlTurnOffExoskeleton->addPremise(prKeyIsOpen);
79        rlTurnOffExoskeleton->addInstigation(INSTIGATION(exoskeleton->mtOff));
80
81 RULE(rlMoveForwardExoskeleton, scheduler, Condition::CONJUNCTION);
82   rlMoveForwardExoskeleton->addPremise(prSimulationIsRunning);
83   rlMoveForwardExoskeleton->addPremise(prExoskeletonIsTurnedOn);
84   rlMoveForwardExoskeleton->addPremise(prJoystickIsUp);
85   rlMoveForwardExoskeleton->addInstigation(INSTIGATION(exoskeleton->mtMoveForward));
86 ...
```

Indo além, essa primeira solução PON do primeiro cenário também fez uso do Compilador PON (também chamado PON-Compilador) em ativo desenvolvimento pelo grupo de pesquisa do PON [Ferreira *et al.*, 2013] e da respectiva linguagem LingPON. A especificação formal da linguagem está no Anexo A - Especificação Formal da Linguagem PON. O Algoritmo 5 ilustra o *FBE exoskeleton* e as mesmas *Rules* do Algoritmo 4.

Algoritmo 5: Código-fonte em LingPON em exoskeleton.pon

```
  1 fbe Exoskeleton
  2     attributes
  3         integer atExoskeletonState 0
  4         integer atExoskeletonPositionX 0
  5         integer atExoskeletonPositionY 0
  6     end_attributes
  7     methods
  8         method mtOn(atExoskeletonState = 0)
  9         method mtOff(atExoskeletonState = 1)
 10         method mtMoveForward(atExoskeletonPositionY = atExoskeletonPositionY + 1)
 11         method mtMoveRight(atExoskeletonPositionX = atExoskeletonPositionX + 1)
 12         method mtMoveBack(atExoskeletonPositionY = atExoskeletonPositionY - 1)
 13         method mtMoveLeft(atExoskeletonPositionX = atExoskeletonPositionX - 1)
 14     end_methods
 15 end_fbe

... ...

 45 rule rlTurnOnExoskeleton
 46     condition
 47         subcondition A1
 48                 premise prSimulationIsRunning simulation.atSimulationState == 1 and
 49                 premise prExoskeletonIsTurnedOff exoskeleton.atExoskeletonState == 0 and
 50                 premise prKeyIsClosed event.atEventState == 2
 51         end_subcondition
 52     end_condition
 53     action
 54         instigation inOn exoskeleton.mtOn();
 55     end_action
 56 end_rule
 57
 58 rule rlTurnOffExoskeleton
 59     condition
 60         subcondition A2
 61                 premise prSimulationIsRunning simulation.atSimulationState == 1 and
 62                 premise prExoskeletonIsTurnedOn exoskeleton.atExoskeletonState == 1 and
 63                 premise prKeyIsOpen event.atEventState == 1
 64         end_subcondition
 65     end_condition
 66     action
 67         instigation inOff exoskeleton.mtOff();
 68     end_action
 69 end_rule
 70
 71 rule rlMoveForwardExoskeleton
 72         condition
 73                 subcondition A3
 74                         premise prSimulationIsRunning simulation.atSimulationState == 1 and
 75                  premise prExoskeletonIsTurnedOn exoskeleton.atExoskeletonState == 1 and
 76                  premise prJoystickIsUp event.atEventState == 3
 77         end_subcondition
 78     end_condition
 79     action
 80         instigation inForward exoskeleton.mtMoveForward();
 81     end_action
 82 end_rule
 83
 84 ...
```

(II) Solução POE via despachante – *Dispatcher Handler* (*event loop*)

Ao seu turno, em POE, na fase de projeto do *software*(*Design*), também se realiza uma atividade de análise (assim como em PON). Nesse sentido, o objetivo é criar, associar e compor elementos computacionais para atender os requisitos necessários do problema, *i.e.*

tratamento de eventos, assim construindo o *software* em Orientação a Eventos. O diagrama de sequência (conceitual) da Figura 18 apresenta o tratamento de um evento gerado por um dispositivo (*e.g.* chave ou *joystick*) pelo *software,* que deve executar uma das ações de *Transport* (*Exoskeleton*).

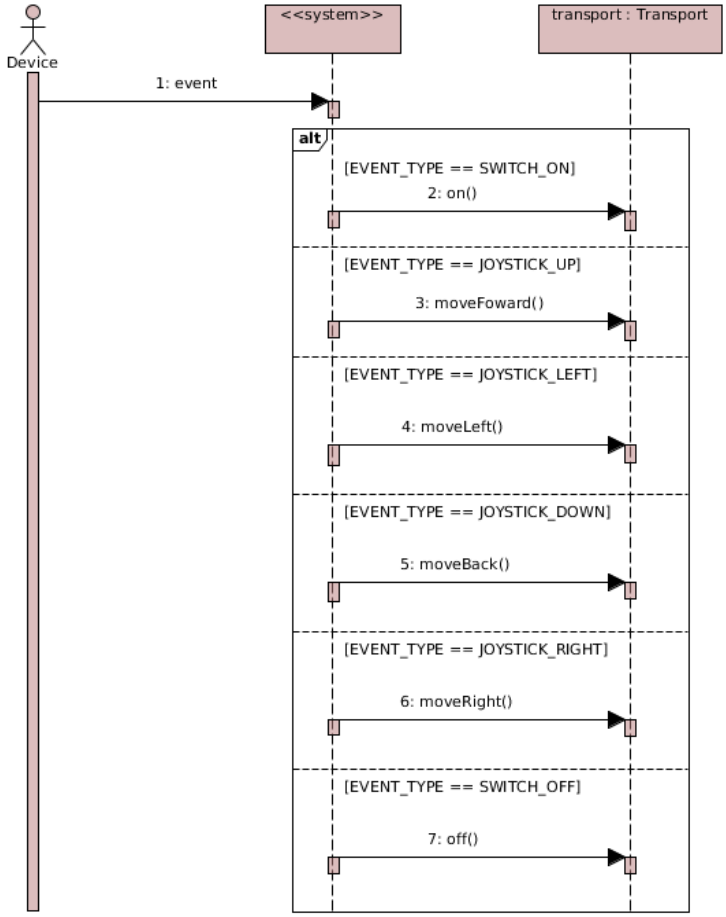

Figura 18: Diagrama de Sequência relacionando dispositivo gerador de eventos, *software* e *Transport*.

Na solução técnica aqui adotada em POE – *Handler-Dispatcher*–, o objeto transmissor é associado indiretamente ao objeto receptor interessado. Dessa forma, a relação entre os objetos transmissores e objetos receptores é indireta, mediante um objeto (ou componente) despachante – *Dispatcher Handler* [Ferg, 2006][Samek, 2008][Hansen e Fossum, 2010]. Faison também chama esse componente de Coordenador, do inglês *Coordinator*. Este elemento possui um propósito específico, que é coordenar as ações de um ou mais componentes trabalhadores (*i.e. workers*) atribuídos a ele [Faison, 2006]. Segundo Faison, os

componentes que funcionam como *Coordinators* possuem vários nomes na literatura como *handlers*, *relayers*, *mediators*, *adapters*, *dispatchers*, *façades* e *liaisons*.

Dessa maneira, em POE, para realizar o primeiro cenário do caso de estudo Simulador de Transporte Individual foram construídas três classes em C++: *Transport* (*Exoskeleton*); *Dispatcher* (componente despachante ou *Coordinator*), e *DispatcherHandlerSystem* (*i.e.* classe principal de sistema, responsável também por executar o *binding* – relacionamento entre classes e receber os eventos). Além disso, enumerações em C++ (*enumerations)* foram utilizadas para compor os estados dos objetos (*TRANSPORT_STATE* e *EVENT_TYPE*), conforme o diagrama exposto na Figura 19.

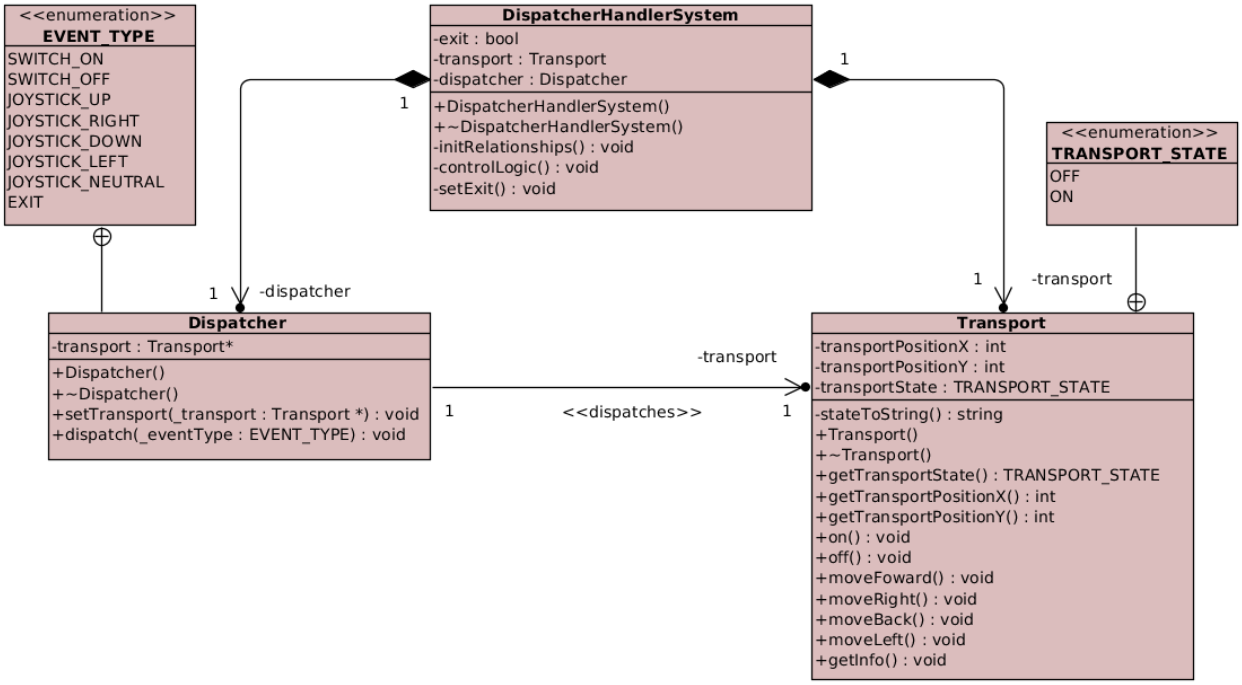

Figura 19: Diagrama de Classe Solução Dispatcher Handler em POE para o Primeiro Cenário. O componente Dispatcher propicia uma relação indireta.

Os principais autores em POE indicam alguns caminhos para o projeto de *software* Orientado a Eventos. Faison aponta que diagramas de sequência são ótimas ferramentas para modelar casos de uso, entretanto são pobres para modelar pontos de decisão em *software* [Faison, 2006]. Em definição análoga à própria UML, Hansen e Fossum instruem que os diagramas de sequência da UML mostram a sequencialidade de eventos e chamadas de métodos durante uma interação [Hansen e Fossum, 2010]. Já Samek preconiza que "*um bom ponto de partida no design de qualquer sistema orientado a eventos é desenhar diagramas de*

sequência para os principais cenários identificados na especificação do problema" [Samek, 2008]. Dessa forma, se percebe uma correlação entre estudar uma sequencialidade de eventos (*e.g.* uma interação) e a construção de *software* POE. Neste sentido, os autores aconselham a modelagem UML via diagrama de sequência.

A partir do diagrama de sequência exposto na Figura 18 e utilizando a técnica *Dispatcher*, se compõe um diagrama de sequência dessa solução, que está apresentado na Figura 20. Este diagrama apresenta uma interação do projeto estrutural para esta solução no primeiro cenário. Desse modo, os objetos transmissores não têm conhecimento dos objetos receptores, gerando desacoplamento. O *Dispatcher* seria o responsável por endereçar e redirecionar as chamadas, desacoplando os eventos das ações ao aplicar um ciclo de reconhecimento de eventos (*event loop*). Assim, o componente *Dispatcher* propicia relação indireta e também concentra o conhecimento do que deve ser executado, funcionando como um demultiplexador [Samek, 2008]. A interação demonstrada executa a seguinte sequência de eventos recebidos: *SWITCH_ON*, *JOYSTICK_UP*, *JOYSTICK_LEFT*, *JOYSTICK_UP*, *JOYSTICK_RIGHT*, *SWITCH_OFF*. Ou seja, liga o exosqueleto, move o exosqueleto para a frente, move o exosqueleto para a esquerda, move o exosqueleto novamente para a frente, move o exosqueleto para a direita, e por fim desliga o exosqueleto.

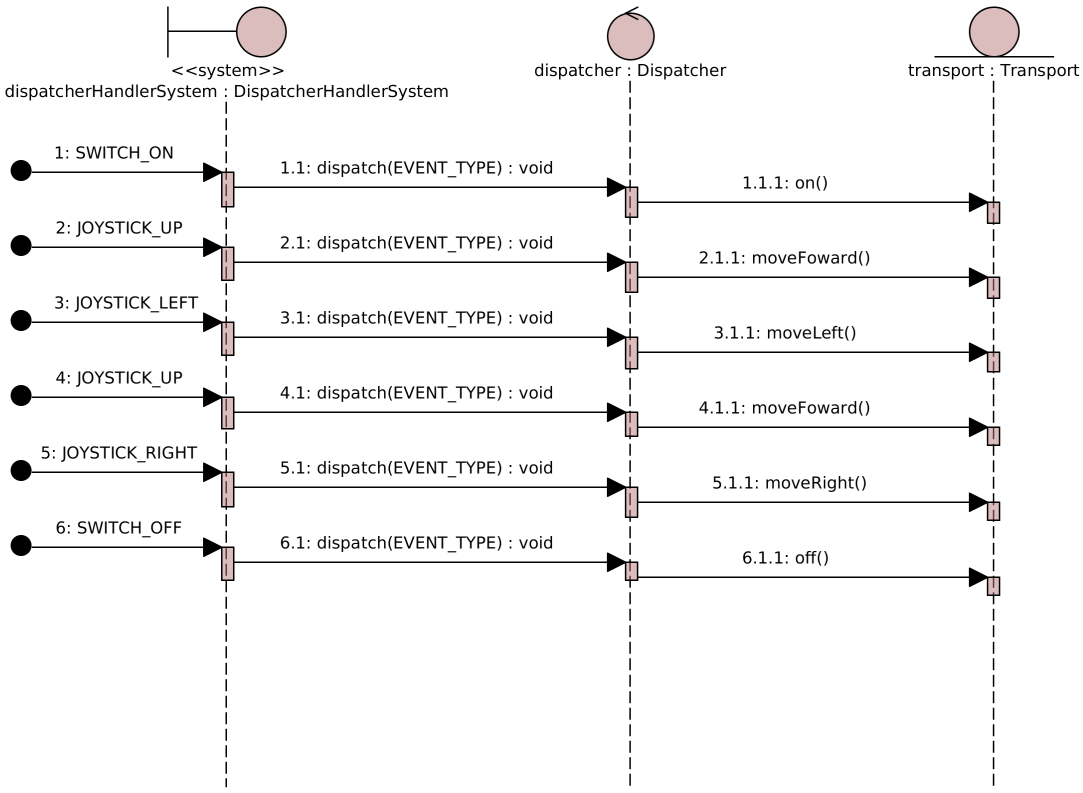

Figura 20: Diagrama de Sequência Solução *Dispatcher Handler* em POE para o primeiro cenário.

Uma possível solução *Dispatcher* é construir uma sequência de avaliações causais para detecção de eventos – barramento comum de eventos (*i.e. channel*) ou demultiplexador – dentro de um ciclo de execução, conforme Algoritmo 6. Essa construção demonstra a sequencialidade imprevisível de comandos em POE. A solução adotada também evidencia redundâncias de avaliações causais. Ainda assim, evita avaliar todas as condições fazendo uso de cláusulas *ELSE* [Banaszewski, 2009].

Algoritmo 6: Primeira versão do código para execução de ações ou retransmissão de mensagem via objeto Despachante em Dispatcher.cpp.

```
...
void Dispatcher::dispatch(Dispatcher::EVENT_TYPE _eventType) {
	if (_eventType == Dispatcher::SWITCH_OPEN) {
		this->transport->off();
	} else if (_eventType == Dispatcher::SWITCH_CLOSE) {
		this->transport->on();
	} else if (_eventType == Dispatcher::JOYSTICK_UP) {
		this->transport->moveForward();
	} else if (_eventType == Dispatcher::JOYSTICK_RIGHT) {
		this->transport->moveRight();
	} else if (_eventType == Dispatcher::JOYSTICK_DOWN) {
		this->transport->moveBack();
	} else if (_eventType == Dispatcher::JOYSTICK_LEFT) {
		this->transport->moveLeft();
	}
}
...
```

Logicamente, existem implementações mais otimizadas que a apresentada, como uma estrutura de controle *SWITCH-CASE* ou o uso de uma tabela *hash* (*i.e.* tupla chave-ação) [Samek, 2008][Banaszewski, 2009]. Assim, a versão definitiva de *Dispatcher* para resolver este caso de estudo em POE se utiliza da estrutura de controle *SWITCH-CASE*, conforme Algoritmo 7. Ainda se mantém a característica a imprevisibilidade de sequência de comandos em POE. Importante frisar a correlação desta estrutura com o funcionamento de uma máquina de estados. Samek elenca a utilização de uma estrutura aninhada de *SWITCH-CASE* como uma das maneiras típicas de compor uma máquina de estado hierárquica e programar *software* que trata eventos. As outras maneiras, segundo o autor, seriam tabela de estados (*i.e.* análogo a uma matriz ou tabela com ponteiros para funções – *state table*), o padrão de projeto Orientado a Objetos *State* [Faison, 1993][Gamma *et al.*, 1995] ou combinações destas técnicas [Samek, 2008].

Algoritmo 7: Versão definitiva do código para execução de ações ou retransmissão de mensagem via objeto Despachante em Dispatcher.cpp utilizando *SWITCH-CASE*.

```
...
void Dispatcher::dispatch(EVENT_TYPE _eventType) {
	switch (_eventType) {
		case (SWITCH_OPEN) : {
			this->transport->off();
			break;
		}
		case (SWITCH_CLOSE) : {
			this->transport->on();
			break;
		}
		case (JOYSTICK_UP) : {
			this->transport->moveForward();
			break;
		}
		case (JOYSTICK_RIGHT) : {
			this->transport->moveRight();
			break;
		}
		case (JOYSTICK_DOWN) : {
			this->transport->moveBack();
			break;
		}
		case (JOYSTICK_LEFT) : {
			this->transport->moveLeft();
			break;
		}
		default : {
			break;
		}
	}
}
...
```

A próxima seção que se segue é 3.1.2 Segundo Cenário com suas duas soluções PON e POE *State*.

3.1.2 Segundo Cenário

Aqui se apresenta o segundo cenário no qual, conceitualmente, um piloto guia um exosqueleto e seu respectivo braço mecânico (*i.e.* um único braço mecânico). Os requisitos para o segundo cenário do *software* Simulador de Transporte Individual são:

- O *software* simulador deverá receber eventos de um *joystick* com um botão e de uma chave liga e desliga.
- A chave liga e desliga (respectivamente chave fechada e chave aberta) o exosqueleto como um todo, incluindo o braço mecânico.

- O botão do *joystick* liga e desliga o braço mecânico do exosqueleto. A Figura 21 mostra graficamente (conceitualmente) uma chave liga-desliga e um *joystick* com um único botão.

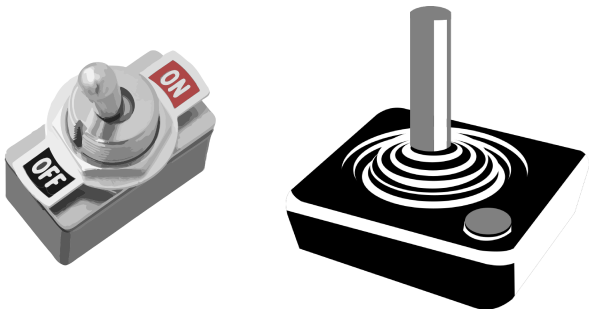

Figura 21: Chave ON/OFF e Joystick com um único botão para mover o exosqueleto (ou o braço mecânico) no segundo cenário.

- O *joystick* movimenta ora o próprio transporte exosqueleto ora o braço mecânico, caso esse esteja ligado. Dessa forma, o *joystick* movimenta o exosqueleto nos eixos X e Y, bem como controla o braço mecânico nos eixos X, Y e Z, conforme Figura 22.

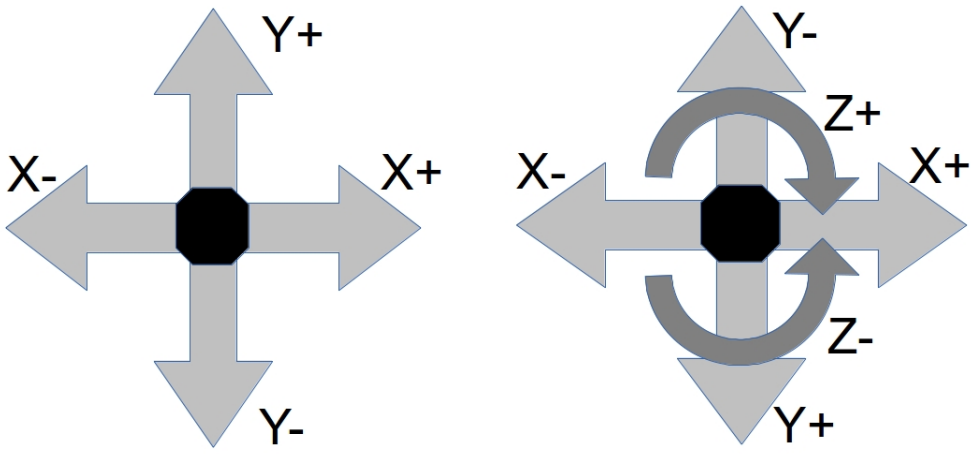

Figura 22: Movimentos do joystick para mover o exosqueleto (esquerda) ou o braço mecânico (direita) no segundo cenário.

A Figura 23 mostra as funções do simulador, neste segundo cenário, por meio de diagramas de caso de uso em UML.

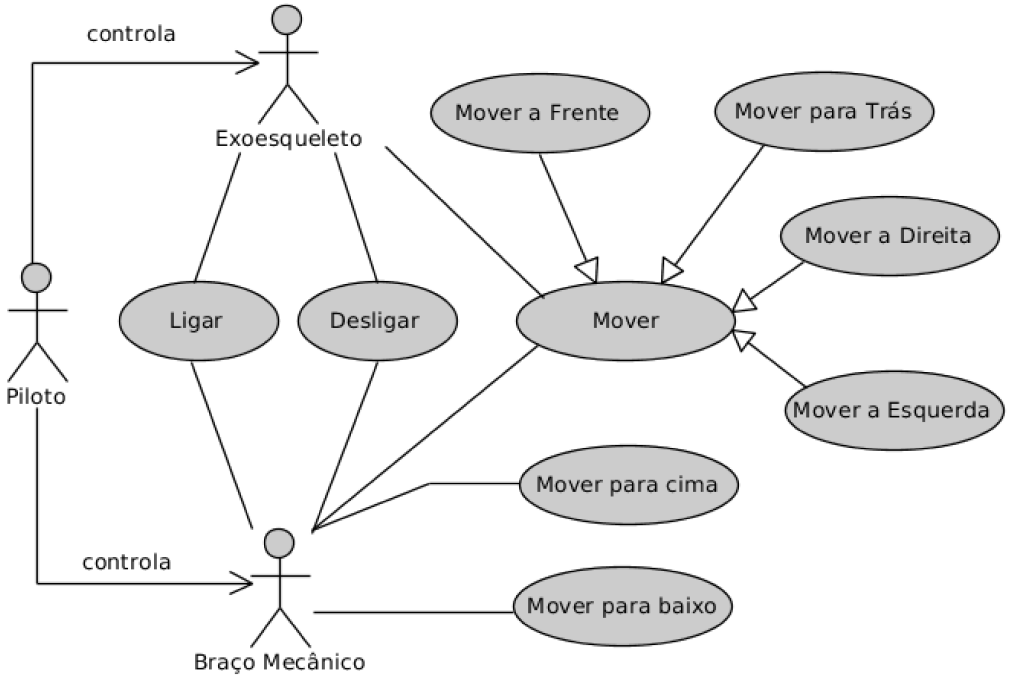

Figura 23: Funções do exosqueleto e braço mecânico para o segundo cenário.

A Figura 24 apresenta o diagrama de atividades em UML para este segundo cenário do caso de estudo em uma visão resumida.

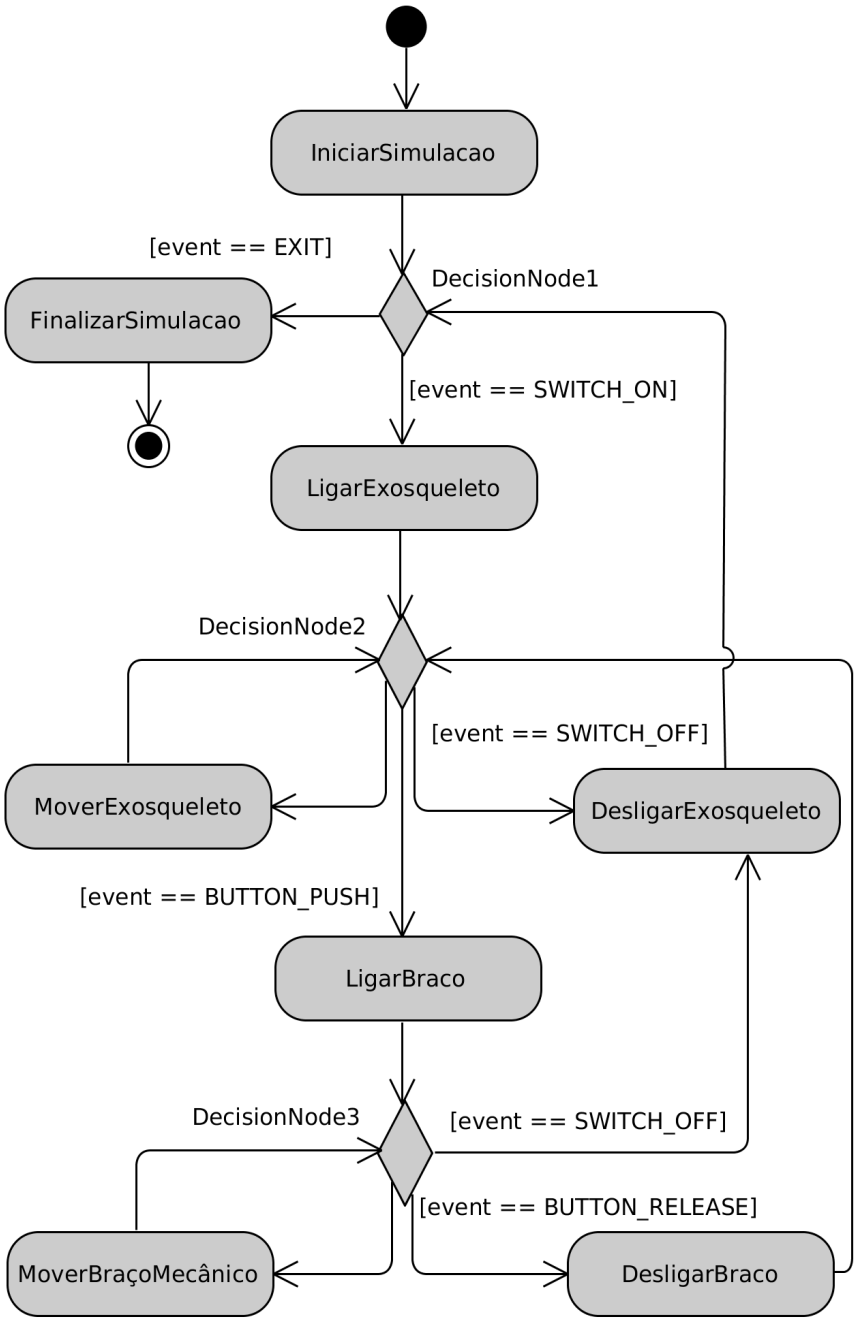

Figura 24: Progressão da simulação em atividades (visão resumida).

A Figura 25 demonstra a visão em detalhe, também em diagrama de atividades UML, para mover o braço mecânico, contendo os eventos necessários para as decisões a partir dos *decision nodes.*

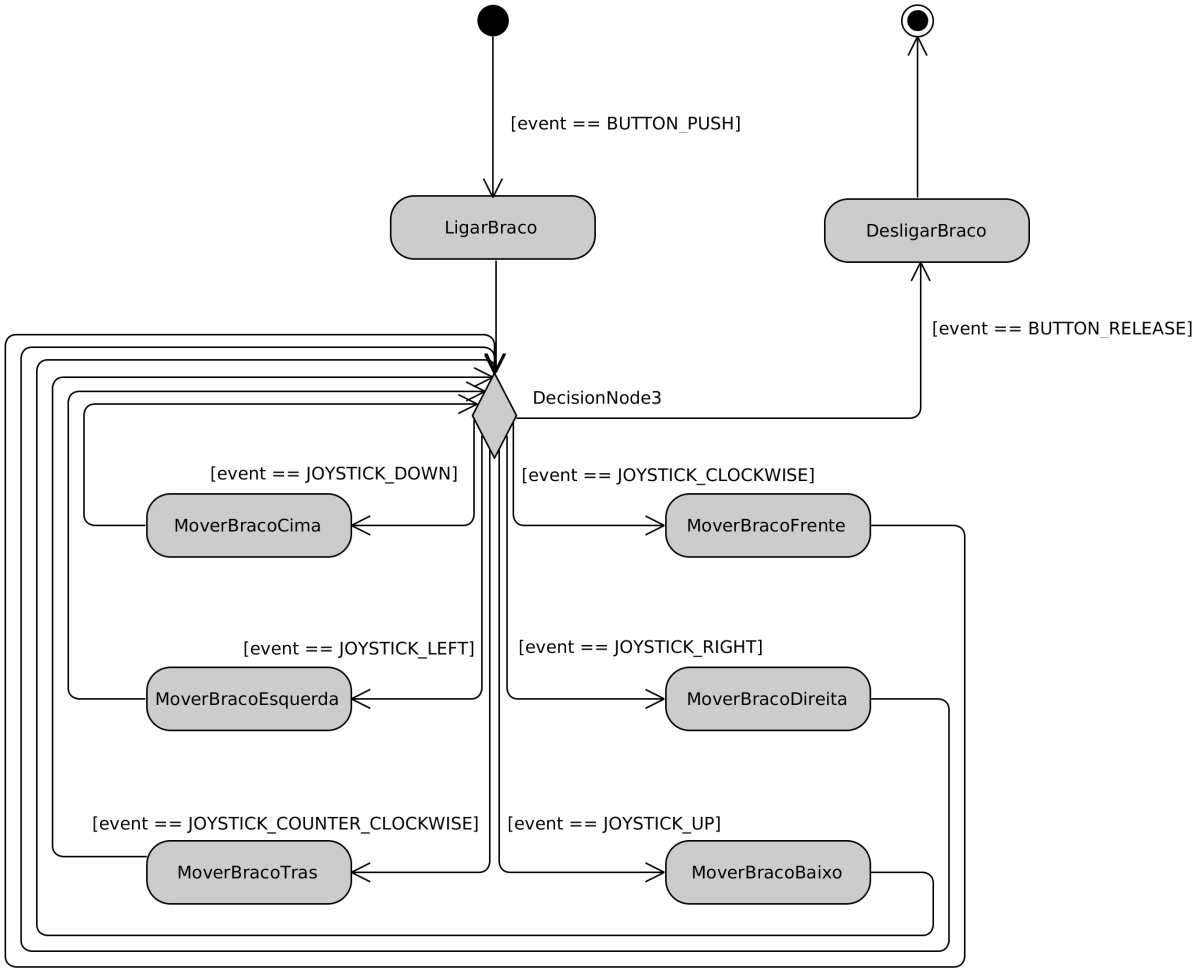

Figura 25: Progressão da simulação em atividades (exclusivamente um braço mecânico).

A Figura 26 apresenta a visão completa do Diagrama de Atividades do Segundo Cenário, contendo os detalhes de eventos necessários para as decisões nos *decision nodes*.

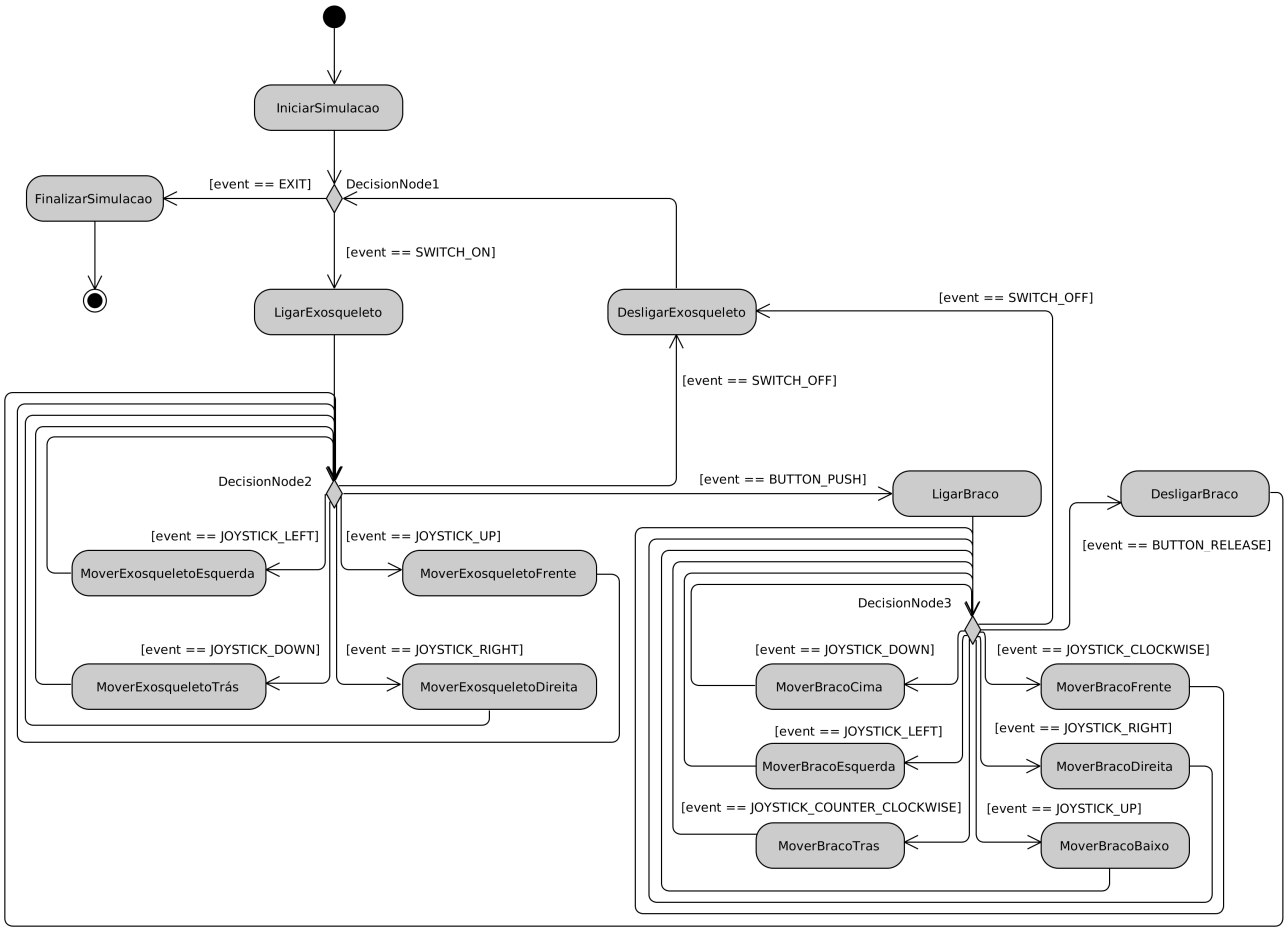

Figura 26: Progressão da simulação em atividades (visão completa).

Neste cenário, os eventos recebidos da chave, do botão e do *joystick* são utilizados pelo *software* para tomada de decisão. Neste caso, é explorada a necessidade de remapear os mesmos geradores de eventos em contextos diferentes, em estados diferentes de um *software*. Um mesmo *joystick* movimenta ora o exosqueleto (nos eixos X e Y), ora o braço do exosqueleto (nos eixos X, Y e Z), dependendo se o braço está ligado ou desligado.

Ademais, como requisito, o movimento do *joystick* corresponde diretamente ao movimento do exosqueleto nos eixos X e Y. Por exemplo, o movimento a frente do *joystick* executa a atividade para o exosqueleto andar no eixo Y positivo, o movimento à esquerda do *joystick* executa a atividade para o exosqueleto andar no eixo X negativo. Esse comportamento é idêntico ao primeiro cenário.

Quando movimentando o braço mecânico, como requisito, o *joystick* funciona para movimentá-lo no eixo Z. Ou seja, o movimento do controle em sentido horário faz com que o

braço se estenda e em anti-horário que o braço se retraia. Ademais, como requisito, no eixo Y o movimento é análogo ao do manche de avião. Dessa forma, o movimento do controle para frente faz com que o braço abaixe (eixo Y negativo) e o movimento para trás faz com que o braço levante (eixo Y positivo). Isto ocorre de forma contrária ao controle de movimento em Y do exosqueleto.

Faz-se necessário investigar a diferença em tratamento de eventos que são remapeados conforme o contexto do *software*. Esta necessidade é comum no atual estado da técnica. Como exemplo elementar um clique do apontador (*mouse*) é remapeado em diversas funções. O objetivo é comparar características do *software* em PON e POE nesta solução, no qual o *joystick* é utilizado tanto para movimentar o exosqueleto quanto o seu próprio braço mecânico (*i.e.* contextos diferentes do *software*).

A seguir são apresentadas as soluções implementadas para este segundo cenário, tanto em PON (Solução I) quanto em POE na Solução (II) – *State*.

(I) Solução PON para o Segundo Cenário

A solução técnica em PON para o segundo cenário se relaciona intrinsecamente à solução PON do primeiro cenário. De modo muito simples, o *software* do primeiro cenário foi estendido para atender os novos requisitos do segundo cenário: um botão a mais, *joystick* com movimento no próprio eixo (corresponde ao eixo Z), um braço mecânico e sua respectiva movimentação. A Figura 27 apresenta o modelo estrutural em PON por meio de um diagrama de classes UML, com os elementos da base de fatos FBE – <<NOP_FBE>> – e a aplicação PON (*i.e.* <<NOP_Application>>).

O diagrama apresenta quatro FBEs, a saber: *Exoskeleton*, *Arm*, *Event* e *Simulation*. *Arm* representa o braço mecânico e a classe *ExoskeletonEventHandlerSystemPON* é a classe principal em PON. Enumeradores também foram utilizados (*EXOSKELETON_STATE*, *ARM_STATE*, *SIMULATION_STATE* e *EVENT_TYPE*) *Exoskeleton* contém o enumerador *EXOSKELETON_STATE* que representa seus estados Ligado e Desligado (*ON* e *OFF*) assim como *Arm* contém o enumerador *ARM_STATE* que representa seus estados Ligado e Desligado (*ON* e *OFF*). Por fim, *Event* utiliza o enumerador *EVENT_TYPE* com os eventos possíveis neste segundo cenário (*SWITCH_ON*, *SWITCH_OFF*, *JOYSTICK_UP*, *JOYSTICK_RIGHT*, *JOYSTICK_DOWN*, *JOYSTICK_LEFT*, *JOYSTICK_NEUTRAL*, *JOYSTICK_CLOCKWISE*, *JOYSTICK_COUNTER_CLOCKWISE*, *BUTTON_PUSH*,

BUTTON_RELEASE, *EXIT*).

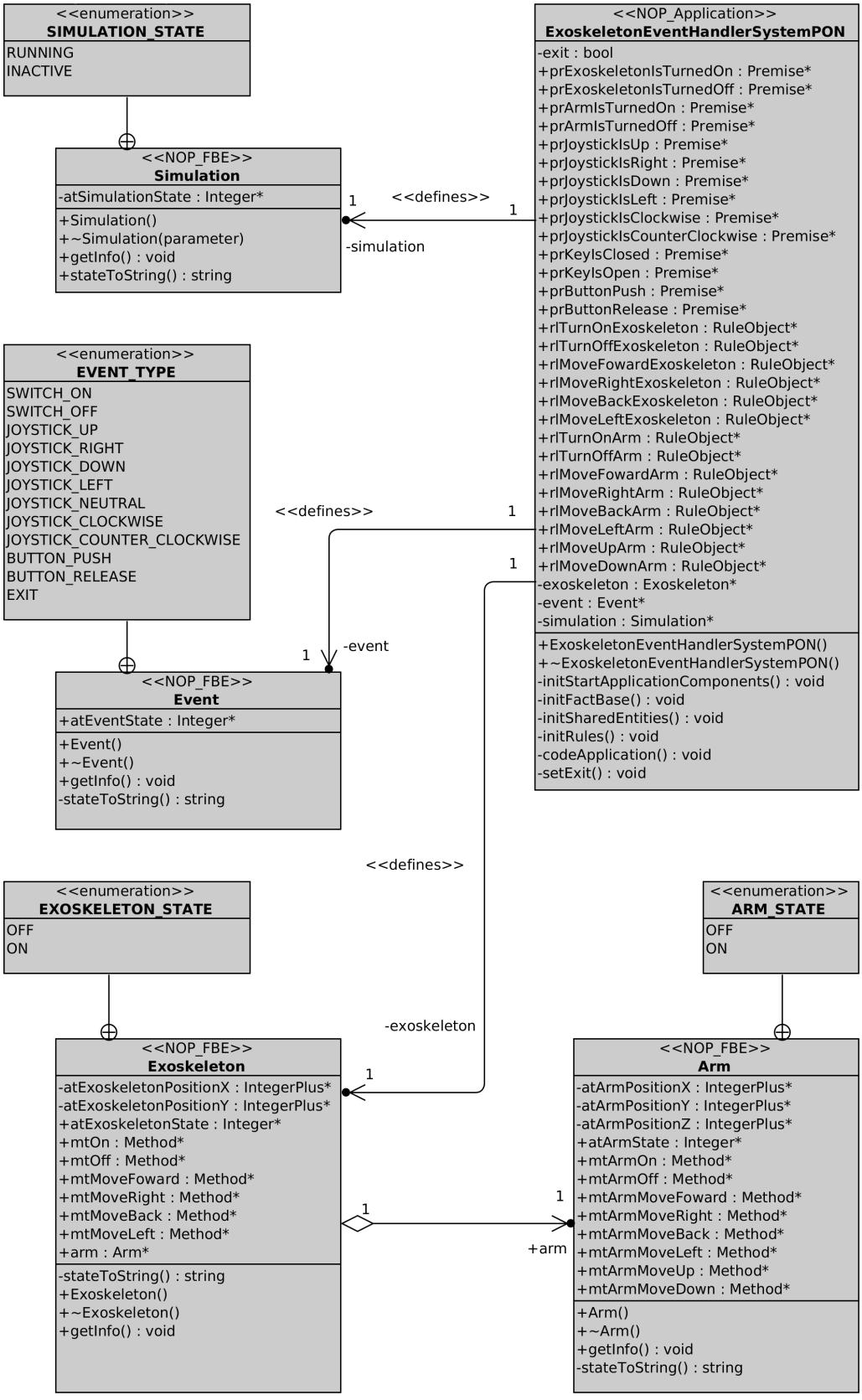

Figura 27: Classes do *software* em PON para o segundo cenário.

Aqui se enumera, a título de exemplo, duas regras utilizando as perguntas de análise neste segundo cenário do caso de estudo:

1) Regra: Ligar o Braço Mecânico

Qual o objetivo da regra? Ligar o Braço Mecânico.

O que precisa acontecer para que a regra seja executada? A simulação deve estar EXECUTANDO, o Exosqueleto deve estar LIGADO, o Braço Mecânico deve estar DESLIGADO e o evento recebido deve ser BOTÃO PRESSIONADO.

O que acontece se a regra for executada? O Braço Mecânico muda para o estado LIGADO.

2) Regra: Mover o Braço Mecânico para a Frente

Qual o objetivo da regra? Mover o Braço Mecânico para a frente.

O que precisa acontecer para que a regra seja executada? A simulação deve estar EXECUTANDO, o Exosqueleto deve estar LIGADO, o Braço Mecânico deve estar LIGADO e o evento recebido deve ser JOYSTICK GIRADO EM SENTIDO HORÁRIO (JOYSTICK_CLOCKWISE).

O que acontece se a regra for executada? O Braço Mecânico se move para a frente (mover no eixo Y positivo).

Assim, para cumprir os requisitos desse cenário foram implementadas quatorze *Rules* listadas na Tabela 3. Foram desenvolvidas duas novas premissas relacionadas para braço ligado ou braço desligado, e novas *Rules* sobre a operação do respectivo braço mecânico, por fim totalizando as quatorze *Rules*. Uma peculiar característica ficou evidenciada em um projeto PON: adicionar funcionalidades é rápido e fácil, provendo *software* muito expansível e modificável (facilidade de manutenção evolutiva).

Tabela 3: *Rules*, *Condition* e suas *Premises* e *Methods* instigados do *software* do segundo cenário exosqueleto e braço mecânico em PON.

Rule	**Nome**	***Condition* e suas *Premises***	***Methods* instigados**
1	rlTurnOnExoskeleton	atSimulationState == RUNNING && atExoskeletonState == OFF && atEventState == SWITCH_ON	Exoskeleton→mtOn

Rule **Nome**	***Condition* e suas *Premises***	***Methods* instigados**
2 rlTurnOffExoskeleton	atSimulationState == RUNNING && atExoskeletonState == ON && atEventState == SWITCH_OFF	Exoskeleton→mtOff Arm→mtArmOff
3 rlMoveForwardExoskeleton	atSimulationState == RUNNING && atExoskeletonState == ON && atArmState == OFF && atEventState == JOYSTICK_UP	Exoskeleton→mtForward
4 rlMoveRightExoskeleton	atSimulationState == RUNNING && atExoskeletonState == ON && atArmState == OFF && atEventState == JOYSTICK_RIGHT	Exoskeleton→mtRight
5 rlMoveBackExoskeleton	atSimulationState == RUNNING && atExoskeletonState == ON && atArmState == OFF && atEventState == JOYSTICK_DOWN	Exoskeleton→mtBack
6 rlMoveLeftExoskeleton	atSimulationState == RUNNING && atExoskeletonState == ON && atArmState == OFF && atEventState == JOYSTICK_LEFT	Exoskeleton→mtLeft
7 rlTurnOnArm	atSimulationState == RUNNING && atExoskeletonState == ON && atArmState == OFF && atEventState == BUTTON_PUSH	Arm→mtArmOn
8 rlTurnOffArm	atSimulationState == RUNNING && atExoskeletonState == ON && atArmState == ON && atEventState == BUTTON_RELEASE	Arm→mtArmOff
9 rlMoveForwardArm	atSimulationState == RUNNING && atExoskeletonState == ON && atArmState == ON && atEventState == JOYSTICK_CLOCKWISE	Arm→mtArmMoveForward
10 rlMoveRightArm	atSimulationState == RUNNING && atExoskeletonState == ON && atArmState == ON && atEventState == JOYSTICK_RIGHT	Arm→mtArmMoveRight
11 rlMoveBackArm	atSimulationState == RUNNING && atExoskeletonState == ON && atArmState == ON && atEventState == JOYSTICK_COUNTER_CLOCKWISE	Arm→mtArmMoveBack
12 rlMoveLeftArm	atSimulationState == RUNNING && atExoskeletonState == ON && atArmState == ON && atEventState == JOYSTICK_LEFT	Arm→mtArmMoveLeft

Rule **Nome**	***Condition* e suas *Premises***	***Methods* instigados**
13 rlMoveUpArm	atSimulationState == RUNNING && atExoskeletonState == ON && atArmState == ON && atEventState == JOYSTICK_DOWN	Arm→mtArmMoveUp
14 rlMoveDownArm	atSimulationState == RUNNING && atExoskeletonState == ON && atArmState == ON && atEventState == JOYSTICK_UP	Arm→mtArmMoveDown

O Algoritmo 8 mostra o código de algumas regras como implementadas no *Framework* em C++, dentro do contexto PON, para a operação do braço mecânico, que se referem respectivamente a desligar o braço mecânico, movê-lo para a frente, e movê-lo para a direita.

Algoritmo 8: Código de três regras PON em ExoskeletonEventHandlerSystemPON.cpp.

```
103 ...
104 RULE(rlTurnOffArm, scheduler, Condition::CONJUNCTION);
105   rlTurnOffArm->addPremise(prExoskeletonIsTurnedOn);
106   rlTurnOffArm->addPremise(prArmIsTurnedOn);
107   rlTurnOffArm->addPremise(prButtonRelease);
108   rlTurnOffArm->addInstigation(INSTIGATION(exoskeleton->arm->mtArmOff));
109
110 RULE(rlMoveForwardArm, scheduler, Condition::CONJUNCTION);
111   rlMoveForwardArm->addPremise(prExoskeletonIsTurnedOn);
112   rlMoveForwardArm->addPremise(prArmIsTurnedOn);
113   rlMoveForwardArm->addPremise(prJoystickIsClockwise);
114   rlMoveForwardArm->addInstigation(INSTIGATION(exoskeleton->arm->mtArmMoveForward));
115
116 RULE(rlMoveRightArm, scheduler, Condition::CONJUNCTION);
117   rlMoveRightArm->addPremise(prExoskeletonIsTurnedOn);
118   rlMoveRightArm->addPremise(prArmIsTurnedOn);
119   rlMoveRightArm->addPremise(prJoystickIsRight);
120   rlMoveRightArm->addInstigation(INSTIGATION(exoskeleton->arm->mtArmMoveRight));
121 ...
```

(II) Solução POE para o Segundo Cenário – *Handler-Dispatcher* e *State Pattern*

Assim como no primeiro cenário, na solução técnica em POE foi aplicada a relação indireta entre os objetos transmissores e objetos receptores empregando o componente despachante – *Handler-Dispatcher*. Ademais, com objetivo de garantir o determinismo do funcionamento da aplicação segundo um dado contexto específico (*e.g.* braço ligado ou desligado), bem como da necessidade de remapear, e até mesmo desprezar, os eventos de um único *joystick* (mover ora o exosqueleto ora o braço), uma Máquina de Estados foi utilizada e implementada por meio de *State Pattern* [Faison, 1993][Gamma et. al, 1995][Samek, 2008].

A Figura 28 apresenta um excerto do projeto estrutural dessa solução usando o padrão de projetos “Máquina de Estados” (*State*) para realizar o remapeamento de eventos do *joystick*.

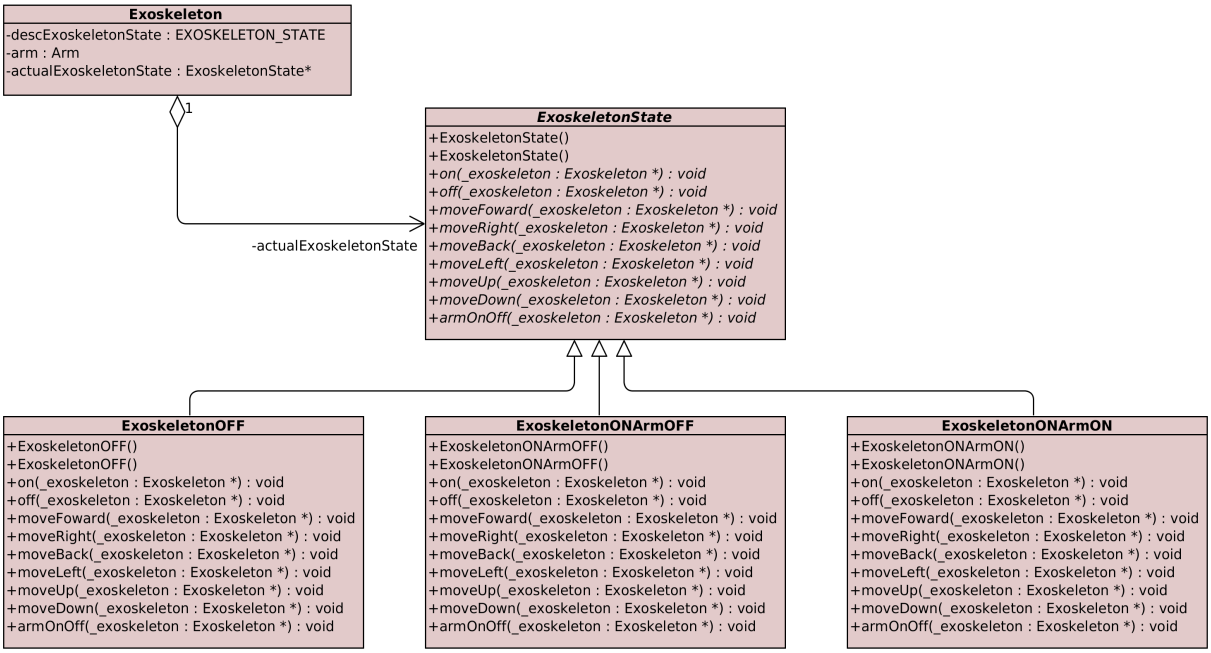

Figura 28: Excerto Diagrama de Classes em POE – *State Pattern.*

O padrão *State* provê uma distribuição da lógica da execução em uma hierarquia de subclasses utilizando polimorfismo. Ou seja, dependendo do estado do exosqueleto, uma respectiva instância dessa subclasse irá executar os métodos correspondentes. Especificamente, um objeto da Classe *Exoskeleton* é associado dinamicamente a instâncias das subclasses da hierarquia *ExoskeletonState*, a saber *ExoskeletonOFF*, *ExoskeletonONArmOFF* e *ExoskeletonONArmON*, que encapsulam os comportamento corretos para cada estado, de acordo com cada contexto de funcionamento do exosqueleto (desligado, ligado braço-desligado, ligado braço-ligado).

A solução *State* também evita (em parte) a necessidade de compor estruturas de controle do tipo SE-ENTÃO [Samek, 2008]. Nesse sentido, caso fosse utilizado somente a estrutura de controle SE-ENTÃO ou *SWITCH-CASE* seria necessário emular três contextos (*i.e.* os mesmos contextos mapeados para cada estado em *State Pattern*). Ou seja, possivelmente triplicando o tamanho do método *Dispatcher.dispatch()* e utilizando variáveis de controle (*i.e. flags*).

Dessa maneira, em POE, para realizar o segundo cenário do caso de estudo Simulador

de Transporte Individual foram construídas nove classes em C++: *Transport (*generalização de *Exoskeleton* e *Arm)*; *StateMachineHandlerSystem* (*i.e.* classe de principal de sistema); e classes das técnicas *Dispatcher* e *State Pattern*, como demonstrado (parcialmente) na Figura 29. Além disso, enumerações C++ (*enumerations)* foram utilizadas para compor os tipos de estado dos objetos (*TRANSPORT_STATE*, *EXOSKELETON_STATE* e *EVENT_TYPE*), conforme também exposto na Figura 29.

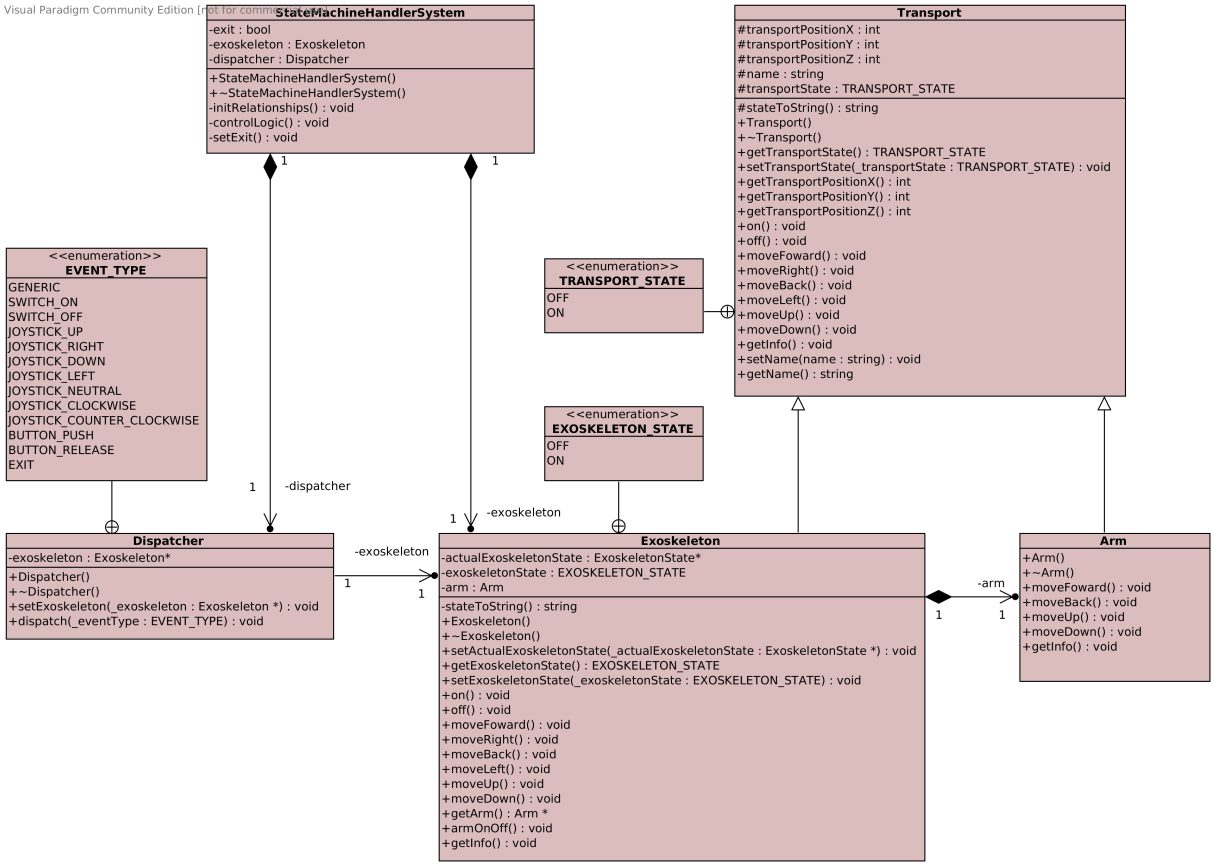

Figura 29: Excerto do diagrama de Classes em POE do segundo cenário.

O diagrama de sequência em *UML* da Figura 30 evidencia uma interação utilizando as técnicas *Dispatcher* e *State Pattern* deste segundo cenário. A interação executa a seguinte sequência de eventos recebidos: *SWITCH_ON*, *JOYSTICK_UP*, *BUTTON_PUSH* e *JOYSTICK_UP*. Ou seja, liga o exosqueleto, move o exosqueleto a frente, liga o braço mecânico, move o braço para baixo. O objetivo é demonstrar a dinâmica, durante a execução, da Máquina de Estados e as respectivas instâncias de *EXOSKELETON_OFF*, *EXOSKELETON_ON_ARM_OFF* e *EXOSKELETON_ON_ARM_ON*. Ressalta-se a criação e destruição desses objetos da hierarquia de *State Pattern* (destruição identificada com um X no final de linha de vida da instância e criação identificada por meio de mensagem em UML 2 com estereótipo <<create>>).

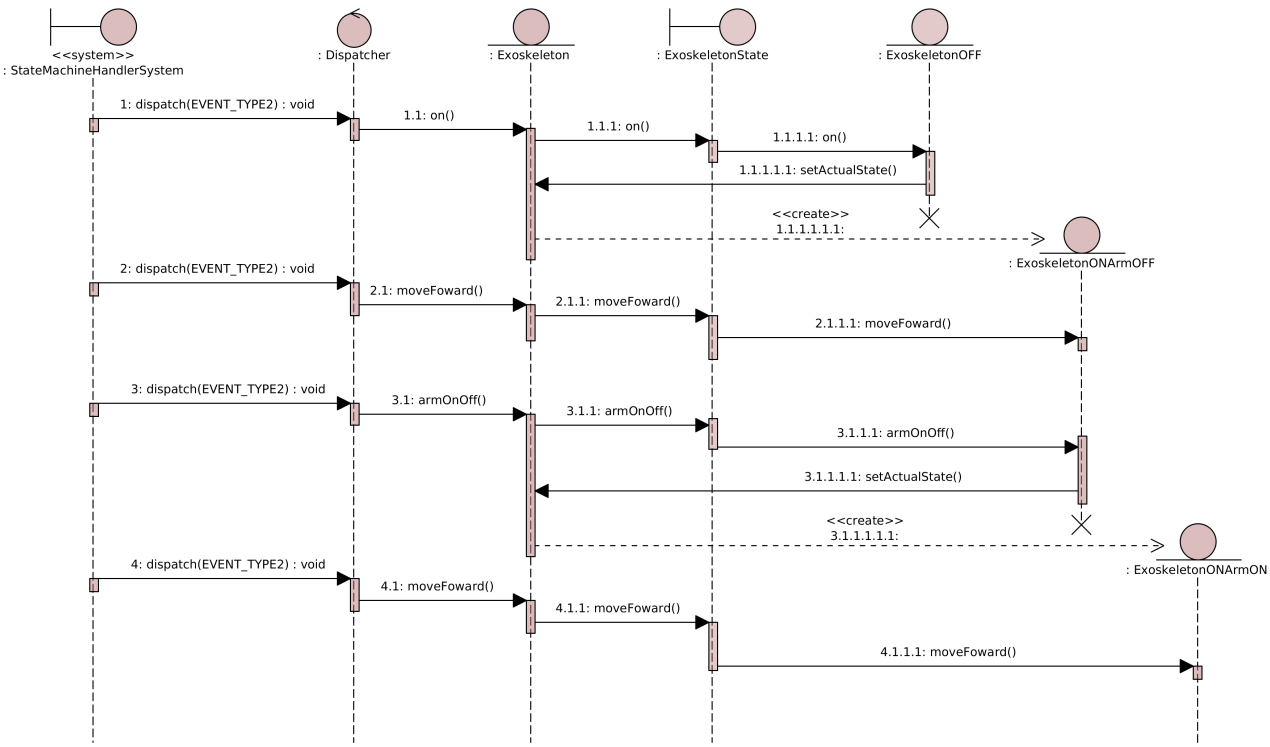

Figura 30: Diagrama de Sequência da solução State Pattern em POE para o segundo cenário.

O Algoritmo 9 mostra o novo método *dispatch()* de *Handler-Dispatcher* (ciclo de reconhecimento de eventos) neste segundo cenário. O algoritmo é muito similar ao método *dispatch()* do primeiro cenário, acrescentando o tratamento de eventos de *JOYSTICK_CLOCKWISE, JOYSTICK_COUNTER_CLOCKWISE* e *BUTTON_PUSH*. Neste cenário, a principal diferença se dá a partir da identificação do tipo de evento pelo *Handler-Dispatcher*.

Algoritmo 9: Código do método *dispatch* em Dispatcher.cpp.

```
...
void Dispatcher::dispatch(EVENT_TYPE _eventType) {
	switch (_eventType) {
		case (SWITCH_OPEN) : {
			this->exoskeleton->off();
			break;
		}
		case (SWITCH_CLOSE) : {
			this->exoskeleton->on();
			break;
		}
		case (JOYSTICK_UP) : {
			this->exoskeleton->moveForward();
			break;
		}
		case (JOYSTICK_RIGHT) : {
			this->exoskeleton->moveRight();
			break;
		}
		case (JOYSTICK_DOWN) : {
			this->exoskeleton->moveBack();
			break;
		}
		case (JOYSTICK_LEFT) : {
			this->exoskeleton->moveLeft();
			break;
		}
		case (JOYSTICK_CLOCKWISE) : {
			this->exoskeleton->moveUp();
			break;
		}
		case (JOYSTICK_COUNTER_CLOCKWISE) : {
			this->exoskeleton->moveDown();
			break;
		}
		case (BUTTON_PUSH) : {
			this->exoskeleton->armOnOff();
			break;
		}
		default : {
			break;
		}
	}
...
```

Os Algoritmos 10, 11 e 12 demonstram os trechos de código-fonte acerca da solução de lógica distribuída para o padrão Máquina de Estados (*State*), para exemplificar o remapeamento de eventos do *joystick*. O primeiro método (Algoritmo 10) é o responsável por mover *Exoskeleton* à frente, de acordo com a hierarquia identificada pelo estado (referência *actualState*).

Algoritmo 10: Código do método *moveForward()* em Exoskeleton.cpp (*State Pattern*).

```
56 ...
57 void Exoskeleton::moveForward() {
58       this->actualExoskeletonState->moveForward(this);
59 }
60 ...
```

O segundo método (Algoritmo 11) é um método virtual da classe de mais alta hierarquia no padrão *State* – *ExoskeletonState* – responsável por garantir um contrato por meio de uma interface (*i.e.* assinatura dos métodos conhecida e acordada).

Algoritmo 11: Código de declaração do método *moveForward()* em ExoskeletonState.h (*State Pattern*).

```
19 ...
20       virtual void moveForward(Exoskeleton* _exoskeleton) = 0;
21 ...
```

O terceiro método (Algoritmo 12) é a implementação do método virtual *moveForward*, no caso da classe *ExoskletonONArmON*, abaixo da hierarquia de *ExoskeletonState*, que representa o estado em que ambos *Exoskeleton* e *Arm* – braço mecânico – estão ligados.

Algoritmo 12: Código do método *moveForward* em ExoskeletonONArmON.cpp (*State Pattern*).

```
28 ...
29 void ExoskeletonONArmON::moveForward(Exoskeleton* _exoskeleton) {
30       _exoskeleton->getArm()->moveForward();
31 }
32 ...
```

A próxima seção apresenta o terceiro cenário deste caso, com suas duas soluções (PON e POE-*Observer)*.

3.1.3 Terceiro Cenário

No terceiro e último cenário deste caso de estudo, um piloto guia um exosqueleto e seus dois braços mecânicos (direito e esquerdo). Os principais requisitos para o terceiro cenário do *software* Simulador de Transporte Individual 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 (respectivamente chave fechada e chave aberta) o exosqueleto como um todo (incluindo os dois braços).
- Os botões ligam e desligam um respectivo braço mecânico (direito ou esquerdo). O botão vermelho comanda o braço esquerdo, em contraparte o botão azul comanda o braço direito. A Figura 31 mostra graficamente (conceitualmente) uma chave liga-desliga e um *joystick* com os dois botões.

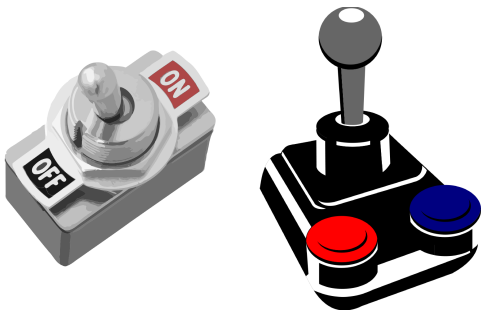

Figura 31: Chave ON/OFF e Joystick com dois botões para mover o exosqueleto ou o(s) braço(s) mecânico(s) no terceiro cenário.

- 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, conforme Figura 32.

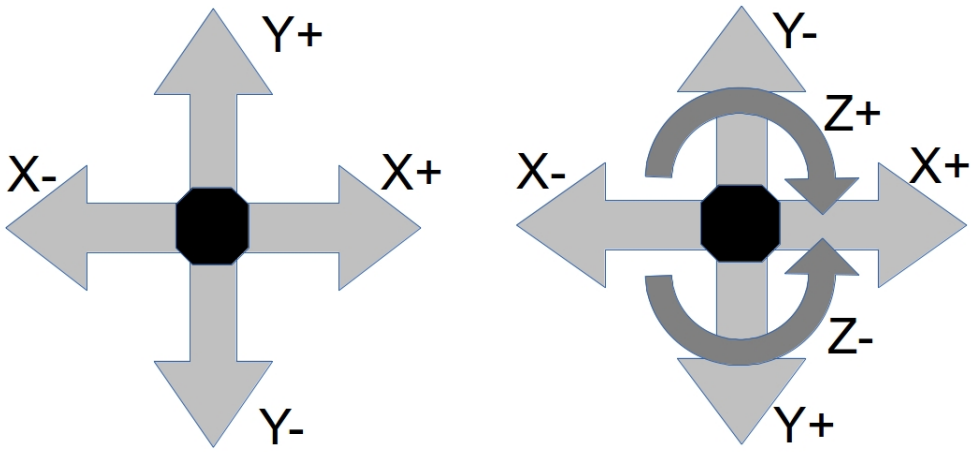

Figura 32: Movimentos do *joystick* para mover o exosqueleto (esquerda) ou os braços mecânicos (direita) no terceiro Cenário (análogo ao segundo cenário).

A Figura 33 mostra as funções do exosqueleto e braços mecânicos, neste terceiro cenário, por meio de um diagrama de caso de uso em UML.

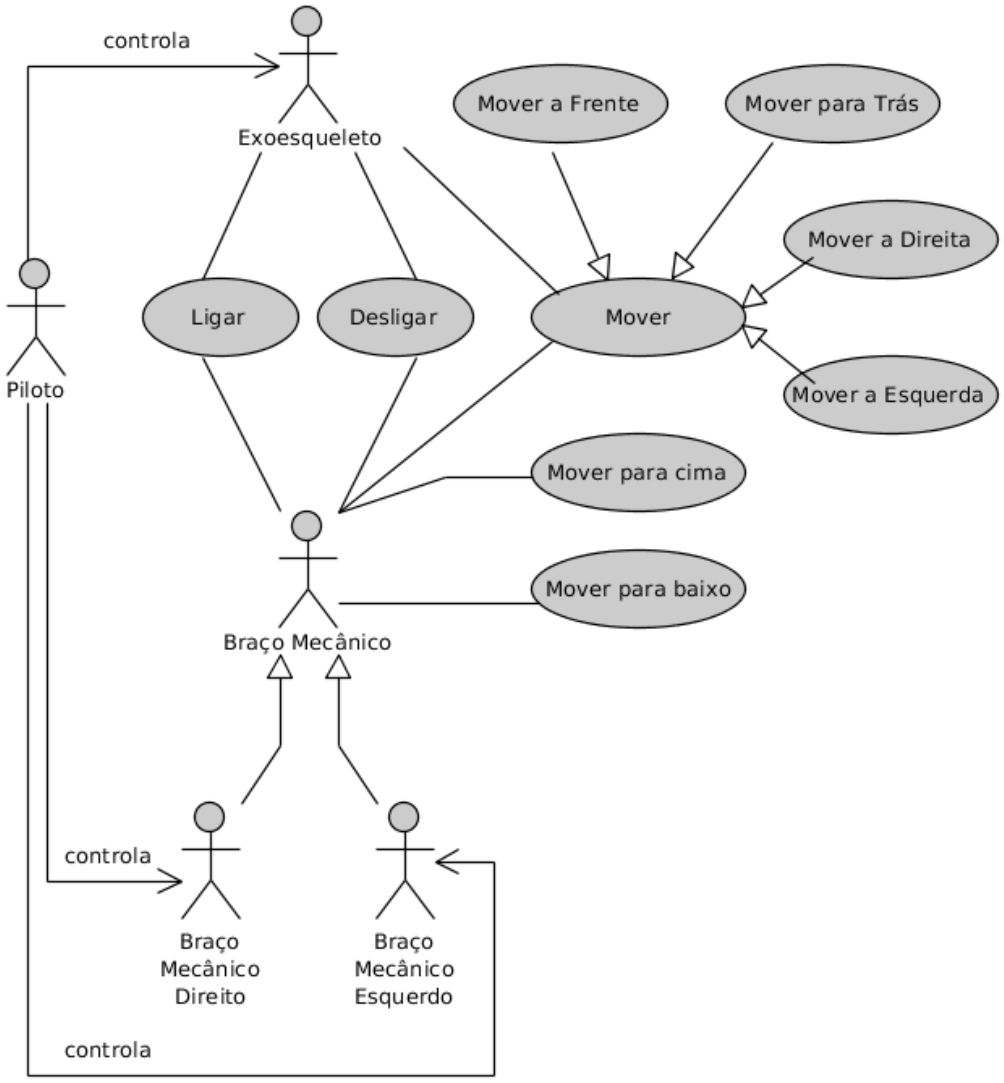

Figura 33: Funções do exosqueleto e dos braços mecânicos para o terceiro cenário.

Neste cenário, de forma análoga ao cenário anterior (segundo cenário), os eventos recebidos da chave, dos botões e do *joystick* são utilizados pelo *software* para tomada de decisão.

Da mesma forma, é explorada a necessidade de remapear os mesmos geradores de eventos em contextos diferentes. Um mesmo *joystick* movimenta ora o exosqueleto nos eixos X e Y; ora o braço direito do exosqueleto nos eixos X, Y e Z; ora o braço esquerdo do

exosqueleto nos eixos X e Y e Z; ora ambos os braços ligados nos eixos X, Y e Z; dependendo se os braços estão ligados ou desligados.

Da mesma maneira, o movimento do *joystick* corresponde diretamente ao movimento do exosqueleto nos eixos X e Y. Por exemplo, o movimento à frente do *joystick* executa a atividade para o exosqueleto andar no eixo Y positivo, o movimento à esquerda do *joystick* executa a atividade para o exosqueleto andar no eixo X negativo. Esse comportamento é idêntico aos primeiro e segundo cenários.

Quando movimentando um braço mecânico, como requisito, o *joystick* funciona para movimentá-lo no eixo Z, em funcionamento idêntico ao segundo cenário. Ou seja, o movimento do controle em sentido horário faz com que o braço se estenda e em anti-horário que o braço se retraia. Ademais, como requisito, no eixo Y o movimento é análogo ao do manche de avião. Dessa forma, o movimento do controle para frente faz com que o braço abaixe (eixo Y negativo) e o movimento para trás faz com que o braço levante (eixo Y positivo). O contrário do controle de movimento do eixo Y do exosqueleto.

Esta construção é relevante pois, além da investigação de diferença em tratamento de eventos que são remapeados conforme o contexto do *software* (algo explorado no segundo cenário), é necessário investigar *software* que trate eventos instigando mais de uma ação ao mesmo tempo. Esta necessidade é comum também no atual estado da técnica, como exemplo elementar um clicar e arrastar do apontador (mouse *dragging*, *i.e.* o *software* rastreia os movimentos do apontador ao arrastar um objeto exibindo em tela), requisito comumente implementado tendo como técnica o padrão *Observer*, *i.e.* os observadores se inscrevem em seus eventos de interesse (*subjects*). Aqui, o objetivo é comparar essas características do *software* em PON e POE nesta solução, em que o *joystick* é utilizado para movimentar o exosqueleto ou seus braços mecânicos.

A Figura 34 apresenta o diagrama de estados de alto nível em UML para este terceiro cenário do caso de estudo (visão resumida).

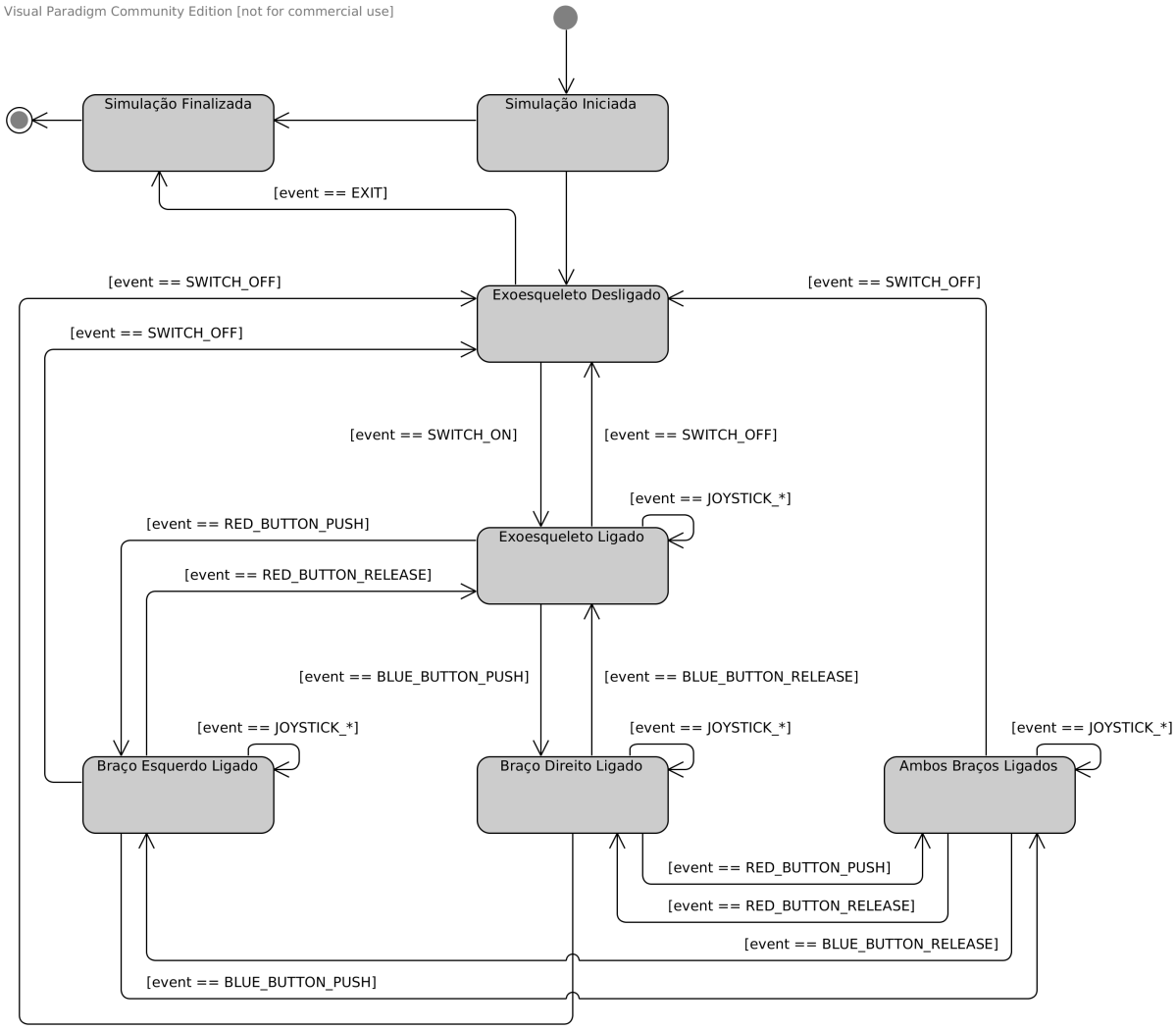

Figura 34: Progressão da simulação em estados do terceiro cenário (visão resumida).

A seguir são apresentadas as soluções implementadas para este terceiro cenário, tanto em PON (Solução I) quanto em POE na Solução (II) – *Observer*.

(I) Solução PON para o Terceiro Cenário

A solução técnica em PON para o terceiro cenário se relaciona intrinsecamente às soluções PON dos dois cenários anteriores. De mesma maneira e de modo muito simples, o

software do terceiro cenário foi estendido para atender os novos requisitos: dois botões, dois braços mecânicos e suas respectivas movimentações.

A Figura 35 apresenta o modelo estrutural em PON por meio de um diagrama de classes UML, com os elementos da base de fatos FBE – <<NOP_FBE>> – e a aplicação PON (*i.e.* <<NOP_Application>>).

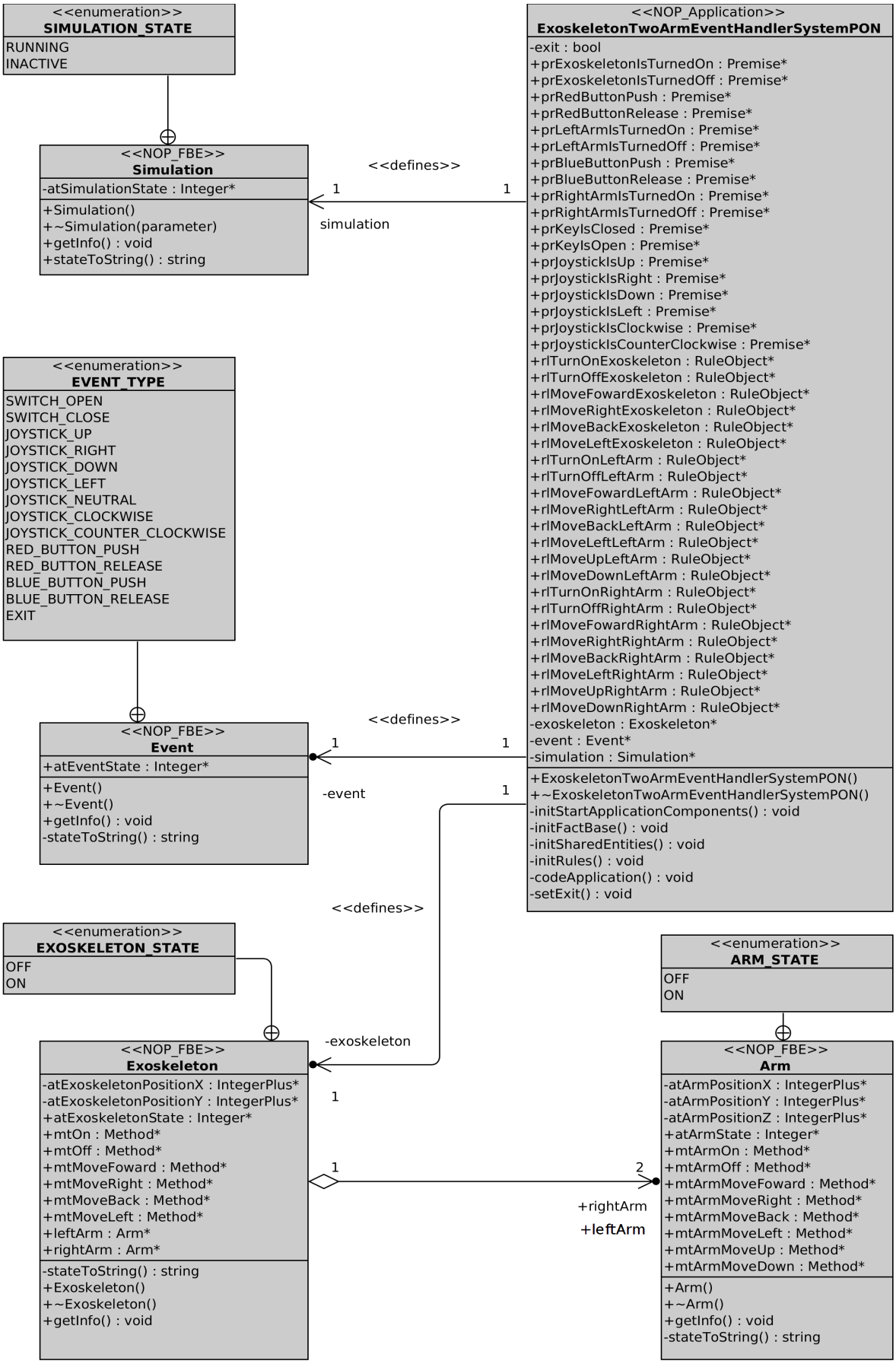

Figura 35: Classes do *software* em PON para o terceiro cenário.

Conforme o diagrama exposto, existem quatro FBEs, a saber: *Exoskeleton*, *Arm*, *Event* e *Simulation*. O FBE *Arm* representa o braço mecânico (as instâncias são *rightArm* e *leftArm*) e a classe *ExoskeletonTwoArmEventHandlerSystemPON* é a classe de sistema em PON (análoga à classe principal em POE). Enumeradores também foram utilizados (*EXOSKELETON_STATE*, *ARM_STATE*, *SIMULATION_STATE* e *EVENT_TYPE*) *Exoskeleton* contém o enumerador *EXOSKELETON_STATE* que representa seus estados Ligado e Desligado (*ON* e *OFF*) assim como *Arm* contém o enumerador *ARM_STATE* que representa seus estados Ligado e Desligado (*ON* e *OFF*). Por fim, *Event* utiliza o enumerador EVENT_TYPE que representa os eventos possíveis para tratamento pelo *software* neste terceiro cenário (*SWITCH_ON, SWITCH_OFF, JOYSTICK_UP, JOYSTICK_RIGHT, JOYSTICK_DOWN, JOYSTICK_LEFT, JOYSTICK_NEUTRAL, JOYSTICK_CLOCKWISE, JOYSTICK_COUNTER_CLOCKWISE, RED_BUTTON_PUSH, RED_BUTTON_RELEASE, BLUE_BUTTON_PUSH, BLUE_BUTTON_RELEASE, EXIT*).

Aqui se enumera, como exemplo, cinco regras utilizando as perguntas de análise deste terceiro cenário do caso de estudo. As regras elencadas a seguir demonstram algumas importantes características de programação em PON.

1) Regra: Ligar o Braço Mecânico Esquerdo

Qual o objetivo da regra? Ligar o Braço Mecânico Esquerdo.

O que precisa acontecer para que a regra seja executada? A simulação deve estar EXECUTANDO, o Exosqueleto deve estar LIGADO, o Braço Mecânico Esquerdo deve estar DESLIGADO e o evento recebido deve ser BOTÃO VERMELHO PRESSIONADO.

O que acontece se a regra for executada? O Braço Mecânico Esquerda muda para o estado LIGADO.

2) Regra: Desligar o Braço Mecânico Direito

Qual o objetivo da regra? Desligar o Braço Mecânico Direito

O que precisa acontecer para que a regra seja executada? A simulação deve estar EXECUTANDO, o Exosqueleto deve estar LIGADO, o Braço Mecânico Direito deve estar LIGADO e o evento recebido deve ser BOTÃO AZUL LIBERADO (*BLUE_BUTTON_RELEASE*).

O que acontece se a regra for executada? O Braço Mecânico Direito se move para a frente (mover no eixo Y positivo).

A análise das regras exemplo 1 e 2 demonstra a possibilidade de programação para tratamento de eventos em paralelo (*e.g.* ligar os dois braços ao mesmo tempo apertando os dois botões ao mesmo tempo).

3) Regra: Mover o Braço Mecânico Esquerdo para Cima

Qual o objetivo da regra? Mover o Braço Mecânico Esquerdo para Cima.

O que precisa acontecer para que a regra seja executada? A simulação deve estar EXECUTANDO, o Exosqueleto deve estar LIGADO, o Braço Mecânico Esquerdo deve estar LIGADO e o evento recebido deve ser JOYSTICK PARA TRÁS (*JOYSTICK_DOWN*).

O que acontece se a regra for executada? O Braço Mecânico Esquerdo se move para cima (mover no eixo Y positivo).

4) Regra: Mover o Braço Mecânico Direito para Cima

Qual o objetivo da regra? Mover o Braço Mecânico Direito para Cima.

O que precisa acontecer para que a regra seja executada? A simulação deve estar EXECUTANDO, o Exosqueleto deve estar LIGADO, o Braço Mecânico Direito deve estar LIGADO e o evento recebido deve ser JOYSTICK PARA TRÁS (*JOYSTICK_DOWN*).

O que acontece se a regra for executada? O Braço Mecânico Direito se move para cima (mover no eixo Y positivo).

A análise das regras exemplo 3 e 4 demonstra a possibilidade de programar *software* compondo regras diferentes que tratem de um único evento (*i.e.* uma única *Premise prJoystickDown*) para execução de ações diferentes ao mesmo tempo (*e.g.* mover os dois braços ao mesmo tempo para cima).

5) Regra: Desligar o exosqueleto

Qual o objetivo da regra? Desligar o exosqueleto como um todo.

O que precisa acontecer para que a regra seja executada? A simulação deve estar EXECUTANDO, o Exosqueleto deve estar LIGADO e o evento recebido deve ser CHAVE DESLIGADA (*SWITCH_OFF*).

O que acontece se a regra for executada? O exosqueleto é desligado, o Braço Mecânico Esquerdo é desligado e o Braço Mecânico Direito é desligado.

A análise da regra exemplo 5 demonstra a possibilidade de programar *software* compondo uma única regra para execução de ações diferentes ao mesmo tempo (*e.g.* desligar todos os aparatos do exosqueleto, incluindo seus dois braços mecânicos) também de maneira

inteligível.

Assim, para cumprir os requisitos desse cenário foram implementadas vinte e duas *Rules* listadas na Tabela 4. Foram desenvolvidas duas novas *Premises* relacionadas para mais um braço mecânico ligado ou desligado, duas novas *Premises* relacionadas para mais um botão de acionamento, bem como novas *Rules* para a operação de um novo braço mecânico, por fim totalizando as vinte e duas *Rules*. Uma peculiar característica ficou novamente evidenciada em um projeto PON: adicionar funcionalidades é rápido e fácil, provendo *software* muito expansível e modificável. Outro destaque é a característica das *Rules* serem orientadas a instância no atual estado da arte e da técnica em PON, pois elas relacionam *Premises* e *Methods* de instâncias de objetos (*i.e.* neste cenário objetos *Exoskeleton*, *LeftArm* e *RightArm*).

Tabela 4: *Rules, Condition* e suas *Premises* e *Methods* instigados do *software* do terceiro cenário exosqueleto e dois braços mecânicos (esquerdo e direito) em PON.

Rule	**Nome**	***Condition* e suas *Premises***	***Methods* instigados**
1	rlTurnOnExoskeleton	atSimulationState == RUNNING && atExoskeletonState == OFF && atEventState == SWITCH_ON	Exoskeleton→mtOn
2	rlTurnOffExoskeleton	atSimulationState == RUNNING && atExoskeletonState == ON && atEventState == SWITCH_OFF	Exoskeleton→mtOff LeftArm→mtArmOff RightArm→mtArmOff
3	rlMoveForwardExoskeleton	atSimulationState == RUNNING && atExoskeletonState == ON && LeftArm→atArmState == OFF && RightArm→atArmState == OFF && atEventState == JOYSTICK_UP	Exoskeleton→mtForward
4	rlMoveRightExoskeleton	atSimulationState == RUNNING && atExoskeletonState == ON && LeftArm→atArmState == OFF && RightArm→atArmState == OFF && atEventState == JOYSTICK_RIGHT	Exoskeleton→mtRight
5	rlMoveBackExoskeleton	atSimulationState == RUNNING && atExoskeletonState == ON && LeftArm→atArmState == OFF && RightArm→atArmState == OFF && atEventState == JOYSTICK_DOWN	Exoskeleton→mtBack
6	rlMoveLeftExoskeleton	atSimulationState == RUNNING && atExoskeletonState == ON && LeftArm→atArmState == OFF && RightArm→atArmState == OFF && atEventState == JOYSTICK_LEFT	Exoskeleton→mtLeft
7	rlTurnOnLeftArm	atSimulationState == RUNNING && atExoskeletonState == ON && LeftArm→atArmState == OFF && atEventState == RED_BUTTON_PUSH	LeftArm→mtArmOn

Rule	**Nome**	***Condition* e suas *Premises***	***Methods* instigados**
8	rlTurnOffLeftArm	atSimulationState == RUNNING && atExoskeletonState == ON && LeftArm→atArmState == ON && atEventState == RED_BUTTON_RELEASE	LeftArm→mtArmOff
9	rlMoveForwardLeftArm	atSimulationState == RUNNING && atExoskeletonState == ON && LeftArm→atArmState == ON && atEventState == JOYSTICK_CLOCKWISE	LeftArm→mtArmMoveForward
10	rlMoveRightLeftArm	atSimulationState == RUNNING && atExoskeletonState == ON && LeftArm→atArmState == ON && atEventState == JOYSTICK_RIGHT	LeftArm→mtArmMoveRight
11	rlMoveBackLeftArm	atSimulationState == RUNNING && atExoskeletonState == ON && LeftArm→atArmState == ON && atEventState == JOYSTICK_COUNTER_CLOCKWISE	LeftArm→mtArmMoveBack
12	rlMoveLeftLeftArm	atSimulationState == RUNNING && atExoskeletonState == ON && LeftArm→atArmState == ON && atEventState == JOYSTICK_LEFT	LeftArm→mtArmMoveLeft
13	rlMoveUpLeftArm	atSimulationState == RUNNING && atExoskeletonState == ON && LeftArm→atArmState == ON && atEventState == JOYSTICK_DOWN	LeftArm→mtArmMoveUp
14	rlMoveDownLeftArm	atSimulationState == RUNNING && atExoskeletonState == ON && LeftArm→atArmState == ON && atEventState == JOYSTICK_UP	LeftArm→mtArmMoveDown
15	rlTurnOnRightArm	atSimulationState == RUNNING && atExoskeletonState == ON && RightArm→atArmState == OFF && atEventState == RED_BUTTON_PUSH	RightArm→mtArmOn
16	rlTurnOffRightArm	atSimulationState == RUNNING && atExoskeletonState == ON && RightArm→atArmState == ON && atEventState == RED_BUTTON_RELEASE	RightArm→mtArmOff
17	rlMoveForwardRightArm	atSimulationState == RUNNING && atExoskeletonState == ON && RightArm→atArmState == ON && atEventState == JOYSTICK_CLOCKWISE	RightArm→mtArmMoveForward
18	rlMoveRightRightArm	atSimulationState == RUNNING && atExoskeletonState == ON && RightArm→atArmState == ON && atEventState == JOYSTICK_RIGHT	RightArm→mtArmMoveRight
19	rlMoveBackRightArm	atSimulationState == RUNNING && atExoskeletonState == ON && RightArm→atArmState == ON && atEventState == JOYSTICK_COUNTER_CLOCKWISE	RightArm→mtArmMoveBack
20	rlMoveLeftRightArm	atSimulationState == RUNNING && atExoskeletonState == ON && RightArm→atArmState == ON && atEventState == JOYSTICK_LEFT	RightArm→mtArmMoveLeft
21	rlMoveUpRightArm	atSimulationState == RUNNING && atExoskeletonState == ON && RightArm→atArmState == ON && atEventState == JOYSTICK_DOWN	RightArm→mtArmMoveUp

Rule	Nome	*Condition* e suas *Premises*	*Methods* instigados
22	rlMoveDownRightArm	atSimulationState == RUNNING && atExoskeletonState == ON && RightArm→atArmState == ON && atEventState == JOYSTICK_UP	RightArm→mtArmMoveDown

O Algoritmo 13 apresenta o código fonte de algumas regras, dentro do contexto do *Framework* C++ para PON, para a operação dos braços mecânicos do exosqueleto. No caso, são as regras para mover para baixo o braço mecânico esquerdo e, respectivamente, na sequência para ligar, desligar e mover para a frente o braço mecânico direito.

Algoritmo 13: Código de quatro regras PON em ExoskeletonTwoArmEventHandlerSystemPON.cpp.

```
151 ...
152 RULE(rlMoveDownLeftArm, scheduler, Condition::CONJUNCTION);
153   rlMoveDownLeftArm->addPremise(prExoskeletonIsTurnedOn);
154   rlMoveDownLeftArm->addPremise(prLeftArmIsTurnedOn);
155   rlMoveDownLeftArm->addPremise(prJoystickIsDown);
156   rlMoveDownLeftArm->addInstigation(INSTIGATION(exoskeleton->leftArm->mtArmMoveDown));
157
158 RULE(rlTurnOnRightArm, scheduler, Condition::CONJUNCTION);
159   rlTurnOnRightArm->addPremise(prExoskeletonIsTurnedOn);
160   rlTurnOnRightArm->addPremise(prRightArmIsTurnedOff);
161   rlTurnOnRightArm->addPremise(prBlueButtonPush);
162   rlTurnOnRightArm->addInstigation(INSTIGATION(exoskeleton->rightArm->mtArmOn));
163
164 RULE(rlTurnOffRightArm, scheduler, Condition::CONJUNCTION);
165   rlTurnOffRightArm->addPremise(prExoskeletonIsTurnedOn);
166   rlTurnOffRightArm->addPremise(prRightArmIsTurnedOn);
167   rlTurnOffRightArm->addPremise(prBlueButtonRelease);
168   rlTurnOffRightArm->addInstigation(INSTIGATION(exoskeleton->rightArm->mtArmOff));
169
170 RULE(rlMoveForwardRightArm, scheduler, Condition::CONJUNCTION);
171   rlMoveForwardRightArm->addPremise(prExoskeletonIsTurnedOn);
172   rlMoveForwardRightArm->addPremise(prRightArmIsTurnedOn);
173   rlMoveForwardRightArm->addPremise(prJoystickIsClockwise);
174   rlMoveForwardRightArm->addInstigation(INSTIGATION(exoskeleton->rightArm->mtArmMoveForward));
175 ...
```

(II) Solução POE para o Terceiro Cenário – *Observer Pattern*

Assim como nos cenários anteriores, nesta solução técnica em POE novamente foi aplicada a relação indireta entre os objetos transmissores e objetos receptores empregando o componente despachante – *Handler-Dispatcher,* dessa maneira desacoplando o recebimento de eventos com o tratamento dos mesmos. Além disso, também foi utilizado em conjunto o padrão de projeto *Observer*, como já citado, que é uma das principais técnicas de construção de *software* para tratamento de eventos na literatura [Ferg, 2006][Faison, 2006][Samek, 2008] [Bainomugisha *et al.*, 2013][Salvaneschi e Mezini, 2014].

Neste cenário, a finalidade de uso de *Observer Pattern* é diminuir o acoplamento (*i.e.* dependência) entre o componente *Dispatcher* e os tratadores de eventos, e também entre os próprios tratadores de eventos. Ou seja, objetiva encapsulamento (componentes mais autocontidos contendo unicamente seus próprios comportamentos e relacionamento preferencialmente somente entre seus dados e comportamentos) e reuso de componentes de *software* (*i.e.* menos dependência entre componentes e maior possibilidade de reuso).

Conforme consta no catálogo de padrões [Gamma *et al.* 1995], se aplica o padrão *Observer*, por conta de três das indicações originais:

- "*Um objeto deve ser capaz de notificar outros objetos sem realizar suposições sobre como esses objetos são, em outras palavras, não se deseja que os objetos sejam altamente acoplados.*"
- "*Uma mudança de estado em um objeto requer mudança em outros objetos, e não se sabe quantos objetos é preciso mudar.*"
- "*É necessária uma propagação de mudança na qual o publicador - conhecido como sujeito (subject) - notifica consumidores registrados - conhecidos como observadores (observers). Quando o estado do publicador muda, os observadores notificados podem realizar qualquer ação que julgue necessária.*"

Outros padrões ou soluções técnicas, ou até mesmo composição de padrões, soluções, técnicas ou estilos de programação, poderiam ter sido utilizadas para resolver em POE este cenário. Todavia o objetivo, como finalidade de pesquisa, é aplicar a solução conforme preconiza a literatura, de forma unívoca, assim de certa forma ortodoxa e purista. Por exemplo, não se utilizou *State Pattern* neste cenário.

Como já exposto, o padrão *State* duplica código (*e.g.* chamadas de métodos) em várias classes, distribuindo a lógica do funcionamento também em várias classes. Dessa maneira, ao não utilizá-lo também se evita (de certa forma) o problema do aumento do número de estados que surgiria. Por exemplo, a solução *State Pattern* utilizaria (acrescentaria) cinco estados (instâncias de classes) para atender os requisitos do cenário (*i.e.* estados *ExoskeletonOFF*, *ExoskeletonONBothArmsOFF*, *ExoskeletonONLeftArmON*, *ExoskeletonONRightArmON*, *ExoskeletonONBothArmsON*). Caso houvesse mais um requisito ao menos parcialmente ortogonal como "*Restringir movimentos do exosqueleto quando houver um sensor ligado*", em um projeto de *software* utilizando *State* mais estados (instâncias de classes) seriam

necessários.

Dessa maneira, em POE, para realizar o terceiro cenário do caso de estudo Simulador de Transporte Individual foram construídas sete classes em C++: *Transport* (generalização de *Exoskeleton* e *Arm)*; *ObserverHandlerSystem* (*i.e.* classe de principal de sistema); e classes das técnicas *Dispatcher* e *Observer Pattern*, como apresentado (parcialmente) na Figura 36. Além disso, enumerações C++ (*enumerations)* foram utilizadas para compor os tipos de estado dos objetos (*TRANSPORT_STATE*, *EXOSKELETON_STATE* e *EVENT_TYPE*), conforme o diagrama da Figura 36.

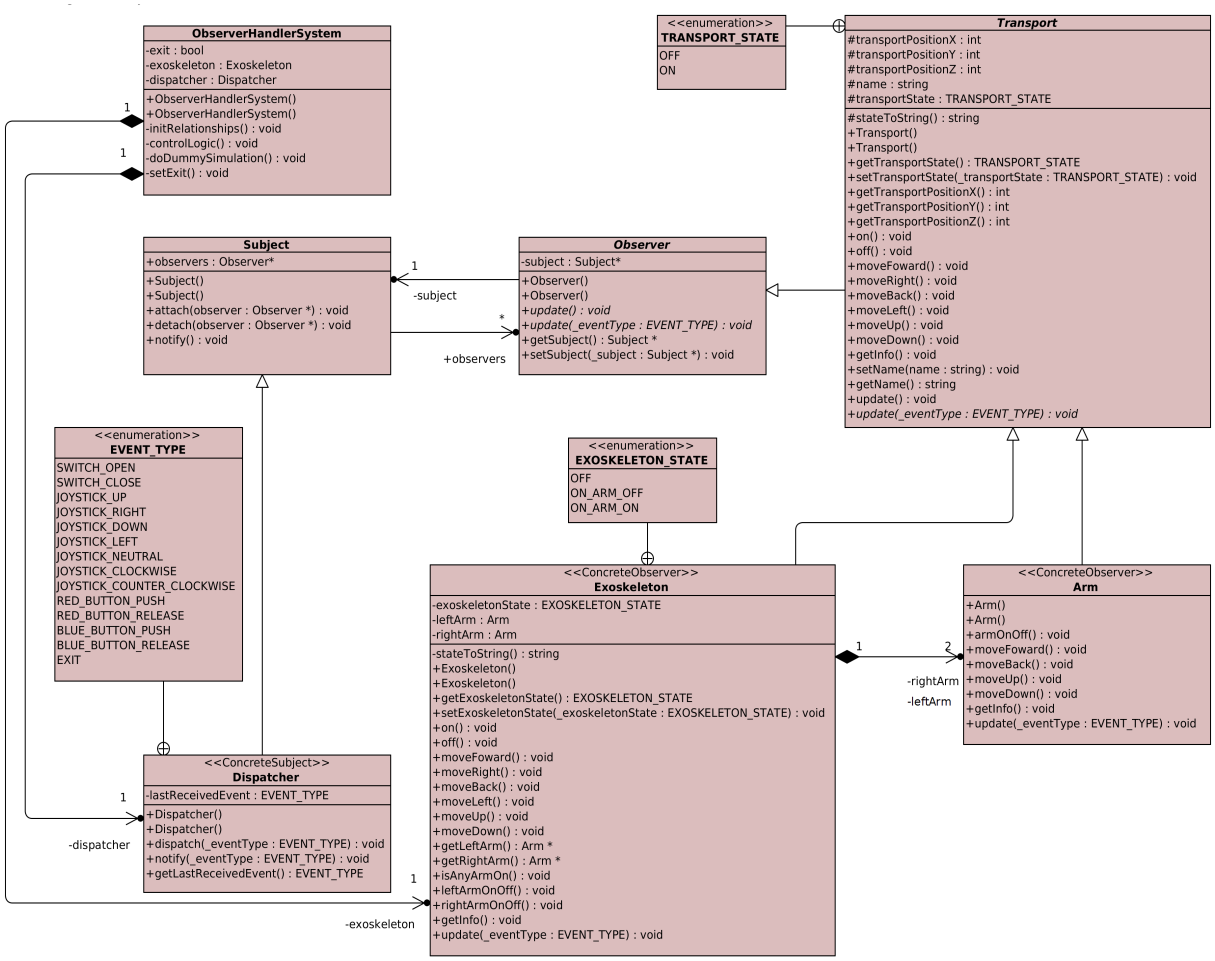

Figura 36: Excerto do diagrama de Classes em POE do terceiro cenário (*Observer Pattern*).

O diagrama de sequência da Figura 37 exemplifica uma interação utilizando as técnicas *Dispatcher* e *Observer Pattern* deste terceiro cenário. A interação executa a sequência de eventos: *SWITCH_ON, JOYSTICK_UP, RED_BUTTON_PUSH, JOYSTICK_UP, BLUE_BUTTON_PUSH*. Ou seja, liga o exosqueleto, move o exosqueleto à frente, liga o braço mecânico, move o braço para baixo. O objetivo é demonstrar a dinâmica, durante a execução, dos Observadores, instâncias de *Exoskeleton* e *Arm*.

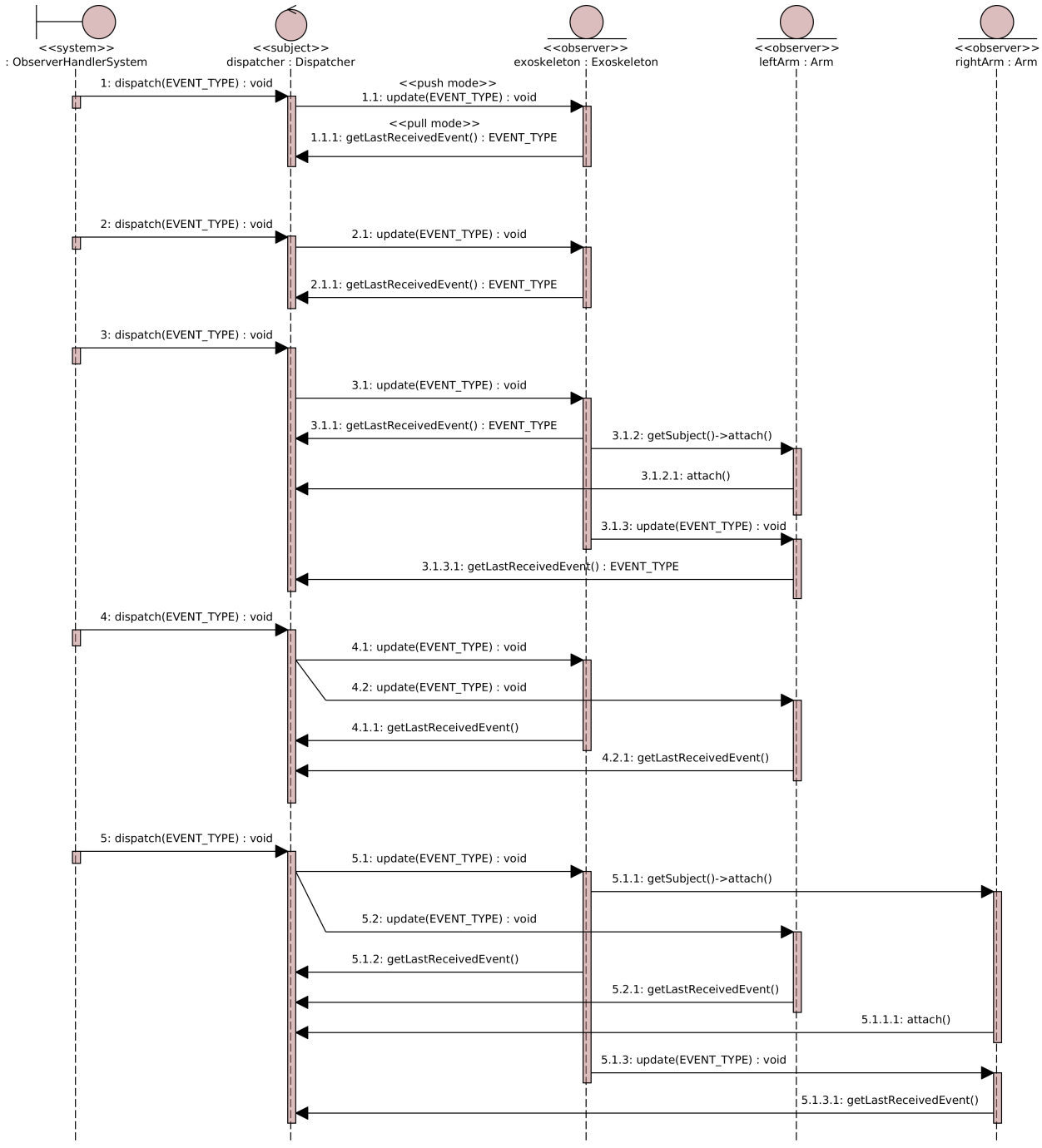

Figura 37: Diagrama de Sequência da solução *Observer Pattern* em POE para o terceiro cenário.

O Algoritmo 14 apresenta os novos métodos *dispatch()* do componente *Dispatcher*, *getLastReceivedEvent* (método que os observadores podem utilizar para se atualizar quanto ao estado de *Subject* em funcionamento - *pull model.* No caso o *Subject* é próprio componente *Dispatcher*) e *notify* (método que notifica os observadores).

Algoritmo 14: Código do método *dispatch()* e *notify()* em *Dispatcher.cpp*.

```
19 ...
20 void Dispatcher::dispatch(EVENT_TYPE _eventType) {
21        this->lastReceivedEvent = _eventType;
22        this->notify(_eventType);
23 }
24
25 EVENT_TYPE Dispatcher::getLastReceivedEvent() {
26        return this->lastReceivedEvent;
27 }
28
29 void Dispatcher::notify(EVENT_TYPE _eventType) {
30        for (int i = 0; i < observers.size(); i++) {
31              observers[i]->update(_eventType);
32        }
33 }
34 ...
```

Em seguida, o Algoritmo 15 apresenta excerto do método *update()* do componente *Exoskeleton*, método que realiza o ciclo de reconhecimento de eventos para este componente. É uma estrutura *SWITCH-CASE* aninhada, análoga a uma máquina de estados hierárquica [Samek, 2008] como no segundo cenário. O método incorpora toda a lógica relacionada com os eventos de *attach()* e *detach()* dos braços mecânicos (*i.e.* ações de anexar e desanexar, ou seja, *binding* dinâmico). De certa forma, é o responsável por centralizar os macroestados de todos os componentes *Observers* da solução (contexto do *software*). Nesse caso, este componente necessariamente nunca deixa de observar realizando uma operação de *detach()* (desanexar). Todos os contextos tratados do *SWITCH-CASE*, atributo *actualState*, estão no Algoritmo 15: *EXOSKELETON_OFF, EXOSKELETON_ON_ARMS_OFF, LEFT_ARM_ON, RIGHT_ARM_ON e BOTH_ARMS_ON.*

Algoritmo 15: Código do método *update()* em *Exoskeleton.cpp*.

```
...
void Exoskeleton::update(EVENT_TYPE _eventType) {

	switch (actualState) {
		case (EXOSKELETON_OFF): {
			switch (_eventType) {
				case (SWITCH_CLOSE): {
					this->on();
					this->actualState = EXOSKELETON_ON_ARMS_OFF;//EXOSKELETON_ALWAYS_ATTACHED;
					break; }
				default: { break; }
			}
		break; }
		case (EXOSKELETON_ON_ARMS_OFF): {
			switch (_eventType) {
				case (RED_BUTTON_PUSH): {
					this->getSubject()->attach(this->getLeftArm());
					this->getLeftArm()->armOnOff();
					this->actualState = LEFT_ARM_ON;
					break; }
				case (BLUE_BUTTON_PUSH): {
					this->getSubject()->attach(this->getRightArm());
					this->getRightArm()->armOnOff();
					this->actualState = RIGHT_ARM_ON;
					break; }
				case (SWITCH_OPEN): {
					//ALWAYS ATTACHED
					this->off();
					this->actualState = EXOSKELETON_OFF;
					break; }
				case (JOYSTICK_UP): {
					this->moveForward();
					break; }
				case (JOYSTICK_RIGHT): {
					this->moveRight();
					break; }
				case (JOYSTICK_DOWN): {
					this->moveBack();
					break; }
				case (JOYSTICK_LEFT): {
					this->moveLeft();
					break; }
				default: { break; }
			}
		break; }
		case (LEFT_ARM_ON): {
			switch (_eventType) {
				case (RED_BUTTON_PUSH): {
					this->getLeftArm()->armOnOff();
					this->actualState = EXOSKELETON_ON_ARMS_OFF;
					break; }
				case (BLUE_BUTTON_PUSH): {

...				...

		case (RIGHT_ARM_ON): {

...		...

		case (BOTH_ARMS_ON): {

...		...
```

Já o Algoritmo 16 mostra excerto do método *update()* do componente *Arm*, método que também realiza o ciclo de reconhecimento de eventos para este componente, neste caso, o braço mecânico, utilizando um *SWITCH-CASE* simples. Este componente deixa de observar eventos, realizando uma operação de *detach()* quando ocorre o evento correspondente "botão pressionado", porém quem realiza o reconhecimento deste evento específico é o componente *Exoskeleton.* Isso ocorre pois o contexto de estados fica organizado e centralizado em *Exoskeleton,* sendo o componente escolhido para ter tal responsabilidade (*i.e.* estados lógicos do *software* que se referem aos estados de *LeftArm e RightArm* ligados ou desligados).

Algoritmo 16: Código do método *update()* em *Arm.cpp*.

```
...
void Arm::update(EVENT_TYPE _eventType) {

	switch (_eventType) {
/*	EXOSKELETON CENTRALIZES THIS CONTROL
		case (BUTTON_PUSH): {
			this->armOnOff();
			break;
		}
*/
		case (JOYSTICK_UP): {
			this->moveUp();
			break;
		}
		case (JOYSTICK_RIGHT): {
			this->moveRight();
			break;
		}
		case (JOYSTICK_DOWN): {
			this->moveDown();
			break;
		}
		case (JOYSTICK_LEFT): {
			this->moveLeft();
			break;
		}
		case (JOYSTICK_CLOCKWISE): {
			this->moveBack();
			break;
		}
		case (JOYSTICK_COUNTER_CLOCKWISE): {
			this->moveForward();
			break;
		}
		default: {
			break;
		}
	}
}
```

Assim, a estrutura de controle para reconhecimento de eventos *SWITCH-CASE* migra de *Dispatcher.dispatch()* para cada objeto observador – *Observer*. Objetiva-se coesão e

encapsulamento, *i.e.* assim cada componente trata seus próprios eventos de interesse. Entretanto, não foi possível desacoplá-los totalmente entre si. Para uma solução correta, é necessário o conhecimento (centralizado) do estado global da solução com a identificação de estados (contexto do *software*) exosqueleto desligado, exosqueleto ligado, braço esquerdo ligado, braço direito ligado e ambos os braços ligados.

Existem variações do padrão *Observer* ou, até mesmo, composição do padrão com outras técnicas ou outros padrões propriamente ditos. Porém neste trabalho se adota a solução demonstrada em [Gamma *et al.*, 1995].

3.1.3 Reflexões do Primeiro Caso de Estudo

Como era objetivo, no primeiro cenário se realizou a tomada de decisão a partir de recebimento de eventos externos. Em PON, a cada evento foi associado um *Atribute*, e as decisões foram compostas em *Rules.* O uso de evento como atributo não é exclusividade desta solução, também ocorre em outras soluções encontradas na literatura [Salvaneschi *et al.*, 2014]. Ao seu turno, em POE foi utilizada a técnica *Handler-Dispatcher.*

No segundo cenário se executou o tratamento de eventos em estados diferentes do *software*. Em PON, as *Premises* realizaram essa diferenciação de tratamento a partir de estados de *Exoskeleton* e *Arm*. Em POE, o padrão de projeto *State* resolveu este cenário por meio de métodos diferentes para o cálculo lógico-causal de *Handler-Dispatcher* (polimorfismo).

Cabe registrar que em POE, ao utilizar a técnica Máquina de Estados aplicando *State Pattern* para implementação dos componentes, a lógica de funcionamento de um componente fica dispersa em instâncias diferentes (*i.e.* objetos). Os comportamentos (I) exosqueleto desligado, (II) exosqueleto ligado e braço desligado, (III) exosqueleto ligado e braço ligado, são instâncias de uma hierarquia de classes. Todos esses comportamentos relacionam-se a um único componente exosqueleto. Ou seja, instâncias de classes diferentes para executar os diferentes comportamentos de um único componente.

Por último, o terceiro cenário trata eventos recebidos que executam mais de uma ação (*e.g.* conceitualmente movem os dois braços ao mesmo tempo). Em PON, as *Premises* partilhadas relacionadas aos eventos recebidos, partilhada em duas *Rules,* cumpriram o requisito. Em POE, ao se utilizar *Observer*, decisões devem ser tomadas pelo programador, como em qual método deve ser colocado qual comportamento (*e.g.* quem centraliza ou delega

o que deve ser feito a partir de um dado evento recebido), indicando que as decisões sobre os comportamentos ficam dispersas pelo *software*. Em contrapartida, no caso do PON, a tomada de decisão fica expressa de forma centralizada ao utilizar *Rules*. Neste sentido, em termos de percepção, o autor (e programador) dos casos de estudo explicita que foi necessário mais tempo para construção do *software* utilizando *Observer*.

Ao final deste capítulo serão realizadas mais discussões do que foi apresentado neste caso de estudo e no capítulo seguinte serão apresentados experimentos quantitativos que se utilizaram deste caso de estudo.

3.2 Caso de Estudo 2: Portão Eletrônico

Nesta seção é apresentado um segundo caso de estudo, em cenário único, de projeto de *software* intitulado "Simulador de Portão Eletrônico", outrora apresentado em [Wiecheteck, 2011] e aprimorado em [Batista, 2013], De maneira geral, é um *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. A Figura 38 demonstra conceitualmente os elementos deste sistema de automação.

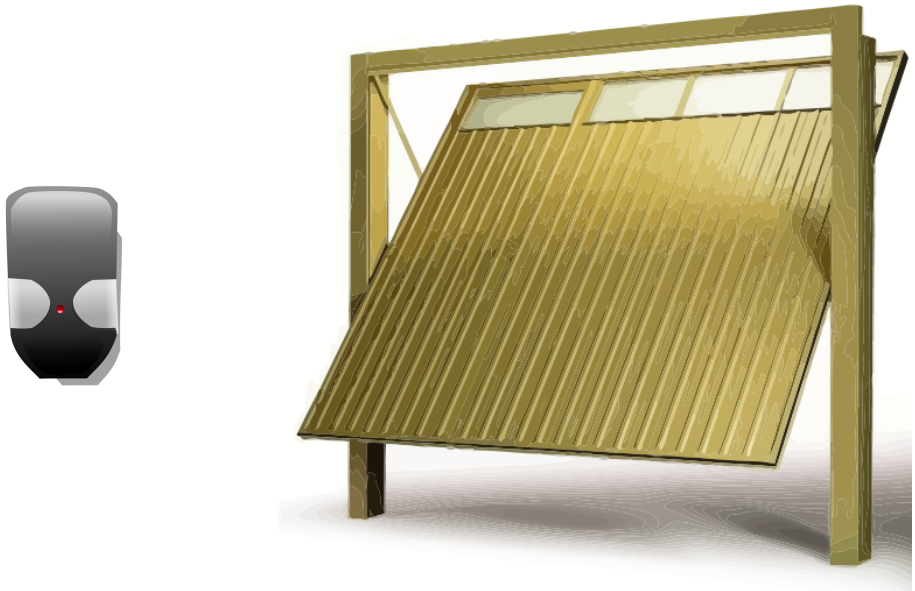

Figura 38: Desenho conceitual de um controle remoto para um portão eletrônico.

Os requisitos do *software* de portão eletrônico são apresentados a seguir [adaptados de Batista, 2013]:

- O *software* simulador deverá receber eventos de um controle remoto (também simulado) acionado conforme escolha do usuário (botão pressionado ou inativo).
- O sistema deverá controlar o processo de abertura e fechamento de um portão eletrônico, sendo que o portão deve iniciar fechado e possuir um contador (*timer*) que inicia zerado e parado.
- O processo de abertura do portão dura 30 segundos (trinta segundos multiplicados por um tempo de referência TR, no caso, um segundo).
- O processo de fechamento do portão dura 30 segundos (30 x TR).
- Estando o portão fechado, o *software* deverá iniciar abertura do mesmo quando o controle remoto for pressionado e, após 30 segundos, o portão estará aberto. O contador deverá ser zerado após a abertura completa do portão.
- Durante a execução da abertura do portão, a qualquer instante, a luz do ambiente onde se encontra o portão eletrônico deverá ser acesa automaticamente.
- Uma vez aberto o portão, e caso o controle remoto seja pressionado novamente, o *software* deverá iniciar o processo de fechamento do portão, que estará completamente fechado após 30 segundos.
- Durante a execução do fechamento do portão, a qualquer instante, a luz do ambiente onde se encontra o portão eletrônico deverá ser apagada.
- Durante a execução de abertura do portão, em que não se passaram ainda os 30 segundos, caso haja um acionamento do controle remoto, o portão para de abrir ficando semiaberto. Acionando-se o controle novamente o portão volta a fechar (*i.e.* cancelamento da abertura).
- O mesmo pode ocorrer durante a execução de fechamento do portão em que não se completaram os 30 segundos. Caso o controle remoto seja pressionado, o portão para de fechar ficando semiaberto. Com um novo acionamento do controle remoto o portão retorna ao processo de abertura (*i.e.* cancelamento do fechamento).

A Figura 39 mostra as funções do Simulador de Portão Eletrônico por meio de um diagrama de caso de uso em UML.

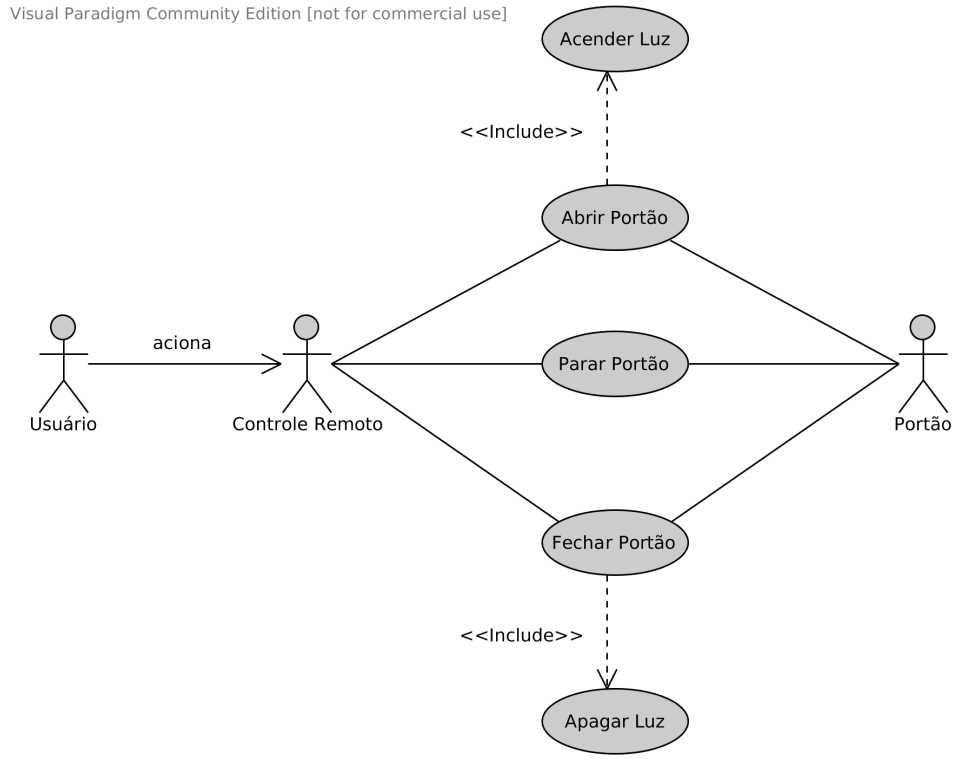

Figura 39: Funções do Simulador de Portão Eletrônico.

A Figura 40 ilustra o Diagrama de Estados (em alto nível) do Simulador de Portão Eletrônico, que contém os eventos necessários para as transições de estado do portão.

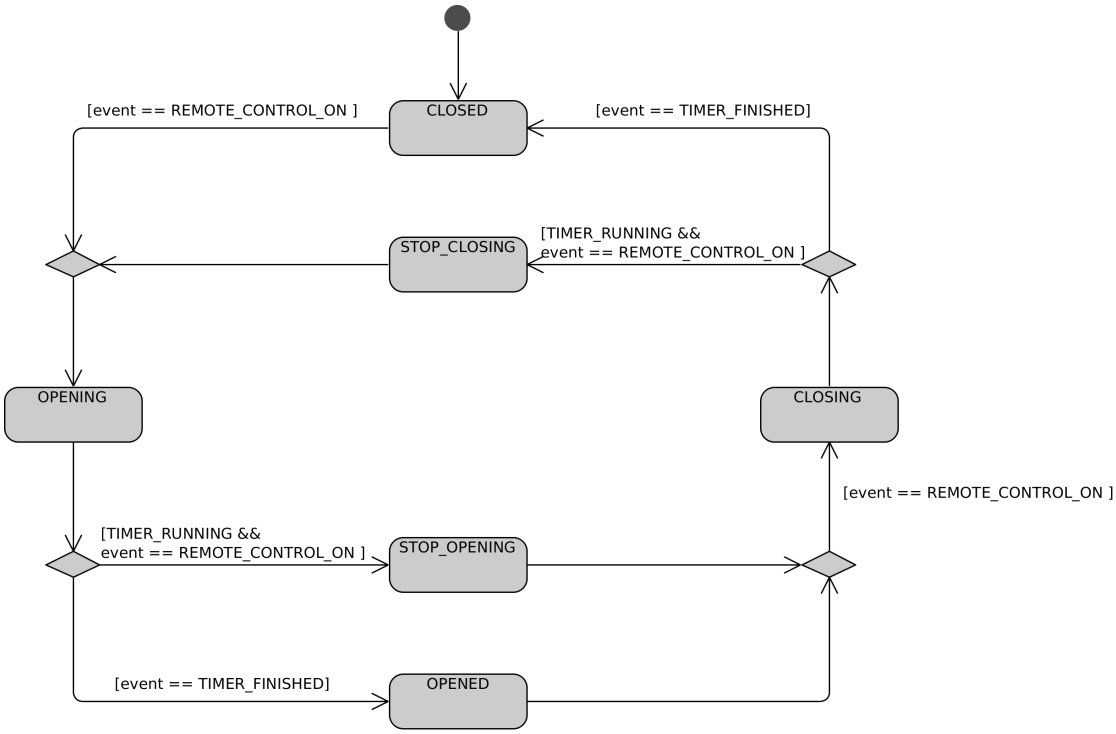

Figura 40: Diagrama de estados do Simulador Portão Eletrônico.

Neste caso de estudo, de forma análoga ao caso de estudo anterior (Simulador de Transporte Individual), os eventos recebidos do controle remoto (eventos externos) são utilizados pelo *software* para tomada de decisão. Neste caso, porém, eventos internos (gerados pelo *timer* - contador de tempo – também são tratados). Assim, o *software* deve tratar eventos levando em conta que a mudança de estado (e respectivo comportamento) não deve ser imediata (deve-se aguardar e reagir somente após um sinal discreto do *timer*).

Esta construção é relevante pois se almeja compor e estudar *software* em PON e POE, neste caso a partir de um sistema já modelado e construído originalmente em PON em outros trabalhos. Em PON, o *software* foi totalmente reconstruído utilizando o *Framework* otimizado em C++ e também em LingPON (linguagem e PON-Compilador embrionário) – (Solução I).

Em POE a construção foi realizada, de forma inédita, nas três técnicas para tratamento de eventos *Dispatcher*, Máquina de Estados usando *State Pattern* e *Observer* (Soluções II, III e IV), conforme as seções que seguem.

(I) Solução PON para o Portão Eletrônico.

Aqui se reimplementa o caso de estudo SPE a partir do avanço obtido com o *Framework* PON otimizado em C++. Originalmente o *software* havia sido construído por [Wiecheteck, 2011], utilizando como plataforma o *Framework* PON Original, proposto por [Banaszewski, 2009]. A Figura 41 ilustra o modelo estrutural obtido a partir da análise de requisitos, do modelo de Caso de Uso e progressão de estados.

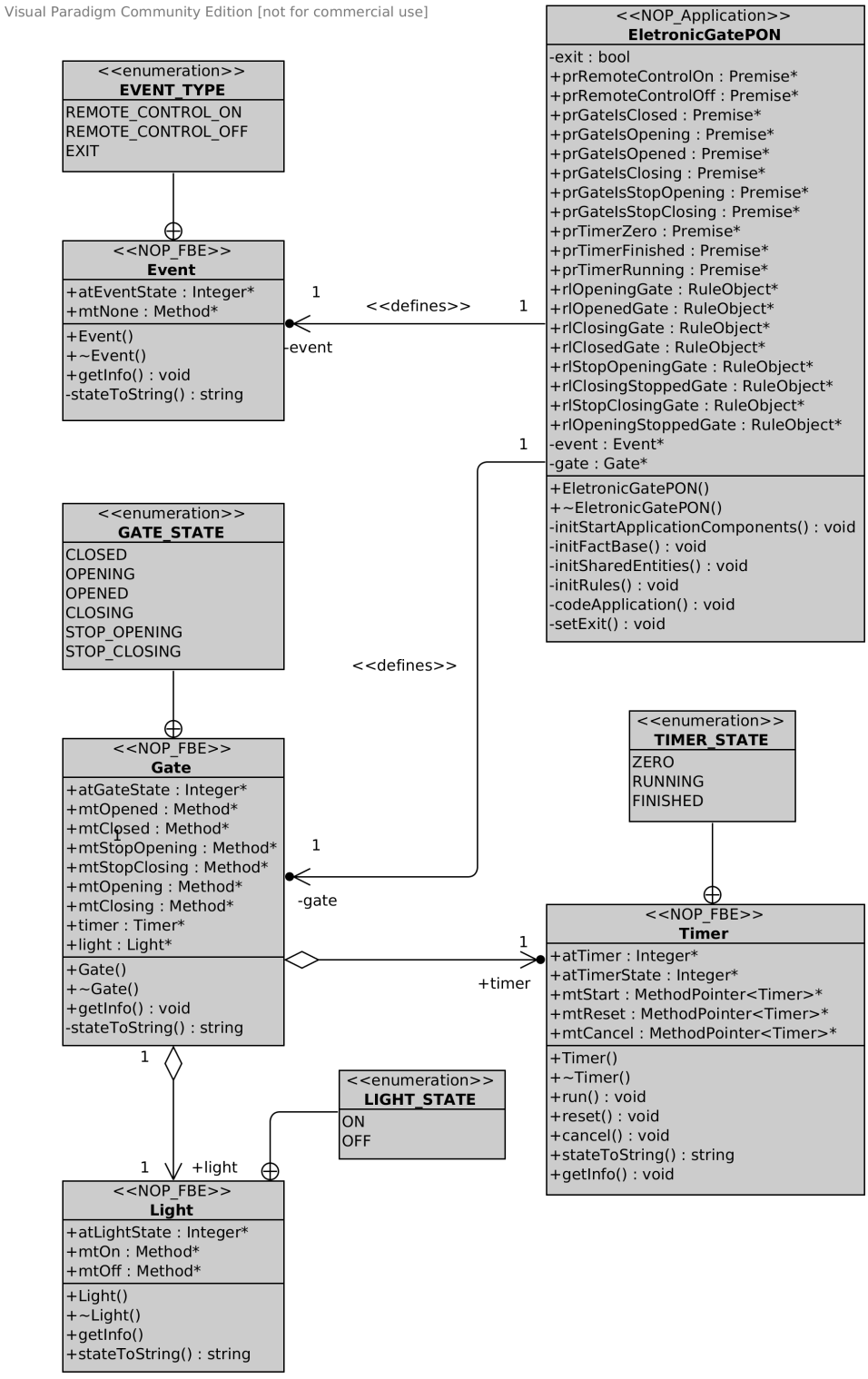

Figura 41: Diagrama de Classes do *software* Simulador de Portão Eletrônico em PON.

O modelo é demonstrado em PON por meio de um diagrama de classes UML, identificando os elementos que compõe a base de fatos *FBE* estereotipado <<NOP_FBE>> e

a aplicação PON estereotipada <<NOP_Application>>.

Conforme o diagrama apresentado na Figura 42, basicamente existem 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. A classe *ElectronicGatePON* é a classe principal em PON. Enumeradores também foram utilizadas para compor os tipos de estado dos objetos (*GATE_STATE, TIMER_STATE, LIGHT_STATE* e *EVENT_TYPE*). *Gate* contém o enumerador *GATE_STATE* que representa seus estados Fechado, Abrindo, Aberto, Fechando, Parou de Abrir, Parou de Fechar (*CLOSED*, *OPENING*, *OPENED*, *CLOSING*, *STOP_OPENING* e *STOP_CLOSING*). *Timer* contém o enumerador *TIMER_STATE* que representa seus estados Zerado, Rodando e Terminado (*ZERO*, *RUNNING* e *FINISHED*). O *FBE Light* contém o enumerador *LIGHT_STATE* que representa seus estados Ligado e Desligado (*ON* e *OFF*). E por fim, *Event* utiliza o enumerador *EVENT_TYPE* que são os eventos possíveis para tratamento pelo *software* neste caso de estudo, que apenas avalia se o botão do Controle Remoto foi pressionado (*REMOTE_CONTROL_ON*).

Como originalmente exposto em [Wiecheteck, 2011] e [Batista, 2013] (e no caso de estudo anterior), também se realiza a criação das regras a partir das perguntas básicas do Desenvolvimento Orientado a Notificações – DON. As perguntas e respectivas regras, para todo o *software*, são listadas a seguir:

1) Regra: Abrir o Portão

Qual o objetivo da regra? Iniciar a execução de abertura do portão.

O que precisa acontecer para que a regra seja executada? O portão tem que estar no estado FECHADO (*CLOSED*), o contador tem que estar zerado (*ZERO*) e o evento recebido deve ser CONTROLE REMOTO PRESSIONADO (*REMOTE_CONTROL_ON*).

O que acontece se a regra for executada? O portão deve mudar para o estado ABRINDO (*OPENING*) e o contador deve EXECUTAR (*RUNNING).*

2) Regra: Acender a lâmpada.

Qual o objetivo da regra? Acender a lâmpada.

O que precisa acontecer para que a regra seja executada? O portão tem que estar no estado FECHADO (*CLOSED*) e o evento recebido deve ser CONTROLE REMOTO PRESSIONADO (*REMOTE_CONTROL_ON*).

O que acontece se a regra for executada? A lâmpada deve ser acesa, mudando o estado

para LIGADA (*ON*).

3) Regra: Terminar de Abrir o Portão

Qual o objetivo da regra? Terminar a abertura do portão.

O que precisa acontecer para que a regra seja executada? O portão tem que estar no estado ABRINDO (*OPENING*) e o contador tem que terminar a contagem (*FINISHED*).

O que acontece se a regra for executada? O portão deve mudar para o estado ABERTO (*OPENED*) e o contador deve ser ZERADO (*ZERO).*

4) Regra: Abrir Portão que Parou de Fechar

Qual o objetivo da regra? Iniciar a execução de abertura do portão caso a execução de fechamento tenha sido interrompida.

O que precisa acontecer para que a regra seja executada? O portão tem que estar no estado PAROU DE FECHAR (*STOP_CLOSING*), o contador tem que estar ZERADO (*ZERO*) e o evento recebido deve ser CONTROLE REMOTO PRESSIONADO (*REMOTE_CONTROL_ON*).

O que acontece se a regra for executada? O portão deve mudar para o estado ABRINDO (*OPENING*) e o contador deve EXECUTAR (*RUNNING).*

5) Regra: Fechar o Portão

Qual o objetivo da regra? Iniciar a execução de fechamento do portão.

O que precisa acontecer para que a regra seja executada? O portão tem que estar no estado ABERTO (*OPENED*), o contador tem que estar ZERADO (*ZERO*) e o evento recebido deve ser CONTROLE REMOTO PRESSIONADO (*REMOTE_CONTROL_ON*).

O que acontece se a regra for executada? O portão deve mudar para o estado FECHANDO (*CLOSING*) e o contador deve EXECUTAR (*RUNNING).*

6) Regra: Apagar a lâmpada.

Qual o objetivo da regra? Apagar a lâmpada.

O que precisa acontecer para que a regra seja executada? O portão tem que estar no estado ABERTO (*OPENED*) e o evento recebido deve ser CONTROLE REMOTO PRESSIONADO (*REMOTE_CONTROL_ON*).

O que acontece se a regra for executada? A lâmpada deve ser apagada, mudando o estado para DESLIGADA *(OFF).*

7) Regra: Terminar de Fechar o Portão

Qual o objetivo da regra? Terminar o fechamento do portão.

O que precisa acontecer para que a regra seja executada? O portão tem que estar no estado FECHANDO (*CLOSING*) e o contador tem que terminar a contagem (*FINISHED*).

O que acontece se a regra for executada? O portão deve mudar para o estado FECHADO (*CLOSED*) e o contador deve ser ZERADO (*ZERO)*.

8) Regra: Fechar o Portão que Parou de Abrir

Qual o objetivo da regra? Iniciar a execução de fechamento do portão caso a execução de abertura tenha sido interrompida.

O que precisa acontecer para que a regra seja executada? O portão tem que estar no estado PAROU DE ABRIR (*STOP_OPENING*), o contador tem que estar ZERADO (*ZERO*) e o evento recebido deve ser CONTROLE REMOTO PRESSIONADO (*REMOTE_CONTROL_ON*).

O que acontece se a regra for executada? O portão deve mudar para o estado FECHANDO (*CLOSING*) e o contador deve EXECUTAR (*RUNNING)*.

9) Regra: Parar de Abrir o Portão

Qual o objetivo da regra? Interromper a execução de abertura do portão.

O que precisa acontecer para que a regra seja executada? O portão tem que estar no estado ABRINDO (*OPENING*), o contador tem que estar RODANDO (*RUNNING*) e o evento recebido deve ser CONTROLE REMOTO PRESSIONADO (*REMOTE_CONTROL_ON*).

O que acontece se a regra for executada? O portão deve mudar para o estado PAROU DE ABRIR (*STOP_OPENING*) e o contador deve ser ZERADO (*ZERO)*.

10) Regra: Parar de Fechar o Portão

Qual o objetivo da regra? Interromper a execução de fechamento do portão.

O que precisa acontecer para que a regra seja executada? O portão tem que estar no estado FECHANDO (*CLOSING*), o contador tem que estar RODANDO (*RUNNING*) e o evento recebido deve ser CONTROLE REMOTO PRESSIONADO (*REMOTE_CONTROL_ON*).

O que acontece se a regra for executada? O portão deve mudar para o estado PAROU

DE FECHAR (*STOP_CLOSING*) e o contador deve ser ZERADO (*ZERO).*

Assim, para cumprir os requisitos desse caso de estudo foram implementadas dez (10) *Rules* listadas na Tabela 5.

Tabela 5: *Rules*, *Condition* e suas *Premises* e *Methods* instigados do *software* Simulador de Portão Eletrônico em PON.

Rule	**Nome**	***Condition* e suas *Premises***	***Methods* instigados**
1	rlOpeningGate	atGateState == CLOSED && atTimerState == ZERO && atEventState == REMOTE_CONTROL_ON	Gate→mtOpening Timer→mtStart
2	rlTurnOnLight	atGateState == CLOSED && atEventState == REMOTE_CONTROL_ON	Light→mtOn
3	rlOpenedGate	atGateState == OPENING && atTimerState == FINISHED	Gate→mtOpened Timer→mtReset
4	rlOpeningStoppedGate	atGateState == STOP_CLOSING && atTimerState == ZERO && atEventState == REMOTE_CONTROL_ON	Gate→mtOpening Timer→mtStart
5	rlClosingGate	atGateState == OPENED && atTimerState == ZERO && atEventState == REMOTE_CONTROL_ON	Gate→mtClosing Timer→mtStart
6	rlTurnOffLight	atGateState == OPENED && atEventState == REMOTE_CONTROL_ON	Light→mtOff
7	rlClosedGate	atGateState == CLOSING && atTimerState == FINISHED	Gate→mtClosed Timer→mtReset
8	rlClosingStoppedGate	atGateState == STOP_OPENING && atTimerState == ZERO && atEventState == REMOTE_CONTROL_ON	Gate→mtClosing Timer→mtStart
9	rlStopOpeningGate	atGateState == OPENING && atTimerState == RUNNING && atEventState == REMOTE_CONTROL_ON	Gate→mtStopOpening Timer→mtCancel
10	rlStopClosingGate	atGateState == CLOSING && atTimerState == RUNNING && atEventState == REMOTE_CONTROL_ON	Gate→mtStopClosing Timer→mtCancel

Compor *software* em PON é relacionar regras para execução de capacidades de *FBEs*. Ou seja, o trabalho de criação, e respectivas escolhas do programador, estão na composição das *Rules.* Dessa forma, agregar ou dividir *Rules* também são escolhas do programador. Como exemplo, otimizar regras, fundir ou dividir regras também. Encapsular métodos também é uma opção do programador. Ou seja, o que é executado pode estar encapsulado em um método ou efetivamente composto na *Regra*. Como exemplo, um *FBE* pode conter somente os métodos *Light->mtOn()* ou *Timer->reset(),* e a execução ficar por cargo de outro método composto na própria *Rule* (dessa forma gerando acoplamento entre métodos). Ou o método pode ficar explícito via *Rule* como construído nesta solução.

O Algoritmo 17 mostra o código fonte de algumas dessas *Rules*, dentro do contexto do *Framework* otimizado em C++ do PON, para a operação do Portão Eletrônico. São as *Rules* para iniciar e terminar a abertura do Portão, e iniciar e terminar o fechamento do mesmo.

Algoritmo 17: Código de quatro *Rules* em ElectronicGatePON.cpp.

```
...
void ElectronicGatePON::initRules() {
	Scheduler* scheduler = SingletonScheduler::getInstance();

	RULE(rlOpeningGate, scheduler, Condition::CONJUNCTION);
	 rlOpeningGate->addPremise(prRemoteControlOn);
	 rlOpeningGate->addPremise(prGateIsClosed);
	 rlOpeningGate->addPremise(prTimerZero);
	 rlOpeningGate->addInstigation(INSTIGATION(gate->mtOpening));
	 rlOpeningGate->addInstigation(INSTIGATION(gate->timer->mtStart));
	 rlOpeningGate->addInstigation(INSTIGATION(event->mtNone));

	RULE(rlOpenedGate, scheduler, Condition::CONJUNCTION);
	 rlOpenedGate->addPremise(prGateIsOpening);
	 rlOpenedGate->addPremise(prTimerFinished);
	 rlOpenedGate->addInstigation(INSTIGATION(gate->mtOpened));
	 rlOpenedGate->addInstigation(INSTIGATION(gate->timer->mtReset));

	RULE(rlClosingGate, scheduler, Condition::CONJUNCTION);
	 rlClosingGate->addPremise(prRemoteControlOn);
	 rlClosingGate->addPremise(prGateIsOpened);
	 rlClosingGate->addPremise(prTimerZero);
	 rlClosingGate->addInstigation(INSTIGATION(gate->mtClosing));
	 rlClosingGate->addInstigation(INSTIGATION(gate->timer->mtStart));
	 rlClosingGate->addInstigation(INSTIGATION(event->mtNone));

	RULE(rlClosedGate, scheduler, Condition::CONJUNCTION);
	 rlClosedGate->addPremise(prGateIsClosing);
	 rlClosedGate->addPremise(prTimerFinished);
	 rlClosedGate->addInstigation(INSTIGATION(gate->mtClosed));
	 rlClosedGate->addInstigation(INSTIGATION(gate->timer->mtReset));
...
```

Parte da implementação em LingPON, apenas com 4 (quatro) *Rules*, é apresentada no Algoritmo 18.

Algoritmo 18: *FBEs Gate* e *Event, e* quatro *Rules* em código LingPON em electronicgate.pon

```
 1 fbe Gate
 2     attributes
 3         integer atGateState 0
 4     end_attributes
 5     methods
 6         method mtClosed(atGateState = 0)
 7         method mtOpening(atGateState = 1)
 8         method mtOpened(atGateState = 2)
 9         method mtClosing(atGateState = 3)
10         method mtStopOpening(atGateState = 4)
11         method mtStopClosing(atGateState = 5)
12     end_methods
13 end_fbe
14
15 fbe Event
16     attributes
17         integer atEventState 0
18     end_attributes
19     methods
20         method mtNone ( atEventState = 0 )
21     end_methods
22 end_fbe

... ...

46 rule rlOpeningGate
47     condition
48     subcondition a1
49             premise prRemoteControlOn event.atEventState == 1 and
50             premise prGateIsClosed gate.atGateState == 0 and
51             premise prTimerZero timer.atTimerState == 0
52     end_subcondition
53     end_condition
54     action
55         instigation inOpening1 gate.mtOpening();
56         instigation inStart1 timer.mtStart();
57         instigation inNone1 event.mtNone();
58     end_action
59 end_rule
60
61 rule rlOpenedGate
62     condition
63     subcondition a2
64             premise prGateIsOpening gate.atGateState == 1 and
65             premise prTimerFinished timer.atTimerState == 2
66     end_subcondition
67     end_condition
68     action
69         instigation inOpened2 gate.mtOpened();
70         instigation inReset2 timer.mtReset();
71     end_action
72 end_rule
73
74 rule rlClosingGate
75     condition
```

```
    subcondition a3
            premise prRemoteControlOn event.atEventState == 1 and
            premise prGateIsOpened gate.atGateState == 2 and
            premise prTimerZero timer.atTimerState == 0
    end_subcondition
    end_condition
    action
        instigation inClosing3 gate.mtClosing();
        instigation inStart3 timer.mtStart();
        instigation inNone3 event.mtNone();
    end_action
end_rule

rule rlClosedGate
    condition
    subcondition a4
            premise prGateIsClosing gate.atGateState == 3 and
            premise prTimerFinished timer.atTimerState == 2
    end_subcondition
    end_condition
    action
        instigation inClosed4 gate.mtClosed();
        instigation inReset4 timer.mtReset();
    end_action
end_rule
```

A seguir são apresentadas as soluções para este caso de estudo em POE.

(II) Primeira Solução POE (*Handler-Dispatcher*) para o Portão Eletrônico.

Assim como no primeiro caso de estudo, em todas as soluções técnicas em POE a relação indireta foi aplicada entre os objetos transmissores e objetos receptores, empregando o componente despachante – *Handler-Dispatcher*. A Figura 42 ilustra o projeto estrutural da primeira solução em POE para o Simulador de Portão Eletrônico.

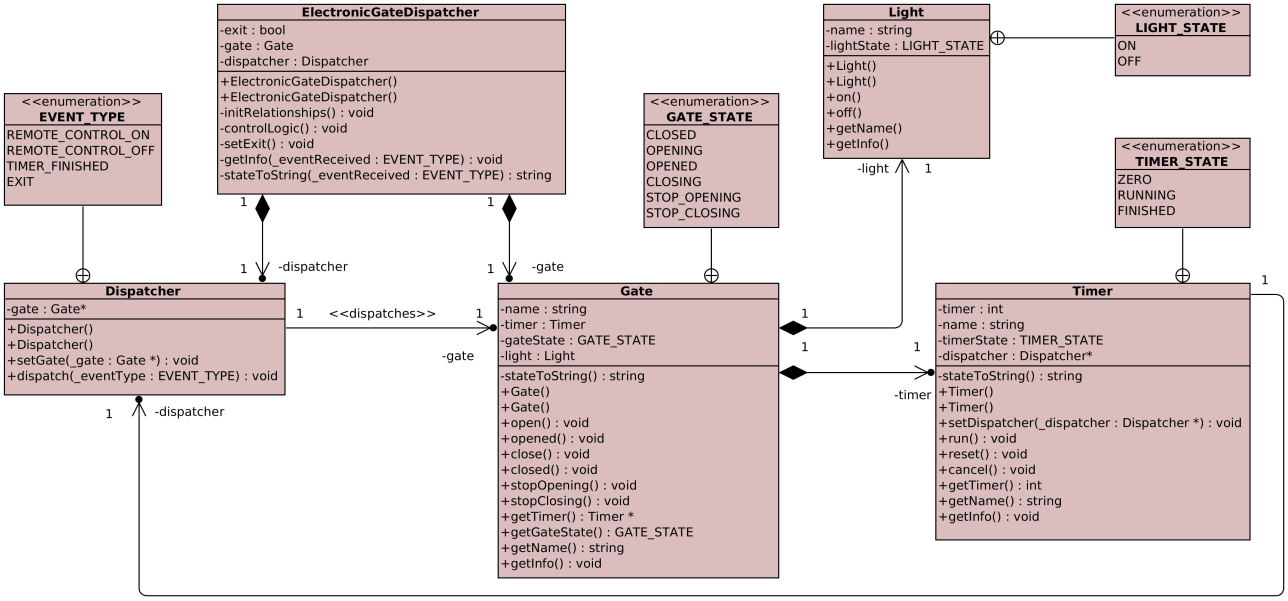

Figura 42: Diagrama de Classes do *software* Simulador de Portão Eletrônico em POE - *Dispatcher*.

Dessa maneira, para realizar a primeira solução do caso de estudo Simulador de Portão Eletrônico em POE *Dispatcher*, foram construídas cinco classes em C++: *Gate*, *Timer* e *Light*; *Dispatcher* (despachante) e *ElectronicGateDispatcher* (*i.e.* classe de principal). Além disso, os enumeradores C++ utilizados foram *GATE_STATE, EVENT_TYPE, LIGHT_STATE* e *TIMER_STATE*.

O diagrama de sequência da Figura 43 apresenta uma interação utilizando a técnica *Handler-Dispatcher*. A interação executa uma sequência de três eventos recebidos de *REMOTE_CONTROL_ON*. Ou seja, o controle remoto é acionado três vezes na sequência. O objetivo é demonstrar a dinâmica, nesta solução, para abrir, parar de abrir e fechar o portão eletrônico, respectivamente os estados *OPENING*, *STOP_OPENING*, *CLOSING* e *CLOSED*.

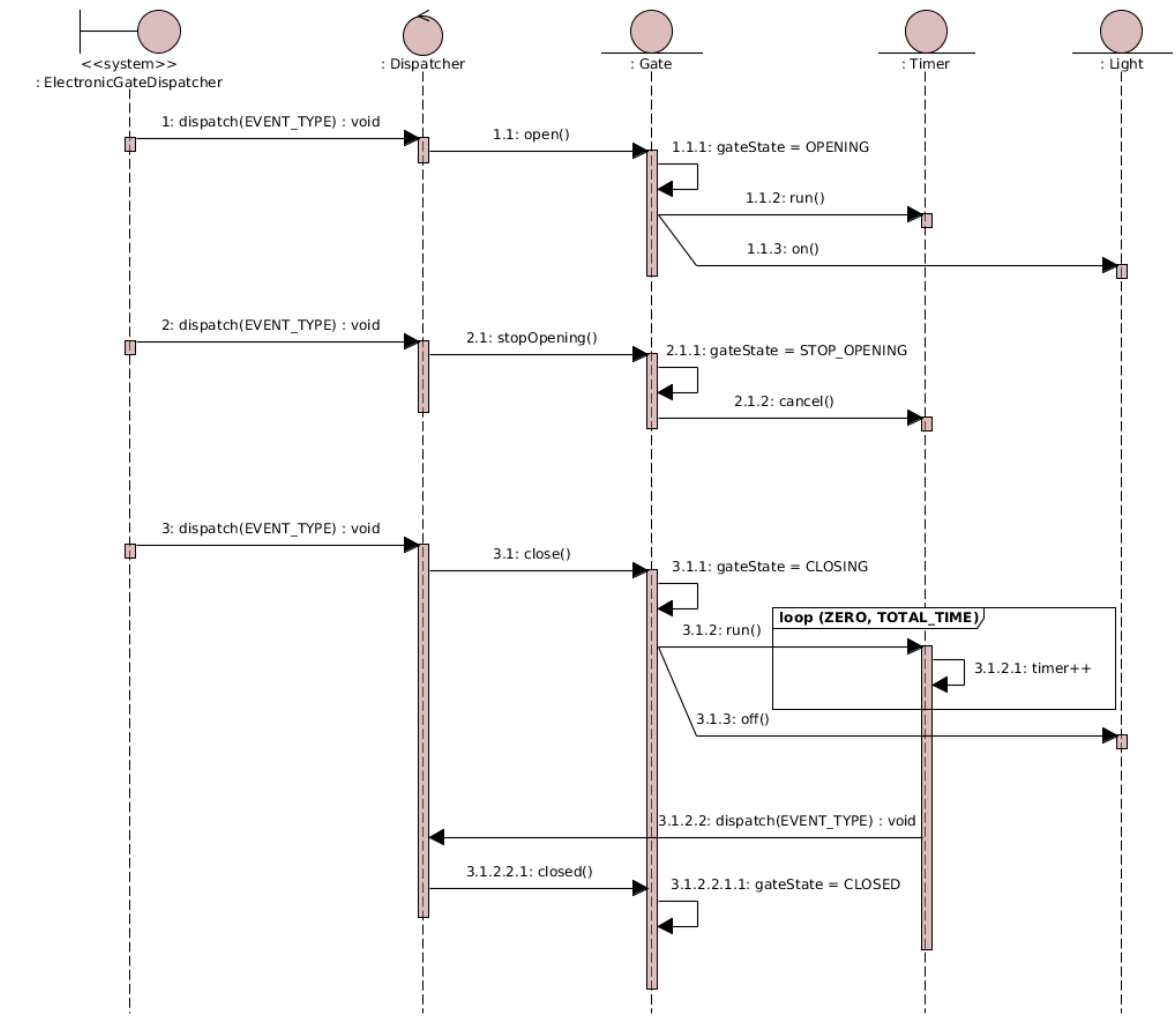

Figura 43: Diagrama de Sequência de *Handler-Dispatcher* em POE para o Simulador de Portão Eletrônico.

O Algoritmo 19 apresenta o método *dispatch()* de *Dispatcher* (ciclo de reconhecimento de eventos ou demultiplexador) neste caso de estudo Portão Eletrônico.

Algoritmo 19: Código para execução de ações via objeto Despachante em *Dispatcher.cpp* utilizando *SWITCH-CASE* do Simulador de Portão Eletrônico.

```
...
void Dispatcher::dispatch(EVENT_TYPE _eventType) {
    switch (gate->getGateState()) {
       case (Gate::CLOSED): {
              switch (_eventType) {
                  case (REMOTE_CONTROL_ON): { this->gate->open();  break; }
                  default: { break; }
              }
       } break;
       case (Gate::OPENING): {
              switch (_eventType) {
                  case (REMOTE_CONTROL_ON): {  this->gate->stopOpening(); break; }
                  case (TIMER_FINISHED): { this->gate->opened(); break; }
                  default: { break; }
              }
       } break;
       case (Gate::OPENED): {
              switch (_eventType) {
                   case (REMOTE_CONTROL_ON): { this->gate->close(); break; }
                   default: { break; }
              }
       } break;
       case (Gate::CLOSING): {
              switch (_eventType) {
                  case (REMOTE_CONTROL_ON): { this->gate->stopClosing(); break; }
                  case (TIMER_FINISHED): { this->gate->closed(); break; }
                  default: { break; }
              }
       } break;
       case (Gate::STOP_OPENING): {
              switch (_eventType) {
                  case (REMOTE_CONTROL_ON): { this->gate->close(); break; }
                  default: { break; }
              }
              } break;
       case (Gate::STOP_CLOSING): {
              switch (_eventType) {
                  case (REMOTE_CONTROL_ON): {  this->gate->open(); break; }
                  default: { break; }
              }
       } break;
       default: { break; }
    }
}
```

(III) Segunda Solução POE (*State Pattern*) para o Portão Eletrônico.

Nessa segunda solução em POE, além do componente despachante – *Handler-Dispatcher* – uma Máquina de Estados também foi utilizada e implementada por

meio de *State Pattern* [Faison, 1993][Gamma et. al, 1995]. A Figura 44 apresenta um excerto do projeto estrutural dessa solução usando o padrão Máquina de Estados (*State*) para realizar o remapeamento do tratamento de eventos pelo portão eletrônico (*Gate*).

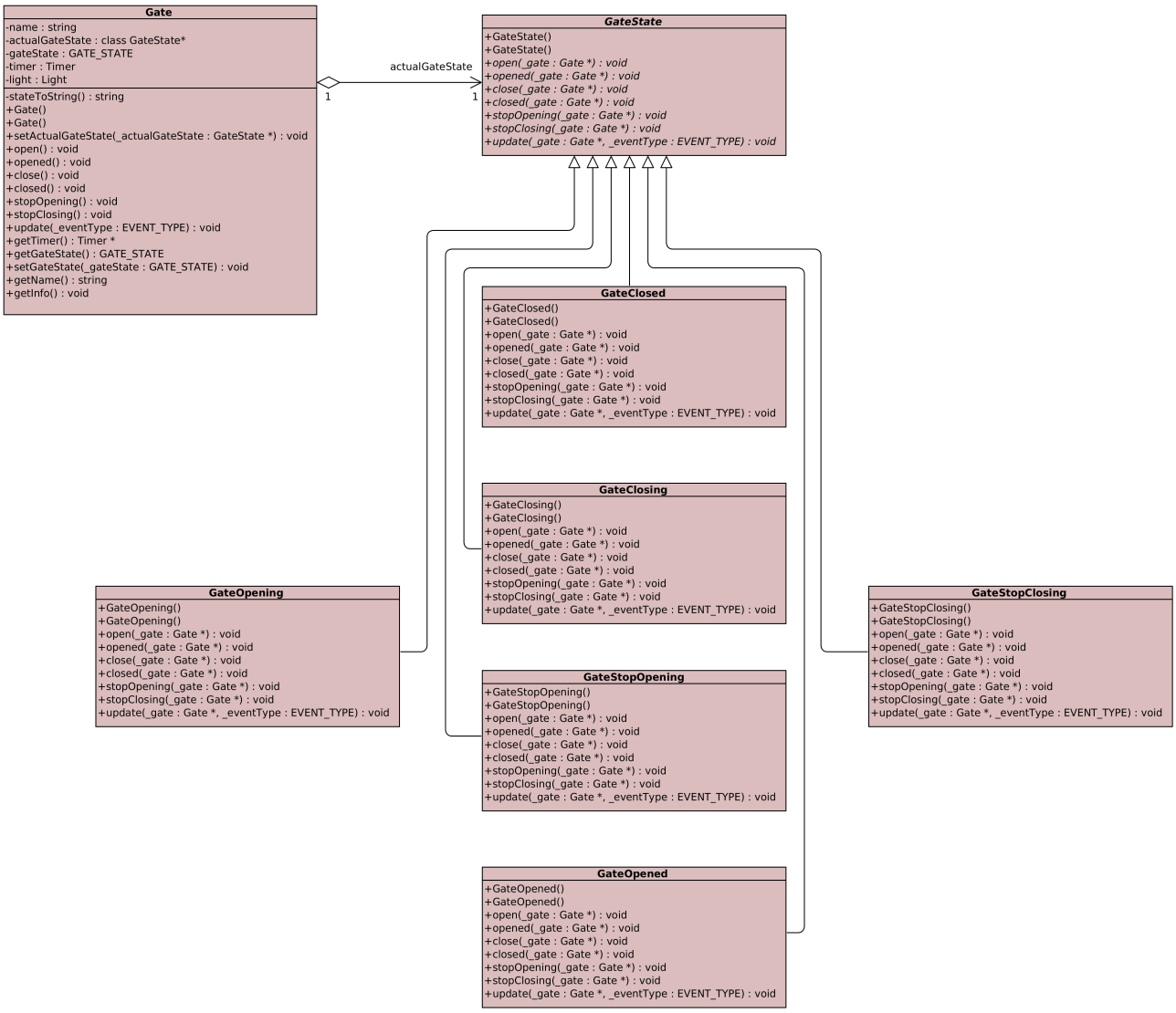

Figura 44: Excerto do Diagrama de Classes do *software* Simulador de Portão Eletrônico em POE – *State Pattern.*

Nessa construção, uma respectiva instância da hierarquia *GateState* e respectivas subclasses devem executar os métodos conforme as especificações apresentadas nos requisitos. As subclasses criadas foram *GateClosed*, *GateOpening*, *GateOpened*, *GateClosing*, *GateStopOpening*, *GateStopClosing*. Estas subclasses encapsulam os comportamentos corretos para cada estado do portão eletrônico, a saber, Fechado, Abrindo, Aberto, Fechando, Parou de Abrir e Parou de Fechar.

Assim, para realizar essa segunda solução em POE do segundo caso de estudo, doze classes foram construídas em C++: *Gate*, *Timer* e *Light*; classes das técnicas *Dispatcher* e *State Pattern* e *ElectronicGateState* (*i.e.* classe de principal), como apresentado (parcialmente) na Figura 45. Além disso, os enumeradores C++ utilizados foram *GATE_STATE*, *EVENT_TYPE*, *LIGHT_STATE* e *TIMER_STATE*.

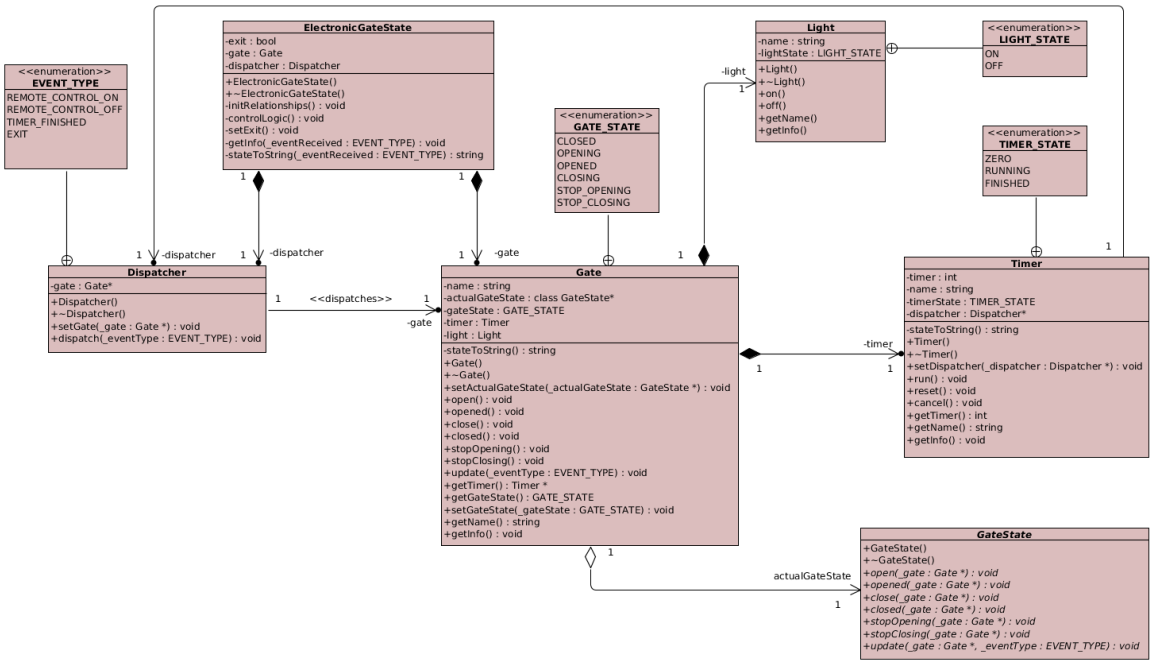

Figura 45: Excerto do Diagrama de Classes do *software* Simulador de Portão Eletrônico em POE.

Com o uso de *State*, a solução *SWITCH-CASE* de *Dispatcher* migrou para cada subclasse da hierarquia, conforme o estado especificado em sua respectiva subclasse. Cada nível, desde o mais alto, da estrutura de controle *SWITCH-CASE*, origina uma subclasse. Os tratamentos de eventos, conforme o contexto (estado), são realizados pelo método *update(_gate : Gate*, _eventType : EVENT_TYPE)*. Esta é exatamente a correlação encontrada na literatura em [Samek, 2008]. Os algoritmos 20, 21, 22, 23, 24 e 25 a seguir ilustram os códigos-fonte dos tratamentos de eventos – método *update()* – em cada subclasse.

Algoritmo 20: Código do método *update()* em GateClosed.cpp (*State Pattern*)

```
41 ...
42 void GateClosed::update(Gate* _gate, EVENT_TYPE _eventType) {
43        switch (_eventType) {
44               case (REMOTE_CONTROL_ON): { this->open(_gate); break; }
45               default : { break; }
46        }
47 }
```

Algoritmo 21: Código do método *update()* em GateClosing.cpp (*State Pattern*).

```
46 ...
47 void GateClosing::update(Gate* _gate, EVENT_TYPE _eventType) {
48        switch (_eventType) {
49               case (REMOTE_CONTROL_ON): { this->stopClosing(_gate); break; }
50               case (TIMER_FINISHED)   : { this->closed(_gate); break; }
51               default : { break; }
52        }
53 }
```

Algoritmo 22: Código do método *update()* em GateOpened.cpp (*State Pattern*)

```
41 ...
42 void GateOpened::update(Gate* _gate, EVENT_TYPE _eventType) {
43        switch (_eventType) {
44               case (REMOTE_CONTROL_ON): { this->close(_gate); break; }
45               default : { break; }
46        }
47 }
```

Algoritmo 23: Código do método *update()* em GateOpening.cpp (*State Pattern*)

```
46 ...
47 void GateOpening::update(Gate* _gate, EVENT_TYPE _eventType) {
48        switch (_eventType) {
49               case (REMOTE_CONTROL_ON) : { this->stopOpening(_gate); break; }
50               case (TIMER_FINISHED) : { this->opened(_gate); break; }
51               default : { break; }
52        }
53 }
```

Algoritmo 24: Código do método *update()* em GateStopClosing.cpp (*State Pattern*)

```
41 ...
42 void GateStopClosing::update(Gate* _gate, EVENT_TYPE _eventType) {
43        switch (_eventType) {
44               case (REMOTE_CONTROL_ON): { this->open(_gate); break; }
45               default : { break; }
46        }
47 }
```

Algoritmo 25: Código do método *update()* em GateStopOpening.cpp (*State Pattern*)

```
41 ...
42 void GateStopOpening::update(Gate* _gate, EVENT_TYPE _eventType) {
43       switch (_eventType) {
44             case (REMOTE_CONTROL_ON): { this->close(_gate); break; }
45             default : { break; }
46       }
47 }
```

(IV) Terceira Solução POE (*Observer Pattern*) para o Portão Eletrônico.

Nesta terceira solução, o componente despachante – *Handler-Dispatcher* – e o padrão de projeto *Observer* [Gamma et. al, 1995] foram utilizados em conjunto. A Figura 46 apresenta um excerto do projeto estrutural dessa solução para realizar o tratamento de eventos pelo portão eletrônico (*Gate*).

Assim, para realizar essa terceira solução em POE do segundo caso de estudo SPE sete classes foram construídas em C++: *Gate*, *Timer* e *Light*; classes das técnicas *Dispatcher* e *Observer Pattern* e *ElectronicGateObserver* (*i.e.* classe de principal), como demonstrado na Figura 46. Enumeradores C++ também foram utilizados: *GATE_STATE, EVENT_TYPE, LIGHT_STATE* e *TIMER_STATE*.

Assim como no terceiro cenário do primeiro caso de estudo, a finalidade de uso de *Observer Pattern* é diminuir o acoplamento (*i.e.* dependência) entre o componente *Dispatcher* e os tratadores de eventos. Neste caso de estudo, na solução proposta, *Dispatcher* unifica o recebimento de eventos, sendo o único *Subject*. E *Gate* é o principal tratador de eventos e executor das principais ações, sendo o principal *Observer* dos estados (eventos) recebidos pelo *software* (*i.e.* componente *Dispatcher*).

Os componentes *Light* e *Timer* são agregados por composição ao componente *Gate.* Caso assim não fosse, seria necessário relacioná-los a uma outra classe (*e.g.* classe principal *ElectronicGateObserver*). Nesse sentido, a solução aqui adotada facilita um algoritmo centralizado com o que deve ser executado.

Da mesma maneira, seriam possíveis outras soluções, como exemplo espalhar os comportamentos em outros componentes, ou até mesmo dispersar o algoritmo de tratamento em outros componentes, ou ainda cada componente poderia ser um *Observer* e respectivamente *Subject* para outros componentes [Gamma *et al.*, 1995]. Contudo, aumentaria a complexidade da solução.

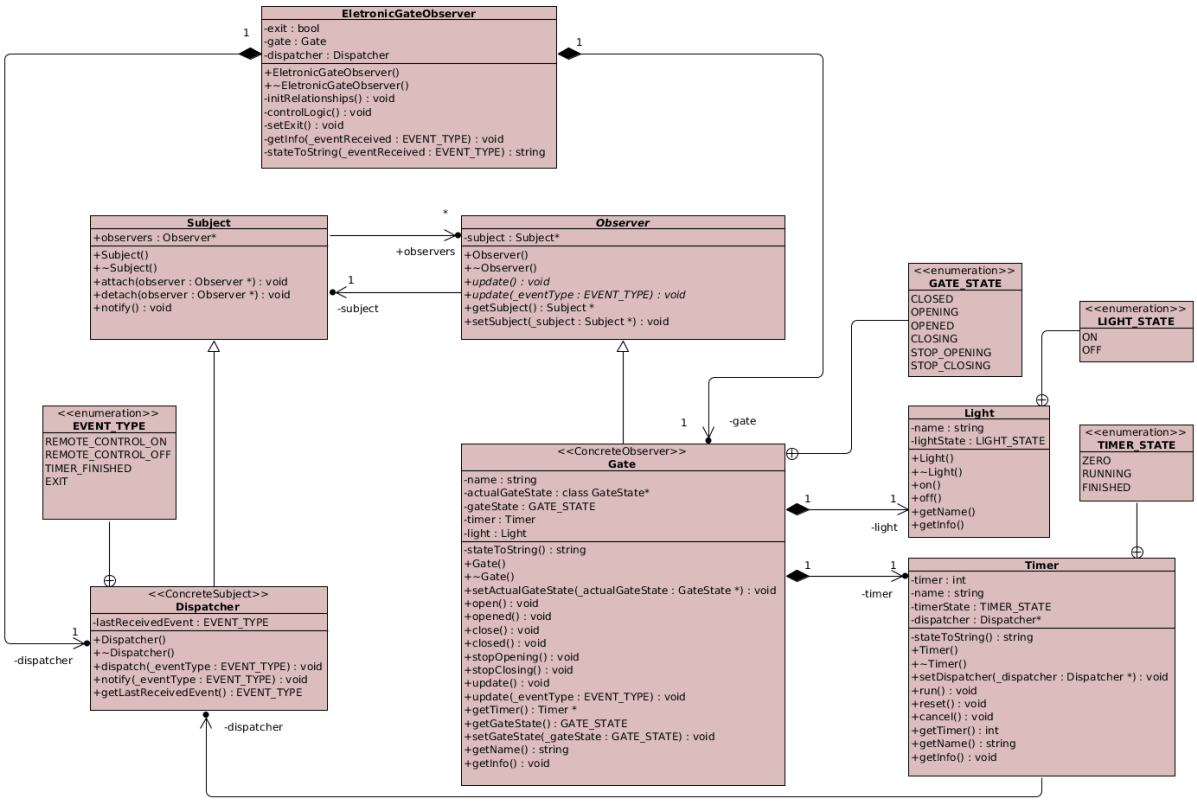

Figura 46: Diagrama de Classes do *software* Simulador de Portão Eletrônico em POE - *Observer*.

A Algoritmo 26 mostra o método *update()* de *Gate* (ciclo de reconhecimento de eventos ou demultiplexador) nesta solução. Assim como no primeiro caso de estudo, a lógica e respectiva estrutura *SWITCH-CASE* migra de *Dispatcher* para o objeto observador – *Observer* – nesse caso o componente *Gate*, de certa maneira o principal *ConcreteObserver* da solução. Conforme os requisitos, é necessário controlar o início, interrupção, fim e reinício do *timer,* mais o controle da luz (ligar e desligar), ou seja, mais de uma ação por evento recebido, conforme o contexto de funcionamento do *software* (estados).

Nesse contexto, é necessário controlar corretamente o funcionamento dessas ações tomadas pelo *software,* executando todas as ações necessárias corretamente em cada estado do funcionamento do portão (como diagramado em Figura 40: Diagrama de estados do Simulador Portão Eletrônico.). Dessa forma, nessa solução adotada, *Gate* é o componente responsável por centralizar as decisões, sempre tratando todos os eventos (*e.g.* de controle remoto externo ao *software* como *REMOTE_CONTROL_ON* e de *timer* interno ao *software* como *TIMER_FINISHED*). Dessa forma, não existe a dinâmica de se anexar ou desanexar, ou seja, o componente *Gate* precisa efetivamente receber sempre todos os eventos.

Algoritmo 26: Excerto de código para execução de ações ou retransmissão de mensagem via objeto Despachante em Gate.cpp (um *ConcreteObserver*) utilizando *SWITCH-CASE* do Simulador de Portão Eletrônico.

```
...
void Gate::update(EVENT_TYPE _eventType) {
	switch (getGateState()) {
		case (Gate::CLOSED): {
			switch (_eventType) {
				case (REMOTE_CONTROL_ON): { this->open(); break; }
				default: { break; }
			}
		} break;
		case (Gate::OPENING): {
			switch (_eventType) {
				case (REMOTE_CONTROL_ON): { this->stopOpening(); break;}
				case (TIMER_FINISHED): {   this->opened(); break; }
				default: { break; }
			}
		} break;
		case (Gate::OPENED): {
			switch (_eventType) {
				case (REMOTE_CONTROL_ON): { this->close(); break; }
				default: { break; }
			}
		} break;
		case (Gate::CLOSING): {
			switch (_eventType) {
				case (REMOTE_CONTROL_ON): { this->stopClosing(); break;}
				case (TIMER_FINISHED): {   this->closed(); break; }
				default: { break; }
			}
		} break;
		case (Gate::STOP_OPENING): {
			switch (_eventType) {
				case (REMOTE_CONTROL_ON): { this->close(); break; }
				default: { break; }
			}
		} break;
		case (Gate::STOP_CLOSING): {
			switch (_eventType) {
				case (REMOTE_CONTROL_ON): { this->open(); break; }
				default: { break; }
			}
		} break;
	}
}

```

Como almejado, neste caso de estudo foi programado *software* que trata eventos de tempo (internos ao *software)*. Essa característica é importante pois o *software* muda de estado conforme os eventos de tempo, que não são gerados externamente. Este caso de estudo foi construído em *Framework* PON, LingPON, *Handler-Dispatcher*, *State* e *Observer., e* avaliações quantitativas são apresentadas no próximo capítulo, nas seções 4.2 Comparações em complexidade de código-fonte de software em PON e POE e 4.3 Comparações em

medidas de execução de software em PON e POE.

Ficou evidenciado no segundo caso de estudo a ocorrência de exagero de engenharia (*Over-Engineering*), particularmente ao se utilizar as técnicas *State* e *Observer* para construir o *software* Portão Eletrônico. Joshua Kerievsky descreve que a *Over-Engineering* é a produção de código mais flexível ou sofisticado do que necessita realmente ser, geralmente acontecendo por excesso de zelo e previsão (muito antecipada) de requisitos futuros do *software* [Kerievsky, 2004]. Percebe-se que a solução *Handler-Dispatcher* compondo o método *dispatch()* com um *SWITCH-CASE* aninhado já é adequada e necessariamente suficiente. O excesso de zelo, nesse caso, pode ser um indício intrínseco à construção de *software* (genericamente), que é agravado conforme mais técnicas (e suas composições) estão disponíveis.

3.3 Reflexões dos casos de estudo

Aqui se encerra esse capítulo com reflexões e discussões. Cabe ressaltar que as análises, escolhas, percepções e reflexões dos casos de estudo (tanto em PON quanto em POE) aqui apresentadas são percepções do autor (e respectivo programador de todas as implementações tanto em PON quanto em POE) deste trabalho.

No primeiro caso de estudo, foi muito simples adicionar novas funcionalidades ao *software* em PON entre os cenários. Novos *FBEs* e novas *Rules* já cumpriram os requisitos necessários. As atividades de expansão, refatoração e manutenção de *software* construído em PON perceberam-se como muito fáceis. O conhecimento expresso e centralizado em regras (*e.g.* PON), na percepção do autor deste trabalho, é mais próximo aos requisitos do *software*. Dessa forma tem melhor relação com o que se deseja que o *software* realize. As perguntas (DON) para criar as *Rules* são um indício dessa característica em PON. Assim, compor *software* em PON é relacionar *Rules* para execução de capacidades de *FBEs* (*e.g.* tecnicamente criar, agregar ou fundir, dividir, encapsular *Methods* são escolhas do projetista/programador).

Ao seu turno, em POE não há necessariamente indicação expressa e centralizada do conhecimento (*i.e.* de certa forma uma Máquina de Estados completa com todos os seus estados pode representar integralmente o conhecimento, entretanto cada estado isoladamente representa somente parcialmente o conhecimento). Assim, o programador escolhe, conforme

sua experiência e capacidade técnica, o "*Como*" expressar o conhecimento, quais componentes utilizar e como dispor cada comportamento (*e.g.* em qual estado da Máquina de Estados dispor o comportamento). Pode-se chamar tal fenômeno de dispersão de regras (*i.e.* comportamentos espalhados por vários componentes dispersos do *software*). Em contraponto, esse fenômeno pode ser visto como um indicativo de maior flexibilidade de escolha para o programador.

Esse fenômeno de dispersão é agravado com a preocupação inata de execução sequencial explícita do *software* com a qual o programador deve gerenciar constantemente (*e.g.* ordem de execução de instruções). Além disso, o diagrama de sequência adotado para modelagem pelos principais autores em POE evidencia essa preocupação constante com a própria sequencialidade ("*Qual*" comportamento é executado "*Quando*").

Sucintamente, aqui se reúnem as percepções e características já citadas durante o Design e respectiva programação dos casos de estudo em PON. Percebeu-se que adicionar funcionalidades foi rápido e fácil. De maneira inteligível (centralizada e assertiva) foi possível programar *software* para as seguintes situações: ocorrência de eventos em paralelo (*e.g.* botões pressionados ao mesmo tempo); uso de regras diferentes que tratam de um mesmo evento (*e.g.* uma única *Premise*) para execução de ações diferentes ao mesmo tempo (*e.g.* mover os dois braços ao mesmo tempo para cima); um evento relacionado a uma única regra que executa várias ações (*e.g.* desligar todos os aparatos). Outro ponto relevante é a característica das *Rules* serem orientadas a instância. Tal característica ainda pode ser experimentada em *software* de tratamento de eventos que requeira um número maior de instâncias.

Por outro lado, em POE, as percepções e características levantadas foram: existem necessidades constantes, por parte do programador, de gerenciar a criação e destruição de objetos que contém os comportamentos que o *software* deve realizar, de organizar a dependências entre objetos (*e.g.* alocação dinâmica e uso de ponteiros); se nota um esforço constante em evitar que a programação leve a um código complexo (isto é, que apresente lógica complicada, ou seja, código intrincado e de difícil compreensão posterior - *spaghetti code*).

Acerca das técnicas de POE, as mesmas foram aplicadas para resolver os requisitos de cada cenário, sendo boas soluções nesse sentido. Por exemplo, sem utilizar o padrão *State* não seria possível resolver o remapeamento de eventos para movimentar ora o exosqueleto ora o braço mecânico sem aumentar razoavelmente a quantidade de avaliações causais ou até mesmo utilizar variáveis globais. Dessa forma, é possível afirmar que são boas soluções, de

boa prática, em POE, e razoavelmente complexas. Não obstante, para cumprir assertivamente os requisitos, novas técnicas tiveram que ser utilizadas (*i.e. State* no segundo cenário e *Observer* no terceiro cenário do primeiro caso de estudo).

Ao se construir *software* utilizando padrão de projeto *Observer* (*e.g.* terceiro cenário e uma solução técnica em Portão Eletrônico) ficaram indícios de que o programador deve se ocupar dos detalhes de implementação como ordem de execução, quais métodos ficam em qual classe e em qual ordem no algoritmo (*e.g.* no ciclo de reconhecimento de eventos), qual objeto é responsável por manter o estado da aplicação (centralizando informações e aumentando o acoplamento), e quem é o componente responsável por anexar e desanexar o *Observer* ao *Subject* - observador ao observável (ações de *attach* e *detach*).

O *Observer Pattern* objetiva encapsulamento, menor acoplamento e reuso de componentes. Entretanto, tal objetivo não foi totalmente atingido, pois foi necessário compor *software* com componente que nunca deixa de observar (*e.g.* cumprindo o requisito do *software* de receber todos os eventos do controle remoto) e componente que também gerencia as relações entre observadores. Também ocorreu a gestão do estado de um componente por outro, indicando acoplamento (*e.g. Exoskeleton* centraliza e reconhece o evento que faz desligar *Arm*). A ressalva é que a utilização do padrão *Observer* para esses requisitos e casos talvez não seja indicada.

Por exemplo, *ChangeManager* poderia ser utilizado ([Gamma *et al.* 1995] cita como uma aplicação de padrão Mediator). Nesse caso, esse componente centralizaria a lógica para reconhecimento de eventos *SWITCH-CASE*, fazendo o mesmo papel de *Exoskeleton* no primeiro caso de estudo. O ganho nesse caso seria imperceptível, seria necessário mais um componente observador que observaria todos os eventos (que nunca deixaria, da mesma forma, de observar realizando uma operação de *detach*). Tal situação seria idêntica no segundo cenário.

Outra possibilidade seria utilizar o padrão *State* ao invés da estrutura de controle, nesse caso ocasionando várias novas classes para compor todo o comportamento da aplicação (*e.g.* como ocorreu no segundo caso de estudo na solução *State*). Uma das consequências seria o próprio aumento de componentes (classes), a lógica da aplicação ficaria dispersa em várias classes, como visto no segundo cenário, e um prognóstico de explosão combinatória de estados com possíveis manutenções evolutivas que aumentassem o número e a complexidade dos requisitos.

Pode-se afirmar que Padrões de *Software* devem ser considerados como receitas de boa prática e não efetivamente com um conjunto de regras a serem seguidas à risca. É

importante considerar que existem famílias e composições de padrões, com seus benefícios e respectivos custos (característica oriunda do inglês *tradeoffs* [Gamma et al, 1995]). Nem toda solução técnica serve para todas as situações.

Por exemplo, *Observer* é efetivo para casos nos quais, realmente, ele somente observa, para atualizar seu próprio estado sem precisar manter estados compartilhados. Nas palavras de [Kerievsky, 2004], o uso de padrões e projeto introduz flexibilidade, mas não se inicia imediatamente colocando padrões em um *software*. Todas essas observações suscitam perguntas que podem se motivadoras para novos trabalhos de pesquisa.

Capítulo 4 - Comparações entre os Paradigmas Orientado a Eventos e Orientado a Notificações

Conforme análise do levantamento bibliográfico e trabalhos correlatos, são listadas a seguir algumas formas usuais de sistematizar uma comparação entre paradigmas de programação:

- Demonstração matemática, como a correlação entre *Lambda Calculi* e Máquina de Turing [Turing, 1937], como por exemplo a demonstração matemática da linguagem Esterel [Berry e Cosserat, 1984];
- Características estruturantes, como uma taxonomia [Van Roy, 2009], tipos abstratos de dados, estruturas de controle [Watt, 2004], e funcionalidades [Jordan *et al.*, 2014];
- Medidas de *software* como medidas CK [Chidamber e Kemerer, 1994] em OO, número de linhas de código (LOC – *lines of code)*, número de *tokens* utilizados, cálculo assintótico, complexidade ciclomática, pontos por função (*i.e.* complexidade de código, complexidade de algoritmo, medida de projeto);
- Medições da execução do *software* como tempo total de execução [Simão *et al.*, 2012c], tempo de resposta, consumo de memória, disponibilidade [Van Roy, 1990] [Brennan *et al.*, 2010].

Neste sentido, a seguir são feitas as comparações PON versus POE, no tocante às suas características estruturantes, medidas de complexidade de código e medições durante execução. Os experimentos utilizaram os programas construídos nos Casos de Estudo. Por fim, são apresentadas as reflexões sobre as comparações. Convém explicar que a escolha desses tipos de comparação (taxonomia, programação de caso de estudo, complexidade de código e desempenho) se utilizou de critérios como análise de melhor aderência quanto à comparação de linguagens e paradigmas diferentes, experiência profissional, potencial de maior utilidade e continuidade de pesquisa.

Sob a visão de *software* executando, a pesquisa levou em conta critérios quantitativos como tempo de resposta e tempo total de execução. Enfatiza-se que a comparação de tempo de resposta é necessária em investigação de tratamento de eventos por *software*.

4.1 Comparação de características estruturantes

Primeira e resumidamente, os principais conceitos de ambos os paradigmas são aqui relacionados. Programar em PON consiste em construir *software* sob um viés de alto nível, por meio de regras, relacionando *Rules* e *FBEs* (característica inspirada em PD/SBR). Nesse contexto, *Rules* notificam *Actions*, que notificam *Methods* para execução, definindo o que executar e não como executar. Em PON a estrutura de execução é um ciclo via notificações, composto de unidades computacionais de pequeno porte que operam em sinergia.

Ao seu turno, o modelo de programas em POE consiste em construir *software* que processe eventos que ocorram em tempo indeterminado. Um evento instiga uma ação em uma unidade computacional (de qualquer tamanho ou complexidade). Ou seja, elaborar tecnicamente o *software* em POE compreende compor componentes computacionais para tratar e processar eventos.

Em seguida, POE e PON são enquadrados de acordo com as cinco principais características dos paradigmas de programação, realizando uma análise sob a perspectiva da taxonomia de Van Roy (2009).

(I) Característica "Registro" (*Record*):

- POE possui mediante suporte dado pelas linguagens de programação hospedeiras do paradigma.
- PON possui. Fundamentalmente herdado de inspirações em POO e SBR, e também possui mediante suporte dado pela linguagem de programação hospedeira C++ (em *Framework* Otimizado). *FBEs* são compostos de propriedades e comportamentos. Por exemplo, o *FBE Exoskeleton* possui a propriedade *Exoskeleton→ atExoskeletonPositionX*. Além disso, *FBEs* podem ser compostos, associados, rearranjados e decompostos dinamicamente. Esta é uma característica inspirada em Orientação a Objetos, *e.g.* objetos podem ser criados, associados e destruídos em

tempo de execução. Assim como em PON as propriedades e comportamentos dos *FBEs* também podem ser criadas e destruídas dinamicamente.

(II) Característica "Recipiente de Escopo Léxico" (*Closures*):

- POE possui mediante suporte dado pelas linguagens de programação hospedeiras do paradigma.
- PON possui, herdado de inspirações em POO e SBR. Cada unidade computacional é uma *closure* em PON. Tanto no nível mais interno (*Attribute*, *Premise*, *Instigation*, *Method*), quanto no nível mais externo (*FBE*, *Condition*, *Action* e *Rule*). *FBE* é a composição de *Attributes* e *Methods*. *Condition* é cálculo condicional notificado pelas *Premises*. A *Rule* compõe a declaração para o cálculo lógico-causal, notificada pelas *Conditions*, e notificam as *Actions*. Além disso, os recipientes de escopo léxico em PON podem ser compostos, associados, rearranjados e decompostos entre si dinamicamente. Em PON, *FBEs, Attributes*, *Methods* são análogos a objetos (de pequeno porte), e suas unidades computacionais *Premise, Condition, Rule, Action, Instigation* são análogas às estruturas de controle (*e.g. if-then-else*) porém como objetos (e também de pequeno porte). Nesse contexto, as unidades computacionais são tão pequenas quanto podem ser.

(III) Características Independência(*Independence*) e Concorrência (*Concurrency*):

- POE não possui nenhumas destas duas características quando implementado em linguagens que não apresentam suporte à independência e concorrência, como *Event Loop,* ou POO sem concorrência. POE possui ambas as características se implementado por meio de POO concorrente, *Publish/Subscribe* e PFR. Em suma, conforme o paradigma hospedeiro.
- PON possui e pode distribuir qualquer parte do programa conforme fundações teóricas e pesquisas em andamento como [Simão *et al.*, 2010][Simão e Stadzisz, 2010] [Belmonte, 2012][Peters, 2012]. Em PON seria possível distribuir um programa com *Attributes*, *Premises*, *Conditions*, *Rules*, *Instigations*, *Actions* e *Methods* em *software* e/ou hardware. Ou até mesmo distribuir os componentes computacionais em quaisquer máquinas, processos, memórias diferentes.

(IV) Característica "Estado Nomeado" (*Named state*):

- POE possui mediante expressividade de passagem de mensagem (*message passing*) [Van Roy, 2009], bem como pelo suporte dado pelas linguagens de programação hospedeiras do paradigma.
- PON possui, herdado de inspirações da OO. Cada unidade computacional guarda seu próprio estado, configurando indícios de baixo acoplamento e alta coesão (*e.g.* não partilham estado e possuem uma única responsabilidade) [Simão *et al.*, 2012c]. Na definição de Van Roy PON implementa célula local (*local cell*), na qual se percebe uma granularidade pequena (*i.e.* cada unidade computacional mantém suas próprias responsabilidades de estado). A Tabela 6 demonstra cada unidade computacional em PON e suas possibilidades de estado (conforme seus fundamentos teóricos e respectivas implementações no atual estado da técnica). Importante destacar que em PON se tem os estados expostos e os estados internos de cada unidade computacional. Os estados expostos representam a semântica da execução do processamento. Em contrapartida, os estados internos são de associação por agregação (relacionamentos) entre as unidades computacionais, para relações de pertencimento (*e.g.* entre um *FBE* e seus *Attributes*), para execução do ciclo de notificações e para execução interna de processamento (*i.e.* estado interno de um *Method*).

Tabela 6: Unidades computacionais e seus respectivos estados em PON [Banaszewski, 2009].

Unidade computacional em PON	**Estados**
FBE	Valor de Denominação Associação por agregação de seus *Attributes* e *Methods*
Attribute	Valores elementares de *Integer*, *Double*, *Char*, *Boolean*, *String* Associação para notificação de suas *Premises*
Premise	Valor lógico VERDADEIRO ou FALSO Associação por agregação de seus *Attributes* e *Operators* Associação para notificação de suas *Conditions*
Condition	Valor lógico VERDADEIRO ou FALSO Associação para validação (*i.e.* notificação) de sua *Rule*
Rule	Valor APROVADA, DESAPROVADA, EXECUTANDO ou EXECUTADA Associação por agregação de suas *Condition* e *Action* Associação para executar (*i.e.* notificar) sua *Action*
Action	Valor EXECUTE, ATIVANDO, ou INATIVA Associação para ativar (*i.e.* notificar) suas *Instigations*
Instigation	Valor ATIVADA, INSTIGANDO ou INATIVA Associação para instigar (*i.e.* notificar) seus *Methods*
Method	Valor INSTIGADO, EXECUTANDO ou INATIVO Estados e operações, dentro do próprio recipiente de escopo léxico, para mudar estado de *Attributes*

(V) Característica "Não-determinismo observável" (O*bservable Nondeterminism*):

- POE: não-determinismo observável ausente em *Event Loop*, POO sequencial e PFR (quando construído em linguagens ou paradigmas que garantem o determinismo). Pode ocorrer não-determinismo observável em POO concorrente, *Publish/Subscribe*. Em suma, conforme o paradigma hospedeiro.
- PON: ocorre não-determinismo observável, que pode ser resolvido conforme as resoluções de conflito *DEPTH*, *BREADTH*, *PRIORITY* e *KEEPER,* propostas por [Banaszewski, 2009] e [Ronszcka, 2012]. Outra solução é a execução do

processamento conforme pesquisa registrada em pedido de patente [Simão e Stadzisz, 2010] que modela a garantia do determinismo em PON (*i.e.* ciclo de inferência via notificações utilizando também contra-notificações).

Por conta da articulação dos conceitos de Registro, Recipiente de Escopo Léxico, Independência, Estado Nomeado e Não-determinismo observável, associando as características próprias do paradigma, as análises dos casos de estudo deste trabalho e demais pesquisas no âmbito do PON como [Banaszewski, 2009] e [Ronszcka, 2012], se identificam as seguinte características marcantes do Paradigma Orientado a Notificações (PON):

- Unidades computacionais de pequeno porte (tão pequenas quanto podem ser): o estado de cada unidade computacional de pequeno porte (***local cell - célula local***) é mantido localmente. Assim, cada elemento de PON que possui estado (*FBE*, *Atribute*, *Condition*, *Premise*, *Rule*) é o único responsável pelo seu estado, sendo que outros elementos interessados neste estado são informados por notificação sobre suas alterações. O conhecimento sobre as influências das mudanças de estado na lógica de toda a aplicação é expresso por meio de regras (característica análoga aos SBRs);
- O PON emprega o conceito de notificação oriundo de Orientação a Eventos em seu modelo de execução; entretanto, o seu uso como único mecanismo responsável pela execução das inferências (*i.e.* "**inferência por notificações**"), tornam o PON único, no sentido que outros paradigmas baseados em regras e fatos utilizam mecanismos de inferência baseados em buscas (*search*), o que não ocorre em PON, no qual os elementos notificantes/notificados realizam diretamente o cálculo lógico-causal em sinergia.

Isto dito, a Tabela 7 inclui PON na taxonomia de Van Roy contrapondo POE. Como previamente apresentado na (Tabela 1), aqui se resumem os paradigmas de programação relacionados aos seus principais conceitos: (I) registro (*record*); (II) recipiente com escopo léxico (*closure*); (III) independência (concorrência), (IV) estado nomeado (*named state*), e (V) não-determinismo observável. São novamente apresentadas linguagens exemplo e características inerentes aos paradigmas. Ademais, as quatro principais características em PON são realçadas com S+, visto que o paradigma possui as características diferenciadas.

Tabela 7: PON incluso na Taxonomia [adaptado de VAN ROY, 2009] (S)IM/(N)ÃO.

Conceito **Paradigma**	**I**	**II**	**III**	**IV**	**V**	**Exemplo de linguagens de programação**	**Conceitos (+)**
Lógico e Relacional	S	S	N	N	N	Prolog, SQL embeddings	+unificação (igualdade) +busca
Funcional	S	S	N	N	N	Scheme, ML	
Funcional Reativo	S	S	S	N	N	Fran, FrTime	+linhas de execução +única atribuição +escolha não-determinística +sincronização em término parcial
Síncrono Discreto	S	S	S	N	N	Esterel, Lustre, Signal	+linhas de execução +única atribuição +escolha não-determinística +sincronização em término parcial +computação cronometrada
Ciclo de eventos (e*vent loop*)	S	S	N	S	N	E in one vat	+porta(canal)
Orientado a Notificações (PON)	S+	S+	S+	S+	S / N[27]	LingPON, *Framework* Otimizado em C++	+célula local(estado) +inferência por notificações
Objeto ativo / Objeto apto	S	S	S	S	S	Publish-Subcribe, E	+porta(canal) +linhas de execução +célula local
OO – Sequencial	S	S	N	S	N	Java, Ocaml	+célula local(estado)
OO – Concorrente	S	S	S	S	S	Smalltalk, Java	+célula local(estado) +linhas de execução

27 Patente [Simão e Stadzisz, 2010] modela a garantia do determinismo em PON.

É possível realizar reflexões da comparação estruturante, assim validando a taxonomia de Van Roy como alicerce único para uma comparação cartesiana de paradigmas. Para tanto, é possível intuir simples perguntas, a partir de cada característica da taxonomia, que assim caracterizam, conceituam e diferenciam paradigmas de programação, elencadas a seguir:

1. Quais os tipos de registros do paradigma de programação? Como se compõem?
2. Quais os tipos de recipientes de escopo léxico (*closure*) do paradigma? Quais os tamanhos? Como se compõe?
3. Qual o tipo de suporte à concorrência?
4. Qual a expressividade de estado do paradigma?
5. É possível não-determinismo observável? Como é evitado o não-determinismo observável?
6. Quais conceitos são singulares do paradigma? (Importante ressaltar que "conceitos singulares" não significa necessariamente exclusividade de um único paradigma, pois também são conceitos que se acumulam com os cinco principais conceitos da taxonomia e podem estar presentes em vários paradigmas).

Ou seja, cada um dos conceitos pode ser expresso em uma perspectiva de questão, cuja resposta descreve as propriedades do paradigma e como este se enquadra segundo a taxonomia. Em seguida, alguns exemplos de opção para respostas em outros paradigmas e as respectivas respostas para o PON são demonstradas.

1. Quais os tipos de Registros do paradigma de programação? Como se compõe?
   Exemplos são estruturas de dados (*e.g. C struct*), *arrays*, listas, cadeia de caracteres, árvores, tabelas *hash*.
   PON: *FBEs* com propriedades, como exemplo o FBE *Exoskeleton* possui uma associação entre a propriedade *atExoskeletonPositionX*. A referência dessa propriedade é *Exoskeleton→atExoskeletonPositionX*. Um *Attribute*, assim como na OO, define uma propriedade de um *FBE*, contendo a tupla <nome-valor>. Podem ser construídos, associados, reorganizados, recompostos e reposicionados dinamicamente.

2. Quais os tipos de recipientes de escopo léxico (*closure*) do paradigma? Quais os tamanhos? Como se compõe?
   Exemplos são procedimentos (Paradigma Procedimental), funções (Paradigma

Funcional), classes e métodos (Paradigma Orientado a Objetos), estruturas de controle (Paradigma Imperativo).

PON: Cada unidade computacional é uma *closure* em PON, desde o nível micro (*Attribute*, *Premise*, *Instigation*, *Method*) até macro (*FBE*, *Condition*, *Action* e *Rule*), sendo unidades computacionais de pequeno porte. Exemplificando, um *Method* (como na OO), define um comportamento de um FBE, referenciando um recipiente léxico contendo <nome-comandos>. Podem ser construídos, associados, reorganizados, recompostos e reposicionados dinamicamente.

3. Qual o tipo de suporte à concorrência?

   Exemplos são linhas de execução diferentes (*e.g. Threads* em POO Concorrente), linguagem síncrona (Paradigma Síncrono Discreto), encapsulamento de funções (PF).

   PON: Distribuição dos próprios componentes, *i.e.* unidades computacionais, por conta do modelo de execução. A distribuição não fica (essencialmente) a cargo da programação de alto nível.

4. Qual a expressividade de estado do paradigma?

   Exemplos de expressividade de estado são ausência de estado (Paradigma Funcional), única atribuição (Paradigma Funcional Reativo), *local cell* (Paradigma Objeto Ativo), variável global (estado compartilhado no Paradigma Imperativo).

   PON: Expressividade de estado presente em cada unidade computacional PON (*local cell*), que gerencia o seu próprio estado e suas próprias responsabilidades.

5. É possível não-determinismo observável? Como é evitado o não-determinismo observável?

   POO concorrente é um exemplo no qual pode ocorrer não-determinismo observável [Van Roy, 2009].

   PON: Sim, é possível não-determinismo observável. As técnicas de resolução de conflito podem mitigar a ocorrência. Não obstante, há pesquisa relativa à garantia do determinismo em [Simão e Stadzisz, 2010]. Esse trabalho indica o uso de contra-notificações entre as unidades computacionais de PON.

6. Quais conceitos são singulares do paradigma?

   Exemplos de conceitos singulares são unificação (igualdade) do Paradigma Lógico

(*e.g.* linguagem PROLOG), busca (*search*) do Paradigma Lógico & Relacional e *log* do Paradigma de Memória Transacional [Van Roy, 2009].
PON: célula local (estado) e inferência por notificações (*inference by notifications*).

E, além disso, são propostas duas novas perguntas e respectivas novas características para acrescer a taxonomia, amplificando e expandindo o trabalho de Van Roy no trabalho de comparação desta pesquisa. As perguntas estão enumeradas a seguir.

7. Qual o modelo de execução?
   Exemplos de modelos de execução são algoritmos de *matching* como RETE, TREATS, LEAPS, HAL, *lambda calculus*, autômato finito ou máquina de Mealy, máquina de Turing, máquina abstrata de Warren.
   PON: ciclo de notificações ou inferência por notificações.

8. Qual a origem técnica ou filosófica do paradigma? Quais problemas o paradigma se propõe a resolver?
   Exemplos: o Paradigma Imperativo é embasado na máquina de Turing e o Paradigma Funcional é embasado no cálculo *lambda* de Church.
   PON: filosófica e teoricamente embasado no Controle Holônico e Redes de Petri [Simão e Stadzisz, 2009][Banaszewski, 2009][Wiecheteck, 2011][Ronszcka, 2012]. Alguns dos problemas que se propõe a resolver são diminuir o desperdício de desempenho em termos de avaliações redundantes, aumentar a coesão e diminuir o acoplamento dos programas, aumentar a flexibilidade e expressividade da programação, facilitar a programação.

A tabela completa de características do PON, com as duas novas características elencadas é apresentada a seguir.

Tabela 8: Características de PON segundo a taxonomia expandida [adaptado de VAN ROY, 2009] (S)IM/(N)ÃO.

Conceito/Característica	**Descrição**
Paradigma	Orientado a Notificações (PON)
(I) Registro	S+
(II) Recipientes de Escopo Léxico	S+
(III) Independência	S+
(IV) Estado Nomeado	S+
(V) não-determinismo Observável	S / N[28]
Exemplos de linguagem de programação	LingPON *Framework* Otimizado em C++
Características	+célula local (estado) +inferência por notificações
Modelo de execução	Ciclo de notificações / Inferência por notificações
Fundação filosófica e/ou teórica	Controle Holônico Redes de Petri
Proposta de resolução de problemas	Diminuir o desperdício de desempenho em termos de avaliações redundantes Aumentar a coesão e diminuir o acoplamento dos programas Aumentar a flexibilidade e expressividade da programação Facilitar a programação

A seguir, é apresentada nesta seção o redesenho da taxonomia de Van Roy como foco no Paradigma Orientado a Notificações (Figura 47) contendo suas características, para melhor visualização nesta dissertação.

28 Patente [Simão e Stadzisz, 2010] modela a garantia do determinismo em PON.

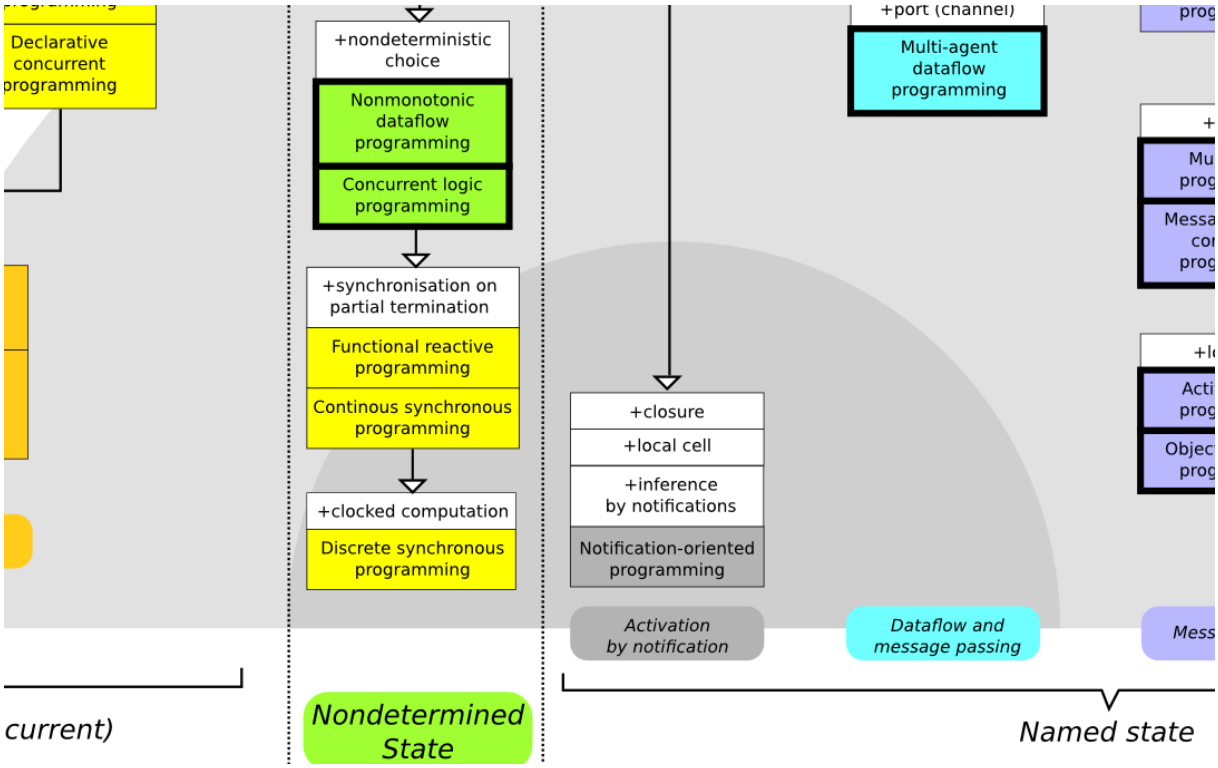

Figura 47: Taxonomia de paradigmas de programação incluindo PON em visão aproximada (excerto) [expandido a partir de Van Roy, 2009]

Primeiramente, a escolha do posicionamento do paradigma (graficamente e semanticamente) se deve pela estrutura intrínseca da própria taxonomia: expressividade de estado (ativação por notificações tem expressividade menor que passagem de mensagem e maior que estado não nomeado), forma de programação (mais declarativa ou mais imperativa – eixo horizontal) e soma de características únicas (eixo vertical). Em segundo lugar, grosso modo, os paradigmas adjacentes são os parentes próximos do paradigma (como já citado, o PON já foi confundido com esses). Por último, é o local mais equidistante entre os paradigmas de alicerce ao PON: Orientação a Objetos e Lógico. A Figura 48 mostra a taxonomia completa.

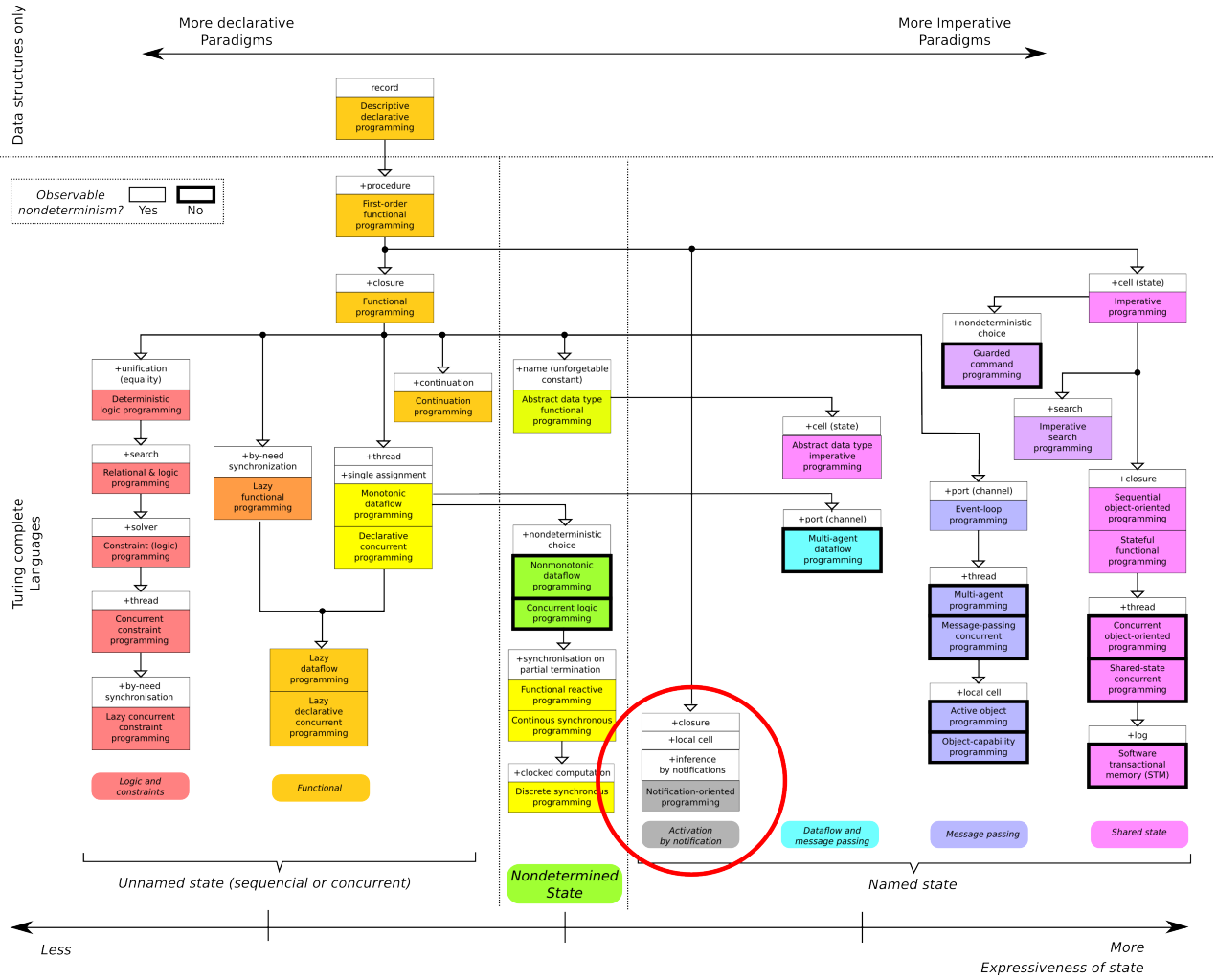

Figura 48: Taxonomia de paradigmas de programação incluindo o PON – em destaque - visão completa da taxonomia [expandido a partir de Van Roy, 2009]

A próxima seção apresenta as medições de complexidade de código para todas as implementações realizadas para os casos de estudo (apresentadas no Capítulo 3 - Casos de estudo).

4.2 Comparações em complexidade de código-fonte de *software* em PON e POE

Para comparações em complexidade de código-fonte, de forma relacionada com outros trabalhos como [Dunkels *et al.*, 2006], [Sant'Anna *et al.*, 2013] e [Salvaneschi *et al.*, 2014b], as métricas foram número de linhas de código (*LOC-lines of code*), número de *closures* (escopos) e número de *tokens* na linguagem (*i.e.* medidas em C++ puro, *Framework* otimizado C++ e LingPON). Os objetivos são medir e comparar código entre programas codificados em diferentes paradigmas, com técnicas diferentes, porém resolvendo problemas idênticos.

Para tanto, foram utilizadas a ferramenta `cloc`[29] e `cloc --diff` para número de linhas de código-fonte, foi realizada contagem manual para número de *closures* (escopos), e para número de *tokens* uso de analisadores léxicos e sintáticos (implementados com as ferramentas `flex` e `bison`), além do uso de contadores de palavras (`wc` e `grep`). Neste contexto, as medições seguiram as seguintes indicações: contagem de todo código-fonte (inclusive códigos acessórios como enumeradores e componente principal – classe *Main*); os nomes foram normalizados para a contagem de número de linhas de códigos modificadas, acrescentadas e removidas (*i.e. diff* entre os *softwares* dos cenários com mesmos nomes de arquivos, e quando pertinente, mesmos nomes para as classes e respectivos métodos); roteiros automatizados para realização das medidas nos casos em que são utilizadas ferramentas (*i.e. shell scripts*).

Para medidas de número de linhas de código, no primeiro caso de estudo as mesmas se deram em maneira global (número de linhas de código-fonte absoluto) e também medindo as diferenças de medidas entre os cenários (número de linhas de códigos modificadas, acrescentadas e removidas).

29 Ferramenta disponível em: http://cloc.sourceforge.net/

A Figura 49 apresenta os dados da medição global. A única medição na qual PON tem maior número de linhas foi no primeiro cenário. É importante enfatizar que a medida engloba todos os códigos-fonte acessórios para funcionamento do *software* (*e.g.* classe principal, métodos de *binding,* criação de componentes, visibilidade de escopo), inclusive de integração com o *Framework* PON, fatos que podem indicar maior número justamente na primeira medida para o PON (cenário 1).

Também se inclui no gráfico a medição de código-fonte feito em LingPON (e respectivo PON-compilador em desenvolvimento) para o primeiro cenário. Cabe enfatizar que a medição em LingPON não inclui as modificações necessárias para funcionamento do *software* (*i.e.* adaptações para receber eventos, configuração inicial dos estados de *Premises* e respectivas correções de comportamento), sendo ilustrativa para demonstrar o número de linhas para a linguagem (os ajustes foram necessários no momento da manufatura deste trabalho pois o PON-Compilador encontra-se em desenvolvimento por outrem do grupo de pesquisa do PON). Concerne também relembrar que a implementação suficiente e completamente funcional em LingPON, *i.e. software* rodando e compilador em PON-Compilador, foi realizada somente para o primeiro cenário do primeiro caso de estudo e para o segundo caso de estudo.

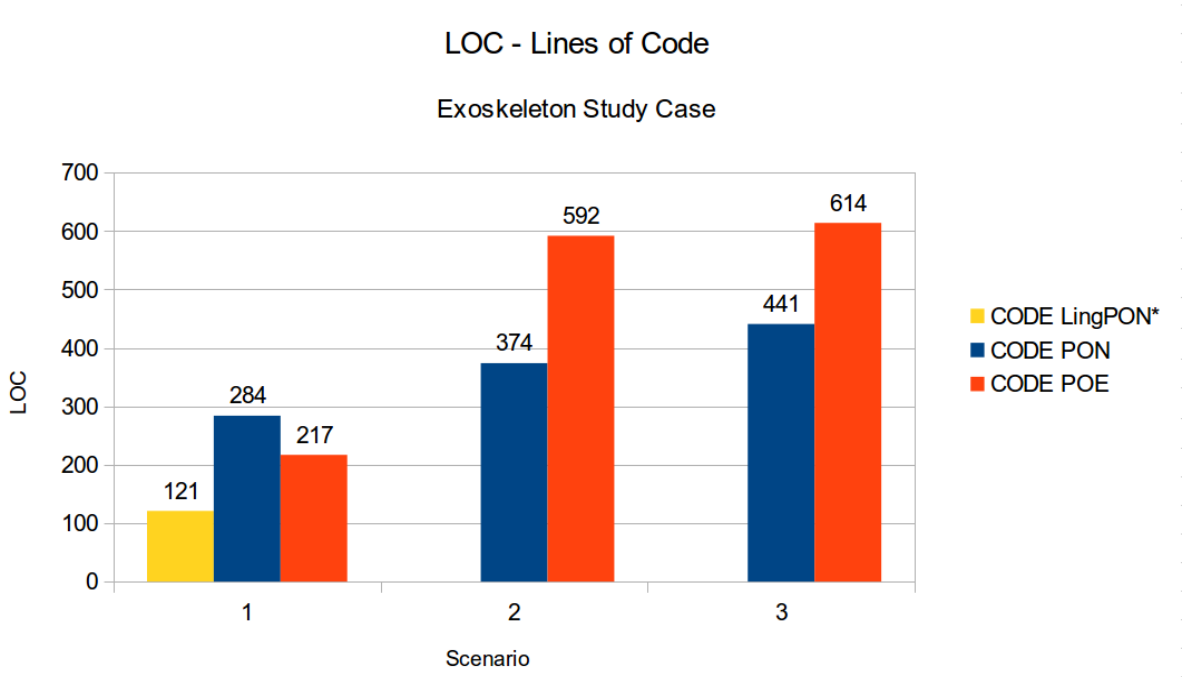

Figura 49: Gráfico linhas de código-fonte PON e POE do primeiro caso de estudo *Exoskeleton*

Os dados das diferenças em número de linhas de código-fonte modificadas, acrescentadas e removidas entre os cenários 1 e 2, e entre os cenários 2 e 3, estão no gráfico da Figura 50.

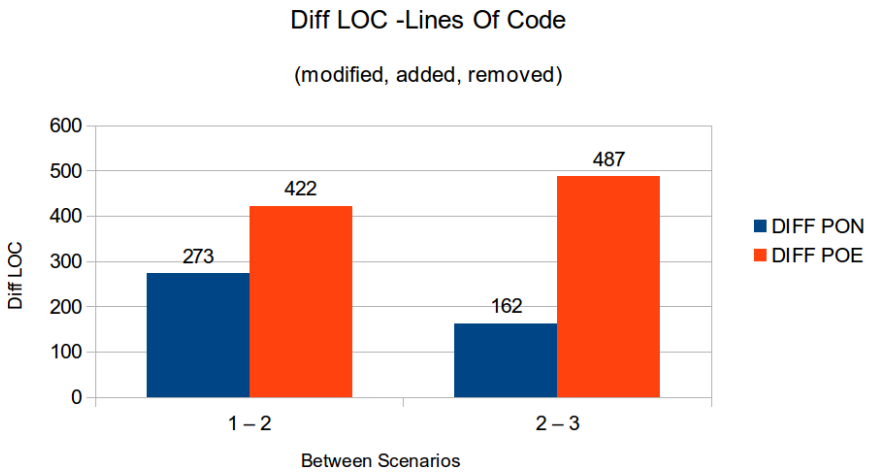

Figura 50: Gráfico de diferenças entre linhas de código-fonte modificadas, acrescentadas e removidas entre cenários, em PON e POE, do primeiro caso de estudo – *Exoskeleton.*

Ambas as medições indicaram expressivamente menor quantidade de linhas de código modificadas, acrescentadas ou excluídas para o PON em relação ao POE. A ressalva da medição global anterior (todos os códigos acessórios e necessários em PON) pode sugerir uma inerente estabilidade maior de código em PON, entretanto essa estabilidade também pode valer para POE (*e.g.* métodos de *binding* e criação de objetos). Ao seu turno também sugere indícios de uma maior estabilidade para modificação, expansão ou refatoração de *software* em PON (que podem servir como hipóteses em trabalhos futuros).

O gráfico na Figura 51 demonstra os dados da medição global para o segundo caso de estudo. Novamente, a única medição na qual o *Framework* PON tem maior número de linhas foi em relação à técnica *Handler-Dispatcher,* bem como o número maior de linhas para as outras técnicas sugerem novos indícios de *Over-Engineering* [Kerievsky, 2004] para programar este caso de estudo nas técnicas *State* e *Observer*. Da mesma forma, no gráfico está inclusa medição de código-fonte em *software* feito em LingPON (e respectivo PON-compilador) para o segundo caso de estudo.

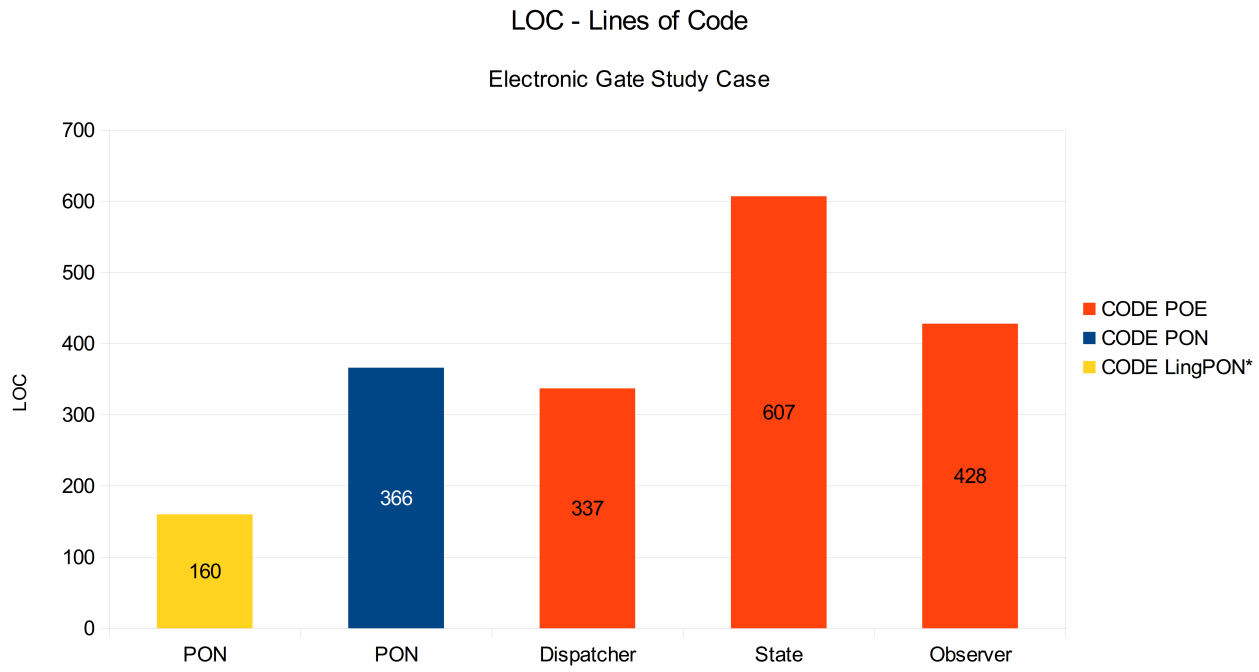

Figura 51: Gráfico de linhas de código-fonte PON e POE do segundo caso de estudo Portão Eletrônico.

Para medidas de *closure* – recipientes de escopo léxico ou simplesmente escopo – em ambos os casos de estudo as mesmas se deram exclusivamente em maneira global (número absoluto). A Figura 52 apresenta os dados dessa medição de *closures* do primeiro caso de estudo em todos os cenários.

PON teve maior número de *closures* em medições no primeiro e terceiro cenários. Ao bem da verdade, PON teve números em relação à *State* – segundo cenário – muito próximos, respectivamente 153 e 159 *closures*. PON também obteve números muito próximos em relação ao padrão *Observer* no terceiro cenário, respectivamente 192 e 186 *closures*. Também

se inclui no gráfico a medição de escopos realizada em LingPON (em amarelo).

Importante enfatizar, por se tratar de uma medição manual, foi possível medir e separar os escopos puros em *Framework* PON, ou seja, somente entidades PON como *FBEs, Atributes, Premises, Conditions, Rules, Actions, Instigations* conforme a Tabela 6 (pg. 129). Assim, ilustram-se os escopos de apoio (identificado na cor cinza no gráfico). Considerando somente os escopos puros, todas as medidas foram favoráveis ao *Framework* PON.

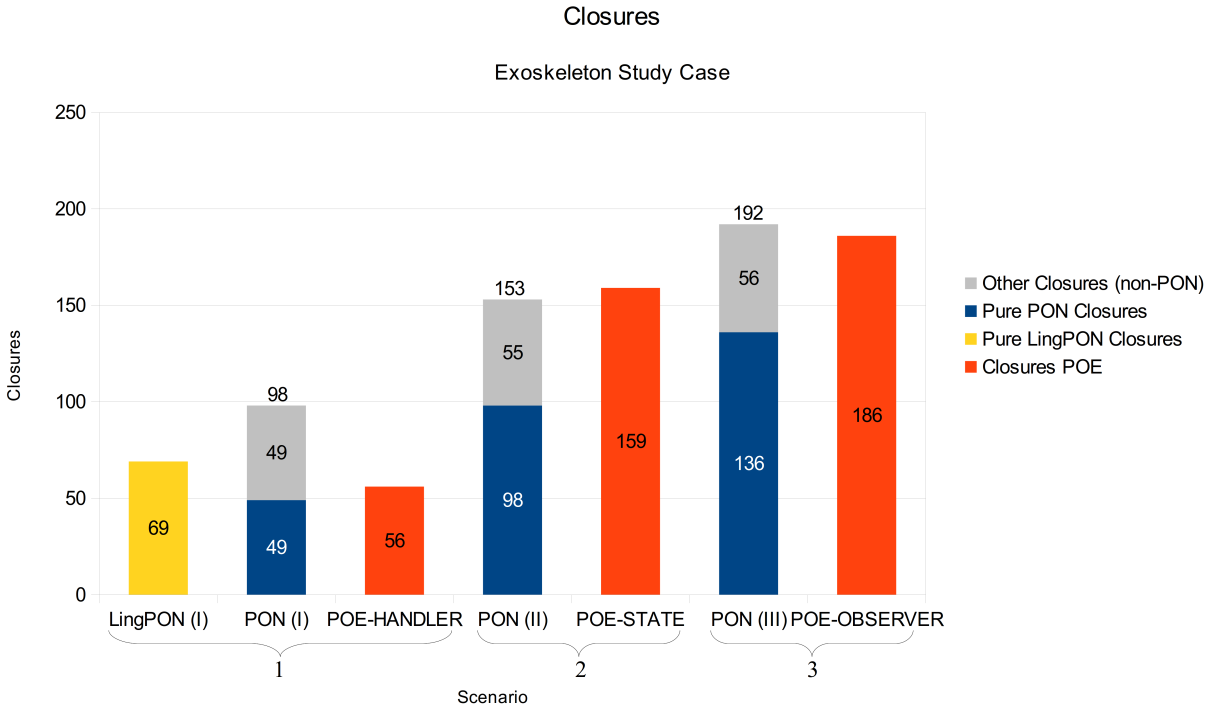

Figura 52: Gráfico número de *closures* PON e POE do primeiro caso de estudo *Exoskeleton* em todos os cenários.

Em LingPON, atualmente é obrigatório o uso de uma *Subcondition* para cada *Rule* e uma *closure* (*i.e.* palavra reservada *methods*) para o conjunto de métodos, o que acaba inflando um pouco a quantidade de escopos no atual estágio da linguagem. Cabe demonstrar o exemplo desses escopos em LingPON, expostos no Algoritmo 27. Por exemplo, um *Method* ou uma *Premise* pode ter escopo de uma linha, e cada *Rule* tem uma *Subcondition, Condition* e uma *Action,* multiplicando-se por quatro o número de escopos (*i.e.* número de *Rules* multiplicado por quatro).

Algoritmo 27: Código em LingPON para demonstrar *closures Methods, Subcondition* e *Premise* (destacadas em negrito) em exoskeleton.pon

```
 1 fbe Exoskeleton
 2     attributes
 3         integer atExoskeletonState 0
 4         integer atExoskeletonPositionX 0
 5         integer atExoskeletonPositionY 0
 6     end_attributes
 7     methods
 8         method mtOn(atExoskeletonState = 0)
 9         method mtOff(atExoskeletonState = 1)
10         method mtMoveForward(atExoskeletonPositionY = atExoskeletonPositionY + 1)
11         method mtMoveRight(atExoskeletonPositionX = atExoskeletonPositionX + 1)
12         method mtMoveBack(atExoskeletonPositionY = atExoskeletonPositionY - 1)
13         method mtMoveLeft(atExoskeletonPositionX = atExoskeletonPositionX - 1)
14     end_methods
15 end_fbe
... ...
45 rule rlTurnOnExoskeleton
46    condition
47        subcondition A1
48          premise prSimulationIsRunning simulation.atSimulationState == 1 and
49          premise prExoskeletonIsTurnedOff exoskeleton.atExoskeletonState == 0 and
50          premise prKeyIsClosed event.atEventState == 2
51        end_subcondition
52     end_condition
53     action
54          instigation inOn exoskeleton.mtOn();
55     end_action
56 end_rule
```

A Figura 53 demonstra o gráfico com os dados dessa medição de *closures* do segundo caso de estudo em todas as técnicas PON, LingPON e POE (*Dispatcher, State* e *Observer*). Novamente, as medições nas quais PON tem maior número de *closures* absoluto foram em comparação às técnicas *Dispatcher* e *Observer* (medidas muito próximas 138 a 132), e em número de escopos puros número expressivamente menor de *closures* para o PON. Em números absolutos, LingPON teve o menor número de *closures* no segundo caso de estudo.

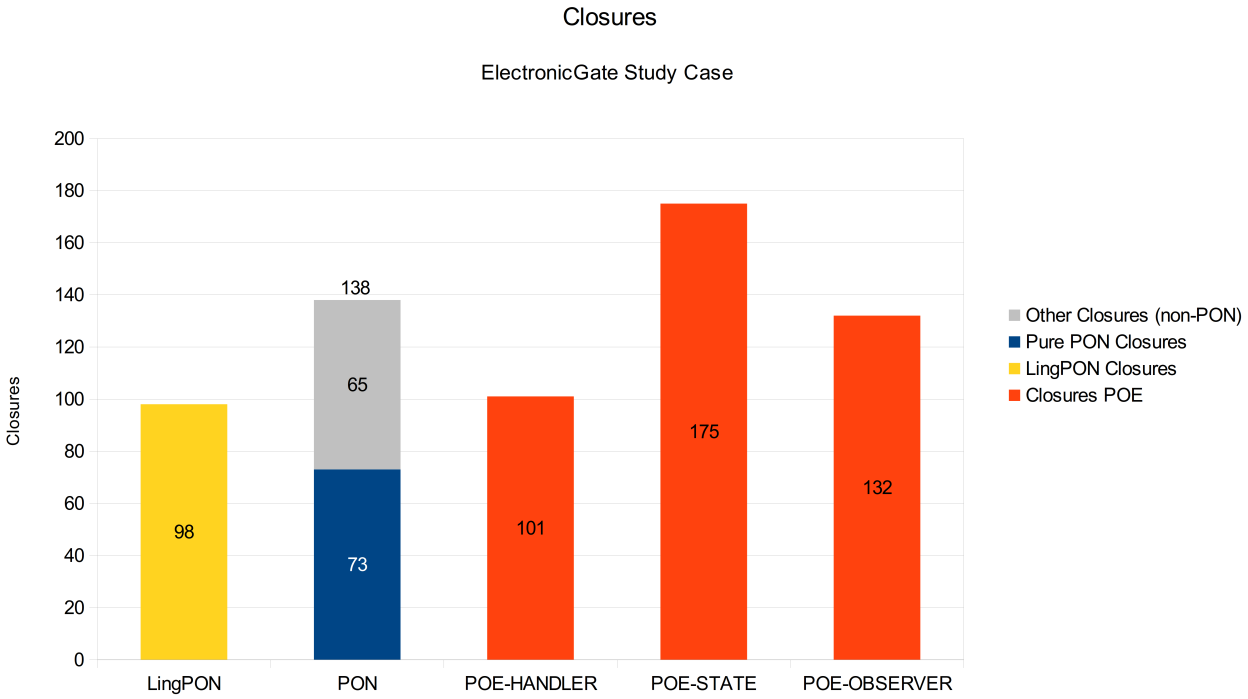

Figura 53: Gráfico número de *closures* PON e POE do segundo caso de estudo Portão Eletrônico.

Para medidas de número de *tokens,* também em ambos os casos de estudo a medição foi número absoluto global. A Figura 54 ilustra os dados dessa medição de *tokens* do segundo caso de estudo. Para o primeiro cenário, LingPON tem um número expressivamente menor de *tokens*. Nos cenários 2 e 3, *Framework* PON tem menos *tokens* que *State* e *Observer* respectivamente.

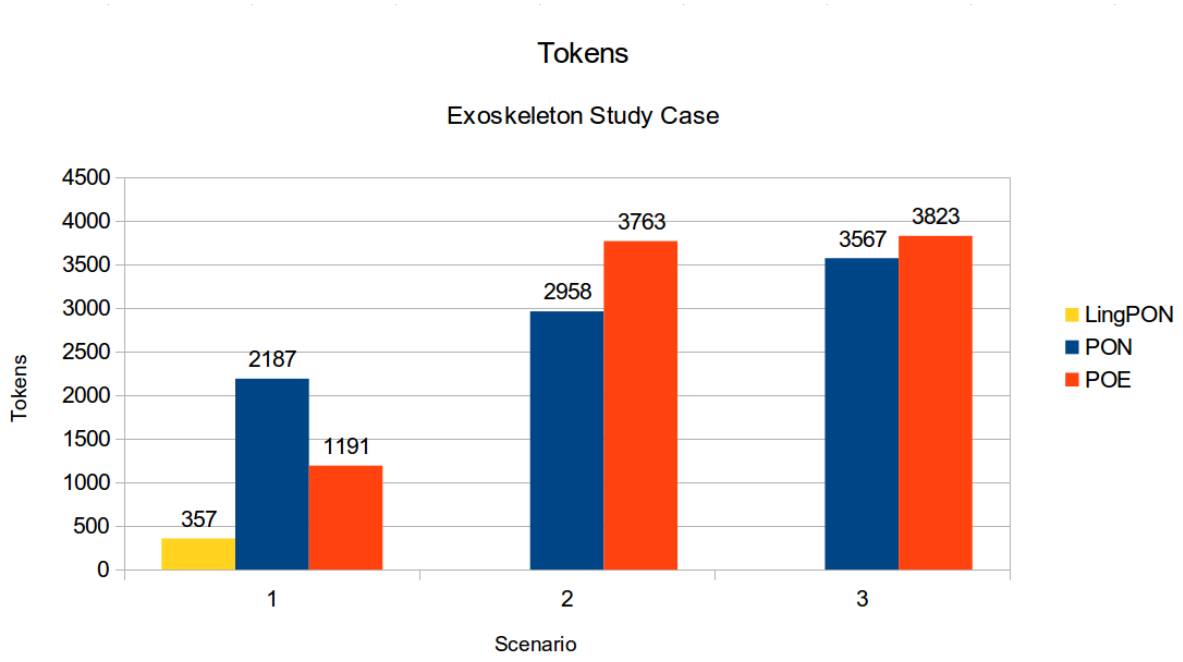

Figura 54: Gráfico número de tokens PON e POE do primeiro caso de estudo Exosqueleto.

Na Figura 55 se ilustram os dados de número de *tokens* do segundo caso de estudo. Em números absolutos, LingPON teve menor número de *tokens* no segundo caso de estudo. A técnica *State* teve mais de mil *tokens* em relação à PON, e quase o dobro em relação à *Handler-Dispatcher*.

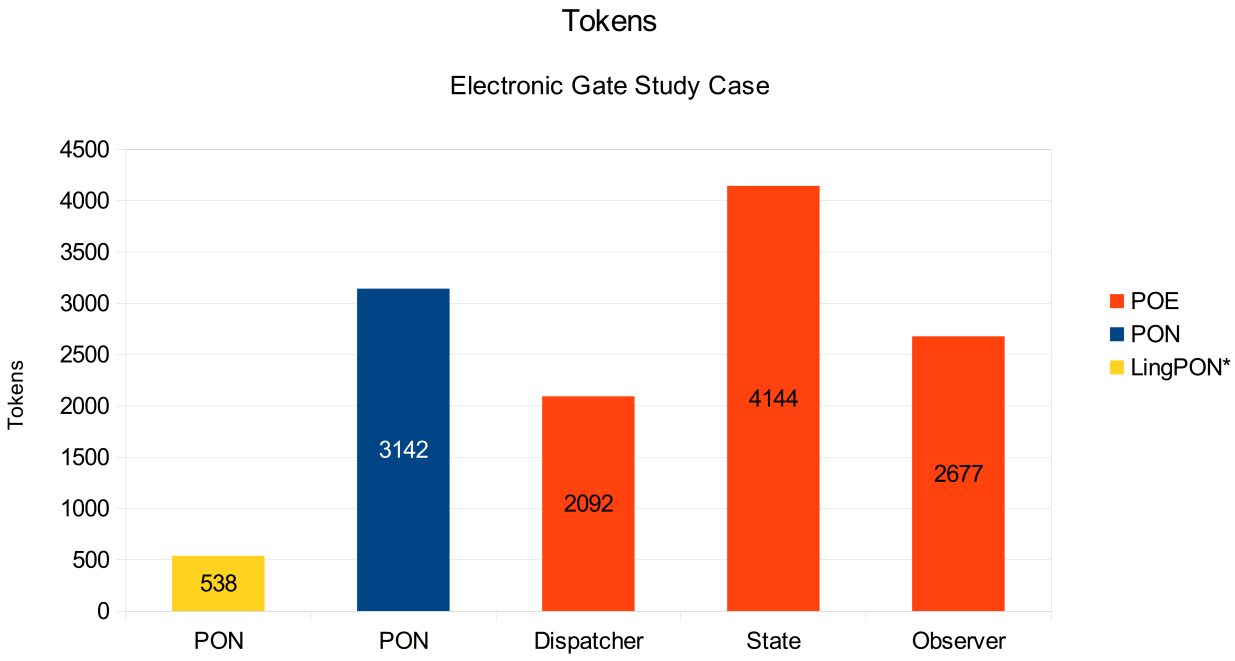

Figura 55: Gráfico número de *tokens* PON e POE do segundo caso de estudo Portão Eletrônico.

Utilizando código formatado, essas medidas proveem uma estimativa de expressividade do modelo de programação, principalmente se forem consideradas em conjunto (número de linhas de código, *closures* e *tokens*). Muito embora os resultados não sejam necessariamente lineares, eles indicam que PON tem um modelo de programação mais expressivo em várias medições.

Como conclusão, acerca das medições de número de linhas, no primeiro caso de estudo PON teve número menor de linhas nos segundo e terceiro cenários, LingPON teve menor número de linhas no primeiro cenário. Em relação a trabalho realizado entre cenários, PON teve menor número de linhas modificadas, acrescentadas e diminuídas. No segundo caso de estudo LingPON teve menor número de linhas (160) e *Framework* PON (366) próximo à *Handler-Dispatcher* (337).

Em medida de *closures*, no primeiro caso de estudo, PON teve números menores de

escopos puros (*i.e.* desconsiderando códigos acessórios) em todos os cenários. Em números absolutos teve números muitos próximos às técnicas *State* e *Observer*. Quanto a LingPON, no uso do primeiro cenário, teve medida 69 escopos versus 56 escopos em *Handler-Dispatcher.* No segundo caso de estudo, LingPON teve o menor número de escopos (em número absolutos), e o *Framework* PON o menor número em escopos puros em PON.

Em medida de *tokens*, no primeiro caso de estudo, PON teve números menores de *tokens* nos segundo e terceiros cenários. No primeiro cenário LingPON teve um número menor de *tokens* (357) em relação ao *Framework* (2187) e *Handler-Dispatcher* (1191). No segundo caso de estudo, a mesma relação, LingPON teve número menor de *tokens* (538) em relação aos número de todos outros *softwares,* e o número de *tokens Framework* PON (3142) foi menor em relação ao *State* (4144). Essas substanciais diferenças em números se devem aos códigos acessórios (tanto para aumentar o número em *Framework* quanto para diminuir em LingPON).

4.3 Comparações em medidas de execução de *software* em PON e POE

Para instrumentação nos experimentos, para medições da execução dos *softwares* dos casos de estudo, tendo como medidas tempo de execução e tempo de resposta, foi utilizada uma classe acessória já presente no *Framework* PON Otimizado C++ (*PerformanceMeter*). Essa classe foi exportada para medir o tempo de execução e o tempo de resposta, de forma idêntica em todas as execuções de cada solução (tanto em PON quanto em POE). A plataforma de execução dos testes foi o sistema operacional e respectivo *kernel* 2.6.32-62-generic #128-Ubuntu SMP x86_64 GNU/Linux, e o processador foi um Intel(R) Core(TM)2 Duo CPU modelo T7700 velocidade de 2.40GHz de 64 bits, em um ambiente livre de preempções ou interrupções de sistema operacional, arquitetura de computador pessoal não virtualizada.

Para as medidas de tempo total de execução, cada *software* (cada solução técnica) foi executado quatro vezes em sequência, descartando a primeira execução, por conta do carregamento do *software* bem como de diferenças nos tempos de execução. Para as medidas de tempo de resposta, a granularidade escolhida foi a combinação do tipo de evento com o roteiro de execução. Dessa forma, medidas de tempo de resposta foram angariadas para

n-passos em relação ao tipo de evento. Assim, foi possível mapear o comportamento e o respectivo tempo de resposta durante os roteiros de execução das soluções.

Em todos os casos de estudo, os objetos de estudo desta pesquisa são *softwares* que tratam eventos. Para o primeiro caso de estudo, o objetivo foi testar e comparar primordialmente tempo de resposta, e em segunda prioridade tempo total de execução. Para tanto, a instrumentação (e construção) da bancada de teste do *software* segue algumas diretrizes:

- *Software* compilado em otimização mínima (`-o0`) para minorar e comparar sem influências (ou mínimo de influência) do compilador nos programas gerados. Aqui se compara programas realizados em paradigmas diferentes, portanto essas otimizações podem influenciar nos resultados, já que elas são projetadas essencialmente para máquinas padrões de mercado atuais (*e.g.* instruções sequenciais, arquitetura de computadores Von Neumann ou Harvard, processadores RISC ou CISC, algoritmos de memória cache, *pipeline*).
- A geração de eventos (*script* de eventos) ocorre em separado do *software* objeto de estudo. Ou seja, outro *software* (pequeno) também programado em C++ realiza essa geração, que se utiliza de parâmetros para geração de eventos (*i.e.* quantidade de *n*-passos, número de execuções, roteiro de testes). Outros estudos anteriores embarcaram a geração de eventos e medição do tempo de execução no próprio *software*. Nos experimentos deste trabalho, são contextos distintos e separados (*i.e.* *software* de geração de eventos, *software* PON, *software* POE).
- O *software* objeto de experimentação é passível de uso pelo usuário, funcionando independentemente de experimentos (mesmo sendo simples *toy problem*).
- Uso de *shell script* (em linha de comando) para realizar o roteiro de testes conforme os critérios de teste como diferentes *n*-passos. Essencialmente o roteiro de linha de comando tem as responsabilidades de executar a geração de eventos, executar o *software* objeto de estudo, e persistir os resultados em arquivo texto.

Os demais detalhes da instrumentação encontram-se no Apêndice B - Dados da instrumentação básica da plataforma dos experimentos. As próximas seções (4.3.1 a 4.3.5) mostram os roteiros de teste e os respectivos resultados.

4.3.1 Primeiro experimento: PON versus *Handler-Dispatcher*

Para fins de simulação e execução em ambas as abordagens POE e PON de forma idêntica, o método utilizado e automatizado, no primeiro cenário do primeiro caso de estudo, contempla os seguintes passos:

1. Estado inicial exosqueleto desligado e *joystick* em posição neutra;
2. Ligar o exosqueleto (chave fechada);
3. Mover o *joystick n*-passos para cada direção (X+, Y+, X-, Y-);
4. Mover o *joystick* para a posição neutra;
5. Desligar o exosqueleto (chave aberta);
6. Mover o *joystick n*-passos para cada direção (X+, Y+, X-, Y-);
7. Mover o *joystick* para a posição neutra.

O número total de eventos gerados é oito multiplicado por *n*-passos (8 x *n*-passos) mais quatro eventos de mover o *joystick* para a posição neutra e eventos da chave para ligar e desligar o exosqueleto. A Tabela 9 informa os dados de execuções para o primeiro cenário para tempo total de execução.

Tabela 9: Tempos de execução em milissegundos (ms) para cada solução, na simulação do primeiro cenário.

POE – Dispatcher	PON – Compilador	PON – Framework	N-steps	Total Events
0,1841	0,1831	0,1838	10	84
0,5510	0,5420	0,5510	100	804
4,2329	4,3330	4,3130	1000	8.004
43,6790	41,8879	150,2600	10000	80.004
692,4040	648,4500	2.123,1400	100000	800.004
6.867,6600	7.059,7800	22.016,0000	1000000	8.000.004

O gráfico da Figura 56 são apresentados os dados de execuções para o primeiro cenário para tempo total de execução, em escala logarítmica.

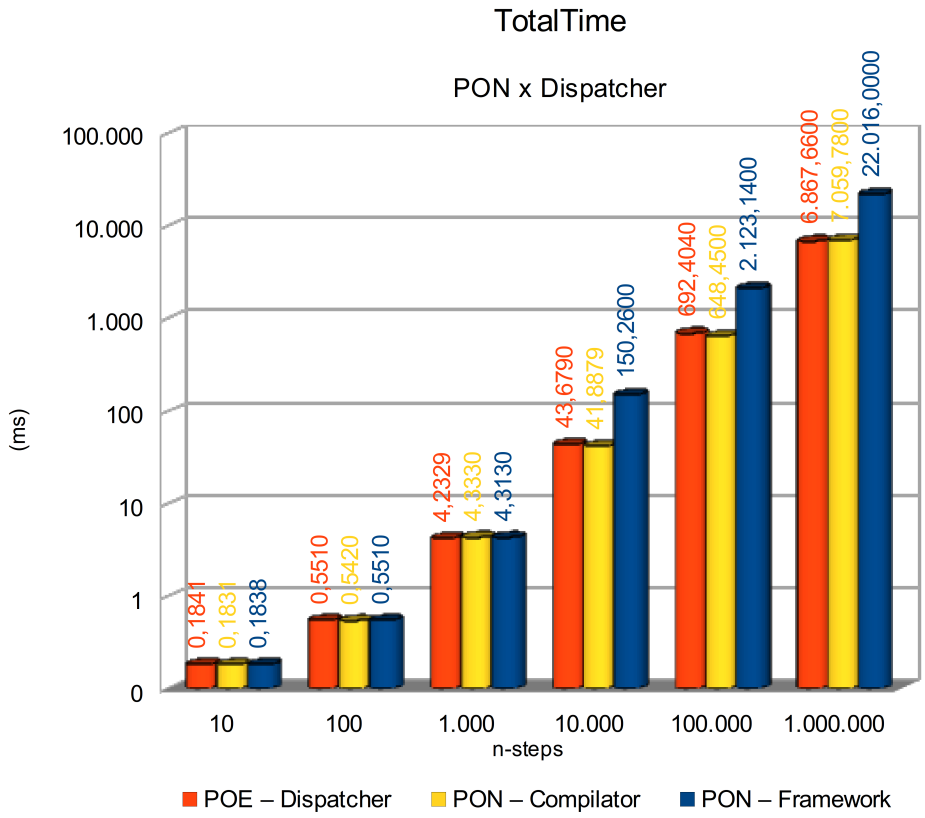

Figura 56: Gráfico com tempos de execução para cada solução, na simulação do primeiro cenário.

Para tempos de resposta, os dados a seguir representam a execução do roteiro – *script* – em PON (*Framework*), PON-Compilador (código intermediário gerado em C++) e POE. O eixo X representa o TIPO DE EVENTO e seta da esquerda para direita indica o transcorrer da execução do *software*. O eixo Y representa o respectivo tempo de resposta para *n-passos* em TIPO DE EVENTO para PON, PON-Compilador e POE. O gráfico da Figura 57 apresenta os dados da execução para o primeiro cenário para tempo de resposta, contrapondo PON, PON-compilador e POE, para *n-steps* igual a 10.000.

Para facilitar a compreensão dos resultados dos experimentos, os gráficos de tempo de resposta como o da Figura 57 possuem uma seta *execution path* (“caminho de execução”), diagramada da esquerda para direita, representando a característica temporal do experimento, ou seja, representando a ordem dos passos do roteiro de execução.

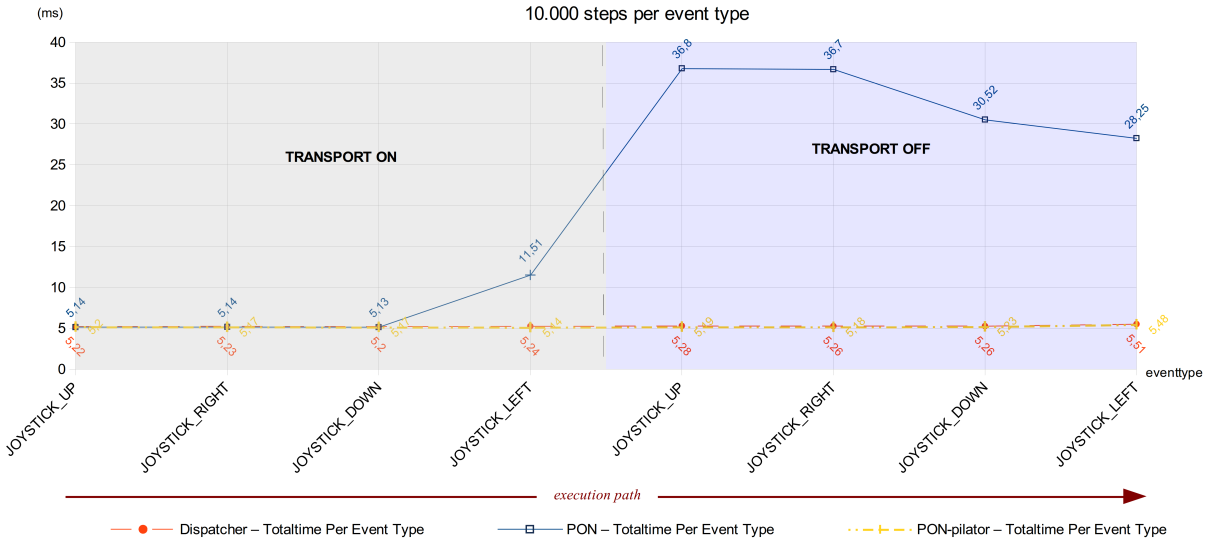

Figura 57: Gráfico com tempos de resposta (ms) por tipo de evento POE, PON e PON-Compilador durante a execução na simulação no primeiro cenário para 10.000 n-*steps*.

Pode-se perceber no gráfico a execução com tempo de resposta para o POE, PON-compilador e PON comparáveis até o terceiro tipo de evento (ou seja, durante a ocorrência de aproximadamente 30.000 eventos), e tanto POE quanto o programa gerado pelo PON-compilador apresentam tempo de resposta similares até o final do experimento. A partir do terceiro tipo de evento, na execução em *Framework* PON, o tempo de resposta se apresentou irregular. Cabe ressaltar que os eventos recebidos a partir do quinto tipo de evento (*JOYSTICK_UP*) são eventos de comportamento desprezado pelo *software* (*TRANSPORT_OFF*). Os três primeiros números em azul (PON), mesmo tendo tempo de resposta menor, estão desenhados acima da linha por limitações da ferramenta de plotagem gráfica (não é possível posicionar individualmente os rótulos de dados de cada ponto, somente por grupo de dados).

Essa anomalia em tempo de resposta durante execução em *Framework* PON suscitou uma investigação descrita resumidamente no Apêndice C - Investigação executada para identificação da anomalia/irregularidade e um novo experimento, descrito na seção 4.3.6,

explora os motivos desta anomalia.

O gráfico da Figura 58 apresenta os dados da execução para o primeiro cenário para tempo de resposta, contrapondo PON, PON-Compilador e POE, para *n-steps* igual a 1.000.000.

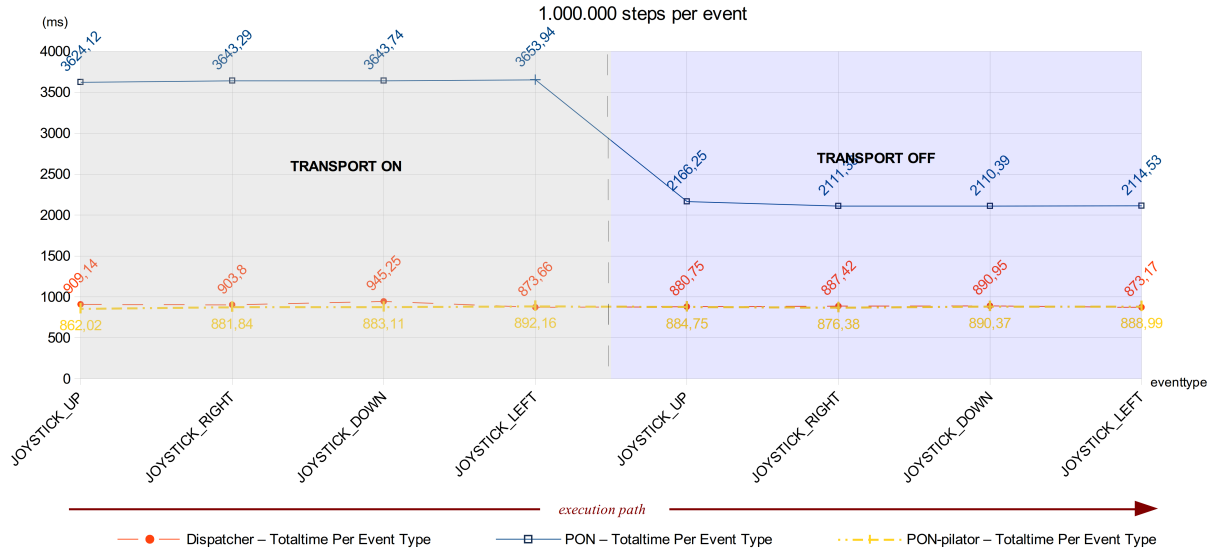

Figura 58: Gráfico com tempos de resposta (ms) por tipo de evento POE x PON durante a execução na simulação no primeiro cenário para 1.000.000 n-*steps*.

Fica demonstrado no gráfico que o tempo de resposta é praticamente idêntico entre o POE e o PON-compilador. Para a execução em *Framework* PON, o tempo de resposta é maior nos eventos que ocorrem enquanto o transporte está ligado (Transport ON), diminuindo quando o mesmo é desligado (Transport OFF).

4.3.2 Segundo experimento: PON versus *StateMachineHandler*

No segundo cenário, foram executados de forma automatizada e idêntica, em ambos os paradigmas POE e PON, os seguintes passos:

1. Estado inicial exosqueleto desligado e *joystick* em posição neutra;
2. Ligar o exosqueleto (chave fechada);
3. Mover o *joystick n*-passos para cada direção (X+, Y+, X-, Y-, Z+, Z-);
4. Mover o *joystick* para a posição neutra;

5. Ligar o braço mecânico (acionar o botão);
6. Mover o *joystick n*-passos para cada direção (X+, Y+, X-, Y-, Z+, Z-);
7. Mover o *joystick* para a posição neutra;
8. Desligar o braço mecânico (acionar novamente o botão);
9. Desligar o exosqueleto;
10. Mover o *joystick n*-passos para cada direção (X+, Y+, X-, Y-, Z+, Z-);
11. Mover o *joystick* para a posição neutra.

O número total de eventos gerados é dezoito eventos multiplicados por *n*-passos (18 x *n*-passos) mais sete eventos para mover o *joystick* para a posição neutra, mais eventos da chave para ligar e desligar o *Exoskeleton,* mais eventos de acionar o botão para ligar e desligar o braço mecânico. A Tabela 10 apresenta os dados de execuções para o segundo cenário para tempo total de execução.

Tabela 10: Tempos de execução em milissegundos (ms) para cada solução, na simulação no segundo cenário.

N-steps	POE – State	PON – Framework	Total Events
10	0,2871	0,2900	187
100	1,1130	1,1379	1.807
1.000	9,4548	9,6011	18.007
10.000	130,7140	590,5130	180.007
100.000	1.561,8200	7.069,5000	1.800.007
1.000.000	15.943,0000	65.974,3000	18.000.007

O gráfico na Figura 59 apresenta os dados de execuções para o segundo cenário para tempo total de execução, em escala logarítmica.

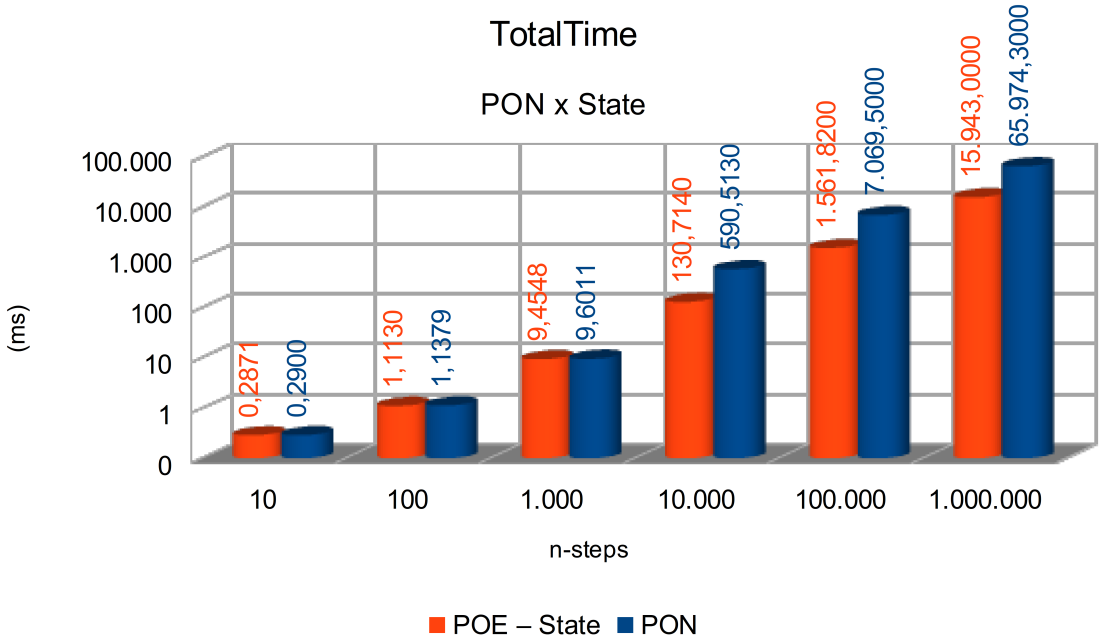

Figura 59: Gráfico com tempos de execução para cada solução, na simulação do primeiro cenário.

Neste segundo experimento, para tempos de resposta, os dados e gráficos e respectivos comportamentos se equipararam (em certa medida) ao primeiro experimento: anomalia a partir de 32k eventos tratados, tempo de resposta comparável entre POE e *Framework* PON até 1.000 *n-steps*, tempo de resposta linear em POE, tempo de resposta menor em eventos que devem ser desprezados em PON. Faz-se necessário registrar que até o momento de escrita desse documento não foi possível realizar os experimentos dos segundo e terceiro cenários em LingPON utilizando o PON-compilador.

4.3.3 Terceiro experimento: PON versus *ObserverHandler*

Neste primeiro experimento no terceiro cenário, foram executados de forma automatizada e idêntica, em ambos os paradigmas POE (*Observer Pattern*) e PON, os seguintes passos:

1. Estado inicial exosqueleto desligado, *joystick* em posição neutra e botões não pressionados;
2. Ligar o exosqueleto (chave fechada);
3. Mover o *joystick n*-passos para cada direção (X+, Y+, X-, Y-, Z+, Z-);
4. Mover o *joystick* para a posição neutra;
5. Ligar o braço mecânico esquerdo (acionar o botão vermelho);
6. Mover o *joystick n*-passos para cada direção (X+, Y+, X-, Y-, Z+, Z-);
7. Mover o *joystick* para a posição neutra;
8. Desligar o braço mecânico esquerdo (acionar novamente o botão);
9. Ligar o braço mecânico direito (acionar o botão azul);
10. Mover o *joystick n*-passos para cada direção (X+, Y+, X-, Y-, Z+, Z-);
11. Mover o *joystick* para a posição neutra;
12. Ligar também o braço mecânico esquerdo (acionar o botão vermelho). Ambos os braços ligados;
13. Mover o *joystick n*-passos para cada direção (X+, Y+, X-, Y-, Z+, Z-);
14. Mover o *joystick* para a posição neutra;
15. Desligar o exosqueleto (desliga também os braços mecânicos);
16. Mover o *joystick n*-passos para cada direção (X+, Y+, X-, Y-, Z+, Z-);
17. Mover o *joystick* para a posição neutra.

O número total de eventos gerados é dezoito multiplicado por *n*-passos (18 x *n*-passos) mais onze (11) eventos para mover o *joystick* para a posição neutra, mais eventos da chave para ligar e desligar o *Exoskeleton,* mais eventos de acionamento dos botões vermelho ou azul para ligar e desligar os braços mecânicos. A Tabela 11 informa os dados de execuções para o terceiro cenário para tempo total de execução.

Tabela 11: Tempos de execução em milissegundos (ms) para cada solução, na primeira simulação no terceiro cenário.

N-steps	POE – Observer	PON	Total Events
10	0,4231	0,4141	311
100	1,8640	1,9570	3.011
1.000	15,6890	16,2468	30.011
10.000	259,1200	2.034,1500	300.011
100.000	2.965,6000	15.348,8000	3.000.011
1.000.000	29.241,5000	152.381,0000	30.000.011

O gráfico na Figura 60 apresenta os dados de execuções para o terceiro cenário para tempo total de execução, em escala logarítmica.

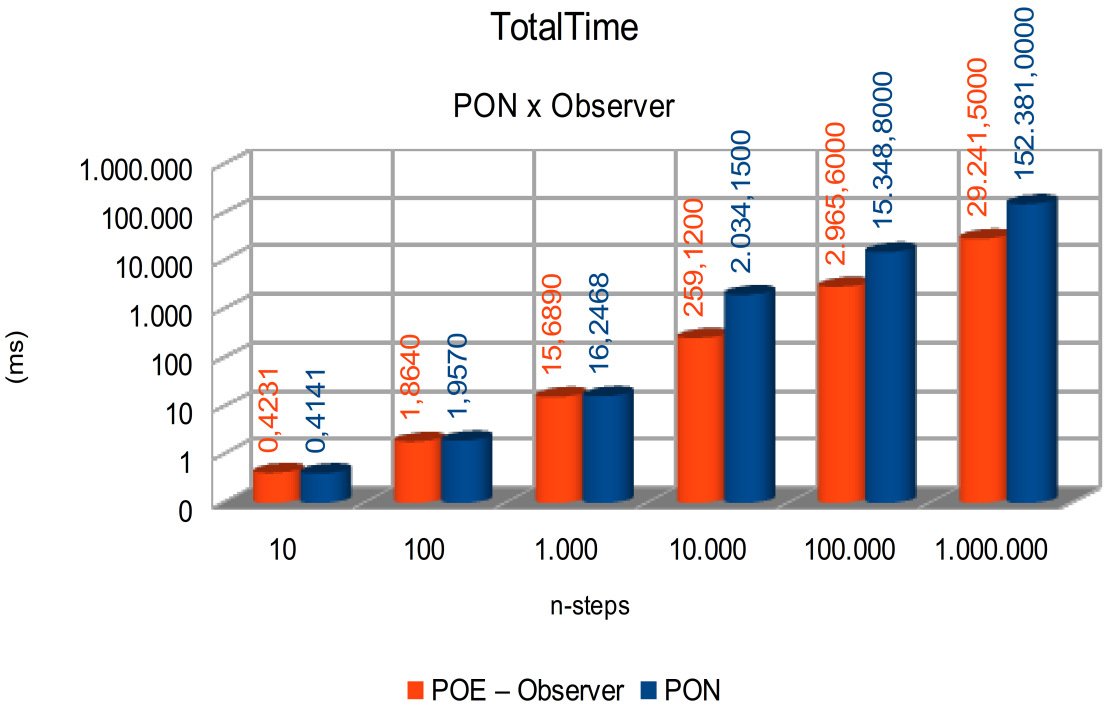

Figura 60: Gráfico com tempos de execução para cada solução, na simulação do primeiro cenário.

Neste terceiro experimento, para tempos de resposta, os dados e gráficos e respectivos comportamentos se equipararam (em certa medida) aos primeiro e segundo experimentos: anomalia a partir de 32 mil eventos tratados em *Framework* PON, tempo de resposta comparável entre POE e *Framework* PON até 1.000 *n-steps*, tempo de resposta linear em POE, tempo de resposta menor em eventos que devem ser desprezados em PON.

4.3.4 Quarto experimento: PON versus *ObserverHandler*

Neste segundo experimento no terceiro cenário, o roteiro almeja unir teste conceitual com a técnica. Assim, foram executados de forma automatizada e idêntica, em ambos os paradigmas POE (*Observer Pattern*) e PON, os seguintes passos:

1. Estado inicial exosqueleto desligado, *joystick* em posição neutra e botões não pressionados;
2. O piloto liga o exosqueleto (chave fechada);
3. O piloto liga ambos os braços mecânicos (para teste);
4. O piloto esbarra por descuido no *joystick* direção para frente (Y+) provocando 10 passos em eventos;
5. O piloto movimenta e depois alinha os braços, movendo o *joystick* 20 passos nas direções (Y-) e 10 passos em (Y+) provocando eventos em cada direção;
6. O piloto desliga os braços mecânicos;
7. O piloto comanda o exosqueleto para andar no roteiro inicial de trabalho (representada pelas setas em vermelho na Figura 61), 500 passos para direção Y+. A cada 500 o piloto “erra” 5 passos movimentado o *joystick* na direção (Z+) gerando eventos que são desprezados;
8. O piloto liga o braço direito e empurra um obstáculo (para desobstruir do caminho). Para tanto, ele movimenta o *joystick* gerando 90 passos para direção (Y-), 90 passos para direção (X+), 90 passos para direção (X-) e 90 passos para direção (Y+). A cada 90 passos o piloto “erra” e “corrige” 5 passos movimentado o *joystick* nas direções (Z+) e (Z-);
9. O piloto desliga o braço direito;
10. O piloto movimenta o *joystick* 500 passos em (Y+), 1.000 passos para direção (X+), 500 passos para direção (Y+) (sempre gerando os mesmos 5 passos de "erro" no *joystick* em Z+ a cada 500 passos);
11. O piloto liga o braço esquerdo e movimenta o *joystick* em (Y-) 80 passos, deixando o braço a frente do exosqueleto;
12. O piloto desliga o braço, e anda para frente, empurrando uma porta vai-e-vem com o braço esquerdo (para desobstruir do caminho), movimentando o *joystick* 3 passos em (Y+);
13. Ao passar pela porta, o piloto alinha o braço movimentando o *joystick* em (Y-) 80 passos e depois desliga o mesmo;
14. O piloto movimenta o exosqueleto até o meio da sala andando 250 passos em (Y+);
15. O piloto "brinca" com o manche do *joystick* no sentido horário e anti-horário, em 10 passos cada um (Z+ e Z-);
16. O piloto movimenta o exosqueleto até a estante vermelha no roteiro, andando 250

passos em (Y+);

17. O piloto liga os dois braços, depois os alinha em paralelo, movimentando o *joystick* 80 passos em (Y-) e 50 passos em (Z+), assim esticando-os, e depois os desliga;
18. O piloto movimenta o exosqueleto um pouco até alinhar em um objeto na estante, andando 1 passos em (Y+);
19. Para levantar o objeto, o piloto liga os dois braços, e em seguida movimenta o *joystick* 5 passos em (Y-), e depois os desliga;
20. O piloto faz todo o caminho contrário no roteiro carregando o objeto até a estante azul (o caminho contrário é indicado no desenho do roteiro em setas azuis na Figura 61): 500 passos para direção (Y-), passa a porta vai-e-vem, mais 500 passos para direção (Y-), 1.000 passos para direção (X-), 500 passos para direção (Y-), passa do lado do obstáculo (caminho já desobstruído), 500 passos para direção (Y-). A cada 500 passos o piloto "erra" 5 passos movimentado o *joystick* na direção (Z+) gerando eventos que são desprezados;
21. O piloto movimenta o exosqueleto um pouco até alinhar o objeto na estante azul, andando 1 passos em (Y-);
22. O piloto liga os dois braços, e movimenta o *joystick* 5 passos (Y+), assim colocando o objeto na estante;
23. Depois o piloto recolhe os braços com 50 passos em (Z-) e 80 passos em (Y+), e logo então os desliga;
24. O piloto "brinca" com o manche do *joystick* no sentido horário e anti-horário, em 10 passos cada um (Z+ e Z-);
25. Ao realizar todas as tarefas do roteiro, o piloto estaciona o exosqueleto desligando-o.

Nesse roteiro ocorrem 6970 eventos para mover o *joystick*, mais vinte (20) eventos da chave para ligar e desligar o *Exoskeleton* e eventos de acionamento dos botões vermelho ou azul para ligar e desligar os braços mecânicos. Assim, neste experimento, o número total de eventos gerados é 6990 multiplicado por *n* vezes com que o roteiro é repetido (*n* x 6990). A Figura 61 demonstra conceitualmente o roteiro deste experimento.

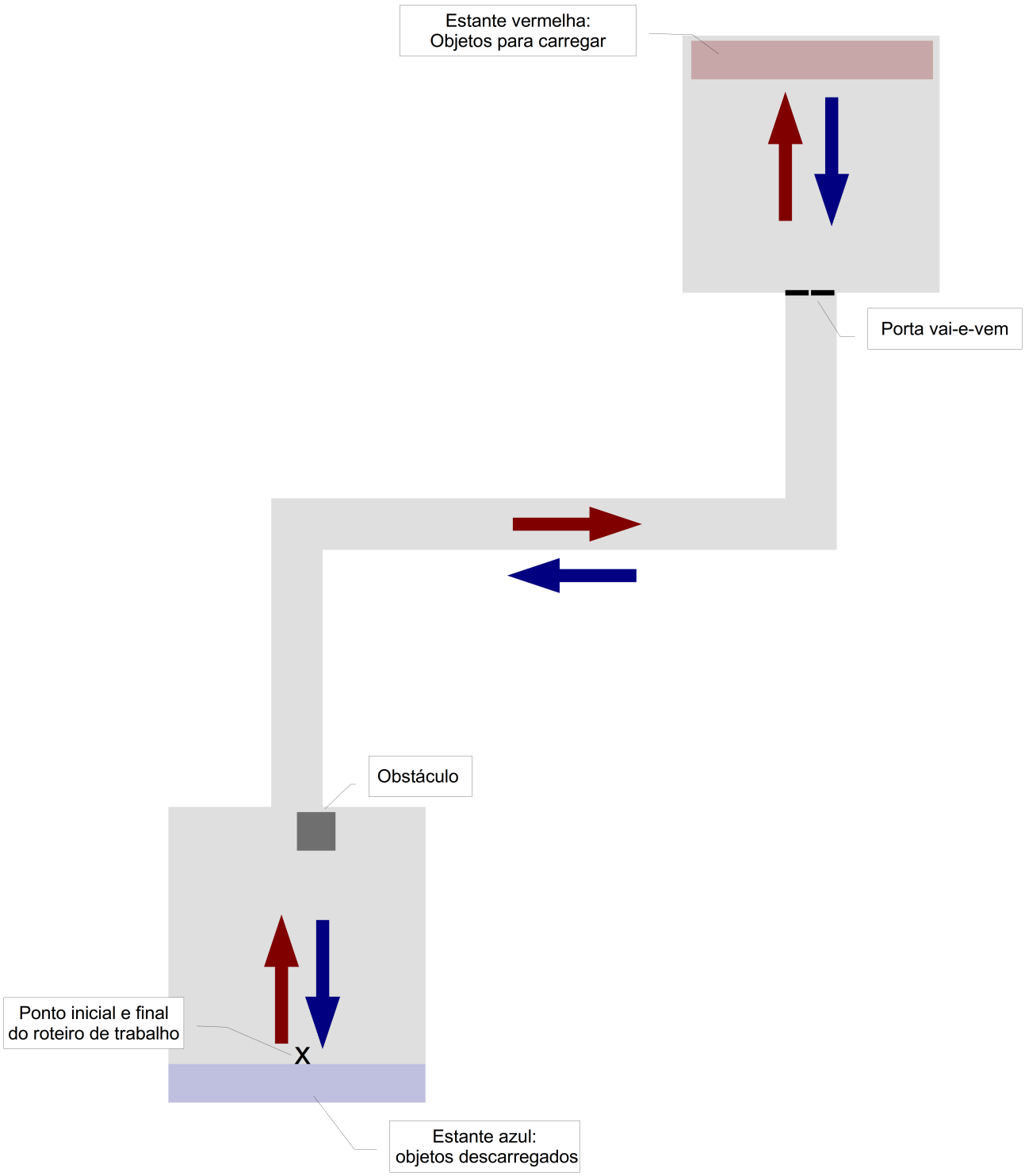

Figura 61: Desenho do roteiro do quarto experimento PON e *Observer*, terceiro cenário do primeiro caso de estudo.

A Tabela 12 apresenta os dados de execuções para o quarto experimento no terceiro cenário (segunda simulação neste cenário) para tempo total de execução.

Tabela 12: Tempos de execução em milissegundos (ms) para cada solução, na segunda simulação no terceiro cenário.

N-steps	POE – Observer	PON – Framework	Total Events
1	4,0012	4,0850	6.990
10	40,1470	183,2830	69.900
100	673,7120	3.026,6100	699.000
1.000	6.587,3100	31.670,5000	6.990.000
2.500	17.387,0000	79.981,2000	17.475.000
5.000	33.843,9000	163.012,0000	34.950.000

O gráfico na Figura 62 apresenta os dados de execuções para o segundo experimento no terceiro cenário para tempo total de execução, em escala logarítmica.

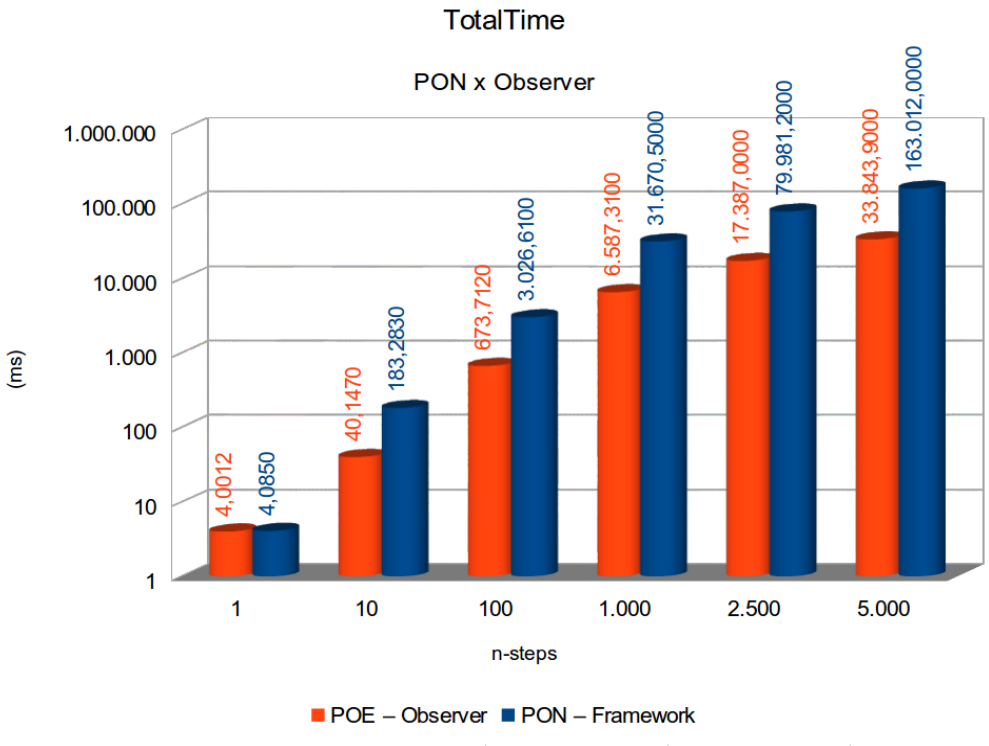

Figura 62: Gráfico com tempos de execução para cada solução, nesta segunda simulação do terceiro cenário do primeiro caso de estudo.

Neste quarto experimento, para tempos de resposta, os dados e gráficos e respectivos

comportamentos se equipararam (em certa medida) aos primeiro, segundo e terceiro experimentos: anomalia a partir de 32 mil eventos tratados em *Framework* PON, tempo de resposta comparável entre POE e *Framework* PON até cerca de 4 *n-steps* (neste roteiro cada *n-steps* tem 6.990 eventos, a partir de 32.000 ocorre aumento de tempo de resposta em *Framework* PON conforme já explanado, ou seja, são comparáveis até cerca 4,68 *n-steps* neste roteiro), tempo de resposta linear em POE, tempo de resposta menor em eventos que devem ser desprezados em PON.

4.3.5 Quinto experimento: Caso de Estudo Portão Eletrônico PON versus POE (*Dispatcher*, *State* e *Observer*)

Para realizar a medição de *software* em execução do segundo caso de estudo foi necessário estabelecer uma forma de experimento em alinhamento com o tipo de *software*. As primeiras hipóteses para medição (e que foram descartadas) eram:

1) Tempos em execução (desempenho) como no primeiro caso de estudo. Tal medição não se mostraria coerente, pois a simples geração de eventos continuamente e repetidamente em sequência (*e.g.* sem intervalos em frequência máxima) não testaria todos os estados do Portão Eletrônico. Se assim fosse executado, o ciclo de estados (*loop*) do Portão Eletrônico ficaria <ABRINDO, PAROU DE ABRIR, FECHANDO, PAROU DE FECHAR> ao serem gerados eventos de controle remoto seguidamente, sem nunca chegar aos estados FECHADO ou ABERTO, assim ficando conceitualmente “parado”.

2) Quantidade de avaliações causais em POE versus número de regras aprovadas em PON. Foi feita a tentativa, entretanto as estruturas *SWITCH-CASE* são transformadas no processo de compilação em tabelas de salto para rótulos – *jump table* para *labels*. Uma *jump table* é um *array* com o índice sendo o valor escalar da condição do *SWITCH-CASE*. Dessa maneira não existe avaliação causal, somente a instrução para direcionar a execução ao bloco correspondente (*jump*) identificado por um rótulo – *label* – que no caso é a própria denominação do *array* com seu respectivo índice (valor escalar).

A alternativa encontrada foi medir a execução conforme um ciclo completo do roteiro

de teste estabelecido (cumprimento dos requisitos executando o roteiro inteiro) versus tempo entre geração de eventos (frequência) e o tempo referência do processo de abertura e fechamento do portão. Assim, testando a execução quando o tempo entre eventos diminui e tempos total para abertura do portão e parcial também diminuem. Importante enfatizar que nesse caso de estudo existem os eventos do controle remoto (externo ao *software*) e do *timer* (interno ao *software*).

Para todas as implementações de Portão Eletrônico, em ambos os paradigmas POE (*Dispatcher*, *State Pattern* e *Observer Pattern*) e PON, foram executados de forma automatizada e idêntica, os seguintes passos:

1. Estado inicial portão fechado e controle remoto desligado;
2. Abrir completamente o portão (acionar o controle remoto);
3. Aguardar o portão abrir (aguardar o tempo total de abertura + 5);
4. Fechar completamente o portão (acionar o controle remoto);
5. Aguardar o portão fechar (aguardar o tempo total de fechamento + 5);
6. Abrir parcialmente o portão (acionar o controle remoto);
7. Aguardar o portão abrir um pouco (aguardar o tempo parcial + 1);
8. Parar de abrir o portão (acionar o controle remoto);
9. Aguardar um pouco (aguardar o tempo parcial);
10. Fechar completamente o portão (acionar o controle remoto);
11. Aguardar o portão fechar (aguardar o tempo total de fechamento + 5);
12. Abrir completamente o portão (acionar o controle remoto);
13. Aguardar o portão abrir (aguardar o tempo total de abertura + 5);
14. Fechar parcialmente o portão (acionar o controle remoto);
15. Aguardar o portão fechar um pouco (aguardar o tempo parcial + 1);
16. Parar de fechar o portão (acionar o controle remoto);
17. Aguardar um pouco (aguardar o tempo parcial);
18. Abrir completamente o portão (acionar o controle remoto);
19. Aguardar o portão abrir (aguardar o tempo total de abertura + 5);
20. Fechar completamente o portão (acionar o controle remoto);
21. Aguardar o portão fechar (aguardar o tempo total de fechamento + 5).

Assim, com o roteiro de execução proposto, o objetivo foi testar todos os estados possíveis do Portão Eletrônico, terminando no estado inicial Portão Fechado. O número total

de eventos gerados é dez eventos de controle remoto acionado mais seis eventos de término do contador de tempo, realizando assim um total de dezesseis eventos.

Os estados que ocorrem do Portão Eletrônico neste roteiro, na sequência, são: FECHADO, ABRINDO, ABERTO, FECHANDO, FECHADO, ABRIR, PAROU DE ABRIR, FECHANDO, FECHADO, ABRINDO, ABERTO, FECHANDO, PAROU DE FECHAR, ABRINDO, ABERTO, FECHANDO, FECHADO. A partir disso se registra toda execução do roteiro de uma solução PON e POE (*Dispatcher*, *State* ou *Observer*) e, a partir do padrão com tempo de referência de 1.000 milissegundos (1 segundo), se compara com os tempos de referência menores para verificar a corretude e completude da execução (ocorrência de todos estados na sequência correta por fim terminando com o portão FECHADO). Foram executados experimentos com os tempos de referência (em milissegundos): 1.000 (1 segundo), 100, 10, 1 e 0.1. Todas as soluções técnicas falharam em corretude e completude (não executaram todos os passos e não passaram por todos os estados do roteiro) a partir do tempo de referência 0,1 milissegundos.

O tempo referência total (TRT) de execução ideal é dado pela seguinte fórmula:

6 multiplicado por (Tempo Total de abertura ou fechamento + 5) +

2 multiplicado por (Tempo Parcial de abertura ou fechamento + 1) +

2 multiplicado por (Tempo Parcial de abertura ou fechamento)

Seja o tempo total de abertura ou fechamento conforme requisito inicial do caso de estudo 30 segundos, ou seja, 30 multiplicado pelo tempo de referência (TR). E o tempo parcial a metade disso, ou seja, 15 segundos multiplicados pelo tempo de referência (TR), originando a fórmula:

$$[6 \times (30 + 5) \times TR] + [2 \times (15 + 1) \times TR] + [2 \times (15) \times TR] = 272 \times TR$$

O gráfico na Figura 63 representa os valores de tempo de referência para o experimento do caso de estudo Portão Eletrônico com tempo de referência 1 ms (o menor tempo de referência em que os experimentos obtiveram sucesso completando o roteiro de forma correta). Os demais gráficos com tempos de referência 1.000, 100 e 10 ms (tempos de referência maiores) se encontram no Apêndice A.

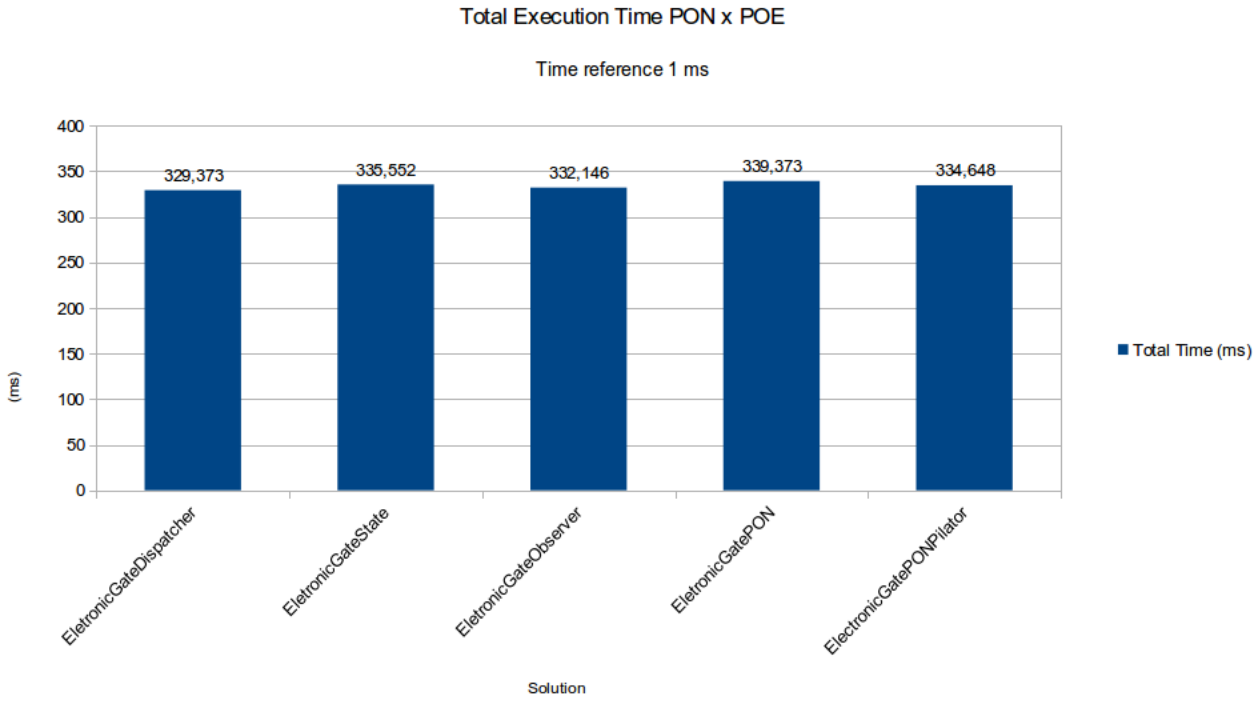

Figura 63: Gráfico de tempo de execução do caso de estudo Portão Eletrônico com tempo de referência 1ms.

A Tabela 13 a seguir representa os dados consolidados de todos os experimentos do caso de estudo Portão Eletrônico, contendo também o Tempo Total de Referência (TRT) e o respectivo Tempo de Referência (TR).

Tabela 13: Tempos totais incluindo tempo ideal de referência dos experimentos do segundo caso de estudo.

TR (*ms*)	**1.000,00**	**100,00**	**10,00**	**1,00**	**0,10**
TRT (272 X TR) (*ms*)	272.000,00	27.200,00	2.720,00	272,00	27,20
EletronicGateDispatcher	272.062,00	27.266,10	2.782,63	329,37	X
EletronicGateState	272.062,00	27.264,80	2.785,82	335,55	X
EletronicGateObserver	272.068,00	27.262,50	2.780,87	332,15	X
EletronicGatePON	272.064,00	27.263,40	2.780,26	339,37	X
ElectronicGatePONPilator	272.059,00	27.260,30	2.781,30	334,65	X

4.3.6 Sexto experimento: comportamento irregular a partir de 32 mil eventos recebidos no *software* em *Framework* PON

Além da investigação executada (conforme Apêndice C - Investigação executada para identificação da anomalia/irregularidade) foi realizado um experimento especial para angariar informações durante a execução de *software*. Para tanto se utilizou do *software* feito no primeiro cenário, em POE, em PON-compilador e *Framework* PON.

Para todas as implementações, em ambos os paradigmas POE (*Observer Pattern*), PON e PON-Compilador, foram executados de forma automatizada e idêntica (o objetivo é executar *n* número de eventos), os seguintes passos:

1. Estado inicial exosqueleto desligado, *joystick* em posição neutra e botões não pressionados;
2. Ligar o exosqueleto (chave fechada);
3. Mover o *joystick n*-passos para uma única direção.

As quantidades de eventos escolhidas foram 0, 1, 11, 32768, 32769, 32770 e 65536. Além do tempo total de execução medido normalmente como nos outros experimentos, a instrumentação utilizou ferramental específico para *profiling* C++ (*e.g.* `perf stat`) para registrar os dados e medições específicas durante a execução. A Tabela 14, apresentada a seguir, apresenta os dados obtidos, com os nomes dos *software*s, a quantidade de eventos, o tempo total de execução, a quantidade de instruções executadas, a quantidade de cargas no cache L1 de instruções do processador, a quantidade de falhas de leitura no cache L1 de instruções do processador, a quantidade de cargas no cache L1 de dados do processador, a quantidade de falhas de leituras no cache L1 de dados do processador. Os tempos marcados com <not counted> (não contados) são os eventos de *hardware* não amostrados por conta da alta velocidade de execução (limitação de amostragem da ferramenta).

Tabela 14: Dados de instrumentação *Framework* PON, PON-Compilador e *Observer* no sexto experimento.

	Events	Total Time (ms)	Instructions	L1-icache-loads	L1-icache-load-misses	L1-dcache-loads	L1-dcache-load-misses
ExoskeletonPonpilatorCPP	0	0,0020	253.123	<not counted>	<not counted>	1.050.845	<not counted>
ExoskeletonTwoArmEventHandlerSystemPON	0	0,0020	28.967.707	<not counted>	<not counted>	11.317.747	<not counted>
ObserverHandlerSystem	0	0,0020	17.558.163	<not counted>	<not counted>	6.949.000	<not counted>
ExoskeletonPonpilatorCPP	1	0,0920	3.180.130	<not counted>	<not counted>	1.090.696	<not counted>
ExoskeletonTwoArmEventHandlerSystemPON	1	0,0920	8.815.901	<not counted>	<not counted>	6.380.032	<not counted>
ObserverHandlerSystem	1	0,0940	4.226.198	<not counted>	<not counted>	1.511.592	<not counted>
ExoskeletonPonpilatorCPP	11	0,1299	3.127.764	<not counted>	<not counted>	1.096.262	<not counted>
ExoskeletonTwoArmEventHandlerSystemPON	11	0,0908	16.278.366	<not counted>	<not counted>	6.383.379	<not counted>
ObserverHandlerSystem	11	0,0901	4.207.848	<not counted>	<not counted>	1.509.094	<not counted>
ExoskeletonPonpilatorCPP	32768	21,8508	22.766.096	<not counted>	<not counted>	43.698.853	14.130
ExoskeletonTwoArmEventHandlerSystemPON	32768	17,1550	647.480.659	400.239.442	195.408	272.393.508	1.112.820
ObserverHandlerSystem	32768	19,5090	99.190.252	<not counted>	<not counted>	58.014.955	20.212
ExoskeletonPonpilatorCPP	32769	17,1990	57.965.319	<not counted>	<not counted>	47.672.979	16.859
ExoskeletonTwoArmEventHandlerSystemPON	32769	17,6809	654.431.684	382.357.971	396.843	267.261.107	1.722.190
ObserverHandlerSystem	32769	17,4941	114.306.496	<not counted>	<not counted>	55.784.710	20.526
ExoskeletonPonpilatorCPP	32770	17,4309	114.821.424	<not counted>	<not counted>	48.717.464	18.428
ExoskeletonTwoArmEventHandlerSystemPON	32770	**17,1350**	647.723.255	382.522.489	401.476	266.398.852	1.115.345
ObserverHandlerSystem	32770	18,0330	119.798.204	<not counted>	<not counted>	56.429.553	20.975
ExoskeletonPonpilatorCPP	65536	35,1179	239.269.947	149.509.215	<not counted>	93.037.075	20.732

Os dados angariados originam as seguintes análises e reflexões sobre o desempenho de execução desses *softwares*:

- As quantidades de eventos que trouxeram evidência de um aumento de tempo maior de execução (mais do que o dobro) em *Framework* PON foram justamente entre 32770 e 65536 eventos (aproximadamente o dobro de eventos), demonstrando tempo total de 17,1350 ms e 163,9470 ms respectivamente. Em POE e PON-Compilador os tempos foram aproximadamente o dobro em tempo de execução;
- Todos os outros dados indicaram que, para os critérios de número de instruções executadas, número de cargas no cache L1 de instruções do processador, número de falhas de leitura no cache L1 de instruções do processador e número de cargas no cache L1 de dados do processador, a quantidade de falhas de leituras no cache L1 de dados do processador, os números praticamente dobraram para PON, PON-Compilador e POE (*Observer*).
- O *Framework* PON apresentou número maior de instruções executadas em todos os experimento. Entretanto o tempo total de execução foi *pari passu* com os outros *softwares* até o número identificado em torno de 32k eventos. Entretanto, não houve aumento expressivo em falhas de leitura no cache L1 de instruções do processador para *software* feito em *Framework* PON.
- Os números de falhas de leituras no cache L1 de dados do processador para PON entre essas quantidades de eventos mais que duplicou (de 1.115.345 para 2.988.606, ou seja, uma relação de 2,679 vezes a mais).
- Sabe-se da latência de leitura maior entre memórias cache L1, L2 e Principal. Ou seja, o aumento de tempo de resposta a partir de 32k eventos durante a execução seria por conta que cada falha de leitura em número a maior, importa em acréscimo em tempo de acesso aos dados, indicando também por consequência os tempos maiores em termos de tempo total de execução em PON.
- Alguns dos possíveis motivos para o comportamento irregular encontrado ao utilizar o *Framework* PON são: problema de alocação de memória ainda não encontrado, problema de programação ainda não encontrado, comportamento intrínseco de arquitetura de *hardware* (*e.g.* atualização e algoritmos de memória cache). Apesar da exaustiva investigação dos reais motivos quanto a esse comportamento irregular, como

demonstrado neste experimento e no Apêndice C - Investigação executada para identificação da anomalia/irregularidade, a mesma foi parcialmente inconclusa até o final deste trabalho de pesquisa indicando possíveis causas do comportamento irregular.

4.4 Resultados/Reflexões das comparações

Como reflexões finais deste capítulo, vale relembrar que PON e POE foram comparados segundo uma taxonomia comum, demonstrando semelhanças e diferenças conceituais entre ambos os paradigmas, dirimindo, de modo imparcial, a imprecisão entre conceitos dos dois paradigmas (PON e POE). Em seguida, comparações em medidas de complexidade de código-fonte mostraram que PON tem código conciso e expressivo e aqui, por fim, são apresentadas reflexões acerca das medições de desempenho.

Os experimentos que utilizaram o primeiro *software* de tratamento de eventos (*Exoskeleton)* e *Framework* PON, os tempos de resposta se equipararam nos primeiro, segundo, terceiro e quarto experimentos, indicando o comportamento irregular a partir de 32 mil eventos tratados em *Framework* PON, e tempo de resposta comparável entre POE e *Framework* PON nos demais casos com tempo de resposta linear em ambos, tempo de resposta menor em eventos que devem ser desprezados em PON, e maiores tempos de resposta ao tratar eventos que requerem mais de uma ação.

No primeiro cenário e respectivo experimento, também se instrumentou *software* programado em LingPON e compilado em PON-compilador (que gera códigos *Framework*, C e C++, neste trabalho, foi escolhida a geração de código em C++). Tanto os tempos totais bem como os tempos de resposta foram muito próximos entre POE e código PON-compilador.

No quinto experimento, que instrumentou *softwares* do caso de estudo Portão Eletrônico, foram comparados em *Framework* PON, LingPON, *Handler-Dispatcher, State* e *Observer*. Nesse contexto, o respectivo roteiro utiliza um tempo de referência comum entre geração do *software* e tempo interno de *timer*. Todos os dados indicaram execuções corretas e completas até o tempo de referência de um milissegundo (1 ms). A partir e menor do que o tempo de referência de 0,1 ms todas as soluções não completaram corretamente o roteiro fictício.

Acerca do experimento de desempenho, ao seu turno, o roteiro de testes engloba a geração de eventos independente do estado do exosqueleto (ligado ou desligado), pois sistemas orientados a eventos, com interação constante com o ambiente externo, necessitam tratar, filtrar ou até mesmo desprezar eventos. Assim, os eventos são recebidos independentemente do estado dos objetos na simulação (*i.e.* em simulação via *software* não existe a ativação elétrica como na operação de equipamentos físicos). Nesses termos, no funcionamento de *software* pode existir desperdício de avaliações ou simplesmente eventos que são ignorados.

Em POE, essas avaliações desperdiçadas ou eventos ignorados são gerenciadas comumente por avaliações causais. Em PON, isso é tratado via *Attributes*, *Premises* ou até mesmo em *Attributes* impertinentes como demonstrado em [Ronszcka, 2012] (*i.e. Premises* que se descadastram do recebimento de notificações de *Attributes* em certas condições). Dessa forma, PON explicita o comportamento de *software*, ou seja, qual objetivo deve cumprir de acordo com um requisito. Como exemplo, uma *Premise* que recebe notificações do *Attribute Exoskeleton→atExoskeletonState*, para o *Exoskeleton* ligado ou desligado, unifica o comportamento de uma *Rule* do *software* em PON, em contrapartida às avaliações causais dispersas em POE.

Por fim, um experimento técnico para instrumentação do comportamento irregular a partir de 32 mil eventos indicou evidências de número maior de falhas de leituras no cache L1 de dados do processador.

No capítulo derradeiro seguinte, são apresentadas as conclusões, contribuições principais, contribuições secundárias e as indicações para trabalhos futuros.

Capítulo 5 - Conclusões e Trabalhos Futuros

Este capítulo apresenta as conclusões deste trabalho, elenca as contribuições (principais e secundárias) e indica possíveis trabalhos futuros.

5.1 Conclusão

Este trabalho teve como principal objetivo comparar os paradigmas POE e PON. Com esse intuito, tanto PON quanto POE foram detalhados segundo os critérios de suas características estruturantes de acordo com a taxonomia proposta por Van Roy. No caso, foi possível estabelecer uma visão conceitual, estruturada e tabular, relacionando e comparando ambos os paradigmas, de acordo com os conceitos elementares de paradigmas: registro, *closure,* independência, estado nomeado, não-determinismo observável e conceitos singulares. Ainda, este trabalho contribuiu redesenhando a taxonomia, incluindo PON e formulando um roteiro de questões que ajudam na comparação de paradigmas de *software* ao estruturar a própria atividade de comparação. Nesse sentido, se afirma que é possível comparar estruturalmente PON também com outros paradigmas de programação, assim como realizado aqui neste trabalho.

Ademais, na comparação estrutural foram identificadas conceitos elementares ao PON dentro da própria taxonomia, como *closure*, célula local e o ciclo de inferência por notificações. Em convergência aos paradigmas OO (dentre outros), possui recipientes de escopo léxico, particularmente *FBEs* análogas aos objetos da OO. Em segundo lugar, possui célula local (cada unidade computacional PON tem sua própria responsabilidade de estado), em convergência aos paradigmas da coluna de passagem de mensagem (conforme Figura 1 – pg. 16) [Van Roy, 2009]. Por último, foi apresentado um conceito particular, que identifica unicamente o Paradigma Orientado a Notificações: inferência por notificações [Simão *et al.*, 2014].

No estado da arte, PON apresenta viés convergente em toda linha de pesquisa com trabalhos em andamento. POE é disseminado profissional e academicamente, entretanto existem discrepâncias em sua terminologia e diversos pontos de vista. Cabe ressaltar que PON tem inspirações em eventos, principalmente por utilizar inferências por notificações entre suas unidades computacionais em seu ciclo de execução.

Para comparar PON e POE em medidas de complexidade de código, utilizando como critérios de número de linhas de código (LOC), número de escopos (recipientes de escopo léxico - *closures*) e número de *tokens* necessários, os dados mostraram números menores de LOC para PON em vários casos, maior estabilidade de código-fonte entre cenários de *software* que foram construídos de modo complementar (manutenções evolutivas), números menores de escopos e números menores de *tokens* nas medidas dos *softwares*. Dessa maneira, indicam uma vantagem em concisão e expressividade de código fonte ao utilizar PON, e ainda maior em LingPON (que ainda está em desenvolvimento por outros pesquisadores do grupo de pesquisa PON).

Já ao comparar PON e POE em medições durante execução, os resultados mostraram:

- Utilizando *Framework* PON o tempo total de execução é comparável às abordagens desenvolvidas em POE até se atingir o número de 32 mil eventos processados, com uma aumento no tempo total de execução a partir desse número, com evidências de número maior de falhas de leituras no cache L1 de dados do processador (resultado exposto na seção 4.3.6 Sexto experimento: comportamento irregular a partir de 32 mil eventos recebidos no software em Framework PON);
- Em PON compilado e respectiva linguagem LingPON os experimentos realizados demonstraram em todos os números de *n-eventos:* tempo comparável em relação ao POE (resultados expostos na seção 4.3 Comparações em medidas de execução de software em PON e POE);
- Em relação ao tempo de resposta, instrumentando o *software* durante a execução e utilizando o *Framework* PON, em altos números de eventos, a curva dos tempos de resposta do próprio PON demonstrou tempos de resposta menores quando os eventos devem ser desprezados, e tempos de resposta (ligeiramente) maiores quando os eventos instigam mais de uma ação (resultados expostos nos experimentos primeiro a quarto, subseções 4.3.1 a 4.3.4, da seção 4.3 Comparações em medidas de execução de software em PON e POE);

- Em relação às comparações entre PON e POE para *software* cujo comportamento característico é mudança de estado (*e.g.* Portão Eletrônico), e utilizando um roteiro de testes que deve ser cumprido totalmente, utilizando um tempo de referência único para geração de eventos e mudança de estado no *software*, todos os *software* em PON, PON compilado e POE tiveram os mesmos resultados, executando o roteiro completamente (em corretude) até o tempo de referência de 1 milissegundo (0,001 segundos) e falharam a partir de 0,1 milissegundos (resultado exposto da seção 4.3.5 Quinto experimento: Caso de Estudo Portão Eletrônico PON versus POE (Dispatcher, State e Observer)).

Por fim, ao construir *software* em ambos os paradigmas, ficou caracterizado que PON tem maior expressividade em programação, algo notório quanto em comparação ao uso da técnica *State* e *Observer*. Com indícios (bem motivados em percepções pessoais do autor do trabalho) em grau de duplicação de código em *State* (informação que se relaciona aos números das medidas de complexidade de código-fonte) e em maior grau de dificuldades e tempo para construir com *Observer*. Em POE houve indícios de exagero de engenharia (*Over Engineering*), ou seja, uso de técnicas mais complexas que o necessário para resolver problemas simples (*i.e. State* e *Observer* no caso de estudo Portão Eletrônico). Ao bem da verdade, programar os casos de estudo em PON (e principalmente em LingPON) foi mais fácil e rápido.

5.2 Contribuições principais deste trabalho

As principais contribuições deste trabalho são:

- Estabelecer que PON é diferente de POE. Os critérios principais são o referencial bibliográfico, em que se identificou uma possível ortogonalidade na conceituação de POE (que tende a ser compreendido como uma forma de se programar e tratar eventos em diversos paradigmas, ao invés de se caracterizar como um paradigma único), a comparação estruturada, construção de casos de estudo e respectivas medições em PON e POE. A taxonomia de Van Roy foi expandida para contemplar o PON e a programação de *software* em PON. O espectro de pesquisa em POE não foi completo

nesta investigação (*e.g.* englobando o lado esquerdo da taxonomia de Van Roy [Van Roy, 2009] e outras linguagens de programação [Bainomugisha *et al.*, 2013]), pois a gama de possibilidades é muito grande para um trabalho científico de curto período (mestrado). Essas possibilidades englobam linguagens de programação, técnicas e estilos de programação, e composição de padrões para resolução de tratamento de eventos em POE.

- Foram estabelecidas comparações em complexidade de código que indicam melhores números em complexidade de código-fonte para o PON (tanto em *Framework* PON quanto LingPON).
- Foram estabelecidas comparações em tempo de resposta e tempo total de execução que indicam que PON apresenta números tão bons quanto POE em vários casos, demonstrando inclusive uma possível adaptação de desempenho durante execução inerente ao Paradigma Orientado a Notificações.
- Propõe-se que PON é uma alternativa para tratamento de eventos em *software*, sendo oportunamente integrável, por exemplo, entre *Framework* C++ e OO C++. As abstrações de evento em *software* podem ser um *Attribute* como demonstrado na programação dos casos de estudo.

5.3 Contribuições secundárias deste trabalho

As contribuições secundárias deste trabalho são:

- Um referencial bibliográfico em POE com indicações para trabalhos futuros em eventos;
- Início de validação empírica da Taxonomia de Van Roy como instrumento de comparação entre paradigmas;
- Perguntas a serem feitas a partir da taxonomia de Van Roy e duas novas perguntas para essa mesma taxonomia de paradigmas de programação (contribuição exposta na seção 4.1 Comparação de características estruturantes);
- Dois novos *software*s, notadamente de tratamento de eventos, para o grupo de pesquisa em PON, sendo um projeto novo Simulador de Transporte Individual e reconstrução (completa e funcional) de Portão Eletrônico;

- Enumeração de algumas formas de comparação de paradigmas de programação, em resumo:
    - Formal, conceitos (relação analítica entre paradigmas), forma de uso e técnica de programação, métricas sob ponto de vista de projeto de *software*, medidas do tempo de resposta e tempo de execução do *software*.
- Novas medições em *software* em PON, a saber, de tamanho do código-fonte (LOCs, *closures* e *tokens);*
- Nova medição de execução de *software* em PON, a saber, tempo de resposta;
- Evidência explícita de influência de arquitetura de *hardware* em execução de *software* PON feito via *Framework* PON.
- Indicações de algumas possibilidades de futuras pesquisas, apresentadas na seção seguinte.

5.4 Trabalhos futuros

Por fim, cabe apresentar indicações para trabalhos futuros, problemas e nova questões de pesquisas obtidas com o presente trabalho de investigação.

5.4.1 Comparações com outros paradigmas

Podem ser realizados, a partir desta pesquisa, futuras comparações. A partir da nova taxonomia com PON incluso e das sentenças de perguntas, pode-se demarcar futuras comparações entre paradigmas como PON versus Programação Funcional Reativo, versus Programação Síncrona, versus Programação de Fluxo de Dados, versus Síncrona Discreta.

5.4.2 Facilidade de programação

A partir do executado neste trabalho investigativo em relação à programação de cenários complementares, novas perguntas de pesquisa foram oportunizadas:

"É mais fácil programar software PON em casos de manutenção de software (erro na especificação ou bug)?"

"É mais fácil refatorar software programado em PON (reformulação do código sem alterar o resultado)?"

"É mais fácil expandir funcionalidade de software programado em PON (a partir de software X acrescentar requisito Y)?"

5.4.3 PON em inovações para Sistemas Auto Adaptativos

Por conta da articulação dos conceitos de Registro, Recipiente de Escopo Léxico, Independência, Estado Nomeado e Não-determinismo observável, associando as características próprias do paradigma, as análise dos casos de estudo deste trabalho e demais pesquisas no âmbito do PON como [Banaszewski, 2009][Ronszcka, 2012], se identificam as seguinte características marcantes do Paradigma Orientado a Notificações (PON):

- Unidades computacionais de pequeno porte (tão pequenas quanto podem ser), estado mantido em cada unidade computacional de pequeno porte (*local cell*) e conhecimento expresso em fatos e regras (característica análoga aos SBRs);
- Reorganização natural – pelos elos desacoplantes (exclusivamente associações por agregação) que ligam todas as partes construídas em PON, é possível rearranjá-las dinamicamente de maneira simples e inata ao paradigma (*e.g.* seria possível um *FBE* ceder alguns de seus *Atributtes* e respectivos *Methods* para outro *FBE* em tempo de execução).
- Recomposição natural – pelos elos desacoplantes (exclusivamente associações por agregação) que ligam todas as partes construídas em PON, é possível rearranjar seus recipientes de escopo léxico (*closures*) dinamicamente de maneira simples e inata ao paradigma. Exemplos: *Premises* podem ser removidas de *Conditions* e novas *Instigations* podem ser acrescentada a uma *Action* em tempo de execução.
- Plasticidade (elasticidade e reposicionamento natural) – pelos elos desacoplantes (exclusivamente associações por agregação) que ligam todas as partes construídas em PON, é possível rearranjá-las em *software* ou *hardware*. A elasticidade é a capacidade

de se distribuir cada unidade computacional em diferentes componentes (*e.g. software* ou *hardware*) ou centralizar em um único componente, análoga e complementar ao reposicionamento natural, na qual qualquer unidade computacional pode ser realocada em qualquer componente conforme a necessidade ou projeto, inclusive dinamicamente. Um exemplo de uso seria executar o reposicionamento (e demonstrando a elasticidade) quando em baixa demanda de processamento todos as unidades computacionais em PON estão centralizadas em um único processador, e quando em alta demanda de processamento cada tipo de unidade computacional PON poderia ser executada em processadores exclusivos.

Essas características marcantes elencadas são propostas como hipóteses de trabalho futuro, pois nesse contexto a suposta exclusividade em PON e a intrínseca naturalidade (*i.e.* característica inata) devem ser analisadas, debatidas, modeladas, testadas e comprovadas, pois outros paradigmas também podem realizar essas características.

5.4.4 PON versus Máquinas de Estado

A hipótese levantada seria: tal qual uma Rede de Petri consegue condensar em si várias Máquinas de Estados, o PON teria a propriedade de condensar aplicações POE, como exemplo, notadamente das técnicas Padrão de Projeto *State* e Máquinas de Estado hierárquicas (*e.g. SWITCH-CASE aninhados*)?

Referências Bibliográficas

[Amagbégnon *et al.*, 1995] Amagbegnon, P., Besnard, L., And Le Guernic, P. "Implementation of the data-flow synchronous language SIGNAL." In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation* (PLDI '95). ACM, New York, NY, USA, 163-173. DOI=10.1145/207110.207134 http://doi.acm.org/10.1145/207110.207134

[Bainomugisha *et al.*, 2013] Bainomugisha, E., Lombide Carreton, A., Van Cutsem, T., Mostinckx, S., De Meuter, W. 2013. "A Survey on reactive programming". ACM Comput. Surv. 45, 4, Article 52 (August 2013), 34 pages. DOI: http://dx.doi.org/10.1145/2501654.2501666

[Banaszewski *et al.*, 2007] Banaszewski, R. F.; Stadzisz, P. C.; Tacla, C. A.; Simão, J. M. "Notification Oriented Paradigm (NOP): A *software* development approach based on artificial intelligence concepts," in Proceedings of the VI Congress of LAPTEC, Santos, Brazil, 2007.

[Banaszewski, 2009] Banaszewski, R. F. "Paradigma orientado a notificações: avanços e comparações". Dissertação de Mestrado - CPGEI/UTFPR, Curitiba-PR, Brasil, 2009 - Disponível em: http://files.dirppg.ct.utfpr.edu.br/cpgei/Ano_2009/dissertacoes/Dissertacao_500_2009.pdf Acessado em: 01/06/2014.

[Batista, 2013] Batista, M. V. Proposta de um Método de Aplicação da Teoria de Projeto Axiomático ao Desenvolvimento de *Software* PON-POR. Dissertação de Mestrado - CPGEI/UTFPR, Curitiba-PR, Brasil, 2013 - Disponível em: http://repositorio.utfpr.edu.br/jspui/handle/1/613. Acessado em: 01/06/2014.

[Batista *et al.*, 2011] Batista, M. V.; Banaszewski, R. F.; Ronszcka, A. F.; Valença, G. Z.; Linhares, R. R.; Stadzisz, P. C.; Tacla, C. A.; Simão J. M.; "Uma comparação entre o Paradigma Orientado a Notificações (PON) e o Paradigma Orientado a Objetos (POO) realizado por meio da implementação de um Sistema de Vendas". III Congreso Internacional de Computación y Telecom. COMTEL, Lima, Peru, Outubro de 2011.

[Belmonte *et al.*, 2012] 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. Volume 8, Nº 84, Dezembro/2012.

[Berry e Cosserat, 1984] Berry, B. and Cosserat, L. “The ESTEREL Synchronous Programming Language and its Mathematical Semantics.” In: *Seminar on Concurrency, Carnegie-Mellon University*, Stephen D. Brookes, A. W. Roscoe, and Glynn Winskel (Eds.). Springer-Verlag, London, UK, UK, 389-448. 1984.

[Berry, 1989] Berry, G. “Real Time Programming: Special Purpose or General Purpose Languages.” Sophia-Antipolis: INRIA, 1989. (Research Report 1065).

[Berry e Gonthier, 1992] Berry, G., Gonthier, G. “The Esterel synchronous programming language: design, semantics, implementation.” Science of Computer Programming, Volume 19, Issue 2, November 1992, pp. 87-152, ISSN 0167-6423, http://dx.doi.org/10.1016/0167-6423(92)90005-V.

[Brennan *et al.*, 2010] Brennan, A., Greer, D., McDaid, K. 2010. “Adaptability performance trade-off: a controlled experiment.” In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (ESEM '10). ACM, New York, NY, USA, Article 56. DOI=10.1145/1852786.1852857 http://doi.acm.org/10.1145/1852786.1852857

[Brookshear, 2012] Brookshear, J. G. “Computer Science: An Overview”. 11o ed. Addison-Wesley.

[Buschmann *et al.*, 1996] Buschmann , F., Meunier , R., Rohnert, H., Sornmerlad , P. and Stal, M. “Pattern-Oriented *Software* Architecture: A System of Patterns”. 1996 by John'Wiley & Sons Ltd.

[Caspi *et al.*, 1987] Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J. A. “LUSTRE: a declarative language for real-time programming.” In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (POPL '87). ACM, New York, NY, USA, 178-188. 1987. DOI=10.1145/41625.41641 http://doi.acm.org/10.1145/41625.41641

[Chidamber e Kemerer, 1994] Chidamber, S.R.; Kemerer, C.F., "A metrics suite for object oriented design," *Software Engineering, IEEE Transactions on*, vol.20, no.6, pp.476,493, Jun 1994doi: 10.1109/32.295895 URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=295895&isnumber=7320

[Cohen e Kalleberg, 2008] Cohen, N. H., Kalleberg, K. T. “EventScript: an event-processing language based on regular expressions with actions.” In: Proceedings of the 2008 ACM SIGPLAN- SIGBED conference on Languages, compilers, and tools for embedded systems. pp. 111–120. LCTES ’08, ACM, New York, NY, USA (2008) . http://doi.acm.org/10.1145/1133373.1133410

[Cooper, 2008] Cooper, G. H. “Integrating dataflow evaluation into a practical higher-order call-by-value language.” Ph.D. thesis, Brown University, Providence, RI, USA. 2008. http://cs.brown.edu/~greg/thesis.pdf

[Czaplicki, 2012] Czaplicki, E. “Elm: Concurrent FRP for Functional GUIs .” Havard Phd Thesis, Advisor: Stephen Chong. 30 March 2012. Disponível em: <http://www.seas.harvard.edu/sites/default/files/files/archived/Czaplicki.pdf>

[Czaplicki e Chong, 2013] Czaplicki, E. e Chong, S. “Asynchronous Functional Reactive Programming for GUIs.” Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. Jun, 2013. pages 411-422, ACM Press, New York, NY, USA. Disponível em: <http://people.seas.harvard.edu/~chong/pubs/pldi13-elm.pdf>

[Dabek *et al.*, 2002] Dabek, F., Zeldovich, N., Kaashoek, F., Mazières, D., Morris, R. “Event-driven programming for robust *software*”. In: Proceedings of the 10th workshop on ACM SIGOPS. European workshop. pp. 186–189. EW 10 (2002). http://doi.acm.org/10.1145/1133373.1133410

[Desai *et al.* 2012] Desai, A., Gupta, V., Jackson, E., Qadeer, S., Rajamani, S. and Zufferey, D. “P: Safe Asynchronous Event-Driven Programming” Microsoft Research. Technical Report MSR-TR-2012-116. November, 2012. Disponível em http://research.microsoft.com/pubs/177118/tr.pdf Acessado em 09/09/2013.

[Dunkels *et al.*, 2006] Dunkels, A. Schmidt, O. Voigt, T. and Ali, M. 2006. “Protothreads: simplifying event-driven programming of memory-constrained embedded systems”. In *Proceedings of the 4th international conference on Embedded networked sensor systems* (SenSys '06). ACM, New York, NY, USA, 29-42. DOI=10.1145/1182807.1182811 http://doi.acm.org/10.1145/1182807.1182811

[Edwards, 2009] Edwards, J. 2009. “Coherent reaction”. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications* (OOPSLA '09). ACM, New York, NY, USA, 925-932. DOI=10.1145/1639950.1640058 http://doi.acm.org/10.1145/1639950.1640058

[Elliott e Hudak, 1997] Elliott, C. Hudak, P. “Functional reactive animation.” *SIGPLAN Not.* 32, 8. August, 1997. 263-273. DOI=10.1145/258949.258973 http://doi.acm.org/10.1145/258949.258973

[Etzion e Niblett, 2011] Etzion, O., Niblett, P. “Event Processing in Action”. Manning Publications Co. 2011.

[Eugster *et al.*, 2003] Eugster, P. T. Felber, P. A. Guerraoui, R. Kermarrec, A. “The many faces of publish/subscribe.” *ACM Comput. Surv.* 35, 2 (June 2003), 114-131. DOI=10.1145/857076.857078 http://doi.acm.org/10.1145/857076.857078

[Faison, 1993] Faison, T. “Object-Oriented State Machines”. *Software* Development Magazine, Sept, 1993. http://www.faisoncomputing.com/publications/articles/OOStateMachines.pdf

[Faison, 2006] Faison, T. “Event-Based Programming: Taking Events to the Limit”. Apress, Berkely, CA, USA (2006).

[Ferg, 2006] Ferg, S. “Event-Driven Programming: Introduction, Tutorial, History.” 2006. Disponível em: http://sourceforge.net/projects/eventdrivenpgm/files/event_driven_ programming .pdf Acessado em 07/09/2013.

[Ferreira *et al.*, 2013] Ferreira, C. A., Ronszcka, A. F., Ioris, P. A. M, Kossoski, C. “Compilador para o Paradigma Orientado a Notificações.” Relatório Técnico da Disciplina Compiladores PPGCA/UTFPR, 2013.

[Fick e Makinson, 1971] Fick, Bruce R. ; Makinson, John B. “Hardiman I Prototype for Machine Augmentation of Human Strength and Endurance” Corporate Author: GENERAL ELECTRIC CO SCHENECTADY NY SPECIALTY MATERIALS HANDLING PRODUCTS OPERATION; http://www.dtic.mil/get-tr-doc/pdf?AD=AD0739735 Report Date: 31 AUG 1971.

[Forgy, 1982] Forgy, Charles L. “Rete: A fast algorithm for the many pattern/many object pattern match problem”. Artificial Intelligence. Volume 19. Issue 1. Pages 17-37. Elsevier. DOI: 10.1016/0004-3702(82)90020-0.

[Gabbrielli e Martini, 2010] Gabbrielli, M., Martini, S. “Programming Languages: Principles and Paradigms. Series: Undergraduate Topics in Computer Science”. 1st Edition, 2010, XIX, 440 p., Softcover.

[Gamma *et al.*, 1995] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. “Padrões de Projeto: Soluções Reutilizáveis de *Software* Orientado a Objetos.” Porto Alegre, Editora Bookman, 2000. Tradução do original de 1995.

[Gasiunas *et al.*, 2011] Gasiunas, V., Satabin, L., Mezini, M., Núñez, A., Noyé, J. “EScala: modular event-driven object interactions in scala.” In: *Proceedings of the tenth international conference on Aspect-oriented software development* (AOSD '11). ACM, New York, NY, USA, 227-240, 2011. http://doi.acm.org/10.1145/1960275.1960303

[Halbwachs *et al.*, 1991] Halbwachs, N.; Caspi, P.; Raymond, P.; Pilaud, D., "The synchronous data flow programming language LUSTRE," Proceedings of the IEEE, vol.79, no.9, pp.1305,1320, Sep 1991. doi: 10.1109/5.97300

[Hansen e Fossum, 2004] Stuart Hansen and Timothy Fossum. 2004. “Events not equal to GUIs”. In Proceedings of the 35th SIGCSE technical symposium on Computer science education (SIGCSE '04). ACM, New York, NY, USA, 378-381. DOI=10.1145/971300.971430 http://doi.acm.org/10.1145/971300.971430

[Hansen e Fossum, 2010] Hansen, S., Fossum , T. V. “Event Based Programming”. Kenosha WI, May 23, 2010. Disponível em: http://www.cs.uwp.edu/staff/hansen/EventsWWW/ e http://www.lulu.com/shop/stuart-hansen/event-driven-programming/ebook/product-17346555.html

[Harel e Pnueli, 1985] Harel, D.; Pnueli, A. “On the Development of Reactive Systems.” In: APT, K. R. (Ed.) Logics and Models of Concurrent Systems. Berlin: Springer-erlag, 1985. (NATO Series, v.F13).

[Harel, 1987] Harel, D. “Statecharts: a visual formalism for complex systems.” Science of Computer Programming, Volume 8, Issue 3, June 1987, Pages 231-274, ISSN 0167-6423, http://dx.doi.org/10.1016/0167-6423(87)90035-9.

[Harel e Politi, 1998] Harel, D. e Politi, M. 1998. “Modeling Reactive Systems with Statecharts: The Statemate Approach.” 1st Ed. McGraw-Hill, New York.

[Holzer *et al.*, 2011] Holzer, A., Ziarek, L., Jayaram, K., Eugster, P. “Putting events in context: aspects for event-based distributed programming.” In: Proceedings of the tenth international conference on Aspect-oriented *software* development. pp. 241–252. AOSD ’11, ACM, New York, NY, USA (2011) . http://doi.acm.org/10.1145/1133373.1133410

[Hughes, 1989] Hughes, J. “Why Functional Programming Matters”. In: Computer Journal, Vol. 32, No. 2, 1989, and in Research Topics in Functional Programming, ed. David Turner, Addison Wesley, 1990. doi:10.1093/comjnl/32.2.98

[Izidoro e Quináia, 2014] Izidoro, V. N. L., Quináia, M. A. "Design Patterns Application in Holonic Control Notification Mechanism," Latin America Transactions, IEEE (Revista IEEE America Latina), vol.12, no.2, pp. 262-268, March 2014 doi: 10.1109/TLA.2014.6749547

[Jordan *et al.*, 2014] Jordan, H., Goetz, G., Noll, J., Butterfield, A., Collier, R. “A feature model of actor, agent, functional, object, and procedural programming languages”. Science of Computer Programming. Available online 20 February 2014. ISSN 0167-6423. http://dx.doi.org/10.1016/j.scico.2014.02.009.

[Kalkman, 1995] Kalkman, C. “Labview: A *software* system for data acquisition, data analysis, and instrument control. ” Journal of Clinical Monitoring. Comput. 11, 1, 51–58. 1995.
<http://www.biomedsearch.com/nih/LabVIEW-*software*-system-data-acquisition/7745456.html>

[Kerievsky, 2004] Kerievsky, J. “Refactoring to Patterns”. Addison-Wesley Professional; 1 edition (August 15, 2004) ISBN-10: 0321213351 ISBN-13: 978-0321213358

[Kossoski et al., 2014] Kossoski, C., Stadzisz, P. C., Simão J. M. “Introdução ao teste funcional de software no Paradigma Orientado a Notificações .” VI Congresso Intern. de Computación y Telecom. - COMTEL, Lima, Peru, 2014.

[Le, 2012] Le, T.G. INI Online (2012), https://sites.google.com/site/inilanguage/

[Le *et al.*, 2012] Le, T.G., Hermant, O., Manceny, M., Pawlak, R., Rioboo, R. “Unifying event-based and rule-based styles to develop concurrent and context-aware reactive applications -toward a convenient support for concurrent and reactive programming.” In: Proceedings of the 7th International Conference on *Software* Paradigm Trends, Rome, Italy, 24 - 27 July, 2012. pp. 347–350 (2012)

[Le *et al.*, 2013] Le, T.G., Fedosov, D., Hermant, O., Manceny, M., Pawlak, R. and Rioboo, R. “Programming Robots with Events”. Embedded Systems: Design, Analysis and Verification. IFIP Advances in Information and Communication Technology Volume 403. pp 14-25. Springer Berlin Heidelberg http://dx.doi.org/10.1007/978-3-642-38853-8_2

[Le Guernic *et al.*, 1986] Le Guernic, P.; Benveniste, A.; Bournai, P.; Gautier, T., "Signal--A data flow-oriented language for signal processing." Acoustics, Speech and Signal Processing, IEEE Transactions on, vol. 34, no. 2, pp. 362-374, Apr. 1986, doi: 10.1109/TASSP.1986.1164809

[Librelotto, 2001] Librelotto, G. R. “Um Compilador para a linguagem RS distribuída.” Dissertação de Mestrado, Orientador: Prof. Dr. Simao Sirineo Toscani, Universidade Federal do Rio Grande do Sul. Instituto de Informática. Programa de Pós-Graduação em Computação. 2001. Disponível em: http://hdl.handle.net/10183/25064

[Linhares *et al.*, 2011] Linhares, R. R.; Ronszcka, A. F. ; Valença, G. Z.; Batista, M. V.; Erig Lima, C. R.; Witt, F. A.; Stadzisz, P. C.; Simão J. M.; “Comparações entre o Paradigma Orientado a Objetos e o Paradigma Orientado a Notificações sob o contexto de um simulador de sistema telefônico”. III Congresso Intern. de Computación y Telecom. - COMTEL, Lima, Peru, Outubro de 2011.

[Linhares *et al.*, 2014] Linhares, R. R., Simão, J. M.; Stadzisz, P. C. “PEDIDO DE PATENTE: Arquitetura de Computador Orientada a Notificações - ARQPON.” 2014, Brasil. Patente: Privilégio de Inovação. Número do registro: BR1020140040706, data de depósito: 21/02/2014, título: "PEDIDO DE PATENTE: Arquitetura de Computador Orientada a Notificações - ARQPON" , Instituição de registro:INPI - Instituto Nacional da Propriedade Industrial. Instituição(ões) financiadora(s): UTFPR.

[Maier e Odersky, 2012] Maier, I., Odersky, M. “Deprecating the observer pattern with Scala.React.” Technical Report. Programming Methods Laboratory (LAMP), *École* Polytechnique Fédérale de Lausanne (EPFL). 2012. Disponível em: <http://infoscience.epfl.ch/record/176887/files/DeprecatingObservers2012.pdf> Acessado em 07/09/2014.

[McKellar, 2012] McKellar, J. “Twisted”. From: “The Architecture of Open Source Applications, Volume II”. Edited by Amy Brown and Greg Wilson. Creative Commons Attribution-Noncommercial 3.0 Unported. Disponível em http://aosabook.org/en/index.html. Acessado em 07/09/2014.

[Melo, 2013] Melo, L. C. V. “Relatório da adaptação do Paradigma Orientado a Notificações - PON para suporte a desenvolvimento de sistemas de lógica fuzzy ”. Programa de Pós-Graduação em Computação Aplicada (PPGCA) Universidade Tecnológica Federal do Paraná (UTFPR) , 2013.

[Miller *et al.*, 2005] Miller, M., Tribble, D., Shapiro, J. 2005. “Concurrency among strangers: Programming in E as plan coordination.” In Proceedings of the Symposium on Trustworthy Global Computing. Lecture Notes in Computer Science, vol. 3705, Springer, 195–229.

[Mühl *et al.*, 2006] Mühl, G., Fiege, L. and Pietzuch, P. “*Distributed Event-Based Systems”* Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Peters, 2012] Peters, E. “Coprocessador para aceleração de aplicações desenvolvidas utilizando paradigma orientado a notificações”. 2012. 94 f. Dissertação (Mestrado em Engenharia Elétrica e Informática Industrial) – Universidade Tecnológica Federal do Paraná, Curitiba, 2012.

[Pucella, 1998] Pucella, R.R., "Reactive programming in Standard ML," Computer Languages, 1998. Proceedings. 1998 International Conference on , vol., no., pp. 48,57, 14-16 May 1998. doi: 10.1109/ICCL.1998.674156

[Ronszcka *et al.*, 2011] Ronszcka, A. F.; Belmonte, D. L.; Valença, G. Z.; Batista, M. V.; Linhares, R. R.; Tacla, C. A.; Stadzisz, P. C.; Simão, J. M.; “Comparações quantitativas e qualitativas entre o Paradigma Orientado a Objetos e o Paradigma Orientado a Notificações sobre um simulador de jogo”. III Congresso Intern. de Computación y Telecom. - COMTEL, Lima, Peru, Outubro de 2011.

[Ronszcka, 2012] Ronszcka, A. F. “Contribuição para a concepção de aplicações no paradigma orientado a notificações (PON) sob o viés de padrões”. Dissertação de Mestrado - CPGEI/UTFPR, Curitiba-PR, Brasil, 2012 - Disponível em: http://files.dirppg.ct.utfpr.edu.br/cpgei/Ano_2012/dissertacoes/CPGEI_Dissertacao_608_2012.pdf. Acessado em: 01/06/2014.

[Rumbaught *et al.*, 1999] Rumbaught, J., Jacobson, I. and Booch, G. “The Unified Modeling Language–Reference Manual”. Addison Wesley.

[Salvaneschi e Mezini, 2014] Salvaneschi, G., Mezini, *M. "Towards Reactive Programming for Object-oriented Applications."* Transactions on Aspect-Oriented *Software* Development, Springer Berlin Heidelberg. 2014. DOI 10.1007/978-3-642-55099-7_7

[Salvaneschi *et al.*, 2014] Salvaneschi, G., Hintz, G., Mezini, M. " REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications". Modularity '14 - 13th International Conference on Modularity – 2014. ACM Press.

[Salvaneschi *et al.*, 2014b] Salvaneschi, G., Amann, S., Proksch, S., Mezini, M. "*An Empirical Study on Program Comprehension with Reactive Programming". TU Darmstadt, Germany.* FSE'14.

[Samek, 2008] Samek, M. "Practical UML Statecharts in C/C++ Event-Driven Programming for Embedded Systems". ISBN: 987-0-7506-8706-5. Newnes, Elsevier, October, 2008.

[Sant'Anna *et al.*, 2012] Sant'Anna, F., Rodriguez, N., Ierusalimschy, R. "Céu: Embedded, Safe, and Reactive Programming." Relatório Técnico, PUC-Rio, 2012, número 12/12, ISSN 0103-9741, Disponível em: ftp://ftp.inf.puc-rio.br/pub/docs/techreports/12_12_santanna.pdf. Acessado em: 01/06/2014.

[Sant'Anna *et al.*, 2013] Sant'Anna, F., Rodriguez, N., Ierusalimschy, R., Landsiedel, O., Tsigas, P. 2013. "Safe system-level concurrency on resource-constrained nodes". In Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems (SenSys '13). ACM, New York, NY, USA, , Article 11 , 14 pages. Disponível em: http://doi.acm.org/10.1145/2517351.2517360. Acessado em: 01/06/2014.

[Schmidt *et al.*, 2000] Schmidt, D., Stal, M., Rohnert, H. and Buschmann , F. "Pattern-Oriented *Software* Architecture, Patterns for Concurrent and Networked Objects Volume 2 ". John Wiley & Sons. 2000.

[Scott, 2008] Scott, M. L. "Programming Language Pragmatics", 3o Edition, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc, 2008.

[Sebesta, 2012] Sebesta, R. W. "Concepts of programming languages" 10th ed. University of Colorado at Colorado Springs . Pearson. 2012.

[Simão e Stadzisz, 2008] Simão, J. M.; Stadzisz, P. C. “Paradigma Orientado a Notificações (PON) – Uma Técnica de Composição e Execução de *Software* Orientado a Notificações”. Pedido de Patente junto ao INPI/Brazil em 2008 e a Agência de Inovação/UTFPR em 2007. No INPI Efetivo PI0805518-1.

[Simão e Stadzisz, 2009] Simão, J. M.; Stadzisz, P. C. “Inference Process Based on Notifications: The Kernel of a Holonic Inference Meta-Model Applied to Control Issues”. IEEE Trans. on Systems, Man and Cybernetics. Part A, Systems and Humans, V.39, I.1, 238-250, doi:10.1109/TSMCA.2008.20066371, 2009.

[Simão *et al.*, 2010] Simão, J. M.; Banaszewski, R. F. ; Tacla, C. A.; Stadzisz, P. C. “PEDIDO DE PATENTE: 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, Brasil. Patente: Privilégio de Inovação. Número do registro: PI10037365, data de depósito: 25/03/2010, Instituição de registro:INPI - Instituto Nacional da Propriedade Industrial. Instituição(ões) financiadora(s): Universidade Tecnológica Federal do Paraná.

[Simão e Stadzisz, 2010] Simão, J. M.; Stadzisz, P. C. “Pedido de patente: Mecanismo de Resolução de Conflito e Garantia de Determinismo para o Paradigma Orientado a Notificações (PON).” 2010, Brasil. Patente: Privilégio de Inovação. Número do registro: PI10002960, data de depósito: 26/02/2010, Instituição de registro:INPI - Instituto Nacional da Propriedade Industrial. Instituição(ões) financiadora(s): Universidade Tecnológica Federal do Paraná.

[Simão *et al.*, 2012a] Simão, J. M.; Banaszewski, R. F.; Tacla, C. A.; Stadzisz, P. C.; "Notification Oriented Paradigm (NOP) and Imperative Paradigm: A Comparative Study," Journal of *Software* Engineering and Applications – JSEA, Vol. 5 No. 6, 2012, pp. 402-416.

[Simão *et al.*, 2012b] Simão, J. M.; Belmonte, D. L.; Ronszcka, A. F.; Linhares, R. R.; Valença, G. Z.; Banaszewski, R. F.; Fabro, J. A; Tacla, C. A.; Stadzisz, P. C.; Batista, M. V; "Notification Oriented and Object Oriented Paradigm comparison via Sale System". Journal of *Software* Engineering and Applications – JSEA, Vol. 5 No. 9, 2012, pp. 695-710.

[Simão *et al.*, 2012c] Simão, J. M.; Belmonte, D. L.; Linhares, R. R.; Banaszewski, R. F.; Tacla, C. A.; Stadzisz, P. C. "Comparações entre duas materializações do Paradigma Orientado a Notificações (PON): *Framework* PON Prototipal versus *Framework* PON Primário ". IV Congresso Intern. de Computación y Telecom. - COMTEL, Lima, Peru, 2012.

[Simão *et al.*, 2014] Simão, J., Renaux, D. Linhares, R., Stadzisz, P. "Evaluation of the Notification Oriented Paradigm applied to Sentient Computing" . Conference: IEEE 17th International Symposium on Object/Component-Oriented Real-Time Distributed Computing, At Reno - USA. DOI: 10.1109/ISORC.2014.54

[Stroustrup, 1996] Stroustrup, B. 1996. "A history of C++: 1979-1991". In *History of programming languages II*, Thomas J. Bergin, Jr. and Richard G. Gibson, Jr. (Eds.). ACM, New York, NY, USA 699-769. DOI=10.1145/234286.1057836 http://dx.doi.org/10.1145/234286.1057836

[Toscani, 1993] Toscani, S. "RS: Uma Linguagem para a Programação de Núcleos Reactivos." Lisboa: Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia. 1993. Tese de Doutorado. Disponível em: http://biblioteca.iscte.pt/resumosindicesteses/informatica/ 01000019740.pdf. Acessado em: 01/06/2014.

[Turing, 1937] *Turing, A. M. "*On Computable Numbers, with an Application to the Entscheidungsproblem." *Proceedings of the London Mathematical Society. (1937) s2-42 (1): 230-265.*

[Valença *et al.*, 2011] Valença, G. Z.; Banaszewski, R. F.; Ronszcka, A. F.; Batista, M. V.; Linhares, R. R.; Fabro, J. A.; Stadzisz, P. C.; Simão, J. M.; "*Framework* PON, Avanços e Comparações". III Simpósio de Computação Aplicada, Passo Fundo - RS, Brasil, 2011.

[Valença, 2012] Valença, G. Z. "Contribuição para materialização do paradigma orientado a notificações (PON) via *Framework* e wizard". Dissertação de Mestrado - PPGCA/UTFPR, Curitiba-PR, Brasil, 2012- Disponível em: http://repositorio.utfpr.edu.br/jspui/handle/1/393. Acessado em: 01/06/2014.

[Van Roy, 1990] Van Roy, P. L. "Can Logic Programming Execute as Fast as Imperative Programming?" Ph.D. in Computer Science, UC Berkeley, December 1990. http://www.info.ucl.ac.be/~pvr/Peter.thesis/Peter.thesis.html

[Van Roy e Harefidi, 2004] Van Roy, P, Haridi, S. “Concepts, Techniques, and Models of Computer Programming”. MIT Press, 2004.

[Van Roy, 2009] Van Roy, P. “Programming Paradigms for Dummies: What Every Programmer Should Know .” In: New Computational Paradigms for Computer Music. p. 9-47. G. Assayag and A. Gerzso (eds.), IRCAM/Delatour France, 2009.

[Watt, 2004] Watt, D. A. “Programming Language Design Concepts”. J. Willey & Sons, 2004.

[Weiser 1991] Weiser, M. “The computer for the 21st century”. Scientific American, vol 265, no 3, pp 94-104, 1991. Disponível em <http://cs.lth.se/english/sde/phd_courses/ pervasive_systems/literature/>. Acessado em: 19/03/2014.

[Wiecheteck, 2011] Wiecheteck L. V. B.; “Método para Projeto de *Software* usando o Paradigma Orientado a Notificações – PON”. Dissertação M. Sc., CPGEI/UTFPR Curitiba-PR Brasil, 2011.

[Wiecheteck, Stadzisz e Simão, 2011] Wiecheteck, L. V. B.; Stadzisz, P. C.; Simão, J. M.; “Um Perfil UML para o Paradigma Orientado a Notificações (PON)”. III COMTEL, Lima, Peru, Outubro de 2011.

[Witt *et al.*, 2011] Witt, F. A. De; Simão, J. M.; Linhares, R. R.; Stadzisz, P. C.; Lima, C. R. E. “Comparação entre o Paradigma Orientado a Objetos (POO) e o Paradigma Orientado a Notificações (PON) em um Controle Discreto em Lógica Reconfigurável”. XVI Seminário de Iniciação Científica e Tecnológica da UTFPR. Ponta Grossa-PR Brasil, Setembro 2011.

[Xavier *et al.*, 2014] Xavier, R. D.; Fabro, J. A.; Stadzisz, P. C.; Simão, J. M. “Paradigmas de desenvolvimento de *software*: comparação entre abordagens orientada a eventos e orientada a notificações”. Revista SODEBRAS Publicação: Volume 9, Nº 101, Maio/2014. ISSN - 1809-3957.

Apêndice A - Gráficos de desempenho

Neste Apêndice, seguem os gráficos de desempenho dos experimentos da seção 4.3 Comparações em medidas de execução de software em PON e POE deste trabalho.

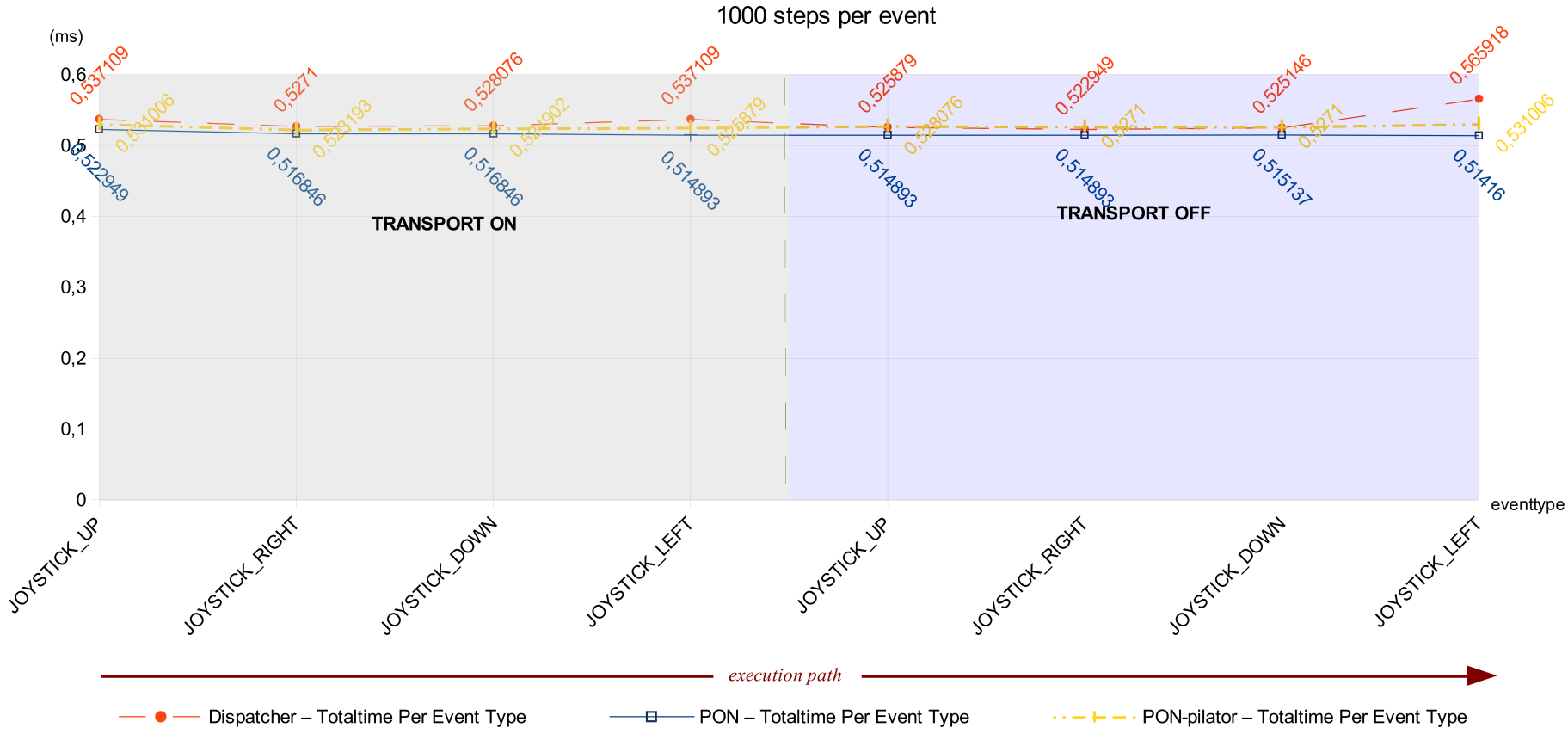

Figura 64: Gráfico com tempos de resposta (ms) por tipo de evento POE x PON durante a execução na simulação no primeiro cenário para 1.000 n-*steps*.

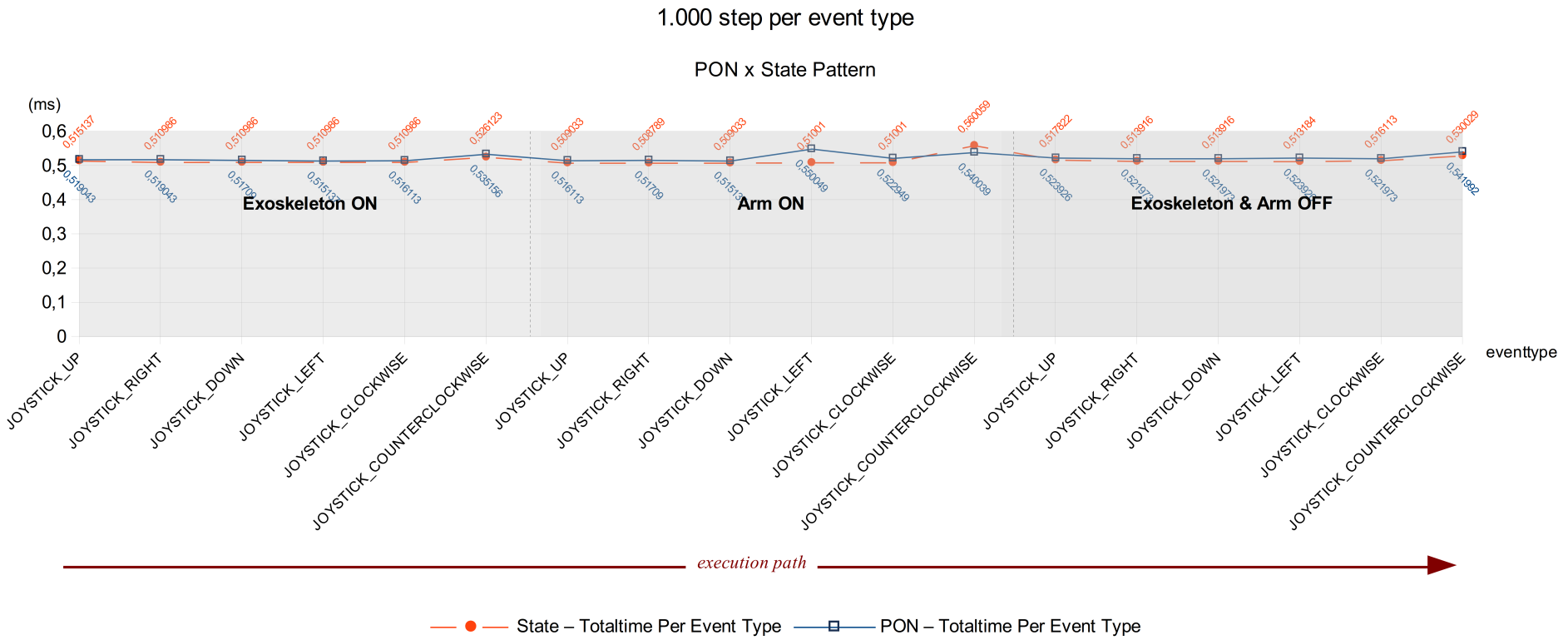

Figura 65: Gráfico com tempos de resposta (ms) por tipo de evento POE x PON durante a execução na simulação no segundo cenário para 1.000 n-*steps*.

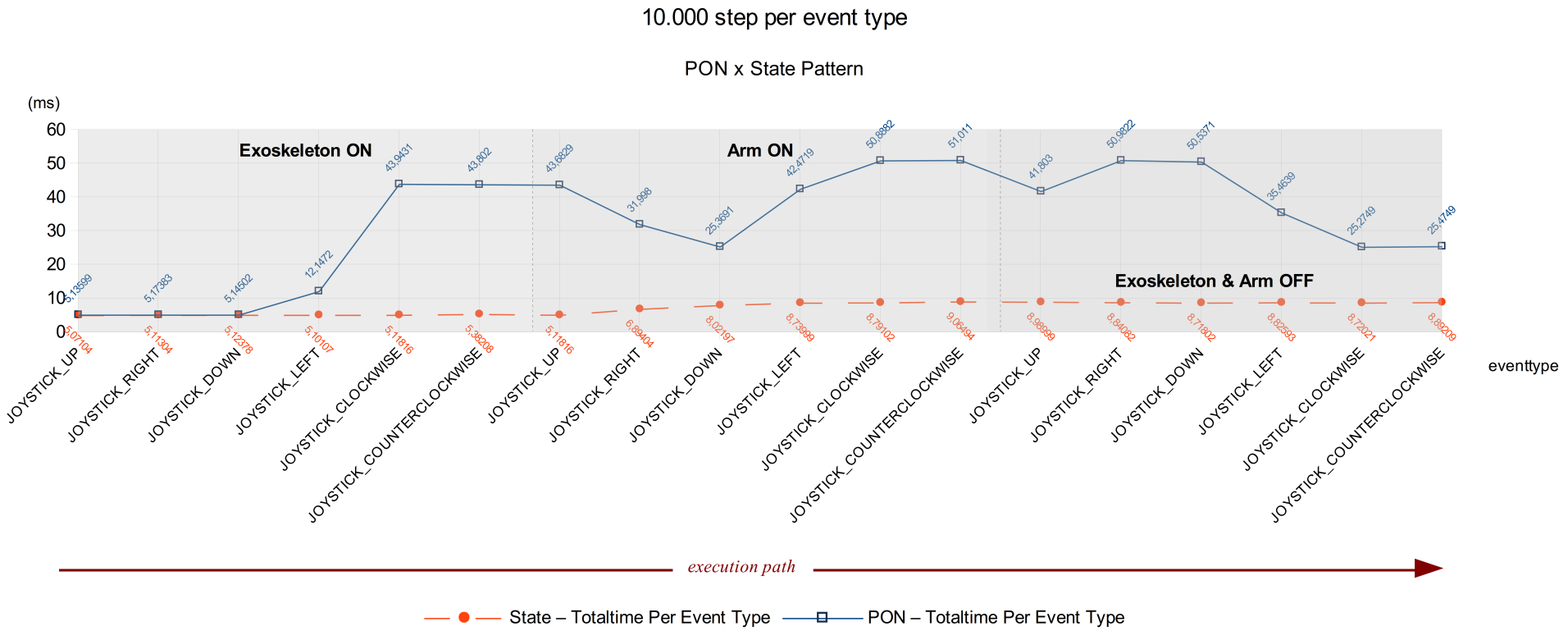

Figura 66: Gráfico com tempos de resposta (ms) por tipo de evento POE x PON durante a execução na simulação no segundo cenário para 10.000 n-*steps*.

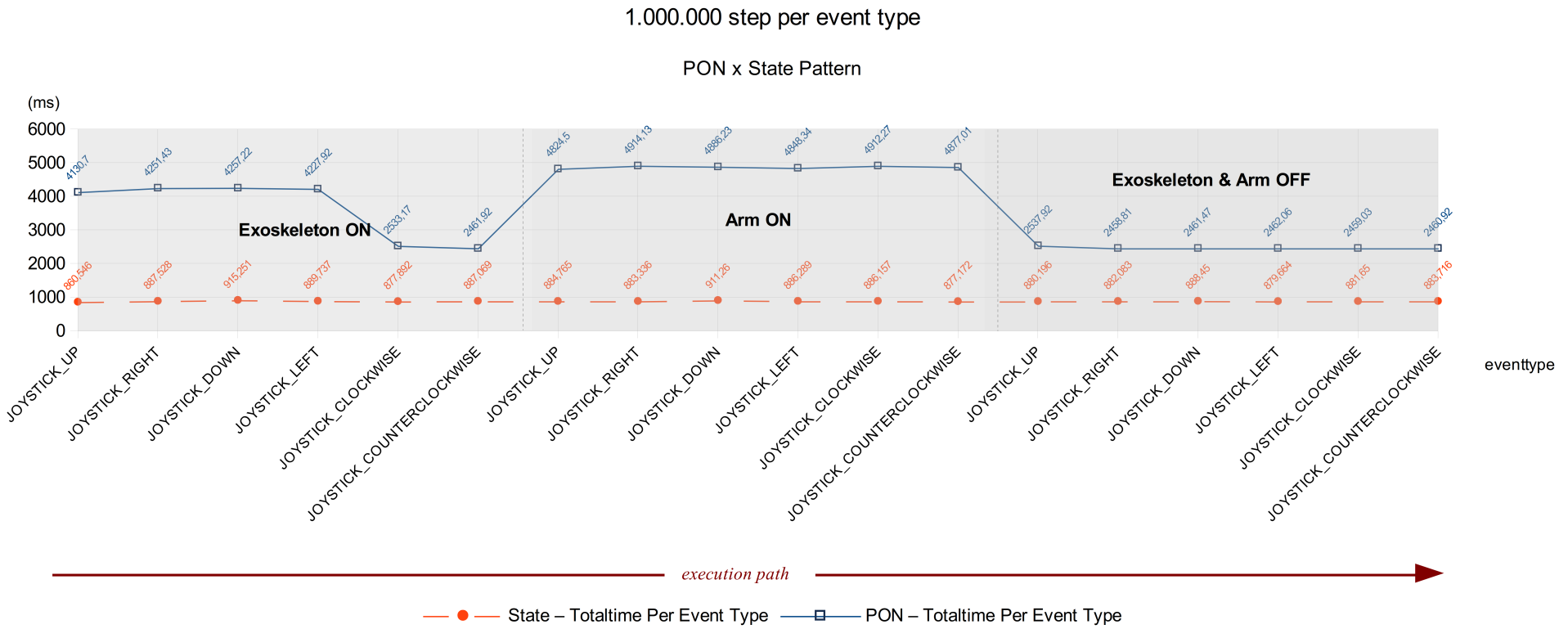

Figura 67: Gráfico com tempos de resposta (ms) por tipo de evento POE x PON durante a execução na simulação no segundo cenário para 1.000.000 n-*steps*.

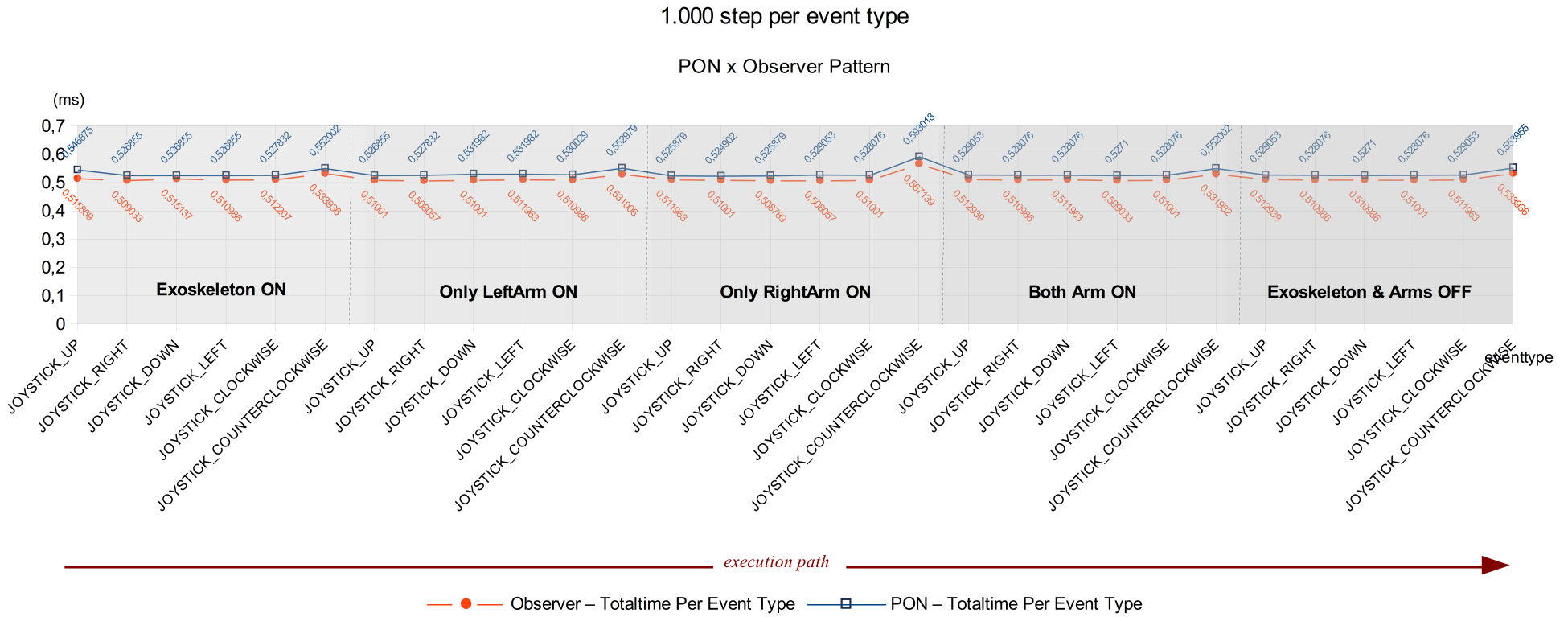

Figura 68: Gráfico com tempos de resposta (ms) por tipo de evento POE x PON durante a execução na simulação no terceiro cenário para 1.000 n-*steps*.

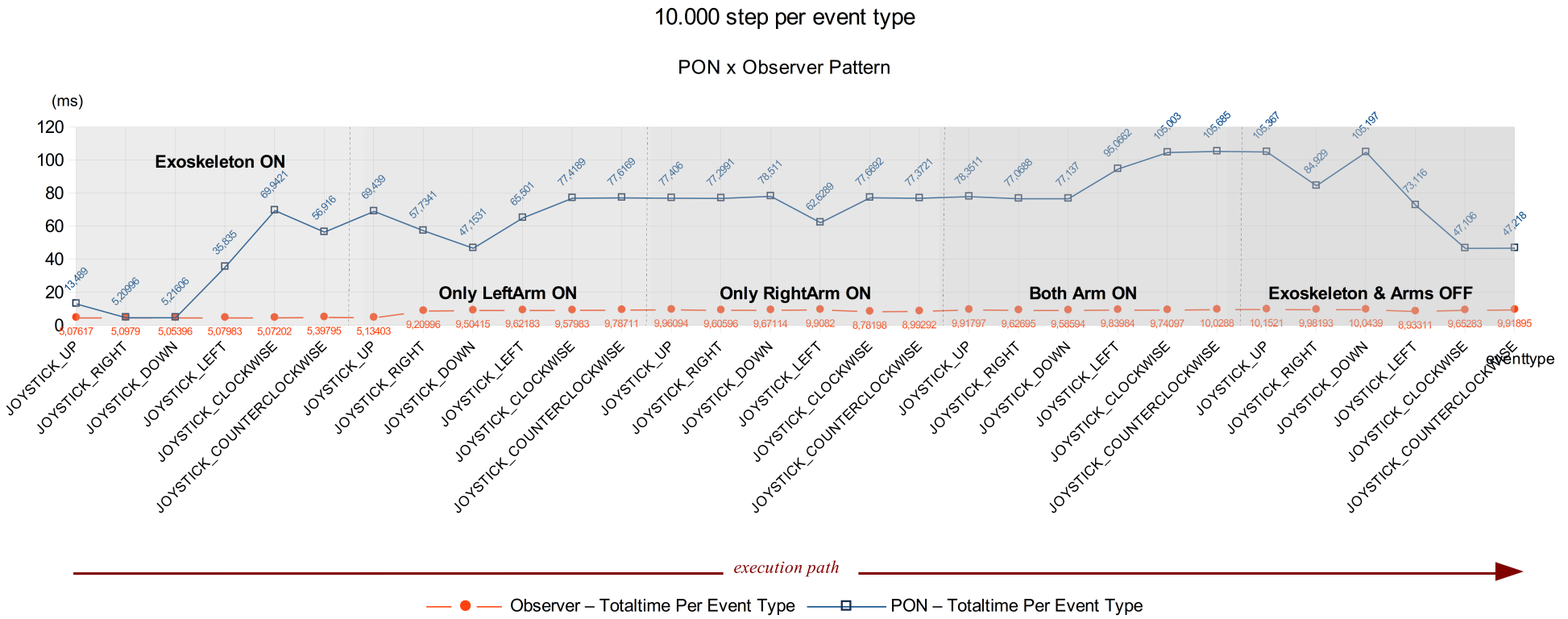

Figura 69: Gráfico com tempos de resposta (ms) por tipo de evento POE x PON durante a execução na simulação no terceiro cenário para 10.000 n-*steps*.

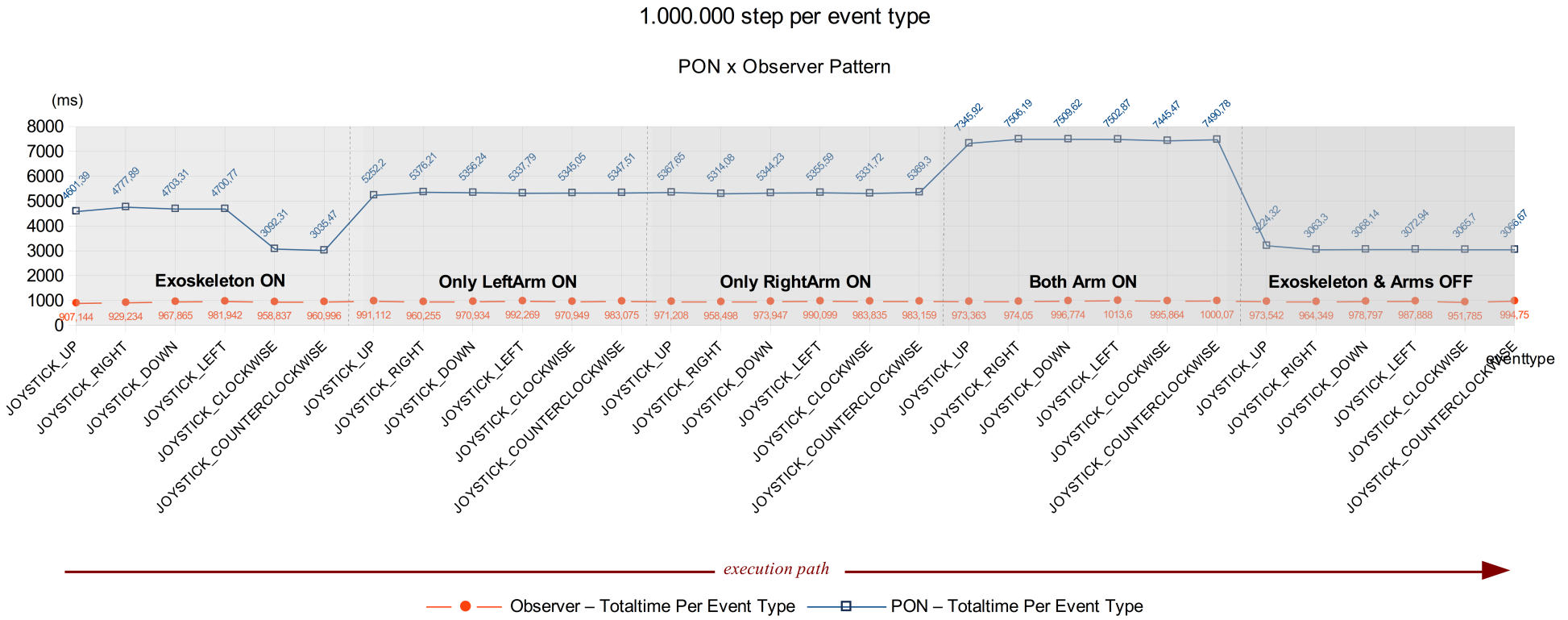

Figura 70: Gráfico com tempos de resposta (ms) por tipo de evento POE x PON durante a execução na simulação no terceiro cenário para 1.000.000 n-*steps*.

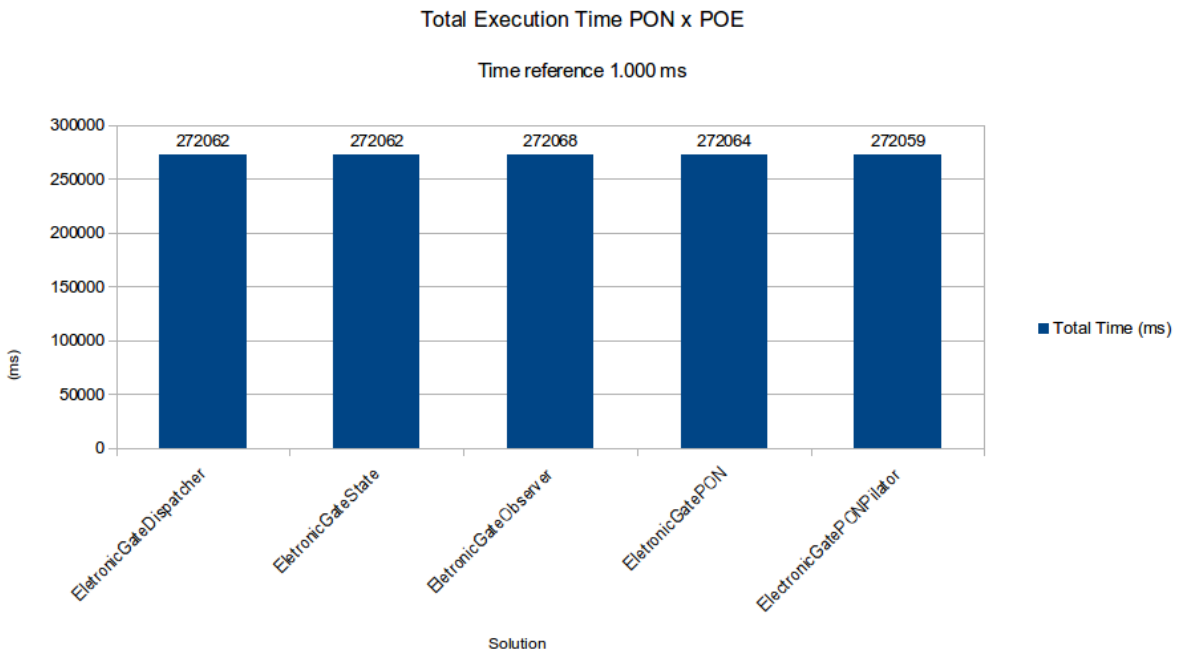

Figura 71: Gráfico de tempo de execução do caso de estudo Portão Eletrônico com tempo de referência 1,000ms.

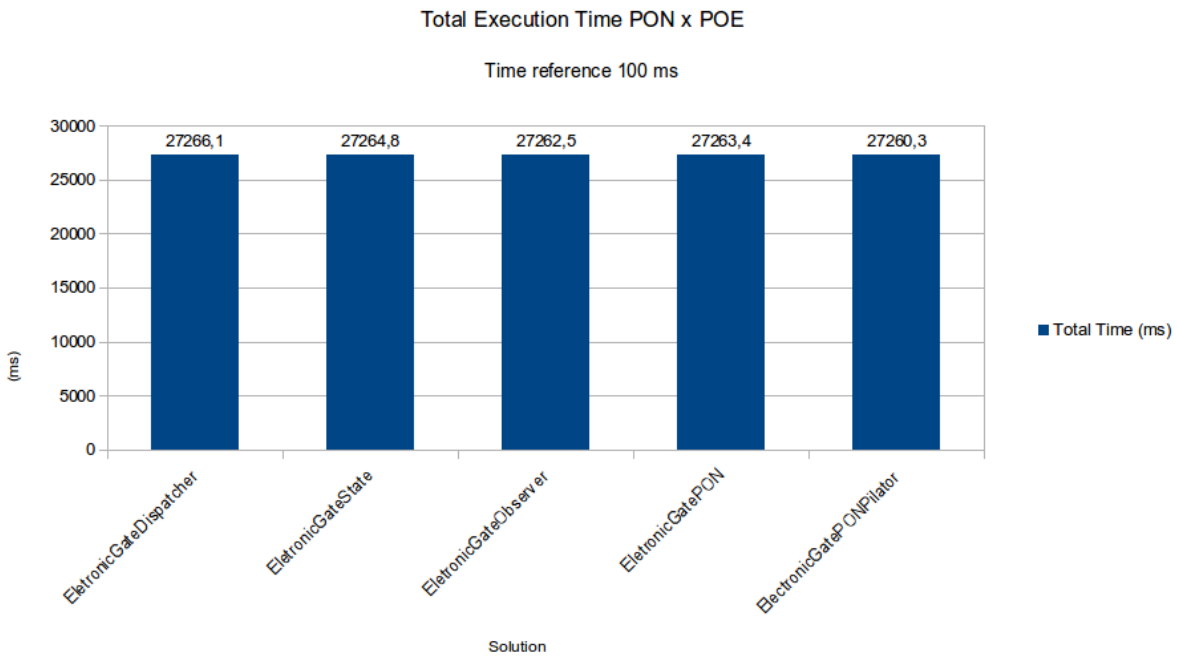

Figura 72: Gráfico de tempo de execução do caso de estudo Portão Eletrônico com tempo de referência 100ms.

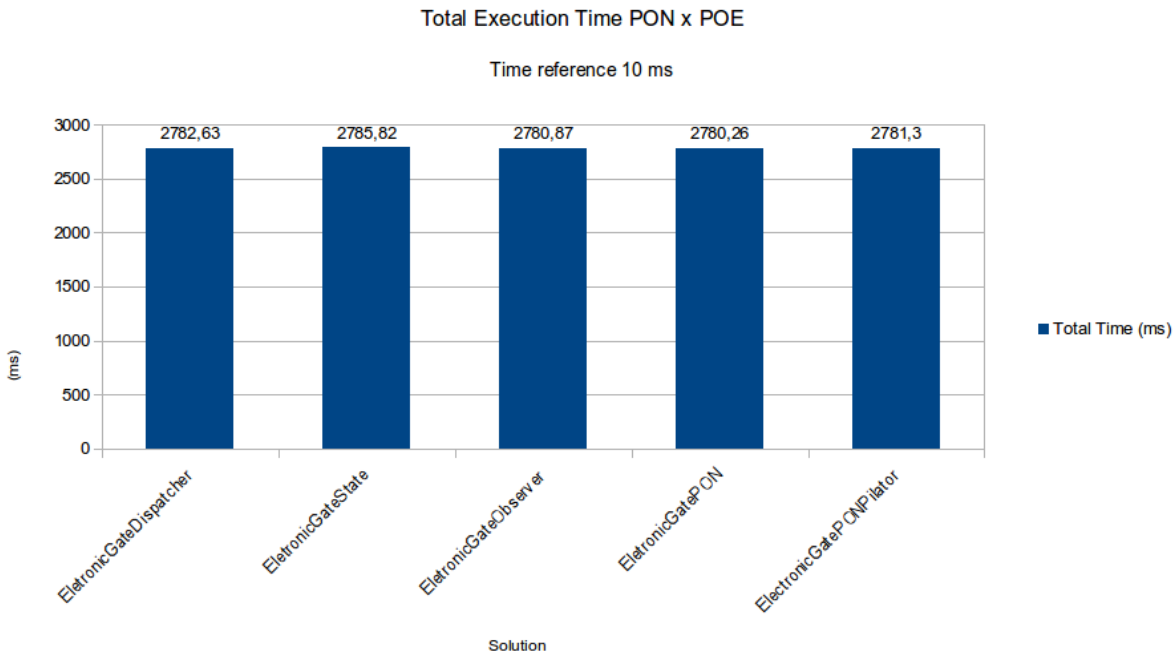

Figura 73: Gráfico de tempo de execução do caso de estudo Portão Eletrônico com tempo de referência 10ms.

Apêndice B - Dados da instrumentação básica da plataforma dos experimentos

Para compor a instrumentação, ligando o *software* gerador de eventos com o *software* tratador de eventos, foram utilizados *Linux Pipes* (representado e executado pelo símbolo de barra vertical "|") e redirecionamento de saída para arquivo para persistir os resultados (representado e executado pelo símbolo relacional de maior que ">").

Genericamente, o uso de Pipes funciona da seguinte maneira:

```
$ comando1 parametro1 | comando 2 | comando 3 > arquivoresultado.txt
```

Resumidamente, um "comando1" (iniciado com "parametro1"), gera saídas para o "comando2", que gera saídas para o "comando3", que por fim escreve sua saída (*stream*) em um arquivo nomeado "arquivoresultado.txt" Cada comando é um *software*.

Um exemplo deste experimento do comando de shell script (*software* roteiro - *script* - de linha de comando) para a medição é:

```
$ ./ScriptSimple 100000 1 0 | ./SimpleEventHandlerSystemPON  >
SimpleEventHandlerSystemPON100000.txt
```

O *software ScriptSimple* gerador de eventos, com parâmetros (n-passos = 100000, número de execuções = 1, roteiro de eventos = 0), gera eventos para o *software* SimpleEventHandlerSystemPON, e o resultado é persistido no arquivo texto SimpleEventHandlerSystemPON100000.txt.

Todos os códigos-fonte (arquivos de *shell script*, *software* gerador de eventos e *software* tratador de eventos objeto de estudo em PON e POE) encontram-se disponíveis no repositório de versionamento compartilhado do grupo de pesquisa do PON.

Além, por conta das dificuldades de medição em execução de *software* (e de anomalias encontradas conforme dados apresentados), também foram utilizadas ferramentas de profiling C++ do próprio Linux, como *time*, *valgrind*, *perf stat*, *gprof*, e o próprio depurador (GDB - GNU Debugger) do GCC – compilador C++ .

Apêndice C - Investigação executada para identificação da anomalia/irregularidade

Este apêndice tem objetivo registrar as tentativas executadas para elucidar os motivos da anomalia que ocorre a partir de *n-steps* com *n* igual a 10.000. Ou seja, tempo de resposta irregular a partir da geração de 10k eventos.

(I) Investigação em ponto de vista do *Framework*:

- Mudança de todos os atributos NOP_INTEGER de *double* para o tipo primitivo *int* C++ (equivalente ao *long int*);
- Retirada de todos os acessórios de impressão em tela (std::iostream::cout);
- Retirada de todas as classes acessórias ou não utilizadas;
- Depuração minuciosa;

A investigação não logrou sucesso, apenas pequena diminuição no tempo de execução.

(II) Visão do *software* (objeto de estudo – primeiro caso de estudo):

- Retirada de todas as impressão em tela (std::iostream::cout);
- Uso de diferentes escalonadores, até ficar sem resolução de conflito ao utilizar o escalonador NO_ONE;
- Uso das estruturas de dados NOP_VECTOR e NOP_HASH;
- Depuração minuciosa;

A investigação não logrou sucesso, apenas ínfima diminuição no tempo de execução.

(III) Visão de ambiente operacional (bancada de teste):

- Ambiente livre de preempções ou interrupções de S.O.;
- Somente ferramentas e processos de S.O. essenciais rodando;
- Desligamento de *hardware* (mouse, touchpad, vídeo);

- Monitoramento de SO de tempo *idle*;
- Carregamento do *software* várias vezes antes da medição;
- Emulação de disco em memória (RAMDISK) para minorar ou anular tempo de acesso a disco (tempo de I/O);
- Experimentos com mudança no número de *n*-passos;
- Indícios de ocorrência do comportamento irregular a partir do número 32K eventos gerados. A irregularidade na realidade se inicia em 32K e não 10K eventos;
- *Profiling* de *software* C++ com ferramental específico;
- Provável correlação do número 32K eventos com memória cache L1 do processador;
- Realizado testes de experimento em outra máquina;
- Realizado experimento em processador com outra quantidade de memória cache;
- Não realizado experimento em máquina que seja possível desligar a memória cache para todos os *software*s

Apêndice D - Ferramentas Utilizadas

Ambiente de Implantação (*deploy*) – Linux Ubuntu 10.04

IDE de desenvolvimento - Eclipse

IDE de modelagem – Visual Paradigm

Controle de Versão - git

Ferramentas de C++ *profiling* – valgrind, perf stat, time, gmon, gprof

Redação da documentação - LibreOffice

Anexo A - Especificação Formal da Linguagem PON

A especificação da LingPON que consta no relatório técnico [Ferreira *et al.*, 2013].

A) Símbolos Terminais

```
RULE = “rule”;
CONDITION = “condition”;
ACTION = “action”;
PREMISE = “premise”;
INSTIGATION = “instigation”;
SUBCONDITION = “subcondition”;
FBE = “fbe”;
ATTRIBUTES = “attributes”;
METHODS = “methods”;
METHOD = “method”;
INST = “inst”;

END_RULE = “end_rule”;
END_CONDITION = “end_condition”;
END_ACTION = “end_action”;
END_SUBCONDITION = “end_subcondition”;
END_FBE = “end_fbe”;
END_ATTRIBUTES = “end_attributes”;
END_METHODS = “end_methods”;
END_INST = “end_inst”;

INTEGER = “integer”;
BOOLEAN = “boolean”;
REAL = “real”;
STRING = “string”;

AND = “and”;
OR = “or”;
TRUE = “true”;
FALSE = “false”;

LP = “(”;
RP = “)”;
LB = “{”;
RB = “}”;
LC = “[”;
RC = “]”;

ASSIGN = “=”;

EQ = “==”;
NE = “!=”;
```

```
      LT = "<";
      GT = ">";
      LE = "<=";
      GE = ">=";

      SEMICOLON = ";";
      COMMA = ",";

      PLUS = "+";
      MINUS = "-";
      MULT = "*";
      DIV = "/";

      POINT = ".";

      NUMBER = "0..9";
```

B) Símbolos não Terminais

```
PROGRAM     =
      fbes
      inst
      rules

inst  =
      INST declarations END_INST

declarations =
      declaration
      declaration declarations

declaration =
      type ids

ids =
      id
      id COMMA ids

rules =
      rule
      rule rules

rule =
      RULE rule_body END_RULE
      RULE id rule_body END_RULE

rule_body =  decl_condition decl_action

decl_condition =
      CONDITION condition_body END_CONDITION
      CONDITION id condition_body END_CONDITION

condition_body =
      subcondition operator condition_body
      subcondition

operator  =
      AND
      OR
```

```
subcondition =
      SUBCONDITION subcondition_body END_SUBCONDITION
      SUBCONDITION id subcondition_body END_SUBCONDITION
      subcondition_body

subcondition_body =
      premise AND subcondition_body
      premise

premise  =
      PREMISE exp
      PREMISE id exp

exp   =
      fator comp fator

comp  =
      EQ
      NE
      LT
      GT
      LE
      GE

fator =
      id
      NUMBER
      boolean

boolean =
      TRUE
      FALSE

decl_action =
      ACTION action_body END_ACTION
      ACTION id action_body END_ACTION

action_body =
      action_elements action_body
      action_elements

action_elements   =
      instigation
      method_use
      exp SEMICOLON

instigation =
      INSTIGATION method_use
      INSTIGATION id method_use

method_use  =
      id LP RP SEMICOLON

id    =
      ID
      ID POINT ID

fbes  =
      fbe
      fbe fbes
```

```
fbe   =
      FBE fbe_body END_FBE
      FBE id fbe_body END_FBE

fbe_body    =
      decl_attributes decl_methods

decl_attributes         =
      ATTRIBUTES attributes END_ATTRIBUTES

attributes  =
      attributes_body
      attributes_body attributes

attributes_body   =
      type id value

type  =
      BOOLEAN
      INTEGER
      REAL
      STRING
      id

value =
      NUMBER
      boolean
      id

decl_methods =
      METHODS methods END_METHODS

methods
      method_body
      method_body methods

method_body       =
      METHOD id LP id ASSIGN value RP
```