

**UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO APLICADA**

FELIPE DOS SANTOS NEVES

***FRAMEWORK PON C++ 4.0:
CONTRIBUIÇÃO PARA A CONCEPÇÃO DE APLICAÇÕES NO
PARADIGMA ORIENTADO A NOTIFICAÇÕES POR MEIO DE
PROGRAMAÇÃO GENÉRICA***

DISSERTAÇÃO

**CURITIBA
2021**

FELIPE DOS SANTOS NEVES

***FRAMEWORK PON C++ 4.0:
CONTRIBUIÇÃO PARA A CONCEPÇÃO DE APLICAÇÕES NO
PARADIGMA ORIENTADO A NOTIFICAÇÕES POR MEIO DE
PROGRAMAÇÃO GENÉRICA***

**Framework NOP 4.0:
Contribution to the development of applications in the Notification
Oriented Paradigm through Generic Programming**

Dissertação apresentado(a) como requisito para obtenção do título(grau) de Mestre em Computação Aplicada, do Programa de Pós-Graduação em Computação Aplicada, da Universidade Tecnológica Federal do Paraná (UTFPR).

Orientador: Prof. Dr. Robson Ribeiro Linhares

Coorientador: Prof. Dr. Jean Marcelo Simão

CURITIBA

2021



[4.0 Internacional](#)

Esta licença permite compartilhamento, remixe, adaptação e criação a partir do trabalho, mesmo para fins comerciais, desde que sejam atribuídos créditos ao(s) autor(es).

Conteúdos elaborados por terceiros, citados e referenciados nesta obra não são cobertos pela licença.



FELIPE DOS SANTOS NEVES

FRAMEWORK PON C++ 4.0: CONTRIBUIÇÃO PARA A CONCEPÇÃO DE APLICAÇÕES NO PARADIGMA ORIENTADO A NOTIFICAÇÕES POR MEIO DE PROGRAMAÇÃO GENÉRICA

Trabalho de pesquisa de mestrado apresentado como requisito para obtenção do título de Mestre Em Computação Aplicada da Universidade Tecnológica Federal do Paraná (UTFPR). Área de concentração: Engenharia De Sistemas Computacionais.

Data de aprovação: 03 de Setembro de 2021

Prof Adriano Francisco Ronszcka, Doutorado - Sem Vinculo Oficial

Prof Fernando Schutz, - Universidade Tecnológica Federal do Paraná

Prof Herve Panetto, Doutorado - Universite de Lorraine

Prof Jean Marcelo Simao, Doutorado - Universidade Tecnológica Federal do Paraná

Prof Joao Alberto Fabro, Doutorado - Universidade Tecnológica Federal do Paraná

Documento gerado pelo Sistema Acadêmico da UTFPR a partir dos dados da Ata de Defesa em 03/09/2021.

Dedico esse trabalho aos meus pais, que me
deram a dádiva da educação, sem o qual nada
seria possível.

AGRADECIMENTOS

Sou eternamente grato a toda a minha família, especialmente aos meus pais, Fernando e Soraya, que acreditaram sempre no meu potencial e me apoiam em todas as minhas decisões. Agradeço também a minha esposa Thaís, por todo o amor, carinho e principalmente paciência durante os períodos em que mais precisei.

Agradeço também a todos os meus professores e colegas de trabalho que tiveram um impacto na minha carreira, em particular a Felipe Calliari e Luiz Duma que também acreditaram em meu potencial e me deram a oportunidade de iniciar minha carreira como desenvolvedor de *software*.

Agradeço aos professores orientadores Dr. Jean Marcelo Simão e Dr. Robson Ribeiro Linhares por todo o apoio, dedicação e paciência durante o desenvolvimento deste trabalho. Por fim, agradeço também aos membros da banca Dr. João Alberto Fabro, Dr. Fernando Schütz, Dr. Adriano Francisco Ronszcka e Dr. Hervé Panetto, por disponibilizar seu valioso tempo na avaliação deste trabalho.

É imensurável a minha gratidão a todas as pessoas que me ajudaram nessa jornada. Deixo registrado também meu agradecimento a todos aqueles que, de alguma forma ou outra, contribuíram para a realização deste trabalho.

*“Each of us is carving a stone,
erecting a column, or cutting
a piece of stained glass
in the construction of something
much bigger than ourselves.”*

(Adrienne Clarkson)

*“Cada um de nós está a esculpir
uma pedra, a erguer uma coluna,
ou a cortar um pedaço de vidro
manchado na construção de algo
muito maior do que nós próprios.”*

(Adrienne Clarkson)

RESUMO

NEVES, Felipe dos Santos. **Framework PON C++ 4.0: Contribuição para a concepção de aplicações no Paradigma Orientado a Notificações por meio de Programação Genérica.** 2021. 330 f. Dissertação (Mestrado em Computação Aplicada) – Universidade Tecnológica Federal do Paraná. Curitiba, 2021.

O Paradigma Orientado a Notificações (PON) é uma nova abordagem para a construção de sistemas computacionais. O PON propõe a computação por meio de um modelo de entidades reativas desacopladas que interagem por meio de notificações pontuais, dentre as quais se divide e separa a computação facto-execucional da computação lógico-causal. Com isso é possível reduzir ou eliminar redundâncias temporais e estruturais, comuns em outros paradigmas de programação, que podem afetar o desempenho dos programas. Ainda, o desacoplamento intrínseco entre as entidades do PON facilita a construção de sistemas concorrentes e/ou distribuídos. Além disso, a estrutura orientada a regras do PON tende a facilitar o desenvolvimento por permitir programar em alto nível de abstração. O PON apresenta várias materializações em *software*, sendo as mais maduras tecnologicamente aquelas que se dão por meio de *frameworks*, desenvolvidos em diferentes linguagens de programação. Dentre estes *frameworks* o que apresenta o maior grau de maturidade e estabilidade é o *Framework PON C++ 2.0*. Entretanto, o *Framework PON C++ 2.0* ainda apresenta certas limitações, como excessiva verbosidade, baixa flexibilidade de tipos e baixa flexibilidade algorítmica. Nesse contexto este trabalho propõe o desenvolvimento de um novo *framework*, denominado *Framework PON C++ 4.0*, com o objetivo de remover as limitações presentes no *Framework PON C++ 2.0*, bem como as imperfeições do *Framework PON C++ 3.0* que envolve *multithread/multicore*, de forma a melhorar a usabilidade do PON e seu desempenho neste âmbito. O *Framework PON C++ 4.0* é desenvolvido utilizando técnicas de programação genérica, por meio de recursos adicionados nas versões do padrão ISO C++11, C++14, C++17 e C++20, bem como aplicando o método de desenvolvimento orientado a testes. Esta dissertação de mestrado apresenta os resultados obtidos com a implementação do *Framework PON C++ 4.0* por meio de um conjunto de aplicações pertinentes, tanto em ambiente *single thread* quanto *multithread/multicore*. Tais aplicações são um sistema de sensores e uma aplicação de controle automatizado de tráfego, oriundos do grupo de pesquisa, e dos algoritmos *Bitonic Sort* e *Random Forest* oriundos da literatura. Tais aplicações foram executadas e comparadas em termos de desempenho com as mesmas aplicações implementadas no *Framework PON C++ 2.0*, *Framework PON Elixir/Erlang* e também implementações no Paradigma Orientado a Objetos (POO) em linguagem de programação C++ e Paradigma Procedimental (PP) em linguagem de programação C. Como resultado destas comparações, o novo *Framework PON C++ 4.0* se mostra superior ao *Framework PON C++ 2.0* tanto em tempo de execução como consumo de memória nos cenários avaliados, além de apresentar balanceamento de carga comparável aos do *Framework PON Elixir/Erlang* em ambiente *multicore*. As melhorias na usabilidade são adicionalmente avaliadas e atestadas pelo *feedback* de desenvolvedores do PON.

Palavras-chave: Paradigma Orientado a Notificações. *Framework PON C++ 4.0*. Programação Genérica. C++ Moderno. Desenvolvimento Orientado a Testes.

ABSTRACT

NEVES, Felipe dos Santos. **Framework NOP 4.0: Contribution to the development of applications in the Notification Oriented Paradigm through Generic Programming.** 2021. 330 p. Dissertation (Master's Degree in Applied Computing) – Universidade Tecnológica Federal do Paraná. Curitiba, 2021.

The Notification-Oriented Paradigm (NOP) is a new approach to the construction of computer systems. The NOP proposes computability by means of reactive and decoupled entities model that interact by means of punctual notifications, separating fact-executional from logic-causal computing. With this it is possible to reduce or eliminate temporal and structural redundancies, common in other programming paradigms, which can affect program performance. Still, the intrinsic decoupling between NOP entities facilitates the construction of concurrent and/or distributed systems. Moreover, the rule-oriented structure of the NOP tends to ease development by allowing programming at a high level of abstraction. NOP presents several materializations in software, being the most mature technologically those that occur through frameworks, developed in different programming languages. Among these frameworks, the one that presents the highest degree of maturity and stability is the C++ Framework NOP 2.0. However, the C++ Framework NOP 2.0 still has certain limitations, such as excessive verbosity, low type flexibility and low algorithmic flexibility. In this context, this work proposes the development of a new framework, named C++ Framework NOP 4.0, with the objective of removing the limitations present in the C++ Framework NOP 2.0, as well as the imperfections of the C++ Framework NOP 3.0 that involves multithread/multicore, in order to improve the usability of the NOP and its performance in this regard. The C++ Framework NOP 4.0 is developed using generic techniques, by means of features added in the ISO C++11 C++14, C++17 and C++20 and applying the test-driven development methodology. This master's thesis presents the results obtained with the implementation of the C++ Framework NOP 4.0 through a set of relevant applications, such as the sensor application, traffic light control application, and the algorithms Bitonic Sort and Random Forest, both in a single thread and multithread/multicore environment. These applications were executed and compared in terms of performance against the same applications implemented with the C++ Framework NOP 2.0, Elixir/Erlang Framework NOP and also implementations in the Object-Oriented Paradigm (OOP) in the C++ programming language and Procedural Paradigm (PP) in the C programming language. As a result of these comparisons, the new C++ Framework NOP 4.0 proves to be superior to C++ Framework NOP 2.0 in both runtime and memory consumption in the evaluated scenarios, besides presenting CPU utilization levels comparable to the Framework Elixir/Erlang Framework NOP multicore environment. Usability improvements are additionally evaluated and attested by feedback from NOP developers.

Keywords: Notification-Oriented Paradigm. C++ Framework NOP 4.0. Generic Programming. Modern C++. Test Driven Development.

LISTA DE ILUSTRAÇÕES

Figura 1 – Classificação dos paradigmas de programação	18
Figura 2 – Taxonomia dos paradigmas e linguagens	19
Figura 3 – Linguagens, paradigmas e conceitos	20
Figura 4 – Classificação dos paradigmas de programação	22
Figura 5 – Evolução dos paradigmas de programação	23
Figura 6 – Estrutura de <i>Rule</i> no PON	24
Figura 7 – Diagrama de classes das entidades do metamodelo do PON	26
Figura 8 – Representação do fluxo de notificações do PON	27
Figura 9 – Fluxo de desenvolvimento no TDD	33
Figura 10 – Classificação dos paradigmas de programação com os paradigmas emergentes	43
Figura 11 – Conhecimento representado em regras no PL	50
Figura 12 – Arquitetura interna de um Sistema Baseado em Regras	50
Figura 13 – Estrutura do algoritmo RETE	51
Figura 14 – Diagrama de componentes	53
Figura 15 – Processo de detecção de eventos	55
Figura 16 – Interação entre as entidades do PON e ciclo de notificações	61
Figura 17 – Cálculo assintótico do mecanismo de notificações	63
Figura 18 – Notificações baseadas em lista encadeada e tabela <i>hash</i>	64
Figura 19 – Diagrama de atividades do processo de renotificação	65
Figura 20 – <i>Rule</i> que depende do mecanismo de renotificações	66
Figura 21 – Alterações de estado com <i>Attribute</i> impertinente ativo	67
Figura 22 – Alterações de estado com <i>Attribute</i> impertinente desativado	68
Figura 23 – Alterações de estado com <i>Attribute</i> impertinente reativado	68
Figura 24 – Declaração de <i>Premises</i> do exemplo do alarme	70
Figura 25 – Declaração de <i>Rules</i> utilizando <i>Premises</i> compartilhadas	70
Figura 26 – Modelo centralizado de resolução de conflitos na estratégia <i>Breadth</i>	72
Figura 27 – Exemplo de aplicação de <i>Master Rule</i>	73
Figura 28 – Representação de uma <i>Formation Rule</i>	74
Figura 29 – Exemplo de aplicação de <i>FBE Rules</i>	75
Figura 30 – Cenário do Mira ao Alvo	78
Figura 31 – Estrutura do <i>framework C++ 1.0</i>	80
Figura 32 – Comparação do desempenho do <i>framework C++ 1.0</i> com o <i>framework C++ Prototipal</i>	81
Figura 33 – Estrutura do <i>framework C++ 2.0</i>	82
Figura 34 – <i>Wizard PON</i>	83
Figura 35 – Comparação do desempenho do <i>Framework PON C++ 2.0</i> com o <i>Framework PON C++ 1.0</i>	84
Figura 36 – Estrutura do <i>framework C++ 3.0</i>	86
Figura 37 – Diagrama de atividades do controle de entidades do <i>framework C++ 3.0</i>	87
Figura 38 – Visão geral do método <i>LobeNOI</i>	87
Figura 39 – Estrutura do NeuroPON	88
Figura 40 – Taxa de utilização dos núcleos da CPU no treinamento de RNA MLP com método BP	89

Figura 41 – Tempos de execução (em milissegundos) do treinamento de ANN MLP com método BP	90
Figura 42 – Diagrama de classes do JuNOC++	91
Figura 43 – Gráfico dos resultados de experimento com o JuNOC++	92
Figura 44 – Tela principal do ambiente de simulação do futebol de robôs	94
Figura 45 – Diagrama de classes do futebol de robôs	94
Figura 46 – <i>Screenshot</i> do jogo desenvolvido	96
Figura 47 – Diagrama de classes do jogo desenvolvido	97
Figura 48 – Comparação de desempenho das aplicações em C++, Java e C#	101
Figura 49 – Diagrama simplificado de classes do <i>Framework PON C# IoT</i>	102
Figura 50 – Diagrama de atividades do PON C# IoT	103
Figura 51 – Diagrama de componentes PON C# IoT	103
Figura 52 – Resultados de testes do PON C# IoT	105
Figura 53 – Modelagem UML dos elementos do PON enquanto mircro-atores	106
Figura 54 – Detalhamento dos ambientes do experimento do <i>Framework PON Elixir/Erlang</i>	108
Figura 55 – Taxa média de ocupação por núcleo em ambiente VM02	108
Figura 56 – Taxa média de ocupação por núcleo em ambiente VM02	108
Figura 57 – Taxa média de ocupação por núcleo em ambiente VM08	108
Figura 58 – Taxa média de ocupação por núcleo em ambiente VM16	108
Figura 59 – Tempos médios de execução NeuroPON de uma RNA MLP para a função XOR no <i>Framework PON Elixir/Erlang</i>	109
Figura 60 – Estrutura de atores em Akka.NET	110
Figura 61 – Modelo de atores na aplicação do portão eletrônico em Akka.NET	110
Figura 62 – Comparação entre o <i>Framework PON Akka.NET</i> e C++ 3.0	112
Figura 63 – Método MCPON	120
Figura 64 – Estrutura do Grafo PON	121
Figura 65 – Sistema de compilação do PON	122
Figura 66 – Classes de equivalência e análise de valores limite	125
Figura 67 – Fase de testes unitários do PON	126
Figura 68 – Fluxo básico e fluxos alternativos de eventos em um caso de uso	126
Figura 69 – Fase de testes de integração do PON	127
Figura 70 – Gráfico com contagem de artigos publicados a cada ano	129
Figura 71 – Gráfico com contagem de teses e dissertações publicadas a cada ano	129
Figura 72 – Número de páginas nas diferentes versões do padrão ISO C++	130
Figura 73 – Resultado de teste com Google Benchmark	146
Figura 74 – Diagrama de classes do pacote <i>Core</i> do <i>Framework PON C++ 2.0</i>	152
Figura 75 – Diagrama de classes do <i>Framework PON C++ 4.0</i>	153
Figura 76 – Diagrama de classes de <i>logs</i> do <i>Framework PON C++ 4.0</i>	176
Figura 77 – Relatório dos testes do <i>Framework PON C++ 4.0</i>	182
Figura 78 – Testes de desempenho da aplicação do sensor	193
Figura 79 – Tempos de execução da aplicação do sensor com o <i>Framework PON C++ 4.0</i> relativo ao <i>Framework PON C++ 2.0</i>	194
Figura 80 – Consumo de memória POO em C++	194
Figura 81 – Consumo de memória <i>Framework PON C++ 2.0</i>	194
Figura 82 – Consumo de memória <i>Framework PON C++ 4.0</i>	195
Figura 83 – Processo de ordenação com <i>Bitonic Sort</i>	196
Figura 84 – <i>Rules</i> para comparador do <i>Bitonic Sort</i> em PON	197

Figura 85 – <i>Rules</i> para implementação mais eficiente do comparador do <i>Bitonic Sort</i> em PON	198
Figura 86 – Testes de desempenho da aplicação do <i>Bitonic Sort</i> com diferentes implementações em PON	199
Figura 87 – Consumo de memória para a aplicação do algoritmo <i>Bitonic Sort</i> com o <i>Framework PON C++ 4.0</i> na implementação original	200
Figura 88 – Consumo de memória para a aplicação do algoritmo <i>Bitonic Sort</i> com o <i>Framework PON C++ 4.0</i> na implementação em estágios	200
Figura 89 – Testes de desempenho da aplicação do algoritmo <i>Bitonic Sort</i>	200
Figura 90 – Comparação da paralelização na aplicação do algoritmo <i>Bitonic Sort</i>	202
Figura 91 – Tempos de execução do algoritmo <i>Bitonic Sort</i> com o <i>Framework PON C++ 4.0</i> paralelizado relativo ao sequencial	202
Figura 92 – Utilização de CPU durante execução do algoritmo <i>Bitonic Sort</i> com o <i>Framework PON C++ 4.0</i> sequencial	203
Figura 93 – Utilização de CPU durante execução do algoritmo <i>Bitonic Sort</i> com o <i>Framework PON C++ 4.0</i> paralelizado	203
Figura 94 – Árvores de decisão do algoritmo <i>Random Forest</i>	204
Figura 95 – Testes de desempenho da aplicação do algoritmo <i>Random Forest</i>	206
Figura 96 – Comparação da paralelização na aplicação do algoritmo <i>Random Forest</i>	207
Figura 97 – Tempos de execução do algoritmo <i>Random Forest</i> com o <i>Framework PON C++ 4.0</i> paralelizado relativo ao sequencial	207
Figura 98 – Utilização de CPU durante execução do algoritmo <i>Random Forest</i> com o <i>Framework PON C++ 4.0</i> sequencial	208
Figura 99 – Utilização de CPU durante execução do algoritmo <i>Random Forest</i> com o <i>Framework PON C++ 4.0</i> paralelizado	208
Figura 100 – Ambiente de simulação	209
Figura 101 – Estados do CTA com estratégia independente	210
Figura 102 – Estados do CTA com estratégia CBCF	210
Figura 103 – Consumo de memória para a aplicação de semáforo com estratégia independente com o <i>Framework PON C++ 4.0</i>	212
Figura 104 – Consumo de memória para a aplicação de semáforo com estratégia CBCF com o <i>Framework PON C++ 4.0</i>	212
Figura 105 – Uso de CPU durante execução do Semáforo com estratégia independente com o <i>Framework PON C++ 4.0</i>	213
Figura 106 – Uso de CPU durante execução de semáforo com estratégia CBCF com o <i>Framework PON C++ 4.0</i>	213
Figura 107 – Tempo de execução da aplicação de semáforo com o <i>Framework PON C++ 4.0 Framework PON Elixir/Erlang</i>	213
Figura 108 – Diagrama de classes do jogo desenvolvido com o <i>Framework PON C++ 4.0</i>	215
Figura 109 – Resultado da pesquisa de avaliação da melhoria do <i>Framework PON C++ 4.0</i> sobre o <i>Framework PON C++ 2.0</i>	216
Figura 110 – Resultado da pesquisa de avaliação da melhoria do <i>Framework PON C++ 4.0</i> sobre o <i>Framework PON C++ 2.0</i>	217
Figura 111 – Comunicação distribuída com o PON	226
Figura 112 – Aplicação mira ao alvo	297
Figura 113 – Resultado do experimento mira ao alvo com o <i>Framework PON C++ 1.0</i>	298
Figura 114 – Resultado do experimento mira ao alvo com o <i>Framework PON C++ 4.0</i>	299

LISTA DE TABELAS

Tabela 1 – Índice TIOBE de linguagens de programação	21
Tabela 2 – Propriedades do PON materializadas pelos <i>frameworks</i> do PON	38
Tabela 3 – Demandas de desenvolvimento de <i>software</i> atendidas pelos Paradigmas Dominantes	59
Tabela 4 – Resumo de conceitos de programação do PON	76
Tabela 5 – Propriedades elementares contempladas nas materializações do PON	113
Tabela 6 – Conceitos do PON contemplados nas materializações do paradigma	114
Tabela 7 – Caso de teste para <i>Premise</i>	125
Tabela 8 – Problemas a serem endereçados pelo <i>Framework PON C++ 4.0</i>	149
Tabela 9 – Padrões de projeto aplicados no <i>Framework PON C++ 4.0</i>	154
Tabela 10 – Exemplo de <i>logs</i> do <i>Framework PON C++ 4.0</i>	179
Tabela 11 – Objetivos atingidos pelo <i>Framework PON C++ 4.0</i>	183
Tabela 12 – Propriedades elementares contempladas nas materializações do PON	186
Tabela 13 – Conceitos do PON contemplados nas materializações do paradigma - com adição do <i>Framework PON C++ 4.0</i>	188
Tabela 14 – Número de elementos em relação ao número de árvores	205
Tabela 15 – Linhas de código para a composição do jogo NOPUnreal	215
Tabela 16 – Linhas de código para a composição do <i>framework</i>	217

LISTA DE ABREVIATURAS, SIGLAS E ACRÔNIMOS

SUMÁRIO

1	INTRODUÇÃO	16
1.1	CONTEXTUALIZAÇÃO	17
1.1.1	Paradigmas de Programação Dominantes	17
1.1.1.1	Paradigma Dominantes vis-à-vis PI & PD	18
1.1.2	Paradigmas de Programação Emergentes	21
1.1.3	Paradigma Orientado a Notificações — PON	23
1.1.4	Programação Genérica	28
1.1.5	Desenvolvimento Orientado a Testes — <i>Test Driven Development</i> (TDD)	32
1.2	MOTIVAÇÃO	34
1.3	JUSTIFICATIVA	36
1.4	OBJETIVO GERAL	40
1.5	OBJETIVOS ESPECÍFICOS	40
1.6	ORGANIZAÇÃO DO TRABALHO	41
2	REVISÃO DO ESTADO DA ARTE E DA TÉCNICA	42
2.1	PARADIGMAS DE PROGRAMAÇÃO ATUAIS	42
2.1.1	Paradigmas Imperativos	43
2.1.1.1	Paradigma Procedimental (PP)	44
2.1.1.2	Paradigma Orientado a Objetos (POO)	45
2.1.1.3	Considerações Sobre os Paradigmas Imperativos	46
2.1.2	Paradigmas Declarativos	47
2.1.2.1	Paradigma Funcional (PF)	48
2.1.2.2	Paradigma Lógico (PL)	48
2.1.2.3	Considerações Sobre os Paradigmas Declarativos	51
2.1.3	Paradigmas Emergentes	52
2.1.3.1	Paradigma Orientado a Componentes (POC)	52
2.1.3.2	Paradigma Orientado a Eventos (POE)	54
2.1.3.3	Paradigma Orientado a Aspectos (POAs)	56
2.1.3.4	Paradigma Orientado a Agentes (POAg) / Atores (POAt)	56
2.1.4	Considerações Sobre os Paradigmas Dominantes e Emergentes	58
2.2	PARADIGMA ORIENTADO NOTIFICAÇÕES (PON)	60
2.2.1	Constituintes do PON	60
2.2.2	Complexidade temporal	62
2.3	CONCEITOS DE PROGRAMAÇÃO OU DESENVOLVIMENTO EM PON	64
2.3.1	Reatividade das entidades	64
2.3.2	Renotificações	65
2.3.3	<i>Keeper</i>	66
2.3.4	Entidades impertinentes	67
2.3.5	Compartilhamento de entidades	69
2.3.6	Resolução de conflitos	70
2.3.7	<i>Master Rule</i>	72
2.3.8	<i>Formation Rules</i> - Regras de Formação	73
2.3.9	<i>FBE Rules</i>	75

2.3.10	Agregação entre <i>FBEs</i>	75
2.3.11	Resumo dos Conceitos de Programação do PON	76
2.4	MATERIALIZAÇÕES DO PON EM SOFTWARE	77
2.4.1	<i>Frameworks PON C++</i>	77
2.4.1.1	<i>Framework PON C++ Prototíp</i>	77
2.4.1.2	<i>Framework PON C++ 1.0</i>	79
2.4.1.3	<i>Framework PON C++ 2.0</i>	81
2.4.1.4	<i>Framework PON C++ 3.0</i>	84
2.4.1.5	JuNOC++	89
2.4.1.6	Implementações realizadas com o <i>Framework PON C++ 2.0</i>	92
2.4.1.6.1	<i>Futebol de Robôs</i>	93
2.4.1.6.2	<i>Jogo NOPUnreal</i>	95
2.4.1.7	Considerações sobre os <i>frameworks</i> do PON em C++	98
2.4.2	<i>Frameworks PON Java/C# 1.0</i>	99
2.4.2.1	<i>Framework PON Java 1.0</i>	99
2.4.2.2	<i>Framework PON C# 1.0</i>	100
2.4.2.3	Comparações	100
2.4.3	<i>Framework PON C# IoT</i>	101
2.4.4	<i>Framework PON Elixir/Erlang</i>	104
2.4.5	<i>Framework PON Akka.NET</i>	109
2.4.6	Reflexões sobre os <i>frameworks</i> do PON	112
2.4.6.1	<i>Frameworks</i> do PON vis-à-vis sua teoria	112
2.4.6.2	<i>Frameworks</i> do PON enquanto estado da técnica	115
2.4.6.3	Ponderações gerais sobre a pertinência de <i>Frameworks</i> do PON	117
2.4.7	Tecnologia LingPON	119
2.5	MÉTODO DE TESTE DE SOFTWARE PARA O PON	124
2.5.1	Teste unitário em PON	124
2.5.2	Teste de integração em PON	125
2.6	LEVANTAMENTO DOS TRABALHOS REALIZADOS NO PON	127
2.7	LINGUAGEM DE PROGRAMAÇÃO C++ CONTEMPORÂNEA / C++ MODERNO	128
2.7.1	<i>Smart Pointers</i>	130
2.7.2	Expressões <i>lambda</i>	132
2.7.3	<i>Templates</i>	133
2.7.3.1	<i>Variadic templates</i> e <i>Fold expressions</i>	135
2.7.3.2	Expressões constantes	136
2.7.4	Programação concorrente	137
2.7.4.1	Tarefas assíncronas	138
2.7.4.2	Políticas de execução	140
2.7.4.3	Mecanismos de sincronização	141
2.8	TDD E FRAMEWORKS DE TESTE	142
2.8.1	Google Test	144
2.8.2	Google Benchmark	145
2.9	REFLEXÕES SOBRE OS PROBLEMAS EM ABERTO	146
3	FRAMEWORK PON C++ 4.0	149
3.1	ESTRUTURA	149
3.2	MECANISMO DE NOTIFICAÇÕES	154

3.3	<i>ATTRIBUTE</i>	159
3.4	<i>PREMISE</i>	160
3.5	<i>CONDITION</i>	160
3.6	<i>RULE</i>	163
3.7	<i>ACTION, INSTIGATION E METHOD</i>	164
3.8	ESCALONADOR	167
3.9	FACILITADORES DE DESENVOLVIMENTO	170
3.9.1	Abstrações	171
3.9.2	<i>Builders</i> de entidades compartilhadas	172
3.9.3	<i>Logger</i>	176
3.10	OPÇÕES DE COMPILAÇÃO	179
3.11	TESTES UNITÁRIOS E DE INTEGRAÇÃO	180
3.12	REFLEXÕES SOBRE O <i>FRAMEWORK PON C++ 4.0</i>	183
4	EXPERIMENTOS COM O <i>FRAMEWORK PON C++ 4.0</i> E SEUS RESULTADOS	189
4.1	TESTES DE DESEMPENHO	189
4.1.1	Aplicação de sensores	190
4.1.2	Aplicação <i>Bitonic Sort</i>	195
4.1.3	Aplicação <i>Random Forest</i>	203
4.1.4	Aplicação de Controle de Tráfego Automatizado (CTA)	208
4.2	JOGO NOPUNREAL COM <i>FRAMEWORK PON C++ 4.0</i>	214
4.3	PESQUISA DE OPINIÃO DE DESENVOLVEDORES	215
4.4	REFLEXÕES SOBRE OS RESULTADOS OBTIDOS	216
5	CONCLUSÕES E TRABALHOS SUBSEQUENTES	220
5.1	CONCLUSÃO	220
5.2	TRABALHOS SUBSEQUENTES	224
5.2.1	Desenvolvimento de aplicações com o <i>Framework PON C++ 4.0</i>	225
5.2.2	Compilador LingPON com alvo <i>Framework PON C++ 4.0</i>	225
5.2.3	<i>Framework PON C++ 4.0</i> Distribuído	226
5.2.4	NeuroPON	227
5.2.5	Técnicas de depuração em PON	227
	REFERÊNCIAS	229
	APÊNDICES	240

1 INTRODUÇÃO

Este trabalho de mestrado está inserido em um projeto de pesquisa sobre o Paradigma Orientado a Notificações (PON), que é um novo paradigma de desenvolvimento de sistemas computacionais (SIMÃO, 2005). As materializações atualmente mais estáveis do PON para programação de *software* são os arquétipos ou *frameworks* dele sobre linguagens de programação correntes, como C++, C#, Java, Akka e Erlang/Elixir. Naturalmente, a elaboração de um *framework* PON em uma dada linguagem viabiliza nela uma nova abordagem para o desenvolvimento de programas, que é justamente a orientação a notificações, visto que as linguagens são inicialmente desenvolvidas de acordo com um ou mais paradigmas existentes (ROY, 2012).

A proposta apresentada por este trabalho consiste no desenvolvimento de uma nova e ímpar materialização do PON por meio de um *framework* distinto, utilizando o padrão C++20 da linguagem de programação C++. A intenção é manter o desempenho de *frameworks* desenvolvidos com padrões anteriores da linguagem C++, como o padrão C++98 utilizado no caso do *Framework* PON C++ 2.0, e ter as vantagens de desenvolvimento obtidas com *framework* feitos em linguagens ditas modernas como C# e Java. Neste sentido, propõem-se ademais um conjunto de melhorias sobre as materializações atuais do PON na forma de *frameworks*, culminando em um novo arquétipo chamado *Framework* PON C++ 4.0.

Tais melhorias no âmbito do *Framework* PON C++ 4.0 são inclusive possibilitadas pela nova versão do C++ em voga (C++20) com novos conceitos e recursos, como ponteiros inteligentes (*smart pointers*), *variadic templates*, expressões *fold* e expressões *lambda*. As melhorias são igualmente inclusive de orientação sistêmica, fazendo particularmente a aplicação de conceitos de programação genérica e de desenvolvimento orientado a testes. Ainda, os meios de se expressar código em PON são facilitados no proposto *Framework* PON C++ 4.0, aproximando-se do alto nível da Linguagem de Programação do PON, a qual faz parte do estado da arte, mas ainda não do estado da técnica.

Este capítulo primeiramente apresenta a contextualização dos principais conceitos explorados ao longo do desenvolvimento do trabalho. Subsequentemente são apresentadas as motivações, as justificativas e os objetivos que compõem a proposta desse trabalho.

1.1 CONTEXTUALIZAÇÃO

Nesta contextualização é introduzido o Paradigma Orientado a Notificações (PON) sob a ótica de paradigmas de programação, assim como uma revisão básica de seus conceitos de modo a prover o embasamento necessário ao entendimento desse trabalho. Também na contextualização são apresentados os conceitos de programação genérica e desenvolvimento orientado a testes, os quais foram aplicados no desenvolvimento deste trabalho referente ao PON.

Em tempo, é natural que os trabalhos que exploram o PON apresentem uma revisão sobre paradigmas de programação e, particularmente, sobre o PON. Portanto, para o desenvolvimento desta sucinta revisão se aproveita e referencia os conteúdos desenvolvidos anteriormente em outros trabalhos do grupo de pesquisa do PON da UTFPR, salientando os trabalhos de Banaszewski (2009), Santos (2017) e Ronszcka (2012, 2019).

1.1.1 Paradigmas de Programação Dominantes

Um paradigma de programação é tido como um modelo ou método de se desenvolver programas, seguindo modelos linguístico-matemáticos e, ademais, um determinado conjunto de princípios. Os paradigmas são definidos de forma a resolver determinados problemas, à luz dos seus modelos e princípios. Assim, cada paradigma tem seu próprio conjunto de regras que o torna melhor ou pior na resolução de cada problema em específico (ROY, 2012).

As regras definidas pelos paradigmas de programação ou mesmo desenvolvimento servem como guia para o estabelecimento de padrões (*e.g.*, de programação e execução) e ferramentas de desenvolvimento de soluções nas respectivas linguagens de programação e mesmo de projeto (ou *design*). Tais regras também servem como uma orientação aos desenvolvedores sobre como estruturar melhor os seus programas e também projetos que os precedem (ROY; HARIDI, 2004).

No tocante a programação, dentre os principais paradigmas de programação destacam-se quatro, o Paradigma Procedimental (PP), Paradigma Funcional (PF), Paradigma Orientado a Objetos (POO) e o Paradigma Lógico (PL) (SCOTT, 2000; WATT, 2004; BROOKSHEAR, 2006). Esses paradigmas são referenciados como Paradigmas Dominantes. Ainda, pode-se considerar que PP e POO fariam parte de um paradigma maior chamado Paradigma Imperativo (PI), bem como PF e PL fariam parte de um paradigma maior chamado Paradigma Declarativo (PD) (GABBRIELLI; MARTINI, 2010). Naturalmente, pode-se ter interseção dos subparadigmas em

PD e PI, salientando o subparadigma PF a título de exemplo.

Em todo caso, o PP e o POO do PI e o PF e o PL do PD, são conhecidos como Paradigmas Dominantes por estarem bem estabelecidos e em pleno uso prático. Apesar de sua ampla utilização estes paradigmas ainda apresentam certas deficiências, como a presença de redundâncias (estruturais e temporais) e a tendência a acoplamento no PI. Essas deficiências também estão presentes no PD em certa medida, pois ele utiliza mecanismos de execução que se baseiam em buscas sobre elementos passivos¹. Normalmente, o PD é implementado sob linguagens em PI, na sua forma usual ademais (BANASZEWSKI, 2009; RONSZCKA, 2019).

1.1.1.1 Paradigma Dominantes vis-à-vis PI & PD

Os paradigmas dominantes podem ser classificados como subconjuntos dos dois paradigmas maiores, nomeadamente o PI e o PD (BANASZEWSKI, 2009). Essa relação entre os paradigmas nesta visão dada é ilustrada na Figura 1. Essa divisão hierárquica é útil, ainda que imperfeita, pois os paradigmas apresentam características em comum entre si, tendendo a ficar mais perto de um paradigma dominante que de outro. Em suma, esta classificação facilita comparações com relação a estas características em comum (RONSZCKA, 2019).

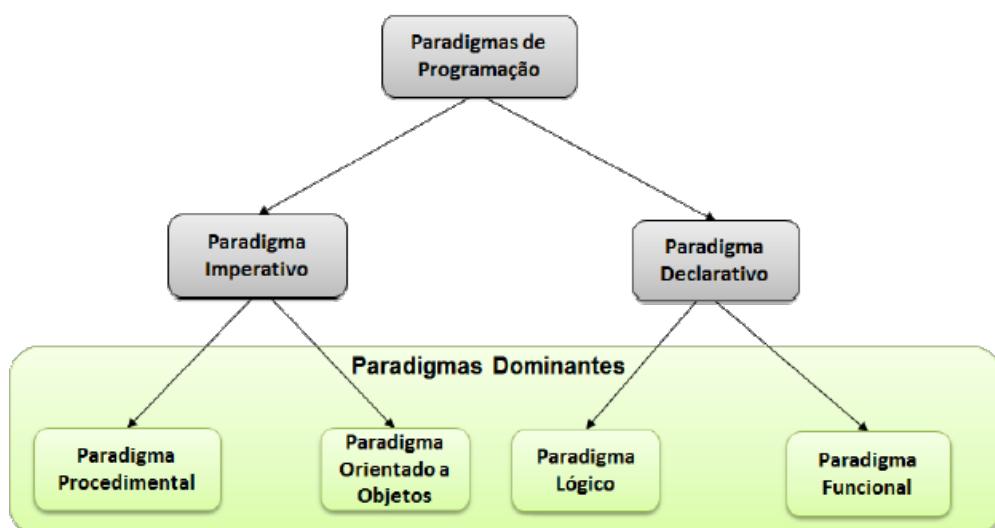


Figura 1 – Classificação dos paradigmas de programação
Fonte: Adaptado de Ronszcka (2019)

Segundo Peter Van Roy, cada paradigma tem seu conjunto de técnicas e conceitos de programação que, conjuntamente, definem sua forma de estruturar o pensamento na construção de programas (ROY, 2012). Com base nesses conceitos ele elaborou uma taxonomia para

¹ A Seção 2.1 apresenta e discute em maiores detalhes os paradigmas mencionados.

paradigmas de programação, conforme apresentada na Figura 2. Neste diagrama os paradigmas de programação são organizados em um grafo que apresenta o relacionamento conceitual entre eles. Neste contexto, setas entre dois quadros representando paradigmas representam a adição de novos conceitos, de forma que o paradigma derivado contempla os conceitos dos paradigmas anteriores, acrescidos de um ou mais novos conceitos que, conjuntamente, os definem como paradigmas distintos (ROY, 2012).

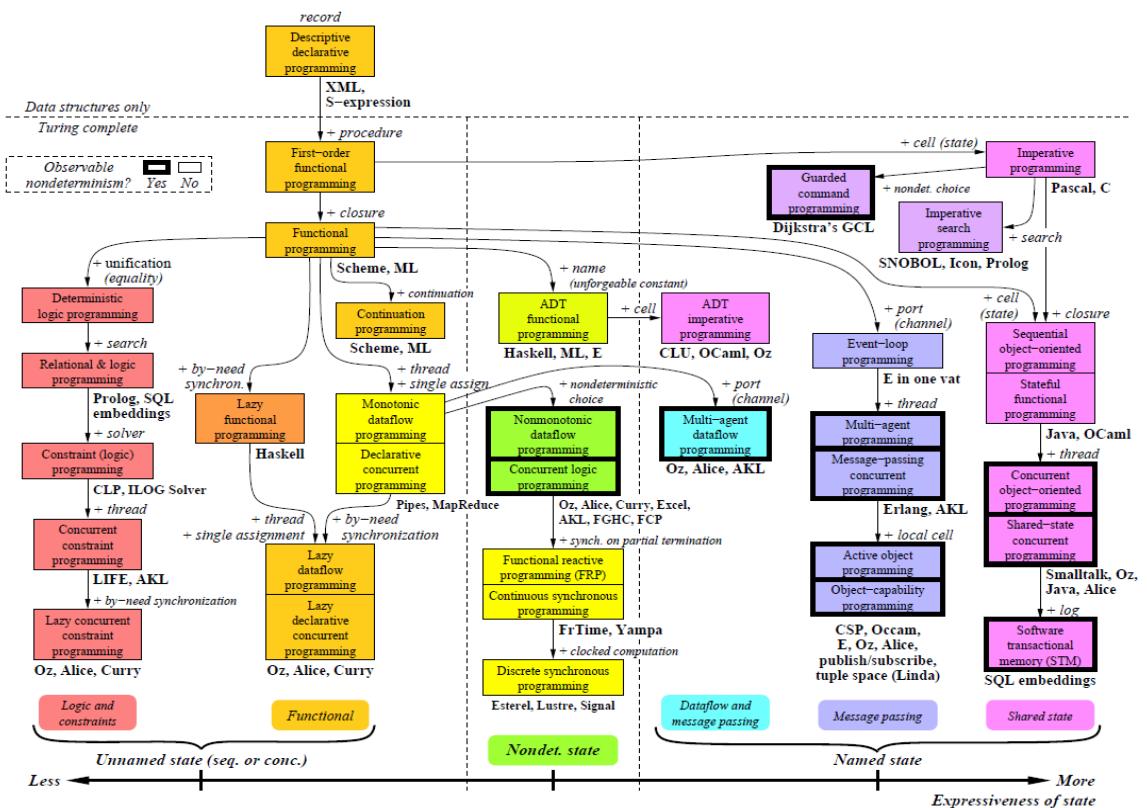


Figura 2 – Taxonomia dos paradigmas e linguagens

Fonte: Roy (2012)

Cada linguagem de programação materializa um ou mais paradigmas, sendo que cada paradigma é composto por um conjunto de conceitos (ROY, 2012). Essa relação entre as linguagens, paradigmas e conceitos é ilustrada na Figura 3. Na prática, quanto mais paradigmas de programação uma linguagem suportar, melhor seria, pois, em dados termos isso provê ao desenvolvedor um conjunto ainda maior de ferramentas. Em havendo mais ferramentas, elas podem ser escolhidas para serem utilizadas de forma a melhor se encaixar no problema específico, visto que nenhum paradigma por si só seria a melhor solução para todos os problemas (ROY; HARIDI, 2004).

Em verdade, no desenvolvimento de uma única aplicação podem sim, ser encontrados

diversos paradigmas, sendo eles aplicados em partes individuais do programa e fidedignas a cada paradigma empregado. Esse estilo de programação é naturalmente chamado de multiparadigma. Entretanto e finalmente, a utilidade de uma linguagem multiparadigma depende naturalmente de quão bem os diferentes paradigmas estão integrados (STROUSTRUP, 1995; ROY; HARIDI, 2004).

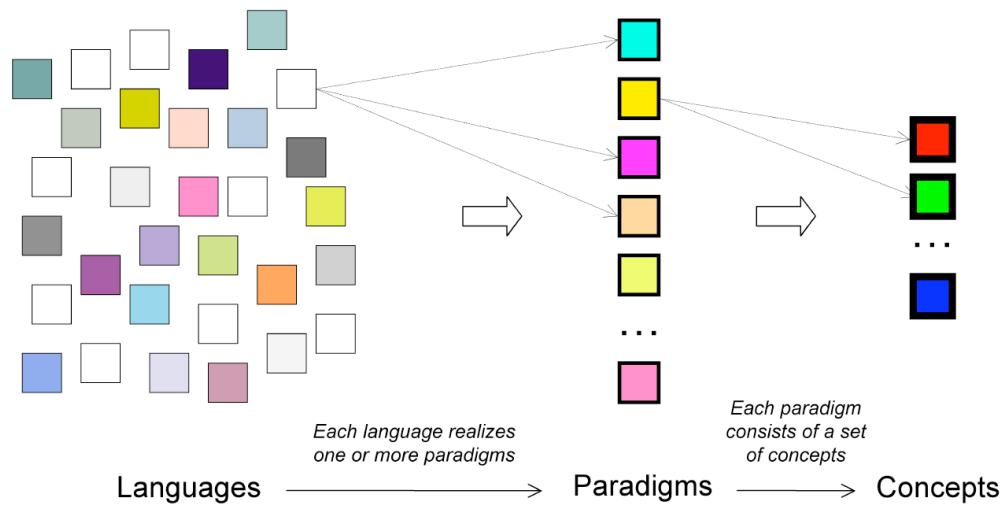


Figura 3 – Linguagens, paradigmas e conceitos

Fonte: Roy (2012)

Dentre os paradigmas dominantes citados, o POO está entre os mais utilizados. Essa conclusão pode ser tirada ao se observar a Tabela 1, em que 4 das 5 linguagens mais populares em 2021 materializam o POO (Java, Python, C++ e C#), somando um total de 35,07% no Índice TIOBE² da Comunidade de Programação (TIOBE, 2021). Essas linguagens são as conhecidas Java, Python, C++ e C#.

Em tempo, o Índice TIOBE é um indicador de popularidade de linguagens de programação, atualizado mensalmente, utilizando dados disponíveis nas ferramentas de pesquisa online. Este índice não reflete qualitativamente sobre as linguagens de programação, mas sim quantitativamente sobre o volume de código escrito com as mesmas.

Além de suas vantagens técnicas, a dominância do POO entre os paradigmas pode ser atribuída também aos sucessos comerciais das linguagens que os materializam, à luz dos ganhos que a POO traz vis-à-vis ao PP, sendo justamente o PP o segundo paradigma mais relevante neste mesmo índice de popularidade. Neste âmbito do domínio do POO, Java, C++ e Kotlin dominam o mercado de desenvolvimento Android, enquanto Swift e Objective-C dominam o de iOS, de

² A sigla TIOBE vem de *The Importance of Being Earnest* (i.e., A Importância de Ser Zeloso), título de uma peça de comédia de 1895 de Oscar Wilde, que é a inspiração para nome da companhia holandesa homônima que mantém o Índice TIOBE.

Tabela 1 – Índice TIOBE de linguagens de programação
Fonte: Adaptado de TIOBE (2021)

Linguagem	Índice
C	12,54%
Python	11,84%
Java	11,54%
C++	7,36%
C#	4,33%

forma que é muito difícil desenvolver *software* para plataformas *mobile* sem dominar o POO. Da mesma forma, também tem se tornado importante conhecer o POO para o desenvolvimento web, salientando aqui as linguagens JavaScript, Python e Ruby (GWOSDZ, 2020).

Dentre essas linguagens salientadas, destaca-se C++ em alguns aspectos, cuja grande popularidade é atribuída por prover a velocidade de execução do C enquanto dá suporte ao POO efetivamente, tendo permitido a transição do PP para o POO por suportar ambos. De fato, desde a sua concepção, ela foi tida, na verdade, como uma linguagem desenvolvida para suportar diversos estilos de programação e, por sua vez, paradigmas, diferindo de linguagens que focam no suporte de apenas um destes estilos (STROUSTRUP, 1995). Com essa perspectiva de ampliar o suporte a múltiplos paradigmas destaca-se a adição de expressões lambda no padrão C++11, que ampliou o suporte do C++ ao paradigma de programação funcional (STROUSTRUP, 2020).

Mesmo uma linguagem como C++, entretanto, ainda herda os problemas de sequencialidade da busca e orientação a pesquisas sobre elementos passivos por meio de laços de repetição, advindo da computação sequencial. Na verdade, estes problemas afetam desde linguagens imperativas até declarativas, sendo que estas últimas, na prática, são implementadas com base nas primeiras finalmente, conforme já discutido. Esses problemas trazem redundâncias temporais e estruturais que podem causar prejuízos em desempenho e também em desacoplamento, dificultando tanto a modularização do código como a execução de forma paralela ou com paralelismos enfim, além da distribuída (SIMÃO *et al.*, 2009; SIMÃO *et al.*, 2012; RONSZCKA, 2019).

1.1.2 Paradigmas de Programação Emergentes

Conforme explicado anteriormente, o PP e o POO do PI e o PF e o PL do PD, são conhecidos como Paradigmas Dominantes por estarem bem estabelecidos e em pleno uso prático. Além dos Paradigmas Dominantes também existem diversos outros paradigmas, ainda com menor grau de importância na prática industrial ou pelo menos não ainda de mesmo impacto que os dominantes, dado que são propostas mais recentes, mas que sim contribuem com novas

formas de se conceber programas.

Estes novos paradigmas por sua vez são referenciados como Paradigmas Emergentes. Dentre os Paradigmas Emergentes estão o Paradigma Orientado a Eventos (POE), Paradigma Orientado a Componentes (POC), o Paradigma Orientado a Aspectos (POAs) e o Paradigma Orientado a Agentes (POAg) (BANASZEWSKI, 2009). A Figura 4 ilustra essa organização em Paradigmas Dominantes e Emergentes, indicando justamente a precedência dos Paradigmas Dominantes com relação aos Paradigmas Emergentes.

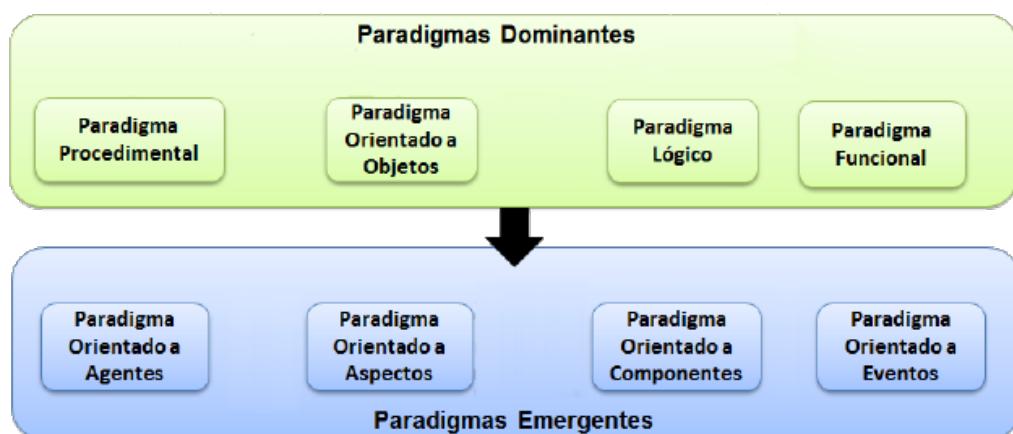


Figura 4 – Classificação dos paradigmas de programação
Fonte: Adaptado de Ronszeka (2019)

Os ditos Paradigmas Dominantes já estão estabelecidos há muito tempo, como o PP, cuja origem foi na década de 50, enquanto os ditos Paradigmas Emergentes, em sua maior parte, têm sua concepção mais recente, após a década de 90. Neste contexto, a Figura 5 apresenta uma linha temporal com os Paradigmas Dominantes, exemplificando com as linguagens de programação nas quais os paradigmas são aplicados e mesmo os Emergentes. Isto permite entender por quê os Paradigmas Emergentes apenas iniciam sua trajetória para alcançar maior impacto na prática industrial (BANASZEWSKI, 2009).

Os ditos Paradigmas Emergentes citados buscam propor soluções a problemas existentes dos Paradigmas Dominantes, porém ainda herdam os problemas de sequencialidade pela orientação a buscas/percorimentos do PI e POO, já mencionados, ainda que atenuados em certa medida no Paradigma Orientado a Eventos (POE), devido à interação entre objetos que ocorre por meio de eventos, alterando o fluxo de execução do programa (FERG, 2006). O detalhamento tanto do POE como dos outros paradigmas mencionados neste capítulo é feito na Seção 2.1.

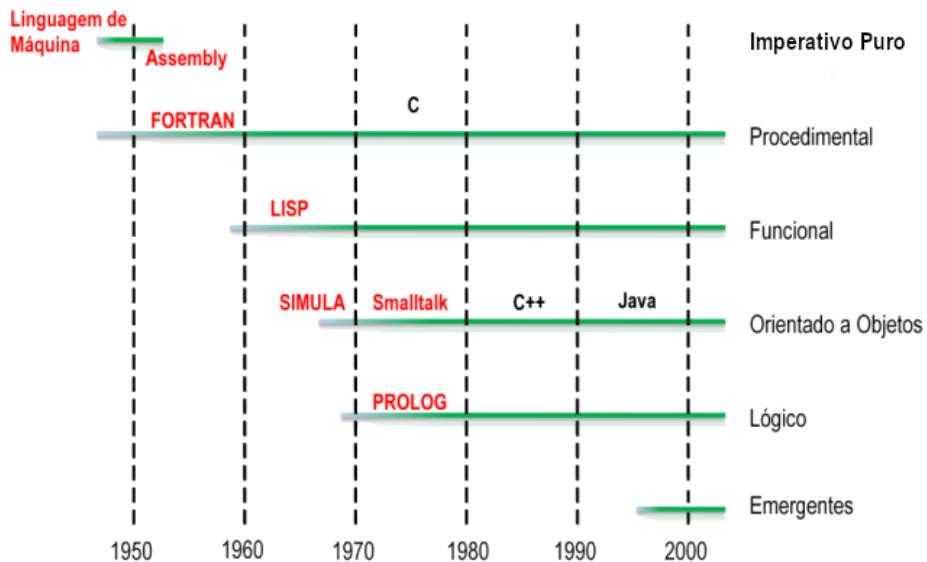


Figura 5 – Evolução dos paradigmas de programação

Fonte: Adaptado de Banaszewski (2009)

1.1.3 Paradigma Orientado a Notificações — PON

À luz do apresentado nas seções anteriores, é introduzido o Paradigma Orientado a Notificações (PON). O PON pode ser inserido nesse contexto como um paradigma emergente, mas *sui generis*. O PON surge embrionariamente em 2001 por meio da proposta de um metamodelo para a solução de problemas de controle discreto, o qual evoluiu como uma solução geral de inferência. Subsequentemente, a solução evolui para um paradigma de programação e mesmo de desenvolvimento em geral (SIMÃO; STADZISZ, 2002; SIMÃO, 2005; SIMÃO; STADZISZ, 2008; SIMÃO *et al.*, 2009; RONSZCKA, 2019). O PON, como paradigma de programação/desenvolvimento, tem os objetivos de tornar menos árdua a tarefa do desenvolvimento de sistemas por permitir concepção em alto nível, tornar o código e sua execução mais eficiente por evitar redundâncias (estruturais e temporais) e, por fim, tornar a sua execução paralelizável/distribuível por garantir o desacoplamento (SIMÃO *et al.*, 2009).

O PON até encontra inspirações no PI, tais como a flexibilidade algorítmica e a abstração em forma de objetos do POO e mesmo a reatividade da programação dirigida a eventos, entretanto ambas postas de modo distinto. Ele também até aproveita conceitos próprios do PD, como a facilidade de programação em alto nível e a representação do conhecimento em regras, mas também com suas próprias idiossincrasias que o permitem ser algo outro. Assim, neste âmbito, o PON provê a possibilidade do uso de partes de ambos os estilos de programação em alguma medida, apresentando, contudo, revoluções em seu modelo no tocante aos seus constituintes, à

organização deles e ao seu processo de inferência ou cálculo lógico-causal (SIMÃO; STADZISZ, 2008; SIMÃO *et al.*, 2009; RONSZCKA, 2019).

Como o próprio nome sugere, a principal característica estrutural do PON é que ele é construído com base em notificações entre suas entidades (*i.e.*, *Attributes*, *Premises*, *Conditions*, *Rules*, *Actions*, *Instigations* e *Methods*), havendo, portanto, um mecanismo para tal. Cada elemento constituinte do PON pode enviar e/ou receber notificações pontuais, fazendo com que as avaliações lógicas e causais somente sejam realizadas quando ocorre uma notificação. Na verdade, o PON propõe a divisão da computabilidade em dois grandes grupos de entidades, um grupo facto-execucional e outro grupo lógico-causal, relacionados entre si por meio de notificações de seus constituintes. O primeiro grupo se constitui dos *Fact Base Elements (FBEs)*, enquanto o segundo se constitui em entidades chamadas de *Rules*.

Em suma, *FBEs* e *Rules* se compõem de elementos menores que permitem realizar o fluxo de notificações entre *FBEs* e *Rules* e vice-versa. No PON, cada *Rule* é a entidade capaz de tratar uma expressão lógico-causal. Assim, as *Rules* gerenciam o conhecimento sobre qualquer comportamento lógico-causal no sistema. O conhecimento lógico-causal de uma *Rule* provém normalmente de uma regra se-então, o que é uma maneira natural de expressão deste tipo de conhecimento. Por sua vez, o *FBE* é uma entidade usada para descrever estados e serviços de entidades reais ou cibernéticas em um problema computacional (BANASZEWSKI, 2009).

Na Figura 6, é apresentado um exemplo de entidade *Rule*, justamente na forma de uma regra lógico-causal, já com a indicação de suas entidades constituintes menores (NEVES *et al.*, 2021). A *Rule* ilustrada seria parte de um sistema de tomada de decisão relativa a um dado sensor. Este sensor compõe a etapa facto-execucional do sistema na forma de um *FBE*, enquanto a *Rule* compõe a etapa lógico-causal.

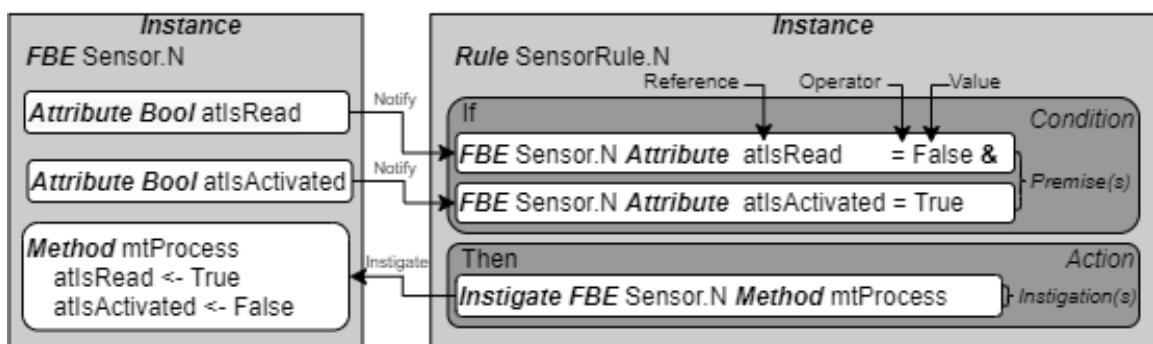


Figura 6 – Estrutura de Rule no PON
Fonte: Neves *et al.* (2021)

Mais precisamente, o sensor é representado pelo *FBE Sensor* com seu *Method mtProcess*

e seus *Attributes* *atIsRead* e *atIsActivated*. A *Rule*, por sua vez, apresenta uma *Condition* que é composta por duas *Premises* que fazem as seguintes verificações sobre o *FBE*: a) *O sensor foi lido?* b) *O sensor foi ativado?*. A aprovação desta *Condition* leva a execução da *Rule*, que por sua vez aciona a sua *Action*, a qual é composta uma *Instigation*. A aprovação da *Rule* resulta na execução do *Method mtProcess* que implementa a funcionalidade de processar no *FBE Sensor*, atribuindo valores aos seus *Attributes* que levam a mudança de estado de cada um deles.

Em suma, a *Condition* da *Rule* em questão lida com a decisão de processamento do sensor a partir de duas *Premises* que avaliam estados dos *Attributes* do *FBE Sensor*. Uma vez que a decisão seja positiva, a *Action* da *Rule* é responsável pela instigação da *Instigation*, a qual instiga o *Method* da *Rule*. Isto posto, por generalização, percebe-se que todo o processamento lógico-causal e facto-execucional é efetuado por elementos constituintes das *Rules* e *FBEs*, tidos como entidades pequenas ou mínimas (SIMÃO; STADZISZ, 2008; SIMÃO *et al.*, 2009; KERSCHBAUMER, 2018).

O diagrama de classes da Figura 7 apresenta as relações entre todos os elementos do chamado metamodelo do PON (SIMÃO *et al.*, 2009; RONSZCKA, 2012; SIMÃO *et al.*, 2012). Assim sendo, cada um dos elementos constituintes do PON é mais precisamente e mesmo sucintamente detalhado conforme segue:

- *Fact Base Element (FBE)*: Entidade de *Elemento Base de Fatos* que contém os *Attributes* e *Methods*, podendo ser considerada similar aos objetos do POO em termos simplistas.
- *Attribute*: Entidade de *Atributo* que representa, cada qual, uma propriedade de um *FBE*, sendo responsável por armazenar um valor discreto-factual que representa estados. Um *Attribute* difere de uma variável do PP ou atributo do POO tradicional no sentido de que possui a capacidade de notificar *Premises* relacionadas quando seu estado é alterado.
- *Premise*: Entidade de *Premissa* que realiza a avaliação lógica de estados de dois *Attributes* por meio de determinado operador de comparação (*e.g.*, igual, diferente, maior e menor) e notifica *Conditions* relacionadas quando muda de estado lógico.
- *Condition*: Entidade de *Condição*, que realiza a avaliação lógica de *Premises* por meio de determinado operador lógico (*e.g.*, conjunção ou disjunção), e notifica *Rules* relacionadas quando muda de estado, como de aprovada para reprovada e vice-versa.

- *Rule*: Entidade de *Regra*, relacionada a uma *Condition*. Tipicamente, cada *Rule* executa sua *Action* quando é aprovada.
- *Action*: Entidade de *Ação*, cada *Rule* é relacionada a uma *Action*, que se relaciona com uma ou mais *Instigations*. Ela ativa todas as suas *Instigations* quando é executada uma vez notificada pela *Rule*.
- *Instigation*: Entidade de *Instigação*, responsável por instigar os *Methods* relacionados quando é ativada pela *Action* relacionada.
- *Method*: Entidade de *Método* do PON, de forma análoga a funções membro (ou métodos) do POO, sendo executado quando notificado/instigado pela *Instigation*.

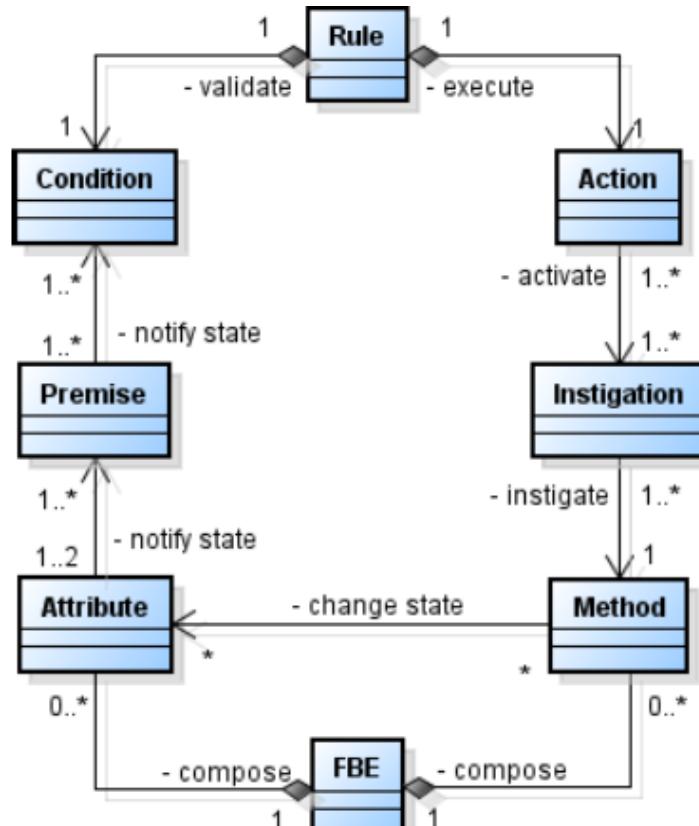


Figura 7 – Diagrama de classes das entidades do metamodelo do PON
Fonte: Ronszcka (2012)

O fluxo de execução, ilustrado na Figura 8, ocorre em função da mudança de estado de um *Attribute* de determinado *FBE*. A mudança de estado no *Attribute* notifica as suas *Premises*, que por sua vez reavaliam os seus estados lógicos. Se o valor lógico da *Premise* se altera ocorre então uma notificação para as *Conditions* interessadas no estado desta *Premise*. Consequentemente, cada *Condition* avalia seu valor lógico segundo as *Premises* relacionadas.

Quando a condição da *Condition* é satisfeita, isto resulta na aprovação da sua respectiva *Rule* para a execução. Quando a *Rule* é aprovada, sua *Action* é ativada. Uma *Action* é conectada a uma ou mais *Instigations*, que por sua vez acionam a execução de alguma funcionalidade de um *FBE* por meio dos seus *Methods*. As chamadas para um *Method* podem causar alteração nos estados dos *Attributes* e o ciclo de notificação recomeça (BANASZEWSKI, 2009).

O fluxo de iterações das aplicações em PON é realizado por meio de uma cadeia de notificações pontuais entre as entidades do PON. Essa cadeia de notificações é transparente ao desenvolvedor, porque as notificações são realizadas de forma autônoma pelas entidades reativas do PON. O fluxo de iterações do PON difere do fluxo de iterações de aplicações do PI, no qual o desenvolvedor deve modelá-lo de maneira explícita por meio do uso de laços de repetição. No PON esse fluxo acontece de forma natural na perspectiva de execução da aplicação, conforme exemplificado na Figura 8 e esboçado no diagrama de classes da Figura 7.

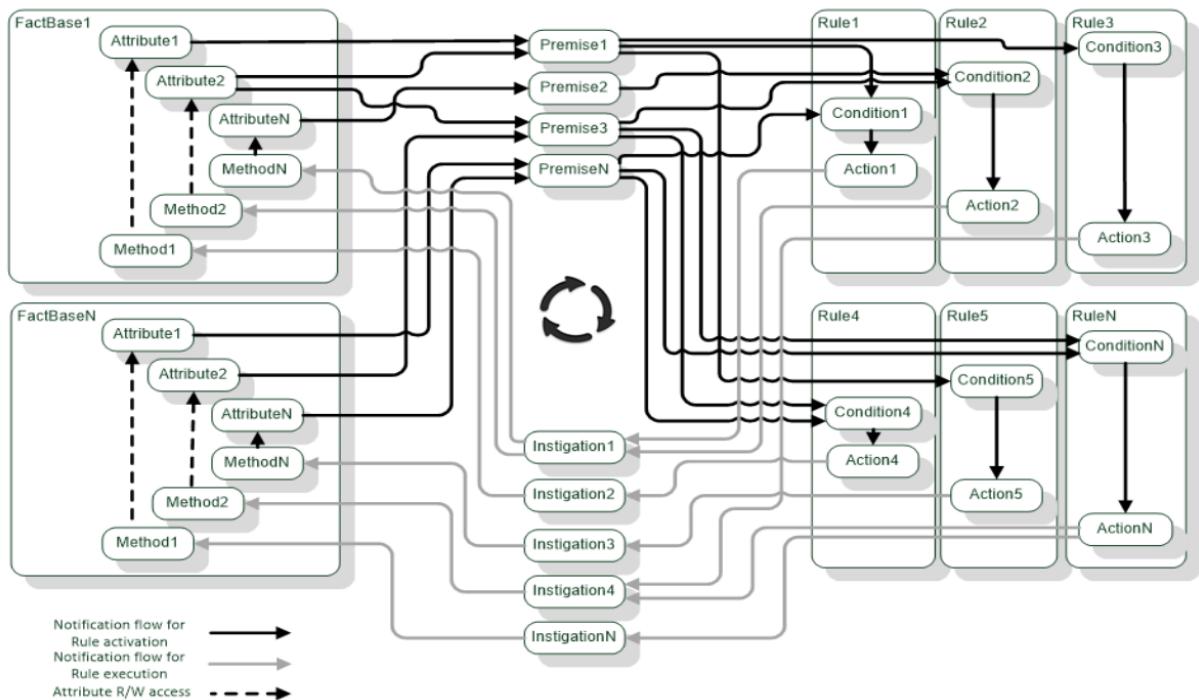


Figura 8 – Representação do fluxo de notificações do PON
Fonte: Linhares (2015)

No PON, as entidades pequenas são reativas e efetivamente desacopladas, sendo que colaboram entre si por meio de notificações pontuais de modo a realizar o cálculo lógico-causal de forma precisa, nos termos já explicados, evitando implicitamente redundâncias de processamento. Essas notificações ditam, portanto, o fluxo de execução das aplicações. Eis que essa nova maneira de concepção de programas tende a proporcionar melhora no desempenho das

aplicações, potencialmente facilitando a sua concepção, tanto para ambientes não distribuídos como para ambientes distribuídos (SIMÃO; STADZISZ, 2008; SIMÃO *et al.*, 2009; SIMÃO *et al.*, 2012).

Esta seção introduz o PON de maneira breve, sendo que nas Seções 2.2, 2.6 e 2.3 é feita uma revisão mais aprofundada do estado da arte do PON. Além disso, o PON não está vinculado a nenhuma plataforma específica, possuindo implementações em diversas plataformas. Uma revisão detalhada sobre as diversas materializações disponíveis do PON é feita na Seção 2.4.

No tocante às implementações em plataformas de *software*, particularmente dos *frameworks*, uma lacuna destes é a falta de uso extensivo de programação genérica, bem como a falta de orientação a testes ostensiva nos seus desenvolvimentos. Nesse sentido, a Seção 1.1.4 apresenta os conceitos de programação genérica que podem ser aplicados em materializações do PON.

1.1.4 Programação Genérica

A programação genérica pode ser caracterizada como uma técnica de programação que foca na abstração dos seus algoritmos na sua forma mais genérica, mas sem perda de eficiência (STEPANOV; ROSE, 2015). A aplicação das técnicas de programação genérica consiste em escrever código que consegue suportar e atender adequadamente diferentes tipos de dados, no qual o tipo é interpretado como um argumento que só é de fato aplicado quando o código for instanciado utilizando um tipo concreto (DEHNERT; STEPANOV, 1998).

Essa técnica é frequentemente utilizada para resolver um problema inerente de linguagens de programação com tipagem estática, que é a pouca flexibilidade de tipos em comparação com as linguagens com tipagem dinâmica (CARDELLI; WEGNER, 1985). Nas linguagens com tipagem estática, os tipos das variáveis são resolvidos na etapa de compilação, enquanto nas linguagens com tipagem dinâmica os tipos são resolvidos apenas durante a execução. Tem-se como exemplo o C++ enquanto uma linguagem com tipagem estática, bem como a linguagem Python enquanto uma linguagem com tipagem dinâmica (HURD, 2021).

Mais precisamente, um problema clássico que a programação genérica resolve é a duplicidade de código, principalmente em linguagens com tipagem estática. É muito comum haver a necessidade de realizar operações ou algoritmos, por meio de funções, utilizando variáveis de diversos tipos ao longo do código. Um exemplo usual são estruturas de dados, como listas encadeadas, que atuam sobre entidades computacionais de diversos tipos. Nesse tipo de caso,

pode haver a duplicação de código ao declarar funções diferentes, ainda que similares, para o tratamento de cada tipo (ALEXANDRESCU, 2001; STEPANOV; ROSE, 2015).

Os códigos e algoritmos desenvolvidos com programação genérica precisam, entretanto, alcançar um desempenho tão bom quanto algoritmos escritos com código em tipagem específica, caso contrário seria muito difícil justificar seu uso em casos nos quais o desempenho seja uma característica importante (STEPANOV; ROSE, 2015). A substituição dos tipos em tempo de compilação faz com que o compilador gere o código especializado para cada tipo instanciado pelo programa, de forma que tanto o desempenho como as propriedades fornecidas pela tipagem estática são mantidas mesmo em um quadro de código genérico (ALEXANDRESCU, 2001). Neste âmbito, caso o compilador tente instanciar o código genérico utilizando um operador ou conversão não suportada pelo tipo instanciado isso vai resultar em um erro de compilação (BOS, 2019).

No caso do C++ em particular, o principal recurso utilizado para a aplicação de programação genérica são os *templates*. Com os *templates*, declara-se um tipo genérico, o qual é utilizado na definição do código em desenvolvimento. Subsequentemente, o compilador se encarrega de substituir os tipos conforme as chamadas deste código sejam instanciadas utilizando tipos específicos. Essa substituição de tipos é feita na etapa de compilação, de forma que isso não causa prejuízos em desempenho (ALEXANDRESCU, 2001).

Tendo como exemplo uma função simples em C++ que retorna o máximo entre dois valores, enquanto em uma implementação regular seria necessário criar código dedicado para cada tipo utilizado, pode-se ter uma função genérica utilizando *templates*. Tais exemplos são mostrados nos Códigos 1.1 e 1.2, respectivamente. É interessante observar que na implementação do Código 1.1 o código utilizado no corpo das duas funções é igual, enquanto a implementação do Código 1.2 elimina este código repetido.

Código 1.1 – Função GetMax sem templates

```
int GetMax(int a, int b) {
    return (a>b) ? a:b;
}

float GetMax(float a, float b) {
    return (a>b) ? a:b;
}

int main() {
    GetMax(1, 2);
    GetMax(1.5, 2.7);
    return 0;
}
```

Fonte: Autoria própria

Código 1.2 – Função GetMax com templates

```
template <typename T>
T GetMax(T a, T b) {
    return (a>b) ? a:b;
}

int main() {
    GetMax(1, 2);
    GetMax(1.5, 2.7);
    return 0;
}
```

Fonte: Autoria própria

Além da utilização de *templates* para a passagem de argumentos de funções, também é possível utilizar *templates* na construção de classes. Desta forma é possível definir os tipos de variáveis membros como *templates*, possibilitando a reutilização de código. No exemplo do Código 1.3 são definidas classes para a implementação de uma lista simplesmente encadeada com tipo genérico.

Código 1.3 – Classe genérica com *templates*

```
template <typename T>
class ListItem {
    // Definição da classe
    T* item;
    T* next;
}

template <typename T>
class List {
    // Definição da classe
    ListItem<T> head;
};

List<Animal> list_of_animals;
List<Car> list_of_cars;
```

Fonte: Autoria própria

Outro exemplo é apresentado no Código 1.4, onde classes distintas realizam um processo de inicialização por meio do método *init()*. Considerando uma situação na qual as classes A, B e C não são relacionadas, de modo que não seria pertinente a aplicação de herança do POO, e que os métodos *init()* sejam similares, porém não idênticos, pode haver duplicação de código nos métodos *init()*. Neste caso pode ser feita a aplicação de programação genérica para implementar parte destas funcionalidades comuns, conforme apresentado no Código 1.5. Desta forma, é possível eliminar a duplicação de código desnecessária, sem haver quebra de conceitos como herança (THORSEN, 2015). Em um programa sem a aplicação de programação genérica seria necessário escrever o mesmo código duas vezes. Isso traz o problema de duplicar o esforço gasto escrevendo código, assim como abre espaço para o aparecimento de problemas futuros, pois a partir desse momento será necessário fazer a manutenção de dois códigos separados. Portanto, podem acabar surgindo defeitos em um ou outro, e eventualmente esses códigos podem acabar divergindo e apresentando comportamentos diferentes, o que não seria o desejado pela solução inicial (STEPANOV; ROSE, 2015).

Código 1.4 – Exemplo de código sem programação genérica

```
class A {
    void init() {
        // Longo trecho de código complexo;
    };
};

class B {
    void init() {
        // Similar ao init() de A
    };
};

class C {
    void init() {
        // Similar ao init() de A
    };
};
```

Fonte: Adaptado de Thorsen (2015)

Código 1.5 – Exemplo de código com programação genérica

```
template static inline void f1(T* t) {
    // Faz parte do trabalho de init()
    t->doSomething();
}

template static inline void f2(T* t) {
    // Faz outra parte do trabalho de
    // init()
}

template static inline void f3(T* t) {
    // Faz outra parte do trabalho de
    // init()
}
...
void A::init() {
    f1(this);
    f2(this);
    f3(this);
}
void B::init() { // Não utiliza f1
    f2(this);
    f3(this);
}
void C::init() { // Não utiliza f2
    f1(this);
    f3(this);
}
```

Fonte: Adaptado de Thorsen (2015)

Na linguagem de programação C++ existe a biblioteca *Standard Template Library* (STL), o que se constitui em um importante recurso nesta linguagem. A STL é a biblioteca padrão de *templates* que, dentre outros recursos, contém definições de estruturas e algoritmos genéricos implementados com *templates*. Dentre esses, podemos citar as estruturas de listas, vetores e algoritmos de ordenação e busca (ISO/IEC, 2017).

De forma geral, é dito que a programação genérica deve ser utilizada sempre que possível, visto que reduz a duplicidade de código (ALEXANDRESCU, 2001). Porém, como contraponto, ela resulta em um código que é mais complexo, podendo causar dificuldades para programadores iniciantes, que não tenham um domínio tão completo da linguagem e que, portanto, apresentarão maior dificuldade em entender essas construções genéricas mais complexas (STEPANOV; ROSE, 2015).

O suporte à programação genérica foi ampliado nas versões mais atuais do padrão da linguagem de programação C++, com a adição de novos recursos, como *variadic templates* e *fold expressions* (GRIMM, 2020; STROUSTRUP, 2020). Em suma *variadic templates* permitem a utilização de *templates* para a declaração de funções e métodos que recebem um número variável de argumentos, enquanto as *fold expressions* fornecem os mecanismos necessários para se operar sobre esses conjuntos de argumentos variáveis. Na Seção 2.7 são explorados com maiores detalhes os recursos utilizados no escopo deste trabalho, incluindo *variadic templates* e

fold expressions.

Dado esse contexto é possível destacar que estes conceitos de programação genérica podem ser aplicados no desenvolvimento dos *frameworks* do PON em C++ de modo a permitir maior flexibilidade de tipos e flexibilidade algorítmica. A aplicação destes conceitos pode contribuir principalmente no sentido de facilitar a programação em alto nível. A falta disto se constitui em uma das imperfeições dos atuais *frameworks* do PON.

Outrossim, outra imperfeição seria a falta de ostensivo desenvolvimento orientado a testes nos *frameworks*. A carência da realização de testes no processo de desenvolvimento das materializações atuais resulta em dificuldades relativas ao seu uso principalmente no que diz respeito a presença de problemas no código (*i.e., bugs*) que são percebidos apenas durante a sua utilização como, por exemplo, no caso do *Framework PON C++ 2.0* conforme relatado por Xavier (2014) e no caso do *Framework PON C++ 3.0* conforme relatado por Schütz (2019).

1.1.5 Desenvolvimento Orientado a Testes — *Test Driven Development* (TDD)

O método de desenvolvimento chamado Desenvolvimento Orientado a Testes ou, em idioma inglês, o assim chamado *Test Driven Development* (TDD), surgiu no final da década de 90 (LANGR, 2013), tendo seu ciclo de desenvolvimento formal estabelecido por Beck e Kunningham (2002). O TDD é um método que permite aumentar a confiabilidade do *software*, por meio do desenvolvimento de testes.

Como este presente trabalho envolve o desenvolvimento de um novo *framework* para PON em C++ (a chamada versão 4.0), inclusive com maior generalidade e também com maior confiabilidade, há de se ter uma forma de reduzir o número problemas encontrados durante seu uso e execução. Neste âmbito, a aplicação do TDD se torna pertinente a este trabalho.

O uso de TDD evitaria enfim reincorrer em problemas no desenvolvimento de *frameworks* precedentes do PON, como erros de execução e retrabalho para localizar a falha e alcançar as correções devidas. Por exemplo, há um erro reportado no *Framework PON C++ 2.0* em Xavier (2014) que só foi resolvido subsequentemente no *Framework PON C++ 3.0* em Belmonte (2012), que por sua vez tinha imperfeições tratadas apenas em Schütz (2019).

Isto dito, o método do TDD transforma o processo de desenvolvimento do *software*, dando maior foco ao desenvolvimento dos testes. Uma concepção equivocada, porém muito comum, do TDD é que todos os testes devem ser desenvolvidos de uma única vez no começo da etapa de desenvolvimento. Entretanto, na aplicação correta deste método, é dado foco ao

desenvolvimento de um teste por vez (LANGR, 2013).

No ciclo de desenvolvimento estabelecido pelo TDD tem-se em primeiro lugar o desenvolvimento de um teste para a funcionalidade que se deseja implementar. Inicialmente a execução deste teste deve falhar, visto que a funcionalidade ainda não foi implementada. Após a falha inicial do teste é feita a implementação do código e então os testes são executados novamente. Esse ciclo se repete até que todos os testes passem e enfim pode ser dado início ao ciclo de implementação de uma nova funcionalidade (AMBLER, 2006). Esse fluxo é ilustrado na Figura 9.

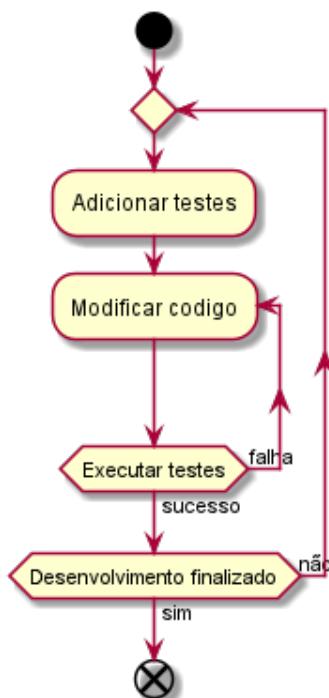


Figura 9 – Fluxo de desenvolvimento no TDD

Fonte: Adaptado de Mody (2017)

Nesse contexto, o TDD recorre a dois tipos de testes, os testes unitários e testes de integração (AMBLER, 2006). Os testes unitários são aqueles que avaliam o funcionamento de uma unidade do código (como classes, objetos ou funções), enquanto os testes de integração avaliam o funcionamento de diferentes partes do sistema operando juntas. Em todo caso, para ambos os tipos de teste, aplica-se o ciclo ilustrado na Figura 9.

Um dos principais benefícios desse método é que, uma vez que os testes já foram desenvolvidos, torna-se possível garantir que a implementação de novas funcionalidades não altera as funcionalidades já existentes no sistema, que foram previamente testadas, contanto que os testes continuem sendo executados com sucesso. O TDD também ajuda o desenvolvedor a

detectar problemas na arquitetura do *software* durante o desenvolvimento (LANGR, 2013).

Em contraponto aos benefícios anteriormente apresentados, uma das maiores dificuldades para a adoção do TDD no processo de desenvolvimento de *software* é que os inúmeros ciclos de desenvolvimento de testes e desenvolvimento de código tendem a aumentar significativamente o tempo de desenvolvimento. Entretanto, certamente, esse tempo gasto na etapa de desenvolvimento é compensado pelo tempo economizado corrigindo problemas de código durante a execução do *software* desenvolvido (LANGR, 2013).

Devido a esse grande número de iterações de testes que acontecem durante o processo, é essencial que os testes sejam fáceis de executar e também rápidos, pois caso isto não ocorra esse processo pode consumir muito tempo durante o desenvolvimento, no limite inviabilizando TDD (LANGR, 2013). Assim, para auxiliar neste processo, existem diversas ferramentas que podem auxiliar o desenvolvedor, sendo que na Seção 2.8 são apresentados *frameworks* para a realização de testes em C++.

Em suma, a aplicação do TDD no contexto do desenvolvimento de um novo *framework* do PON é importante, pois, conforme já explicado, permite garantir a estabilidade do *software* desenvolvido. Isso contribui para melhorar a experiência de uso do *framework* e, por tanto, do paradigma, pois reduziria significativamente o número de problemas encontrados durante o uso do mesmo por outros desenvolvedores.

1.2 MOTIVAÇÃO

Os sistemas computacionais estão se tornando cada vez mais complexos ao longo dos anos, exigindo cada vez mais soluções em *software* e *hardware* que atendam às suas necessidades (BORKAR; CHIEN, 2011; PORDEUS, 2020). Dentre as principais demandas contemporâneas e históricas do desenvolvimento de *software* está o desenvolvimento de programas em alto nível, utilizando ferramentas que facilitem o processo de desenvolvimento. Ademais, em simultâneo, almeja-se que tais programas apresentem alto desempenho, com baixos tempos de execução. Por fim, eles deveriam ainda se adequar a arquiteturas de computação modernas, como é o caso de arquiteturas de computação *multicore*, de forma a aproveitar seu potencial de paralelismo (BELMONTE *et al.*, 2016).

Do ponto de vista de *hardware*, mais especificamente, houve substancial evolução no aspecto da integração de circuitos, sendo que no desenvolvimento de processadores integraram-se múltiplos núcleos em uma única pastilha, o que se constitui nos processadores *multicore*

justamente (ASANOVIC *et al.*, 2009; BORKAR; CHIEN, 2011; PORDEUS, 2020). Nesse âmbito, de modo a atingir alto desempenho, é necessário que o desenvolvimento de *software* seja capaz de fazer proveito da capacidade de paralelismo oferecida por tais processadores, *i.e.*, paralelizando *threads* com granularidade apropriada para tal (HENNESSY; PATTERSON, 2003; LINHARES, 2015; BELMONTE *et al.*, 2016; NEGRINI, 2019).

Nesse contexto todo, considerando as características apresentadas na Seção 1.1.3, o PON se apresenta como uma alternativa viável para o atendimento destas demandas contemporâneas e históricas do desenvolvimento de *software*. À luz de sua teoria, as materializações em PON podem ser evoluídas e utilizadas de forma a permitir o desenvolvimento de *software* em alto nível e com alto desempenho. O desenvolvimento de *software* em PON pode trazer benefícios como redução dos custos em tempo de desenvolvimento ao permitir a programação em alto nível, ao mesmo tempo que apresentaria desempenho compatível ou superior ao das tecnologias atualmente utilizadas. Ademais, o alto nível de desacoplamento das entidades computacionais em PON seria um facilitador para fins de paralelismo e/ou distribuição (SIMÃO *et al.*, 2012; RONSZCKA *et al.*, 2017; RONSZCKA, 2019).

Durante os últimos anos, o PON vem apresentando muitas evoluções, tanto do ponto de vista do estado da arte, com o refinamento dos conceitos que formam o paradigma, assim como do ponto de vista do estado da técnica. Dentre as evoluções técnicas, está o desenvolvimento de *frameworks* que possibilitam a aplicação do paradigma em diversas linguagens de programação como C++ (BANASZEWSKI, 2009; RONSZCKA, 2012), Java (HENZEN, 2015), C# (HENZEN, 2015; OLIVEIRA, 2019), Elixir/Erlang (NEGRINI, 2019), bem como por meio de linguagens ainda prototípicas de programação próprias ao PON no âmbito da chamada Tecnologia LingPON (RONSZCKA, 2019). Esses trabalhos do estado da arte/técnica são explorados com mais detalhes na seção 2.2, sendo salientados os *frameworks* por serem versões assaz estáveis, constituindo o estado da técnica enfim.

Ainda no âmbito do estado da técnica, eis que o paralelismo, justamente uma das principais características do PON, não é implementada em certas versões de *framework*, particularmente naqueles em C++ e tampouco naqueles em C# e Java (RONSZCKA, 2019). Mais precisamente, isto é bem o caso do *Framework PON C++ 1.0* e também do contemporâneo *Framework PON C++ 2.0*. Ainda, mesmo no *Framework PON C++ 3.0*, que justamente visava paralelismo ao nível de *thread*, isto não é adequadamente ou, ao menos, estavelmente implementado (RONSZCKA, 2019; SCHÜTZ, 2019). Por outro lado, *frameworks* PON que tratam mais

apropriadamente deste tipo de paralelismo, como *Framework Akka* e *Framework Elixir/Erlang*, usam linguagens de nível muito alto (orientada a atores) que não primam pelo alto desempenho, como ocorreria se fossem feitos em C++ (RONSZCKA, 2019; NEGRINI, 2019; NEGRINI *et al.*, 2019a; NEGRINI *et al.*, 2019b).

Assim sendo, uma versão nova de *framework*, do PON que resolvesse imperfeições dos precedentes e tivesse usabilidade melhorada, facilitaria o aprendizado e aplicação dos conceitos deste paradigma emergente. Esta nova versão também poderia potencialmente aumentar o volume e qualidade das aplicações desenvolvidas neste paradigma, por meio da natural redução do tempo de desenvolvimento necessário para utilizar o *framework*. Para tal, mais pontualmente, o novo *framework* deve ser mais versátil, inclusive facilitando a integração com bibliotecas e aplicações externas, permitindo o uso de programação genérica e tendo sido desenvolvido à luz de TDD. Quanto maior for a maturidade das materializações do estado da técnica em PON e maior for a facilidade de uso, maior será também a viabilidade do PON, auxiliando assim que este evolua do estágio de Paradigma Emergente, no sentido de começar a alcançar a transição para um paradigma estabelecido, quiçá um dia tornando-se um paradigma vigente.

Ainda no que diz respeito ao desempenho das aplicações, as materializações do PON carecem de aplicações bem definidas que sirvam como *benchmark* bem estabelecido, permitindo a realização de comparações entre os *frameworks* do PON e até mesmo com outros paradigmas. Nesse contexto, Pordeus (2020) propõe a utilização de dois algoritmos conhecidos, o *Random Forest* (que se trata de um algoritmo de aprendizagem de máquina) e o *Bitonic Sort* (um algoritmo de ordenação), que permitiram a avaliação do desempenho do PON em ambientes *mono* e *multicore*. De modo similar, Negrini (2019) também propõe a implementação de algoritmos para o controle automatizado de tráfego. Com esse propósito, uma versão nova de *framework* do PON deveria ser capaz de implementar tais aplicações propostas, se beneficiando também da utilização de ferramentas que permitissem garantir a confiabilidade e reproduzibilidade destes resultados, como por meio do uso de *frameworks* de teste.

1.3 JUSTIFICATIVA

A principal forma disponível e tecnologicamente estável para o desenvolvimento de aplicações no PON ainda é por meio do uso de *frameworks*. Dentre todas as materializações de *frameworks* do PON, aquela que apresenta maior grau de maturidade e estabilidade é o *Framework PON C++ 2.0* (RONSZCKA *et al.*, 2017). Apesar de todos os esforços já realizados

no desenvolvimento dos *frameworks*, inclusive no *Framework PON C++ 2.0*, estes *frameworks* ainda possuem muitas imperfeições e mesmo deficiências. Isto foi, em partes, supra considerado e relatado neste trabalho.

Apesar dos *frameworks* PON em C++ entregarem, do ponto de vista conceitual, toda a funcionalidade necessária para o desenvolvimento de aplicações em PON, eles ainda apresentam algumas limitações importantes, como verbosidade, baixa flexibilidade de tipos, baixa flexibilidade algorítmica, além de particularmente não contemplarem todas as propriedades elementares do PON. Neste sentido, o *Framework PON C++ 2.0* não contempla a propriedade de paralelismo, enquanto a programação em alto nível não é plenamente possível devido principalmente à excessiva verbosidade de expressão de código imposta por ele. O *Framework PON C++ 3.0*, por sua vez, tenta materializar a propriedade de paralelismo ao nível de *thread*, porém não o faz de maneira satisfatória, apresentando diversos problemas de estabilidade, conforme reportado em Schütz (2019) e outros (MARTINI *et al.*, 2019; RONSZCKA, 2019; NEGRINI, 2019).

Além dos *frameworks* em C++, os demais *frameworks* também ainda não atendem de forma satisfatória as demandas mencionadas na Seção 1.2. Os *frameworks* em Java/C# são versões prototípicas e sem contemplar paralelismo, enquanto as materializações em Elixir/Erlang e Akka contemplam paralelismo, mas não possuem o desempenho apropriado possibilitado pelas linguagens de mais baixo nível. Neste contexto, as propriedades atingidas pelos *frameworks* do PON são resumidas na Tabela 2.

Os *frameworks* C++ 2.0, C++ 3.0 e Java/C# possuem o melhor desempenho dentre os *frameworks* do PON. Em termos de tempo de execução, devido à implementação dos *frameworks* em geral usando estruturas de dados baseadas em linguagens do *POO*, isto gera sobrecarga ou *overheads* que distanciam esta implementação do PON de sua teoria, com cálculos assintóticos na ordem de $O(n)$ para o caso médio e $O(n^3)$ para o pior caso. No caso dos *frameworks*, em determinados cenários, o desempenho ainda é inferior quando comparado a aplicações implementadas diretamente no *POO*. Entretanto, é justamente nos *frameworks* em C++ que, em geral, os resultados ficam próximos ao de *POO*, ainda que não sempre o superando como prevê a teoria, mas sim mantendo programação em mais alto nível e com alto nível de desacoplamento ademais (BANASZEWSKI, 2009; VALENÇA, 2012).

Dadas estas demandas, é possível constatar que existe a necessidade de uma materialização que contemple essas demandas de forma mais satisfatória. Conforme já salientado, os *frameworks* em C++ tendem a ter melhor desempenho quando comparado aos outros *frameworks*

Tabela 2 – Propriedades do PON materializadas pelos frameworks do PON**Fonte:** Autoria própria

<i>Framework</i> Propriedade	C++ 2.0	C++ 3.0	Java/C#	C# IoT	Elixir	Akka
Programação em alto nível				✓	✓	✓
Paralelismo		✓		✓	✓	✓
Desempenho	✓		✓			

do PON, entretanto ainda falta neles maior facilidade de programação concorrente e distribuída, assim como de programação genérica. Entretanto, parte das limitações e problemas existentes no *Framework PON C++ 2.0* e *Framework PON C++ 3.0* podem ser considerados decorrentes das limitações da tecnologia utilizada em suas implementações, mais precisamente, o padrão C++98 da linguagem de programação C++. Nesse âmbito, as inovações introduzidas nos padrões mais novos do C++, principalmente o C++17 e C++20, podem contribuir para possibilitar o desenvolvimento de um *framework* capaz de atender a todas essas demandas.

Dado ao excelente equilíbrio entre nível apropriado de abstração e desempenho apropriado, o C++ é uma linguagem de programação apropriada para se realizar a materialização do PON. Conforme constatado em TIOBE (2021), o C++ continua sendo uma linguagem de programação popular, ademais. Naturalmente, a possibilidade de se utilizar o PON em uma linguagem de programação popular facilitaria a aceitação dele por parte de novos desenvolvedores. Adicionalmente, isto também permitiria a integração com outros programas e bibliotecas já existentes em C++ e associados, de forma mais simples, assim possibilitando o desenvolvimento de aplicações completas.

Ainda, apesar do C++ ser tradicionalmente uma linguagem de programação mais usada no contexto da orientação a objetos, novos recursos adicionados nas suas revisões mais recentes facilitam a programação funcional, por meio do uso das expressões *lambda* (STROUSTRUP, 2020). Isto pode ser muito bem utilizado em materializações do PON em C++ para prover maior nível de flexibilidade de programação (facto-execucional). Neste sentido de flexibilizações, com a aplicação da programação genérica do C++ é possível atingir a flexibilidade de tipos no *framework*. Enquanto no *Framework PON C++ 2.0* se precisava definir e implementar estruturas novas para cada tipo de dado a ser utilizado (como *int*, *double*, *string* e *bool*), em versão orientada a *templates/gabaritos* seria possível utilizar qualquer tipo de dado, pois as interfaces serão genéricas, não sendo necessário código específico para implementar um tratamento diferente para cada tipo de dado de entrada.

Outra melhoria que pode ser obtida ao se aplicar a programação genérica é possibilitar

reduzir a quantidade de código necessário para se compor o *framework* em si. Isso traz benefícios, como facilitar a manutenção do código por outros desenvolvedores e/ou pesquisadores, possibilitando assim uma melhor evolução do *framework*. Facilitar-se-ia adicionar novos recursos por não se fazer necessário reescrever uma grande quantidade de código. Ademais, as contribuições da aplicação de programação genérica permitiriam ainda o desenvolvimento de interfaces mais flexíveis e fáceis de utilizar, no sentido de favorecer a programação em alto nível.

Por fim, com o C++17 também foram introduzidos mecanismos que facilitam a programação concorrente, provenientes de mecanismos de execução paralela, como o conceito de políticas de execução³. Esses mecanismos, enquanto recursos nativos da linguagem, tornam transparente ao desenvolvedor a paralelização da execução de *threads* do *software*, o que vai ao encontro do PON no sentido de permitir *threads* com granularidade fina. Esses recursos permitiriam contornar a deficiência do *Framework PON C++ 3.0* no que diz respeito à estabilidade, conforme reportado inclusive em Martini *et al.* (2019). Por agora serem recursos nativos da linguagem C++, estes mecanismos naturalmente apresentam maior estabilidade que implementações customizadas de paralelização, como a desenvolvida no *Framework PON C++ 3.0*.

Desta forma, em suma, observa-se que é factível o desenvolvimento de uma versão de *framework* em C++ melhor que atenda as propriedades do PON e, portanto, vá ao encontro de todas as demandas de desenvolvimento de *software* supra apresentadas. Em suma, a programação genérica poderia ser utilizada para facilitar a programação em alto nível, o bom desempenho já observado nas materializações de *frameworks* do PON em C++ existentes seria mantido e quiçá melhorado e, por fim, as políticas de execução poderiam ser utilizadas para implementar o suporte ao paralelismo de forma simples e estável.

Apesar da maturidade e estabilidade apresentadas pelo *Framework PON C++ 2.0*, as limitações impostas pelas tecnologias utilizadas em sua implementação, como o antigo padrão C++98 da linguagem C++, a excessiva verbosidade e grande quantidade de código redundante dificultam a implementação de melhorias sobre sua base de código. Desta forma, em suma, há espaço para a implementação de uma nova materialização de *framework*, a qual deve ser desenvolvida fazendo proveito dos avanços no estado da técnica introduzidos pelos *frameworks* já disponíveis, porém realizando implementações mais apropriadas em termos de programação contemporânea de *software* aplicando as melhorias acima propostas.

³ As políticas de execução em C++ permitem especificar se a execução de algoritmos e laços de repetição deve ser executada de forma sequencial ou paralela, esse conceito é explorado em maiores detalhes na Seção 2.7

Além disso, a implementação de determinadas aplicações, como os *benchmarks* propostos por Pordeus *et al.* (2020) por meio dos algoritmos *Random Forest* e *Bitonic Sort* são difíceis, ou quiçá impossíveis, de serem implementados de forma satisfatória com o *Framework PON C++ 2.0*. O *Random Forest* é um algoritmo de aprendizagem de máquina, que pode se beneficiar da paralelização da execução de suas árvores de decisão, o que não seria contemplado por implementações com o *Framework PON C++ 2.0*. Por sua vez, o *Bitonic Sort* é um algoritmo de ordenação, sendo composto por várias etapas de decisão avaliando elementos dois a dois, que podem ser implementadas por várias *Rules* do PON, o qual seria difícil de implementar com o *Framework PON C++ 2.0* devido à complexidade da inicialização de suas entidades. Nesse sentido, a implementação de um novo *framework* do PON possibilitaria a realização destes *benchmarks*.

1.4 OBJETIVO GERAL

O objetivo geral deste trabalho é promover avanços ao Paradigma Orientado a Notificações (PON) via um novo *framework* orientado a TDD, programação genérica e recursos modernos da linguagem de programação C++20 (*single thread* e *multithread*), para melhor contemplar as propriedades elementares do PON no estado da técnica, o que deve ser validado via *benchmarks* apropriados (*e.g.*, *bitonic* e *random forest*).

1.5 OBJETIVOS ESPECÍFICOS

Para atingir o objetivo geral, este trabalho apresenta os seguintes objetivos específicos:

- Desenvolver uma nova versão de *framework* em C++, denominado *Framework PON C++ 4.0*, tendo como base o *Framework PON C++ 2.0* enquanto estado da técnica e considerando as melhorias apresentadas nos demais *frameworks* do PON.
- Prover as seguintes melhorias centrais sobre o *Framework PON C++ 2.0*:
 - (a) Adicionar a flexibilidade de tipos aos *Attributes*;
 - (b) Adicionar flexibilidade algorítmica às *Conditions*;
 - (c) Reduzir a verbosidade da utilização do *framework*;
 - (d) Permitir execução com paralelismo.

- Validar o *framework* desenvolvido por meio de técnicas de teste no âmbito do desenvolvimento orientado a testes (TDD — *Test Driven Development*).
- Desenvolver aplicação de simulação de sistema de sensores, oriunda do grupo de pesquisas do PON, para avaliar o desempenho do *Framework PON C++ 4.0* em ambientes *monocore*, comparando com o desempenho das implementações com o *Framework PON C++ 2.0* e com o POO em linguagem C++.
- Desenvolver aplicação com os algoritmos *Bitonic Sort* e *Random Forest*, oriundos da literatura, para avaliar o desempenho do *Framework PON C++ 4.0* em ambientes monocore vis-à-vis o PP em linguagem C, bem como para balanceamento de carga em *multicore*.
- Desenvolver aplicação de simulação de controle de tráfego automatizado, oriunda do grupo de pesquisas do PON, para a avaliar o desempenho do *Framework PON C++ 4.0* em ambientes *multicore*, comparando com o desempenho das implementações com o *Framework PON Erlang/Elixir*.
- Desenvolver aplicação sob a forma de um jogo para avaliar a utilização do *Framework PON C++ 4.0* em integração com bibliotecas externas (*i.e.*, Unreal Engine) em linguagem de programação C++.

1.6 ORGANIZAÇÃO DO TRABALHO

Estra trabalho está dividido em cinco capítulos, incluindo este capítulo de introdução. No Capítulo 2 é feita uma revisão do estado da arte das tecnologias utilizadas no desenvolvimento deste trabalho, apresentando de forma sucinta as materializações do PON, ferramentas de C++ moderno e ferramentas de teste. No Capítulo 3 é apresentado o desenvolvimento realizado neste trabalho. Mais precisamente, é apresentada a modelagem do *Framework PON C++ 4.0* proposto. O capítulo 4, por sua vez, apresenta os resultados de experimentos realizados com o objetivo de avaliar o *Framework PON C++ 4.0* desenvolvido. Por fim, o Capítulo 5 apresenta as conclusões sobre este trabalho, assim como explora as perspectivas para trabalhos subsequentes no âmbito da aplicação do *Framework PON C++ 4.0*.

2 REVISÃO DO ESTADO DA ARTE E DA TÉCNICA

Este capítulo explora o estado da arte e da técnica do Paradigma Orientado Notificações (PON), incluindo os frameworks do PON em C++. Tal qual, o capítulo também explora a linguagem de programação C++, incluindo conceitos de C++ contemporâneo, dito moderno, e ainda os *frameworks* para fins de testes em C++.

Inicialmente na Seção 2.1 é feita uma breve revisão sobre os paradigmas atuais introduzidos no Capítulo 1, seguida por uma revisão do estado da arte do PON na Seção 2.2, que contextualiza os conceitos fundamentais e propriedades elementares. Por sua vez, a Seção 2.3 apresenta os conceitos de programação ou desenvolvimento em PON, que são pertinentes ao desenvolvimento das materializações do PON. A Seção 2.4 discorre sobre as diversas materializações em *software* do PON, salientando os frameworks em C++, também fazendo as devidas reflexões sobre as mesmas no contexto da proposta deste trabalho. Ainda a Seção 2.5 aborda os conceitos de testes unitários e de integração no contexto do PON. Por fim, concluindo a revisão do PON, a Seção 2.6 faz um breve levantamento quantitativo dos trabalhos desenvolvidos no contexto do PON.

No âmbito da linguagem de programação C++, a Seção 2.7 faz uma revisão do estado da arte do C++, na qual são explorados os conceitos de C++ dito moderno. Por sua vez, a Seção 2.5 apresenta os conceitos teóricos de testes no PON. Ainda, a Seção 2.8 faz um levantamento dos *frameworks* de teste, que serão utilizados durante o desenvolvimento do *Framework* PON C++ 4.0. Por fim, a Seção 2.9 faz reflexões sobre os problemas em aberto do PON, com foco no contexto deste trabalho que envolve o de desenvolvimento de um *framework*.

2.1 PARADIGMAS DE PROGRAMAÇÃO ATUAIS

Esta seção tem o objetivo de funcionar como um suporte para familiarizar o leitor com os paradigmas de programação mencionados no Capítulo 1. Isto dito, a leitura do mesmo poderia ser dispensada ou lida transversalmente por conhecedores do assunto. Em suma, a seção introduz conceitualmente cada um dos paradigmas, apresenta um mesmo exemplo de uso em diferentes paradigmas e reflete sobre as principais características deles.

A seção começa com os paradigmas dominantes, nominalmente o Paradigma Imperativo (PI) e o Paradigma Declarativo (PD), bem como seus subparadigmas Procedimental (PP) e

Orientado a Objetos (POO) do PI e o Funcional (PF) e Lógico (PL) do PD. Subsequentemente, apresentam-se os paradigmas emergentes, que não raro se materializam nos dominantes, mas os provendo com novas conotações, estratégias e afins. Na Figura 10 é mostrada a classificação dos paradigmas de programação incluindo paradigmas emergentes, destacando como o Paradigma Orientado a Eventos (POE - *Event-Driven Programming*) é frequentemente materializado com o POO, enquanto a programação baseada em regras (*Rule Based Programming*) é materializada com o PL.

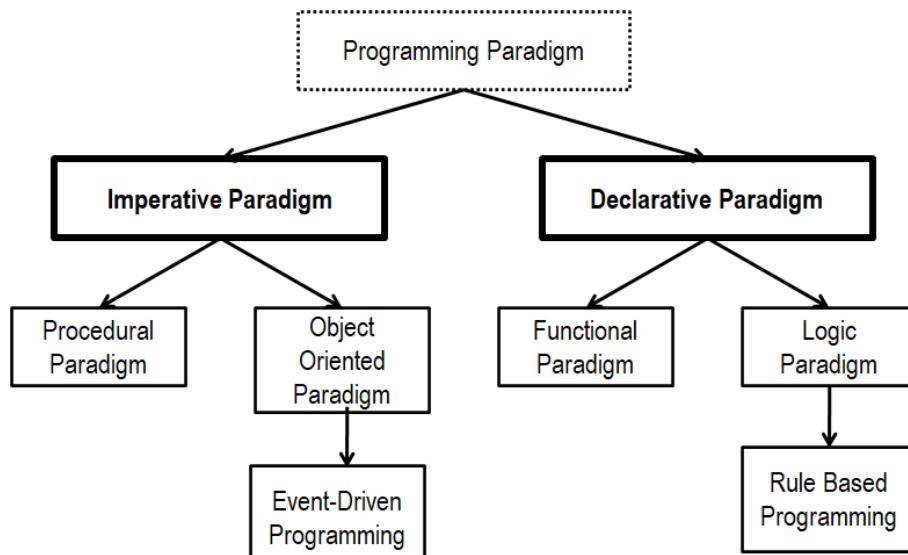


Figura 10 – Classificação dos paradigmas de programação com os paradigmas emergentes
Fonte: Autoria própria

2.1.1 Paradigmas Imperativos

Nesta seção busca-se apresentar os paradigmas imperativos, ou subparadigmas imperativos conforme o ponto de vista, mencionados na Seção 2.1, de modo a estabelecer a sua relação e características em comum, assim como suas peculiaridades. Os subparadigmas que seguem os princípios do Paradigma Imperativo (PI) se caracterizam pela flexibilidade de programação e pela forma explicitamente sequencial pela qual as instruções são executadas (BANASZEWSKI, 2009). O Paradigma Procedimental (PP) foi o primeiro, propriamente dito, paradigma de programação proposto, o qual faz parte do âmago do PI. Posteriormente surge uma forma mais avançada de desenvolvimento em PI, conhecida como Paradigma Orientado a Objetos (POO) (BANASZEWSKI, 2009). O PP e o POO se diferenciam pela forma como os elementos e instruções são organizados, sendo o POO considerado mais rico e estruturado em termos de abstração e expressão do código (RONSZCKA, 2019).

2.1.1.1 Paradigma Procedimental (PP)

O Paradigma Procedimental (PP) define um estilo de programação baseado na organização sequencial de variáveis e comandos, na qual as variáveis representam os estados das entidades e os comandos executam ações sobre estes estados (WATT, 2004; BROOKSHEAR, 2006). O PP ainda vigora pelos motivos de considerável coeficiente de modularização, certa simplicidade de programação em sistemas de menor escala e relativa eficiência, uma vez que a forma de codificação dos comandos condiz com a forma pela qual os mesmos são executados pela máquina. Este fato facilita a geração de executáveis enxutos e eficientes (BANASZEWSKI, 2009). Outro fator que contribui para a permanência deste paradigma é a experiência adquirida há anos pelos programadores neste tipo de linguagem, o que é aqui chamado de inércia cognitiva. Neste sentido, alguns exemplos clássicos de linguagens baseadas no PP são FORTRAN, COBOL, ALGOL, BASIC, PASCAL e C (BANASZEWSKI, 2009).

Como exemplo de programa no PP, o Código 2.1 apresenta um programa que executa a lógica de tomada decisão sobre o estado de um dado uma aplicação de sensor, cujo funcionamento já foi descrito na Seção 1.1.3 sob a forma de *Rules*. Esse exemplo é também utilizado nas seções seguintes de forma a ilustrar o uso dos diversos paradigmas. Esta implementação no PP é feita de forma simples, na clássica linguagem de programação C, implementando o *Sensor* como uma *struct* (estrutura) com duas variáveis membros, sendo declaradas funções auxiliares para alterar os valores destas variáveis da *struct* utilizando ponteiros. Adicionalmente, a verificação dos estados do *Sensor* é feita por meio de um laço de repetição na função *main*.

Em que pese suas vantagens, o PP apresenta certas deficiências, de tal forma que subsequentemente e mais recentemente, o PP perdeu espaço considerável para o POO. O PP é limitado no que diz respeito à sua capacidade de representar abstrações, não possuindo ferramental que realmente induza à coesão de variáveis e procedimentos diretamente relacionados, por exemplo. O PP só possui um único mecanismo explícito de modularização, para favorecer a coesão, que é a divisão do programa em rotinas. Neste sentido, novas ferramentas que aumentam a capacidade de abstração do PP são introduzidas com o Paradigma Orientado a Objetos (POO) (AVACHEVA; PRUTZKOW, 2020).

Código 2.1 – Exemplo de aplicação de sensor em C

```

struct Sensor {
    bool isRead;
    bool isActivated;

    Sensor() {
        isRead = false;
        isActivated = false;
    }
};

void activate_sensor(Sensor* p_sensor) {
    p_sensor->isActivated = true;
    p_sensor->isRead = false;
}

void process_sensor(Sensor* p_sensor) {
    p_sensor->isActivated = false;
    p_sensor->isRead = true;
}

int main() {
    Sensor sensor;
    while (true) {
        if (sensor.isActivated && !sensor.isRead) {
            process_sensor(&sensor);
        }
    }
    return 0;
}

```

Fonte: Autoria própria

2.1.1.2 Paradigma Orientado a Objetos (POO)

O Paradigma Orientado a Objetos (POO) apresenta um nível de abstração mais alto que o PP. Os programas desenvolvidos em POO são compostos por entidades modulares denominadas objetos, que podem ser entendidas como abstrações de uma entidade real ou imaginária, contendo as características pertinentes para sua implementação computacional (POO *et al.*, 2007). Esses objetos agrupam atributos (similar às variáveis do PP) e métodos (similar às funções do PP), organizados de maneira a estimular coesão e desacoplamento, tudo à luz de seus tipos ou classes que ditam o modelo de suas instâncias, incluindo os seus relacionamentos para com outros objetos (WATT, 2004; BROOKSHEAR, 2006; PRESSMAN; MAXIM, 2016; SANTOS, 2017). Exemplos de linguagens que materializam este paradigma são a SIMULA, Smalltalk, C++, JAVA e C# (BANASZEWSKI, 2009).

Como exemplo do POO, no Código 2.2 é apresento um programa em C++ que executa a lógica de tomada decisão sobre o estado de um dado uma aplicação de sensor, cujo funcionamento é descrito na Seção 1.1.3 sob a forma de *Rules*. Esse exemplo foi também apresentado anteriormente como exemplo de implementação para o PP. A diferença entre a aplicação do POO e do PP é justamente o encapsulamento da entidade *Sensor*, agora implementada por meio de uma classe com métodos próprios.

Código 2.2 – Exemplo de aplicação de sensor em C++

```

class Sensor {
public:
    bool isRead{false};
    bool isActivated{false};

    void Activate() {
        isActivated = true;
        isRead = false;
    }
    void Process() {
        isActivated = false;
        isRead = true;
    }
};

int main() {
    Sensor sensor;
    while (true) {
        if (sensor.isActivated && !sensor.isRead) {
            sensor.Process();
        }
    }
    return 0;
}

```

Fonte: Autoria própria

2.1.1.3 Considerações Sobre os Paradigmas Imperativos

Programas desenvolvidos com POO ou com o PP são concebidos como sequências de instruções. Esse mecanismo de execução sequencial consiste em buscas, por assim dizer, ou, mais precisamente, percorimentos sobre entidades passivas (as variáveis, os atributos e afins, vetores destes, estruturas de dados desses e afins) que correspondem aos dados. Isto se dá por meio de comandos de decisão (*e.g.*, *if-else* e *switch* em linguagem C/C++), sendo estes executados em laços de repetição (*e.g.*, *for*, *while*, *do-while* em linguagem C/C++) (SANTOS, 2017).

Devido à sequencialidade da busca/percorimentos e à passividade dos elementos presente nas linguagens do PP e do POO, trechos de código tendem a se tornar interdependentes, levando a acoplamentos, bem como há ainda correlatos problemas de redundância na execução dos programas. Neste âmbito, as expressões causais são avaliadas passivamente ocasionando as chamadas redundâncias temporais e estruturais (BANASZEWSKI, 2009). A redundância temporal, ocorre quando há a reavaliação de expressões causais desnecessárias na presença de estados já avaliados e inalterados, enquanto redundância estrutural ocorre quando o conhecimento sobre um estado resultante da avaliação de uma expressão lógica não é compartilhado entre outras expressões causais pertinentes em diferentes partes do código, causando reavaliações desnecessárias. No pseudocódigo do Código 2.3 a redundância temporal é observada na linha 2, pelo laço de repetição *while(true)*, enquanto a redundância estrutural é exemplificada nas linhas 3 e 12, nas quais o mesmo atributo *attribue_1* do *object_1* é avaliado redundantemente nos dois *ifs*.

Estes problemas podem afetar o desempenho dos programas desenvolvidos (BANASZEWSKI, 2009).

Código 2.3 – Exemplo de redundâncias estruturais e temporais

```

1 ...
2 while (true) do
3   if ((object_1.attribute_1 == 1) and
4     (object_1.attribute_2 == 1) and
5     (object_1.attribute_3 == 1))
6   then
7     object_1.method_1();
8     object_2.method_1();
9     object_2.method_1();
10 ...
11 end_if
12 if ((object_1.attribute_1 == 1) and
13   (object_1.attribute_n == n) and
14   (object_1.attribute_n == n))
15 then
16   object_1.method_n();
17   object_2.method_n();
18   object_2.method_n();
19 ...
20 end_if
21 end_while
22 ...

```

Fonte: Autoria própria

2.1.2 Paradigmas Declarativos

Da mesma forma como foi mostrado na Seção 2.1.1 em relação aos Paradigmas Imperativos, esta seção apresenta os Paradigmas Declarativos, de modo a estabelecer a sua relação e características em comum assim como suas peculiaridades. Os subparadigmas que seguem o PD, por sua vez seriam o Paradigma Funcional (PF) e particularmente o Paradigma Lógico (PL), sabendo que, na verdade, o PF estaria em uma fronteira com PI.

Em todo caso, os subparadigmas PF e PL do PD caracterizam-se e diferenciam-se dos subparadigmas do PI puro ao apresentar modelo menos flexível, porém mais simplificado de programação. Mesmo no PF e certamente no PL, o programador se concentraria mais na organização do conhecimento em alto nível sobre a resolução do problema computacional em si ao invés da implementação mais técnica propriamente dita (RONSZCKA, 2019).

Enquanto o PI em geral impõe pesquisas orientadas a laços de repetições sobre elementos passivos, relacionando dados a expressões causais, o PD possui soluções declarativas que trabalham por recursões e/ou mecanismos de inferência. Inclusive, alguns destes mecanismos no PL permitem evitar parte das redundâncias de execução, conforme discutido no decorrer desta seção (RONSZCKA, 2019).

2.1.2.1 Paradigma Funcional (PF)

O Paradigma Funcional (PF) é baseado no conceito de funções computáveis por meio do cálculo *lambda*, que é um conceito que foi matematicamente introduzido em 1936 por Alonzo Church (BARENDEGT; BARENSEN, 1984). A essência do PF é a construção de programas a partir da manipulação de dados por meio de funções, em um modelo no qual as funções podem invocar outras funções ou até mesmo serem passadas como parâmetros para outras funções (SCOTT, 2000; BANASZEWSKI, 2009). As funções do PF são ditas puras, no sentido que seu resultado depende apenas da sua entrada, devido às funções não possuírem estado compartilhado interno (SCOTT, 2000).

No Código 2.4 o mesmo exemplo da aplicação de sensores é apresentado, desta vez implementado no Paradigma Funcional por meio do uso da linguagem Clojure, que é um dialeto dinâmico e funcional da linguagem de programação Lisp.

Uma das vantagens desse paradigma é permitir a programação em mais alto nível que o PP e POO puras justamente por permitir flexibilidades como a passagem de funções como parâmetros. Além disso, o uso de funções puras facilita o paralelismo, pois não existe estado compartilhado (SCOTT, 2000; BHADWAL, 2020). Em contrapartida, a falta de estado compartilhado combinado com recursão pode e tende a causar redução no desempenho do programa (BHADWAL, 2020). Adicionalmente, ainda que o estilo de programação funcional supostamente facilite escrever as funções, ele dificulta a integração com outras áreas do código, principalmente operações de entrada e saída de dados, como no caso interfaces e arquivos (BHADWAL, 2020).

Alguns exemplos de linguagens baseadas no PF são LISP, Clojure, Miranda, Haskell, Sisal, pH e ML (BANASZEWSKI, 2009). Além destas linguagens, C++11 em diante, C# 3.0 e Java 8 também adicionaram construções que facilitam o desenvolvimento de programas no estilo funcional, ainda que de forma híbrida e dominada por seus paradigmas de origem (BHADWAL, 2020).

2.1.2.2 Paradigma Lógico (PL)

Apesar dos subparadigmas apresentados anteriormente (PP, POO e mesmo PF) se basearem em teorias diferentes, eles acabam compartilhando a mesma forma de execução em alguma medida. Mais precisamente, em todos eles, um programa ou mesmo uma função, recebe

Código 2.4 – Exemplo de aplicação de sensor em Clojure

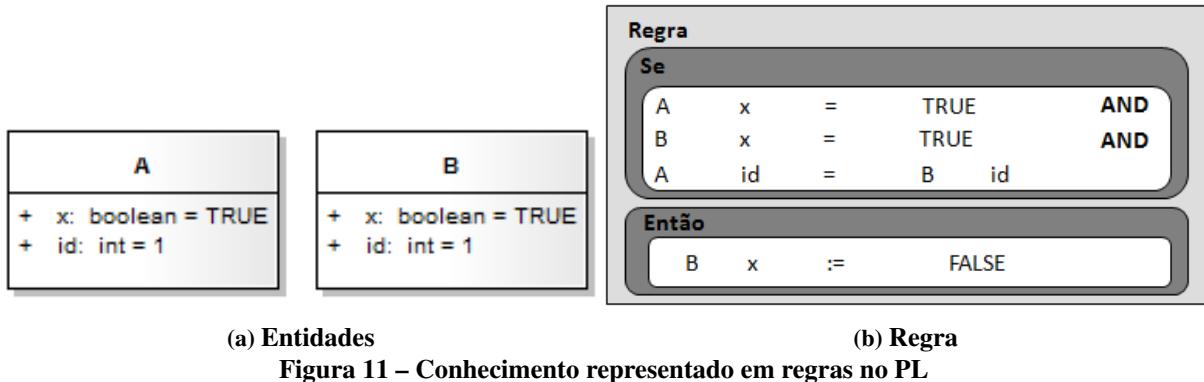
```
;; Autoria: Gustavo Brunholi Chierici - Data 22/06/2021 - Grupo de Pesquisa: PON - UTFPR
(defrecord FPSensor [is-read is-activated]);; Record ("struct") do sensor
(defn create-sensor [] (FPSensor. false false));; Função que retorna um sensor
(defn activate-sensor ;; Função que "ativa" o sensor
  [sensor]
  (assoc sensor :is-activated true :is-read false))
(defn deactivate-sensor ;; Função que "desativa" o sensor
  [sensor]
  (assoc sensor :is-activated false))
(defn read-sensor ;; Função que "lê" o sensor
  [sensor]
  (assoc sensor :is-read true))
(defn check-sensor ;; Função que verifica se o sensor está ativado e o "lê" em caso positivo
  [sensor]
  (if (and (:is-activated sensor) (not (:is-read sensor)))
    (deactivate-sensor (read-sensor sensor)) sensor))
```

Fonte: Autoria própria

dados como entrada, realiza computações e gera uma saída via um mapeamento direto interno. Este processo é definido como um mapeamento da entrada para a saída do programa (WATT, 2004; BANASZEWSKI, 2009).

No PL, além das entidades com suas variáveis e comandos usuais (*i.e.*, base de fatos) em formas como frames (aparentados de objetos, mas com métodos suspostamente simples ou pragmáticos), os comandos lógicos causais também são representados na forma de dados, sob o viés de regras (*i.e.*, base de regras) conforme Figura 11. Não raro, linguagens e ambientes do PL são chamados Sistemas Baseados em Regras (SBRs) justamente. Em todo caso, essa relação entre base de fatos e base de regras, buscando aprovar/desaprovar regras pelas suas confrontações, dá-se pela busca realizada por meio de algum mecanismo de inferência sobre as definições de elementos factuais (e.g., frames) da base de fatos vis-à-vis expressões causais (normalmente regras) da base de regras, conforme Figura 12 (BANASZEWSKI, 2009). Isto dito, alguns exemplos de linguagens que implementam o PL são Oz (ROY; HARIDI, 2004), Prolog, RuleWorks e OPS (BANASZEWSKI, 2009).

Ainda, há linguagens do PL, como OPS e Ilog Rules, que tem como base algoritmos de inferência otimizados como o RETE (FORGY, 1982). Outros exemplos de algoritmos de inferência otimizados são TREAT (MIRANKER, 1987) e o LEAPS (MIRANKER; BRANT, 1990) derivados do RETE, bem como o alternativo o HAL (LEE; CHENG, 2002). Sem a aplicação destas soluções eficientes ciclos de inferência se tornariam muito lentos, inviabilizando a sua utilização em muitos casos. Entretanto, mesmo com algoritmos de inferência considerados



eficientes, ainda há redundância de processamentos e preço de representar base de fatos e base de regras em estrutura de dados (SIMÃO, 2005; BANASZEWSKI, 2009).

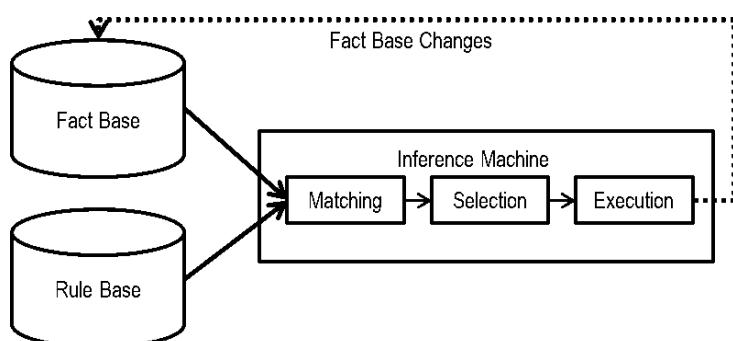


Figura 12 – Arquitetura interna de um Sistema Baseado em Regras
Fonte: Banaszewski (2009)

O RETE em particular implementa um processo de busca otimizado, manipulando elementos da base de fatos e regras sobre uma estrutura baseada em redes ou grafos. Esse processo é ilustrado de forma simplificada na Figura 13. Mais precisamente, o RETE guarda informação sobre avaliações anteriores das regras e também avalia as regras somente quando a base de fatos é atualizada de modo a evitar avaliações redundantes, ou seja, quando um elemento da base de fatos é inserido/modificado/removido da Base de Fatos. Com isto, o RETE resolve o problema das redundâncias temporais do PI. Esta solução se faz possível porque o RETE é composto de duas sub-redes: a α -network é composta por nós sendo cada qual responsável pela avaliação lógica de fatos; a β -network, por sua vez, é composta por outros nós, sendo cada qual responsável pela correlação entre fatos. Se as avaliações e correlações forem satisfeitas para uma regra, a mesma é aprovada sendo, portanto, inserida no conjunto de conflito (FORGY, 1982; DOORENBOS, 1995; BANASZEWSKI, 2009)

No Código 2.5 novamente o exemplo da aplicação de tomada de decisão sobre um sensor é apresentado, desta vez implementado no Paradigma Lógico com a linguagem de programação

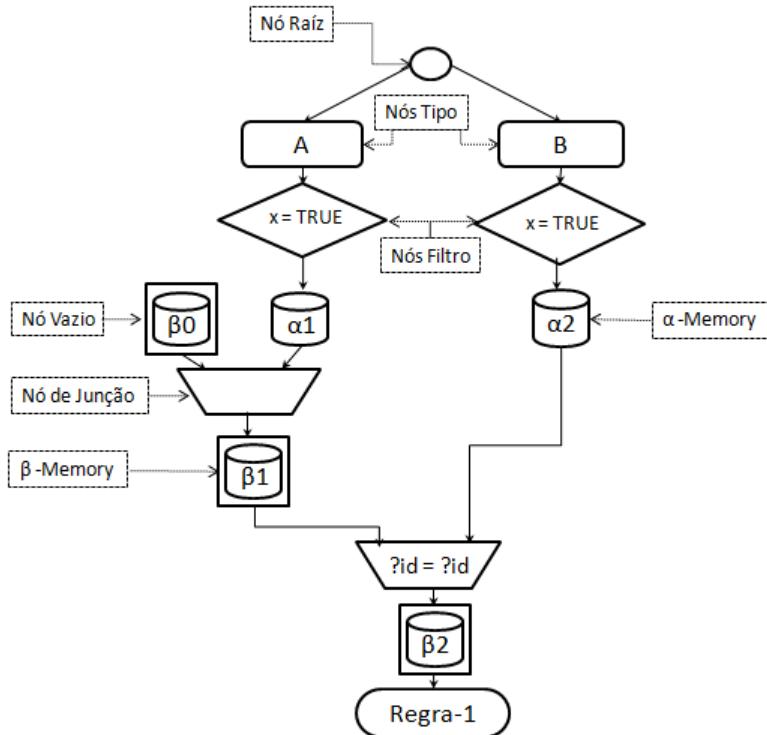


Figura 13 – Estrutura do algoritmo RETE
Fonte: Banaszewski (2009)

RuleWorks. Observa-se que, por se tratar de uma linguagem de programação cuja estrutura se baseia em regras, o código tem apresenta muita similaridade ao representado sob a forma de *Rule* na Figura 6.

Código 2.5 – Exemplo de aplicação de sensor em RuleWorks

```
(object-class sensor
  ^is-read
  ^is-activated
)

(rule process-sensor:sensor
  (sensor ^$ID <the-sensor> ^is-read <false> ^is_activated <true>)
  -->
  (bind <the-sensor ^is-read> (false))
  (bind <the-sensor ^is-activated> (false))
)
```

Fonte: Autoria própria

2.1.2.3 Considerações Sobre os Paradigmas Declarativos

O PD se propõe a resolver algumas das deficiências do PI, principalmente aquelas relacionadas à dificuldade de programação em ambientes *multicore*, atualmente no caso do PL contemporâneo, e redundâncias estruturais e temporais, no caso do PL/SBR com máquinas de inferência apropriadas. Apesar disso, a maioria destas deficiências ainda se repete devido aos

mecanismos de execução tanto do PF como do PL que se baseiam em buscas sob elementos passivos, que por sua vez ou reproduzem estes mesmos problemas de redundâncias estruturais e temporais e/ou os atenuam com algoritmos inferência dito otimizados, mas trocando os problemas por estruturas de dados caras. Assim, normalmente programas em PD são mais lentos que em PI, apesar das redundâncias deste (BANASZEWSKI, 2009).

Além disso, enquanto as linguagens de programação do PD apresentam um modelo de programação mais simplificado, elas também não oferecem a mesma flexibilidade das linguagens do PI. Isto pode dificultar a construção de códigos mais eficientes, visto que limita o controle que o programador tem sobre o *software*. Finalmente, mesmo no caso de PF, tanto em PD quanto PI em geral, a orientação a buscas e/ou percorimentos deles não permite alcançar facilmente módulos desacoplados, particularmente em nível granular, o que dificulta processamentos paralelo e/ou distribuídos (BANASZEWSKI, 2009).

2.1.3 Paradigmas Emergentes

Em suma, como já dito, os paradigmas dominantes PI (com PP e POO) e PD (com PF e PL) tendem a ter problemas similares de redundâncias e/ou processamentos e acoplamentos. Isto dificulta alcançar demandas contemporâneas como a facilitação de programação em alto nível, boa performance de processamento e boa performance do paralelismo/distribuição de processamento em arquiteturas com múltiplos núcleos, por exemplo.

Outrossim, além dos paradigmas dominantes, há os paradigmas emergentes que não raro se materializam sobre os dominantes. Muito embora emergentes conhecidos não tenham resolvidos esses problemas relatados dos dominantes, eles sim trazem novas conotações de desenvolvimento. Exemplos de paradigmas emergentes (talvez mais conhecidos) são como Paradigma Orientado a Componentes (POC), Paradigma Orientado a Eventos (POE), Paradigma Orientado a Aspectos (POA), Paradigma Orientado a Agentes (POAg) e Paradigma Orientado a Atores (POAt).

2.1.3.1 Paradigma Orientado a Componentes (POC)

O Paradigma Orientado a Componentes (POC) se baseia no reuso e combinação de componentes. Um componente consiste em uma parte do programa desenvolvida de forma a ser altamente reutilizável, que pode ser desenvolvida independentemente para ser depois combinada

com outras partes objetivando construir uma unidade maior (D'SOUZA; WILLS, 1998).

Na Figura 14 um diagrama de componentes é exemplificado, no qual os componentes se relacionam para realizar o fluxo de execução do programa. As setas direcionais representam a invocação de serviços oferecidos por estes componentes. Os serviços não são acessados diretamente, mas por meio de entidades descritoras, chamadas de interfaces. O uso de interfaces é uma característica do POC que permite que a definição dos serviços seja feita completamente separada da implementação (CRNKOVIC; LARSSON, 2002)

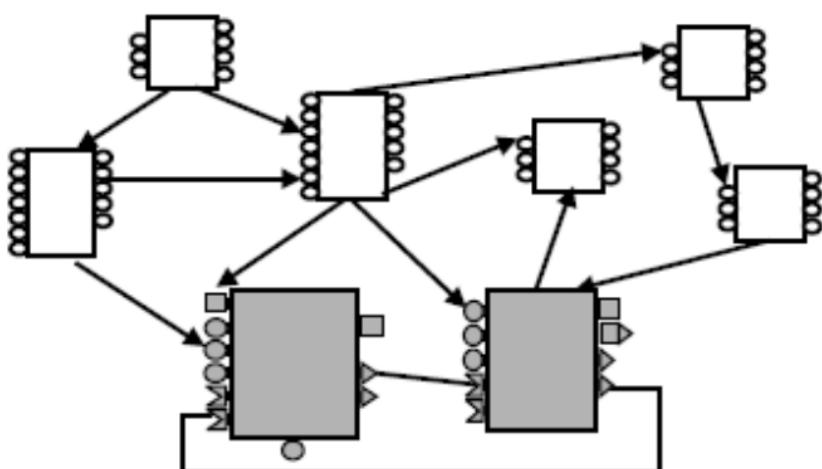


Figura 14 – Diagrama de componentes
Fonte: Crnkovic e Larsson (2002)

O POC é muito utilizado em *frameworks* de programação de *front-end*, como React¹, Angular² e VUE³. No Código 2.6, é mostrado o exemplo de tomada de decisão sobre um sensor em linguagem de programação JavaScript com o *framework* Angular, na qual pode ser observado que a implementação do sensor importa e implementa a funcionalidade do componente *OnInit*.

A principal desvantagem do POC é que o foco na reusabilidade dos componentes tende a aumentar a complexidade deles, de modo que se torna necessário a execução de processos mais cuidadosos de planejamento do sistema e manutenção dos componentes. Quanto mais reutilizável for o componente, mais complexo ele se torna e mais difícil é a sua manutenção (CRNKOVIC; LARSSON, 2002). Por fim, internamente o POC é, na prática, o encapsulamento de módulos criados via paradigmas já existentes.

¹ Para maiores detalhes sobre este *framework*, consultar <https://reactjs.org/>

² Para maiores detalhes sobre este *framework*, consultar <https://angular.io/>

³ Para maiores detalhes sobre este *framework*, consultar <https://vuejs.org/>

Código 2.6 – Exemplo de aplicação de sensor em React no POC

```

/* Autoria: Prof. Dr. Roni Fabio Banaszewski - Data 24/06/2021 - Grupo de Pesquisa: PON -
UTFPR */

import {Component, OnInit} from '@angular/core';
import {BehaviorSubject} from "rxjs";

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent implements OnInit {
  title = 'sensor-app';

  ready: boolean = false;
  active: boolean = true;

  private readySubject = new BehaviorSubject<boolean>(this.ready);
  private activeSubject = new BehaviorSubject<boolean>(this.active);

  ngOnInit() {
    this.readySubject.subscribe(value => {
      this.ready = value;
      this.rule()
    });

    this.activeSubject.subscribe(value => {
      this.active = value;
      this.rule()
    });
  }

  rule() {
    if (this.ready == false && this.active == true) {
      this.readySubject.next(true);
      this.activeSubject.next(false);
    }
  }
}

```

Fonte: Autoria própria

2.1.3.2 Paradigma Orientado a Eventos (POE)

No Paradigma Orientado a Eventos (POE) as entidades interagem por meio da ocorrência de eventos. Um evento consiste em uma condição detectável que pode instigar a execução de um método. O objeto capaz de detectar a condição e gerar o evento é chamado de objeto transmissor, enquanto os objetos interessados que recebem um evento são chamados de objetos receptores (FERG, 2006).

A Figura 15 ilustra a relação entre os objetos transmissores e receptores, que podem ocorrer de forma direta ou indireta. No exemplo à esquerda na Figura 15, um objeto transmissor detecta a ocorrência de um evento e notifica este evento diretamente aos objetos receptores interessados. No exemplo à direita na Figura 15, o evento é transmitido por intermédio do objeto *Dispatcher* (XAVIER, 2014). A transmissão do evento ocorre por meio da passagem de um *token*, o qual pode ser representado por uma mensagem formada por uma cadeia de caracteres

ou um objeto com dados (FERG, 2006).

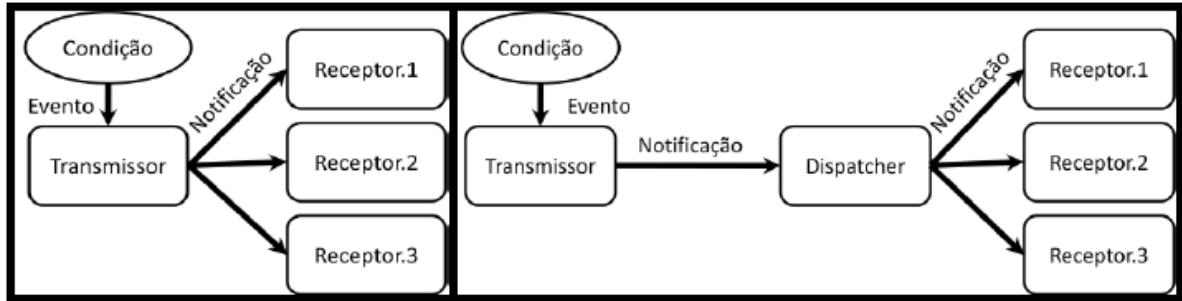


Figura 15 – Processo de detecção de eventos

Fonte: FAISON (2006)

Na verdade, não existe linguagem de programação que implementa especificamente o POE, sendo ele incorporado por *frameworks* e afins nas linguagens dos paradigmas vigentes. Por exemplo, um *framework* muito popular para C++ e Python que implementa a orientação a eventos é o Qt⁴. Em suma, o Qt é um *framework* para a criação de aplicações gráficas de forma intuitiva, usando técnica para tal conhecida como *Rapid Application Development* (RAD). Neste *framework*, um caso muito comum para o uso de eventos é mostrado no Código 2.7, no qual os eventos são enviados para ativar o sensor ao pressionar um botão, ao conectar o evento *button.clicked* às funções *Activate()* e *Process()*.

Código 2.7 – Exemplo de aplicação de sensor com PyQt5 no POE

```

class Sensor(QWidget):
    isRead = False
    isActivated = False
    button = QPushButton('Activate')

    def Activate(self):
        self.isActivated = True
        self.isRead = False
    def Deactivate(self):
        self.isActivated = False
        self.isRead = False
    def Process(self):
        if(self.isActivated and not self.isRead):
            alert = QMessageBox(self)
            alert.setText('Sensor Processed')
            alert.show()
            self.Deactivate()
    def __init__(self):
        super().__init__()
        self.button.clicked.connect(self.Activate)
        self.button.clicked.connect(self.Process)
        self.button.show()

```

Fonte: Autoria própria

Dentre os Paradigmas Emergentes mencionados na Seção 1.1.2, o POE se destaca pelo seu estado de maturidade e extenso uso em diversas aplicações, desta forma podendo

⁴ Para maiores detalhes sobre este *framework*, consultar <https://www.qt.io/>

ser considerado um paradigma em fase de transição de Paradigma Emergente para Paradigma Dominante. Dentre as vantagens do POE estão a flexibilidade e simplicidade de programação devido ao mecanismo de eventos, que facilita a interação entre entidades no código, como o botão *Activate* com o sensor do exemplo. Em contrapartida, o código pode ser considerado confuso, devido ao fluxo de execução do programa, o que também dificulta encontrar erros no programa quando comparado a paradigmas com fluxo de execução mais simples como o do PI ou com lógica em bases de conhecimento unívocas como em SBR/PL do PD.

2.1.3.3 Paradigma Orientado a Aspectos (POAs)

O Paradigma Orientado a Aspectos (POAs) permite separar as funcionalidades principais de funcionalidades secundárias de um código típico orientado a objetos, encapsulando as funcionalidades secundárias em uma unidade modular chamada de aspecto. No contexto do POA, a funcionalidade principal é uma funcionalidade única implementada por determinada função ou método, enquanto as funcionalidades secundárias são genéricas e compartilhadas entre várias funções ou métodos (LADDAD, 2003).

O POAs propõe essa separação por meio do encapsulamento das funcionalidades secundárias, sob a forma de aspectos, para serem invocadas implicitamente (sem a declaração da chamada dos métodos explicitamente no código) pelas funcionalidades principais. Os aspectos são invocados pela chamada dos métodos que implementam as funcionalidades principais (MILES, 2004). A vantagem da aplicação desse mecanismo seria maior incentivo ao reuso de código e maior facilidade de entendimento ou alteração do código (MILES, 2004).

Um exemplo de linguagem de programação na qual pode ser realizada a aplicação do POA é Python, que conta com suporte nativo da linguagem ao POA por meio dos chamados *decorators*. No exemplo do Código 2.8, no qual é implementado o exemplo da aplicação de sensor, essa ferramenta é utilizada para encapsular a verificação dos estados dos atributos do sensor por meio do aspecto *checker_aspect*.

2.1.3.4 Paradigma Orientado a Agentes (POAg) / Atores (POAt)

O Paradigma Orientado a Agentes (POAg) permite construir programas empregando entidades chamadas agentes. Os agentes são entidades computacionais capazes de executar ações de maneira flexível e autônoma, de modo a atingir objetivos estabelecidos no momento da sua

Código 2.8 – Exemplo de aplicação de sensor em Python no POAs

```
# Autoria: Marcos Talau e Felipe Neves - Data 26/06/2021 - Grupo de Pesquisa: PON - UTFPR
def checker_aspect(func):
    def wrapper(self):
        func(self)
        if(self.isActivated and not self.isRead):
            self.Process()
    return wrapper

class Sensor:
    isRead = False
    isActivated = False

    @checker_aspect
    def Activate(self):
        self.isActivated = True
        self.isRead = False
    @checker_aspect
    def Deactivate(self):
        self.isActivated = False
        self.isRead = False
    def Process(self):
        self.Deactivate()

Fonte: Autoria própria
```

concepção (JENNINGS, 1999).

Os agentes são usualmente implementados sobre os conceitos do POO, porém agentes possuem propriedades mais complexas do que um objeto. Diferente dos objetos, que atuam passivamente até que um de seus métodos sejam invocados, agentes podem atuar de forma proativa, tendo objetivos individuais ou coletivos. Um objeto aguarda até que seja lhe indicado o que fazer, enquanto o agente decide por si só quando e o que deve ser feito. Ainda, em POAg as relações entre os entes acontecem na dinâmica do sistema, de maneira fluida e não tudo pré-definido como normalmente ocorre em POO (BANASZEWSKI, 2009).

No Código 2.9, é mostrado o exemplo de tomada de decisão sobre um sensorem linguagem de programação Elixir/Erlang. Neste exemplo simples, pode ser observada a complexidade presente devido às funcionalidades básicas necessárias para a comunicação entre agentes.

O Paradigma Orientado a Atores (POAt) é consideravelmente similar ao POAg, sendo o modelo de agentes uma extensão do modelo de atores. A diferença entre agentes e atores é que os agentes são tipicamente mais complexos e capazes de tomada de decisão lógica (CARDOSO *et al.*, 2013). As principais linguagens do POAt são Akka e Erlang. Ambos os modelos do POAt e POAg trazem a vantagem de facilitar o paralelismo devido ao alto desacoplamento entre as entidades, enquanto uma das principais desvantagens é o alto tempo de execução da comunicação entre os atores/agentes.

Código 2.9 – Exemplo de aplicação de sensor em Elixir no POAg

```
# Autoria: Fabio Negrini - Data 11/08/2021 - Grupo de Pesquisa: PON - UTFPR
defmodule Sensor do use GenServer
  def create() do
    state = init_state()
    {:ok, pid} = GenServer.start_link(__MODULE__, state, name: __MODULE__)
    pid
  end
  defp init_state() do
    %{:is_read => false,
      :is_activated => false}
  end
  @impl true
  def init(stack) do
    {:ok, stack}
  end
  @impl true
  def handle_call(:is_read, _from, state) do
    is_read = Map.get(state, :is_read)
    {:reply, is_read, state}
  end
  def handle_call(:is_activated, _from, state) do
    is_activated = Map.get(state, :is_activated)
    {:reply, is_activated, state}
  end
  @impl true
  def handle_cast(:activate, state) do
    state = Map.put(state, :is_read, false)
    state = Map.put(state, :is_activated, true)
    {:noreply, state}
  end
  def handle_cast(:read, state) do
    state = Map.put(state, :is_read, true)
    {:noreply, state}
  end
  def handle_cast(:deactivate, state) do
    state = Map.put(state, :is_activated, true)
    {:noreply, state}
  end
end
```

Fonte: Autoria própria

2.1.4 Considerações Sobre os Paradigmas Dominantes e Emergentes

Esta seção permitiu contextualizar as principais características, como vantagens e desvantagens dos paradigmas de programação atuais, tanto dos dominantes como dos emergentes. Na Seção 1.2 foram levantadas as principais demandas do desenvolvimento de *software* atuais, que seriam a programação em alto nível, paralelismo e desempenho. Nesse sentido, nenhum dos Paradigmas Dominantes atende de forma simultânea a todas essas demandas. Conforme detalhado na Tabela 3, o PI apesar de apresentar bom desempenho, não possibilita a programação em alto nível e dificulta a implementação de paralelismo, enquanto o PD apresenta tais características, porém o faz em troca de desempenho inferior quando comparado ao PI.

De todo modo os Paradigmas Emergentes mencionados também não por herdarem muitas das deficiências dos Paradigmas Dominantes. Isto é dado por muitos desses paradigmas serem construídos com base nos Paradigmas Dominantes, ou implementados nas linguagens de programação já existentes dos mesmos. Feita esta revisão geral nesta seção, a seção seguinte

Tabela 3 – Demandas de desenvolvimento de *software* atendidas pelos Paradigmas Dominantes
Fonte: Autoria própria

Paradigma Demanda \	Imperativo	Declarativo
Programação em alto nível		✓
Paralelismo		✓
Desempenho	✓	

enfim apresenta em maiores detalhes o Paradigma Orientado Notificações (PON).

2.2 PARADIGMA ORIENTADO NOTIFICAÇÕES (PON)

O Paradigma Orientado a Notificações (PON) é uma solução de desenvolvimento de software que permite, entre outras características, excelente desempenho computacional via entidades enxutas que colaboram por notificações precisas. O inovador PON apresenta propriedades que unem a flexibilidade de programação do Paradigma Imperativo (PI) e a facilidade de programação do Paradigma Declarativo (PD) (RONSZCKA, 2019; OSHIRO *et al.*, 2021).

Em suma e mais precisamente, o PON proporciona uma nova visão de desenvolver, estruturar e executar software por meio de entidades facto-execucionais e lógico-causais que colaboram por notificações precisas e pontuais [Ronszcka 2019]. Com isto, o PON apresenta três propriedades elementares, que consistem em: (a) facilidade de programação orientada a regras em alto nível, (b) evitar redundâncias de código lógico-causal que viabiliza alto desempenho de execução e (c) desacoplamento implícito de construtos que viabiliza paralelismo e distribuição (RONSZCKA, 2019; OSHIRO *et al.*, 2021).

De modo geral, dito de outra forma, o PON resolve certos problemas existentes nos outros paradigmas de programação atuais (como PI e PD com seus subparadigmas dominantes ou emergentes), os quais foram pragmaticamente revisados na Seção 2.1 e detalhadamente revisados na literatura pertinente, como em Roy e Haridi (2004), Scott (2009), Banaszewski (2009), Xavier (2014) e Ronszcka (2019). Exemplos de problemas dos paradigmas de programação vigentes são enfim as redundâncias temporais e estruturais na análise lógico-causal com o consequente mau uso do tempo de processamento, assim como o acoplamento excessivo entre entidades computacionais que dificulta o reaproveitamento e paralelização/distribuição das mesmas (SIMÃO; STADZISZ, 2008; BANASZEWSKI, 2009; LINHARES, 2015; PORDEUS, 2017; KERSCHBAUMER, 2018; RONSZCKA, 2019).

2.2.1 Constituintes do PON

Muito embora o PON já tenha sido apresentado na introdução desta dissertação, ele é aqui reapresentado de forma objetiva para facilitar o encadeamento da leitura, bem como para fins de revisão. Substancialmente, o PON é constituído por dois conjuntos de entidades computacionais: o facto-execucional e o lógico-causal (RONSZCKA, 2019; OSHIRO *et al.*, 2021). A relação entre eles e as entidades constituintes do PON, são ilustradas, no contexto de um sistema de correlação de sensores, pela Figura 16.

Conforme a Figura 16, há entidades facto-execucionais notificantes, os Fact Base Elements (FBE), que representam entidades do mundo e são compostas de sub-entidades Attributes e sub-entidades Methods, as quais respectivamente retratam seus estados e seus serviços/comportamento. Há também entidades lógico-causais notificáveis, as Rules, sendo cada qual composta de uma sub-entidade Condition e uma sub-entidade Action, que respectivamente se associam à sub-entidades Premises e à sub-entidades Instigations. Em tempo de construção das entidades, à luz de suas relações, são estabelecidos conexões para fins de notificações entre elas (RONSZCKA, 2019; OSHIRO *et al.*, 2021). Em suma, o ciclo de notificações funciona assim: cada Attribute de uma instância de um FBE que mudar de estado notifica apenas as Premises efetivamente pertinentes, o que faz com que essas refaçam seus cálculos lógicos. Cada Premise que mudar de estado lógico notifica apenas as Conditions efetivamente pertinentes, fazendo com que essas refaçam seus cálculos lógicos pelos estados notificados contabilizados. Por sua vez, se a Condition for aprovada, ela pode aprovar sua respectiva Rule. Esta, quando aprovada, ativa então sua Action, que notifica suas Instigations. Estas enfim instigam precisamente os Methods. Estes últimos geralmente alteram os estados dos Attributes, reativando o fluxo de notificações (RONSZCKA, 2019; OSHIRO *et al.*, 2021).

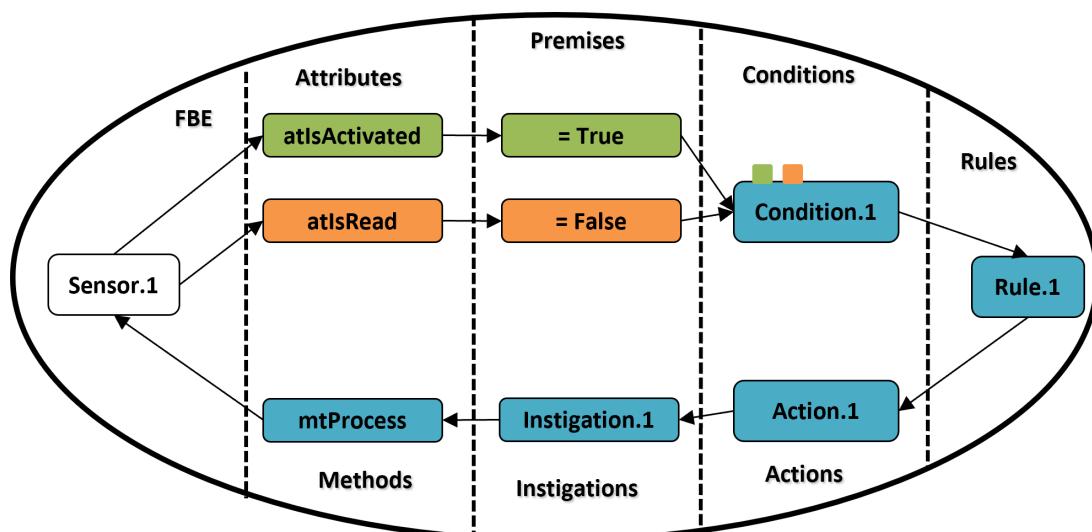


Figura 16 – Interação entre as entidades do PON e ciclo de notificações
Fonte: Autoria Própria

Isto dito, o PON permite o desacoplamento das expressões lógico-causais do resto do programa, por meio da aplicação destas entidades reativas e notificantes justamente. Como as entidades apenas trocam notificações, elas não estão fortemente acopladas entre si, o que é usualmente chamado de desacoplamento. Essa característica favorece o desenvolvimento de aplicações paralelas e distribuídas, ao mesmo tempo que mantém o desempenho das aplicações.

Isso difere do modelo de desenvolvimento usual, como no POO do PI e nos SBR do PD, nos quais as expressões causais são passivas e acopladas a outras partes do programa, conforme anteriormente elucidado (SIMÃO; STADZISZ, 2008; SIMÃO *et al.*, 2009; BANASZEWSKI, 2009; SIMÃO *et al.*, 2012; RONSZCKA, 2019).

No PON, a entidade reativa que trata a expressão causal é a *Rule*, conforme acima explicado. Assim, cada *Rule* gerencia o conhecimento sobre o comportamento lógico-causal de um conjunto de *FBEs* que a fazem reagir pela cadeia de notificação de seus constituintes. Ainda, o conhecimento lógico-causal de uma *Rule* provém normalmente de uma regra se-então, o que é uma maneira natural de expressão deste tipo de conhecimento, enquanto o conhecimento facto-execucional dos *FBEs* provém em uma forma de expressão compatível, similar a objetos ou *frames* (RONSZCKA, 2019).

Uma vez que estas entidades *Rules* e *FBEs* são criadas à luz desses conhecimentos lógico-causais e facto-execucionais, a cadeia de notificações também é criada/conectada na criação das suas entidades constituintes. Essas entidades computacionais notificantes precisamente que possibilitam que o mecanismo de execução ocorra de forma reativa e naturalmente desacoplada, bem como possibilitam a redução de redundâncias temporais e estruturais, viabilizando inclusive boa performance e o paralelismo e/ou distribuição fina de processamento (SIMÃO *et al.*, 2012). A boa performance, particularmente, encontraria lastro no cálculo assintótico da complexidade temporal dessas entidades reativado PON, o que é abordado na seção seguinte.

2.2.2 Complexidade temporal

Com base nas entidades reativas e notificantes do PON é possível calcular a complexidade temporal da execução do mecanismo de notificações. Banaszewski (2009) define a complexidade assintótica para o pior caso representada por $O(n^3)$ ou $O(FactBaseSize * nPremises * nRules)$, onde *FactBaseSize* corresponde ao número máximo de objetos *Attributes*, *nPremises* corresponde ao número máximo de entidades *Premises* notificadas por estes *Attributes* e *nRules* corresponde ao número máximo de entidades *Conditions-Rules* notificadas por estas *Premises* (SIMÃO, 2005; BANASZEWSKI, 2009).

Esse cálculo é ilustrado na Figura 17, na qual *Attributes*, *Premises*, *Conditions* e *Rules* correspondem respectivamente aos símbolos com abreviações *Att*, *Pr*, *Cd* e *Rl*. O caso assintótico considera o pior caso, no qual a alteração do estado do *Attribute* causa alteração no estado de todas as *Premises*, que por sua vez notificam todas as *Conditions*, que também notificam todas

as *Rules*. Entretanto, em um caso típico, a alteração do estado de um *Attribute* nem sempre causa alteração no estado de qualquer *Premise* ou *Condition*, de forma que não são geradas tantas notificações, permitindo um melhor desempenho do que o considerado no cálculo assintótico.

Entretanto, ao invés de se considerar apenas o pior caso, pode ser analisada a complexidade do caso médio. Neste caso, a análise da complexidade é iniciada no começo do processo de notificação do PON, a partir da entidade *Attribute*. Desta forma, as principais variáveis envolvidas na notificação de um *Attribute* seriam $FBat() = n\text{Premises} * n\text{Rules}$. A variável $n\text{Premises}$ é a soma do número de entidades *Premises* a serem notificadas por dado *Attribute* e a variável $n\text{Rules}$ é a soma do número de entidades *Rules* dependentes da mudança de estado de *Premises* consideradas em $n\text{Premises}$. Portanto, se for considerado um ciclo de notificações como a notificação gerada por de um único *Attribute* e n como sendo o número de total de *Attributes* do sistema, a complexidade temporal pode ser definida como $(FBat_1() + \dots + FBat_n())/n$. Assim, o resultado desta média seria uma constante, o que implicaria em uma complexidade linear $O(n)$ para o PON (SIMÃO, 2005; RONSZCKA, 2012; RONSZCKA, 2019).

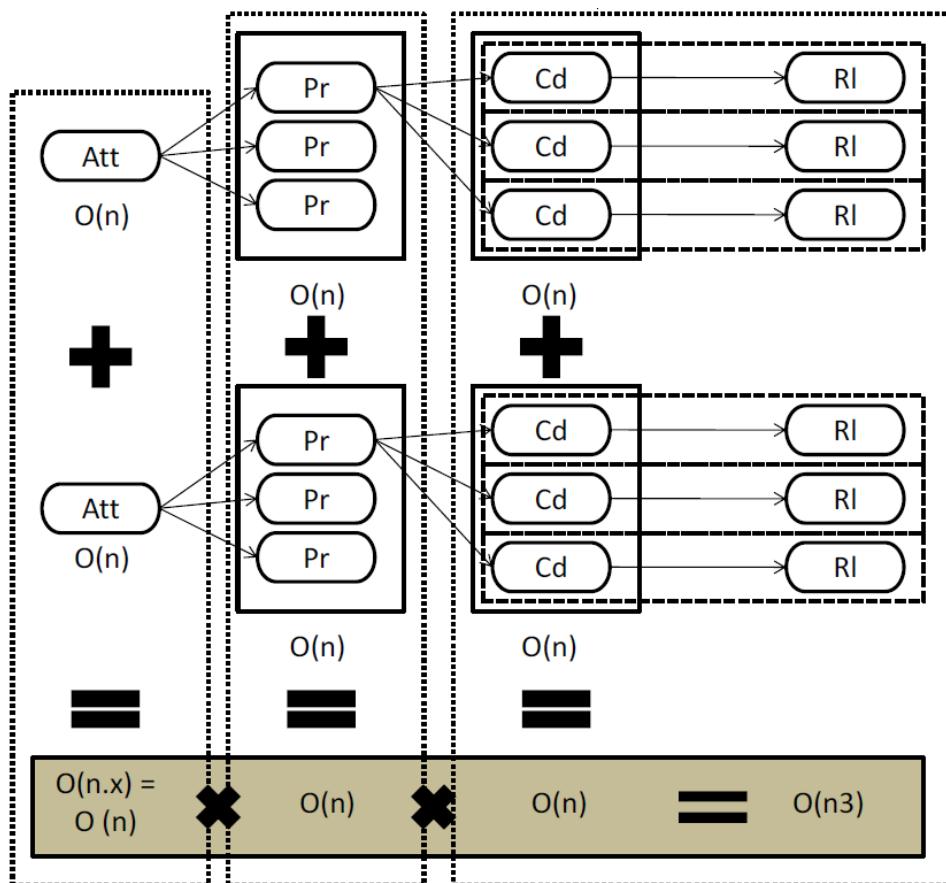


Figura 17 – Cálculo assintótico do mecanismo de notificações

Fonte: Banaszewski (2009)

2.3 CONCEITOS DE PROGRAMAÇÃO OU DESENVOLVIMENTO EM PON

Desde as subseções 2.3.1 até 2.3.10 se apresentam os diversos conceitos de programação ou desenvolvimento em PON, introduzidos com o objetivo de facilitar o desenvolvimento de aplicações em PON, sendo que não necessariamente cada materialização (*i.e.*, implementação) do PON as contemplam na sua totalidade.

2.3.1 Reatividade das entidades

No PON, a reatividade das suas entidades é um conceito central. Neste âmbito, as entidades reativas do PON conseguem gerar as notificações pontuais entre si, de acordo com a alteração dos seus estados, que ditam o fluxo de execução do programa em PON. De maneira geral, a reatividade presente nas entidades definidas no metamodelo do PON proporciona uma execução livre de avaliações lógico-causais redundantes e desnecessárias.

A reatividade das entidades na cadeia de notificações do PON permite evitar a redundância temporal ao avaliar as expressões lógico-causais somente após a mudança de estado dos *Attributes* e suas respectivas *Premises*. Ainda, esta reatividade encontra respaldo no fato de *Conditions* poderem compartilhar a colaboração de *Premises*, o que evita a redundância estrutural nas avaliações lógico-causais. Por fim, a reatividade em PON pode ser melhorada de algumas formas, como por meio de tabelas *hash* nos *Attributes*, permitindo que *Premises* recebam notificações de apenas certos estados de interesse, mas não dos estados que não são de seus interesses (BANASZEWSKI, 2009). Estes cenários de notificação de *Premises* tanto com uma lista encadeada simples como com uma tabela *hash* são ilustrados na Figura 18, onde um *Attribute* *atSignal* tem seu estado alterado de vermelho (*RED*) para verde (*GREEN*), sendo que as setas partindo do *Attribute* representam as notificações realizadas (BANASZEWSKI, 2009).

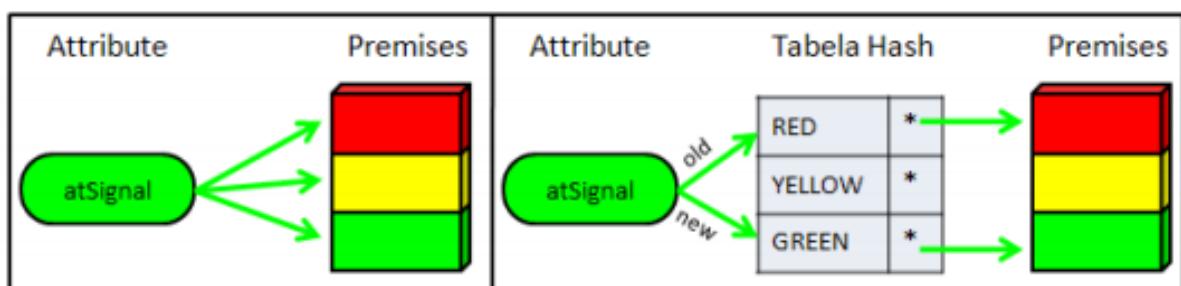


Figura 18 – Notificações baseadas em lista encadeada e tabela hash
Fonte: Banaszewski (2009)

2.3.2 Renotificações

Os *Attributes*, pela sua definição, notificam as *Premises* apenas quando seu estado é alterado. Porém, em determinados cenários, pode haver a necessidade de execução de uma *Rule* mesmo quando não há a alteração no estado dos *Attributes*. Tome-se como exemplo uma *Rule* responsável (via sua *Action* e dada *Instigation*) pela instigação da execução de determinado *Method* que precise ser constantemente reinstigado enquanto a *Condition* daquela *Rule* estiver verdadeira (BANASZEWSKI, 2009).

Neste tipo de contexto que se faz necessário implementar alguma solução, como o chamado mecanismo de renotificações. No diagrama de atividades da Figura 19, é ilustrado o processo de decisão para a geração de notificação em entidades que suportam o mecanismo de renotificação. Com a utilização do mecanismo de renotificações, a notificação é gerada mesmo quando não ocorre uma mudança de estado. À luz da Figura 19, pode-se definir uma frequência para definir (*setar*) estado, permitindo que a cadeia de notificações seja animada com uma dada cadência (BANASZEWSKI, 2009).

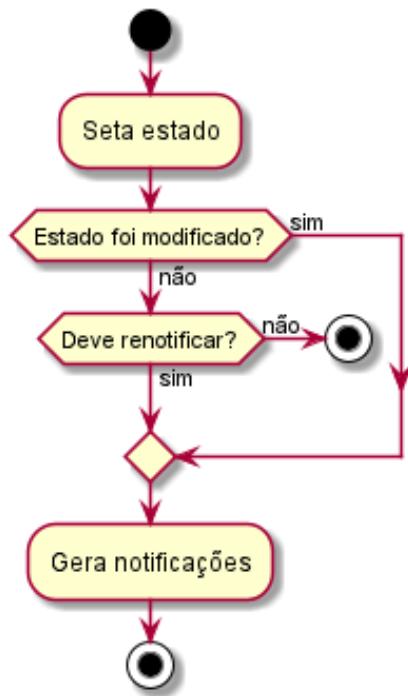


Figura 19 – Diagrama de atividades do processo de renotificação
Fonte: Autoria própria

Para exemplificar mais precisamente esse cenário de renotificações, tome-se o exemplo de *Rule* para um alarme apresentado na Figura 20. Neste caso é considerado que o mecanismo de ativação da sirene, implementado por meio de *mtRingSirenMilliseconds*, permanece ativado por

um tempo limitado após ser instigado (100 ms neste exemplo). Deste modo se torna necessário que a cada nova atribuição de valor para *atStatus* do *FBE Sensor*, seja executada novamente a *Rule*, o que é feito com a utilização do mecanismo de renotificação.

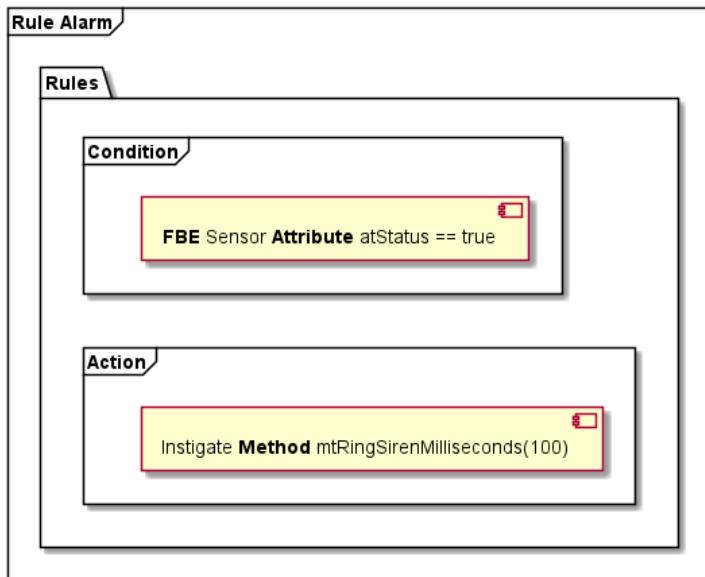


Figura 20 – Rule que depende do mecanismo de renotificações
Fonte: Autoria própria

2.3.3 Keeper

O mecanismo de renotificações, apesar de útil, pode ser bastante computacionalmente custoso, porque ele causa um aumento no número de notificações geradas pelo sistema, aumentando o tempo de execução das aplicações. Desta forma, outra estratégia alternativa desenvolvida de modo a atender a necessidade de execução de uma *Rule* mesmo quando não há a alteração no estado dos *Attributes* foi o padrão *Keeper*. Diferente das renotificações, com o *Keeper* não é necessária uma nova notificação para executar a *Rule*, mas sim a execução da *Rule* pode ser explicitamente requisitada a ser executada quantas vezes for necessário, enquanto a mesma se mantenha aprovada (MUCHALSKI *et al.*, 2012).

Entretanto, o principal problema da adoção dessa estratégia é que a responsabilidade da execução das *Rules* é passada inteiramente ao desenvolvedor, visto que as *Rules* não mais são executadas no momento em que são aprovadas, mas sim apenas quando requisitada explicitamente no código. Isso é necessário de forma a evitar a execução duplicada das *Rules*. O uso do *Keeper* é exemplificado no Código 2.10, onde é possível observar na linha 3 onde a execução da *Rule rlProduct* é explicitamente requisitada.

Código 2.10 – Exemplo de uso do padrão Keeper

```

1 void Result::UpdateProduct() {
2     prProduct->setValue(production->getProductId());
3     rlProcut->execute();
4 }
```

Fonte: Adaptado de Muchalski *et al.* (2012)

2.3.4 Entidades impertinentes

Ainda no contexto da reatividade das entidades, podem existir cenários nos quais as notificações de certas entidades podem ser consideradas desnecessárias. Isso pode ocorrer em situações nas quais um *Attribute*, que apesar de não ser determinante sozinho para a aprovação de uma *Rule* em um dado contexto, apresenta constantes mudanças de estado, disparando o fluxo de notificações novamente a cada variação em seu estado. Nesse caso as notificações desnecessárias impactam negativamente no desempenho da aplicação PON (RONSZCKA, 2012).

A Figura 21 ilustra um cenário de entidades impertinentes, com um exemplo hipotético de controle de temperatura, com dois *Attributes*, nomeadamente *atStatus* e *atTemperature*. Nesse caso, *atTemperature* vai variar constantemente, enquanto *atStatus* vai variar muito ocasionalmente. Porém, a *Condition* da *Rule* somente será aprovada quando *atStatus* possuir o valor *true* e *atTemperature* possuir um dado valor. Nesse cenário dado, as notificações geradas por *atTemperature* seriam impertinentes em sua maioria.

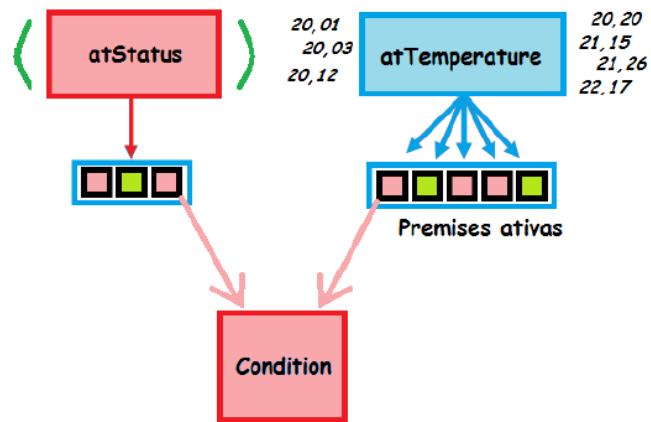


Figura 21 – Alterações de estado com *Attribute* impertinente ativo
Fonte: Ronszcka (2012)

Neste cenário em pauta, o *Attribute* *atTemperature* é categorizado como impertinente por sua mudança estaticamente não aprovar a *Condition* da *Rule*, logicamente em função da mudança largamente mais discreta de seu parceiro de conjunção, o pertinente *atStatus*. Deste modo, para evitar notificações desnecessárias, suas *Premises* não deveriam receber notifica-

ções temporariamente, sendo que as notificações do *atTemperature* para tais *Premises* seriam desabilitadas até segunda ordem. A Figura 22 ilustra o mesmo cenário acima, porém com *atTemperature* sendo tido como um *Attribute* impertinente para com *Premises*, tendo suas notificações temporariamente desabilitadas.

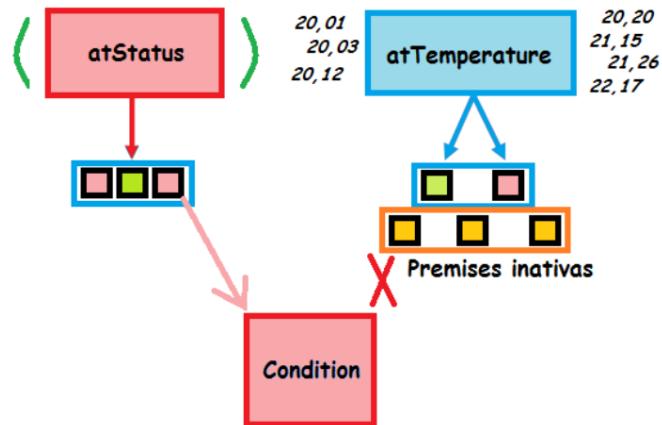


Figura 22 – Alterações de estado com *Attribute* impertinente desativado
Fonte: Ronszcka (2012)

Nesse âmbito a *Condition* fica responsável por reativar temporariamente as *Premises* relativas a *Attributes* impertinentes, de modo a voltar receber notificações destas, quando o conjunto de *Premises* pertinentes para sua aprovação forem satisfeitas. Essa reativação temporária, ilustrada na Figura 23, torna capaz a aprovação da *Condition*. Por fim, após a aprovação e execução da *Rule*, volta-se a desativar o *Attribute* impertinente no contexto da *Rule* em questão. Desta forma, o *Attribute* impertinente voltaria a ignorar *Premises* da *Condition* daquela *Rule* até que fosse requisitado novamente pela *Condition* (RONSZCKA, 2012).

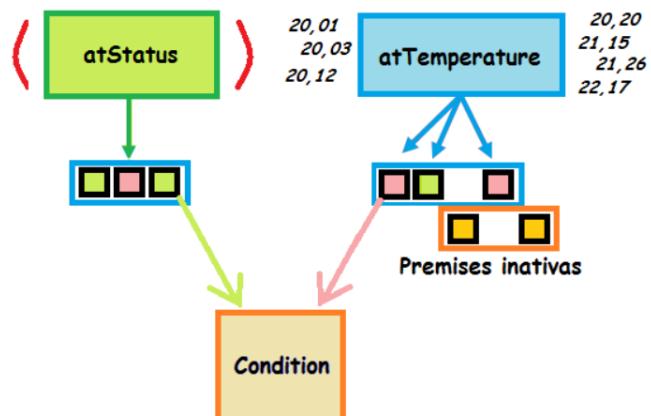


Figura 23 – Alterações de estado com *Attribute* impertinente reativado
Fonte: Ronszcka (2012)

A definição de impertinência das entidades pode acontecer em dois momentos, feita a priori ou em tempo de execução. Esses dois modos de impertinência são chamados de im-

pertinência estática e impertinência dinâmica. A impertinência estática é de responsabilidade do desenvolvedor, de modo que a entidade já é definida como impertinente no momento da compilação, enquanto a impertinência dinâmica não é definida pelo desenvolvedor e sim por um mecanismo em tempo de execução que seja capaz de detectar a impertinência das entidades a luz de dados estatísticos.

2.3.5 Compartilhamento de entidades

O uso do compartilhamento (de colaboração) de entidades pode ser considerado uma boa prática do PON tanto da perspectiva da facilidade de desenvolvimento como para o ganho de desempenho. O compartilhamento de entidades como *Conditions* e *Premises* oferece ganhos de desempenho ao eliminar a criação de estruturas redundantes e, com isso, também reduz o número de notificações desnecessárias (RONSZCKA, 2012). O ganho em facilidade de desenvolvimento, por sua vez, é dado pelo fato do compartilhamento de entidades acontecer de forma natural ao desenvolvedor durante o desenvolvimento das *Rules* do programa em PON. Quando o desenvolvedor escreve o código em PON, seria um cenário comum diferentes *Rules* estarem relacionadas às mesmas *Premises*, *Conditions* ou *Instigations*, as quais seriam compartilhadas já em tempo de construção do programa.

O compartilhamento de entidades é um meio útil para a resolução do problema de redundância estrutural, conceito introduzido na reflexão sobre paradigmas da Seção 2.1. Em suma, a redundância estrutural ocorre, por exemplo, quando o conhecimento sobre um estado resultante da avaliação de dada expressão lógica (*e.g.*, comparação de uma variável com uma constante) não é compartilhado entre outras expressões causais pertinentes, causando reavaliações desnecessárias (BANASZEWSKI, 2009). O compartilhamento de colaboração de entidades evita avaliações redundantes, ao fornecer um meio para se compartilhar os resultados das avaliações lógicas comuns a várias expressões causais (BANASZEWSKI, 2009).

Para exemplificar o compartilhamento de entidades, é expandido o exemplo do alarme, introduzido na Seção 2.2, para ter duas *Rules* e com dois *FBEs* relativos aos sensores, porém apenas um *FBE* para o alarme. Neste caso, existem três *Premises*, mostradas na Figura 24, e duas *Rules*, mostradas na Figura 25. Nesse cenário observa-se que ambas as *Rules*, dependem da mesma *Premise prAlarmOn*, que neste caso é uma entidade cuja colaboração é compartilhada para com as *Rules*, sendo que ambas são notificadas pelas *Premise* em questão quando pertinente.

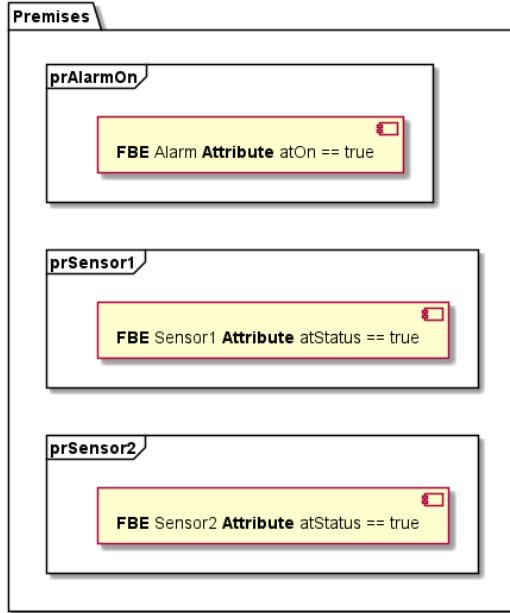


Figura 24 – Declaração de *Premises* do exemplo do alarme
Fonte: Autoria própria

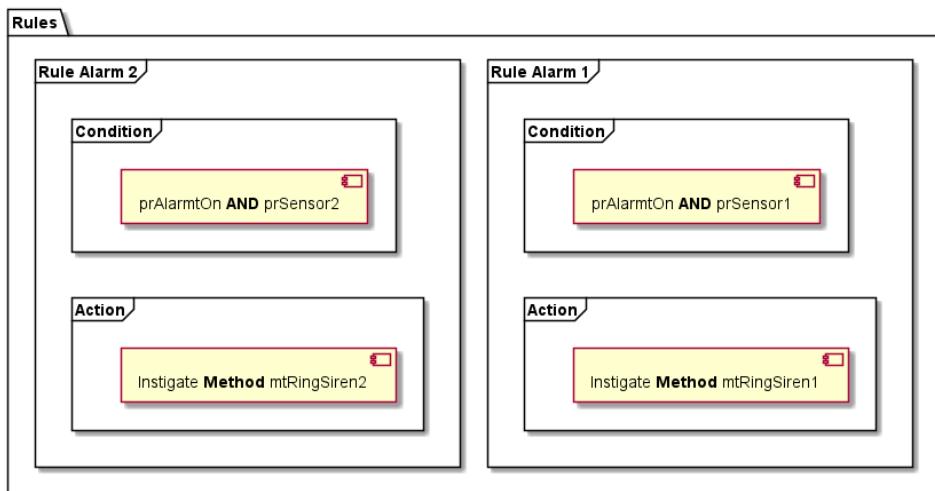


Figura 25 – Declaração de *Rules* utilizando *Premises* compartilhadas
Fonte: Autoria própria

2.3.6 Resolução de conflitos

Um conflito acontece quando duas atividades diferentes dependem de um mesmo recurso compartilhado, o qual deve ser usado exclusivamente (SIMÃO, 2005). Uma das características do PON é ser também orientado a regras, sendo que conflitos entre regras podem surgir quando duas ou mais *Rules* são aprovadas com base em estado de recurso exclusivo. Outra característica do PON é justamente permitir a execução de forma desacoplada e concorrente dos elementos do seu modelo, permitindo paralelismos e distribuições conforme o ambiente em que a aplicação PON seja executada. Nesse âmbito, é importante a identificação e resolução de

conflitos originados dessa execução desacoplada de aprovação e execução de *Rules* (PORDEUS, 2017).

No PON, mais precisamente, o conflito ocorre quando duas ou mais *Rules* referenciam um mesmo *FBE* e demandam exclusividade de acesso a este *FBE* (como para o caso de modificar *Attributes* deste *FBE*). Neste caso, apenas uma das *Rules* poderia ser executada por vez, fazendo o acesso exclusivo a este *FBE*, o que leva ele a ser um *FBE* exclusive neste contexto dado. Ainda, neste âmbito de conflitos, a resolução de conflitos em ambientes monoprocessados ocorre de modo a estabelecer a ordem de execução das *Rules*. Já em ambientes multiprocessados, a resolução de conflitos é necessária para evitar o acesso concorrente ao mesmo recurso (VALENÇA, 2012; BANASZEWSKI, 2009).

Nos ambientes monoprocessados pode ser aplicado um escalonador de *Rules* que utiliza estruturas de dados lineares (*e.g.*, pilha, filha ou lista) (BANASZEWSKI, 2009). Esse modelo, exemplificado na Figura 26, recebe as *Rules* na ordem em que são aprovadas e as executa de acordo com a estratégia de resolução de conflito escolhida, o que seria assaz similar a resolução de conflitos em geral de sistemas baseados em regras (BANASZEWSKI, 2009). Isto dito, as diferentes estratégias de resolução de conflitos para ambientes monoprocessados são listadas abaixo:

- *Breadth* ou largura: escalonamento *First In First Out* (FIFO), primeiro a entrar é o primeiro a sair, no qual as *Rules* são executadas na mesma ordem que são aprovadas. Pode ser usada uma estrutura de dados do tipo fila para este propósito.
- *Depth* ou profundidade: escalonamento *Last In First Out* (LIFO), último a entrar é o último a sair, no qual as *Rules* são executadas começando sempre pela última *Rule* aprovada. Pode ser usada uma estrutura de dados do tipo pilha para este propósito.
- *Priority* ou prioridade: escalonamento baseado na prioridade definida para cada *Rule*.
- *No one* ou nenhum: a *Rule* é executada imediatamente ao ser aprovada.

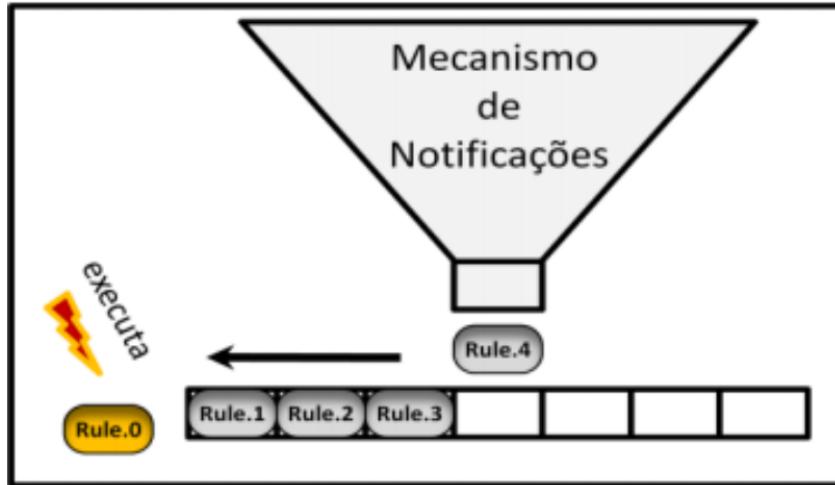


Figura 26 – Modelo centralizado de resolução de conflitos na estratégia *Breadth*
Fonte: Banaszewski (2009)

Apesar deste modelo ser uma solução eficiente para ambientes monoprocessados, ele não se mostra adequado para ambientes *multicore* por não apresentar um mecanismo que permita o escalonamento apropriado da execução das *Rules*. Banaszewski (2009) então propõe tal mecanismo que execute o escalonamento de forma eficiente. Para que este mecanismo de escalonamento seja integrado ao modelo de resolução de conflitos, é necessário alterar a forma pela qual as *Rules* são executadas neste modelo. Assim, ao invés de uma *Rule* aprovada instigar a execução imediata da mesma, essa deve repassar o controle da execução da respectiva *Rule* para o componente escalonador de *Rules*. Quando a *Rule* é aprovada, ao invés de ser executada imediatamente, ela é adicionada a uma fila de execução, considerando a prioridade de execução das respectivas *Rules*. Por sua vez, essas *Rules* são executadas de forma escalonada por uma *pool* de executores, com um número de executores pré-definidos, que permite a execução simultânea de múltiplas *Rules*.

2.3.7 Master Rule

Este conceito estabelece um mecanismo de dependência entre *Rules*. Podem existir cenários nos quais diferentes *Rules* dependem do mesmo conjunto de *Premises* compartilhadas para sua aprovação, as quais notificariam todas as entidades interessadas em suas mudanças de estado. Nesse caso as notificações desnecessárias podem ser evitadas caso essas *Premisses* notifiquem uma única *Rule*, ao invés de todas as interessadas. Essa *Rule*, quando aprovada, fica responsável por notificar as demais *Rules* interessadas e, portanto, dependentes daquela. Nesse cenário é criada uma dependência entre as *Rules*, na qual a *Master Rule* é capaz de

notificar outras *Rules* (RONSZCKA, 2012). Esse mecanismo traz benefícios em questões de desempenho e facilidade na composição de aplicações, porque a dependência entre as *Rules*, por meio do conceito de *Master Rule*, reduz as notificações geradas pelas *Premises* ao direcionar as notificações apenas para a *Master Rule* (RONSZCKA, 2012).

Para exemplificar o conceito de *Master Rule* é expandido o exemplo do alarme explorado na Seção 2.3.5, com duas *Rules* e com dois *FBEs* para os sensores, porém agora considerando que uma das *Rules*, chamada *rlAlarm2*, depende de três *Premises* (*prAlarmOn*, *prSensor1* e *prSensor2*), sendo que duas destas *Premises* fazem parte da composição de outra *Rule*, chamada *rlSensor1*, de modo que este cenário pode ser construído por meio do uso de uma *Master Rule*. Considere-se as mesmas três *Premises* da Figura 24, aplicadas nas *Rules* da Figura 27. Pela aplicação da *Master Rule*, a aprovação da *rlAlarm2* só acontece após a *rlAlarm1* também ser aprovada.

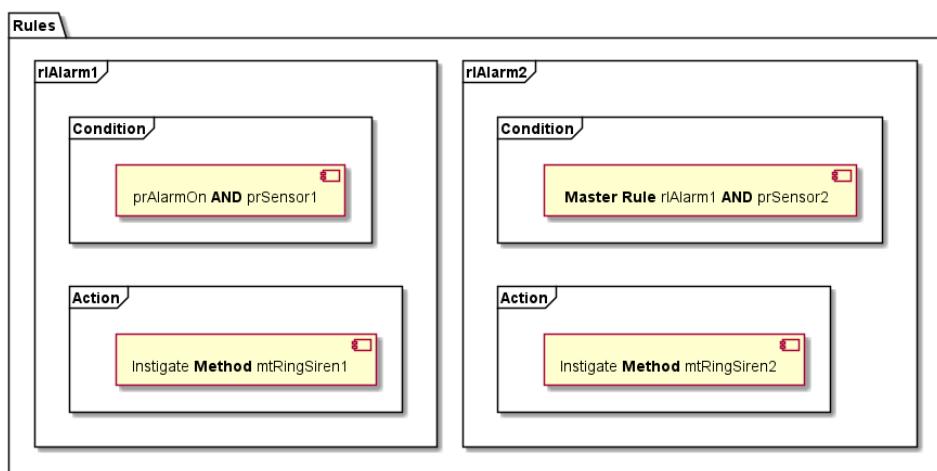


Figura 27 – Exemplo de aplicação de *Master Rule*
Fonte: Autoria própria

2.3.8 *Formation Rules* - Regras de Formação

O conceito de *Formation Rules* (ou Regra de Formação) do PON é um reaproveitamento das *Formation Rules* do Controle Orientado a Notificações (CON), o predecessor do PON (SIMÃO, 2001). Cada *Formation Rule* permite a criação de *Rules* específicas com base na representação genérica de uma *Rule*. Esse conceito é útil quando o conhecimento causal de uma *Rule* é comum para diferentes conjuntos de instâncias de *FBEs*, ou seja, um conjunto de *Rules* específicas se diferencia apenas nas instâncias referenciadas (RONSZCKA, 2019).

A título de exemplo tome-se um cenário hipotético de um sistema de alarmes, no qual

existem as entidades usuários e alarmes (*FBE User* e *FBE Alarm*). A responsabilidade do sistema seria de avisar cada usuário sobre a ativação de cada alarme conectado. Para isso seria necessário replicar essa condição para cada combinação de usuário e alarme (e.g., *user1 x alarm1*, *user1 x alarm2*, *user2 x alarm1*, *user2 x alarm2* etc.), o que torna o processo custoso e propenso a erros (RONSZCKA, 2019).

Com a aplicação do conceito de *Formation Rules* se torna possível a declaração do conhecimento lógico-causal da *Rule* de forma genérica, com base nos tipos dos *FBEs*, ao invés das instâncias pontuais de cada um. Assim, em tempo de compilação, seria feita a composição das *Rules* para cada combinação de instâncias (RONSZCKA, 2019). Isso permite tornar a replicação de *Rules* uma tarefa automática, minimizando a possibilidade de erros e facilitando o trabalho do desenvolvedor, além de promover maior legibilidade do código em geral, ao remover a duplicação de código similar repetido (RONSZCKA, 2019).

A Figura 28 ilustra uma representação de uma *Formation Rule* que pode ser adotada para gerar as instâncias das *Rules* referentes a um cenário de automatização do controle de veículos. Neste caso a *Formation Rule* define a *Rule* que deve ser aplicada a todos os carros e cruzamentos, onde o veículo só pode acelerar caso o sinal esteja no estado verde e não hajam pessoas andando no cruzamento.

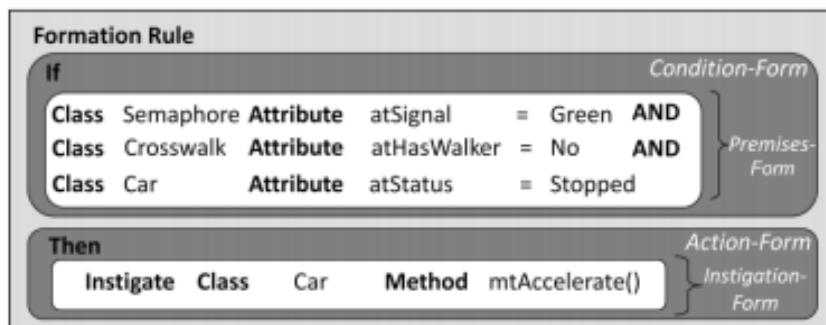


Figura 28 – Representação de uma *Formation Rule*
Fonte: Banaszewski (2009)

Uma *Formation Rule* é mais genérica do que uma *Rule* tradicional. Para sua descrição é utilizado o sufixo *Form*, que define as lógicas que é utilizada como base para instanciar de forma efetiva cada uma das entidades do PON. As suas entidades *Form* somente analisam se o *FBE* é de uma determinada classe. Uma *Formation Rule* filtra e cria combinações de *FBEs*, onde cada combinação resulta na criação de uma *Rule* independente. Em suma, uma *Formation Rule* aplica conceitos genéricos a fim de criar *Rules* específicas, mantendo a capacidade de notificação entre os objetos colaboradores. A função principal da *Formation Rule* é gerar automaticamente a partir

de um modelo várias instâncias de *Rules* que compartilham a semântica deste modelo.

2.3.9 FBE Rules

De maneira geral, o conceito de agregações de *Rules* em *FBEs* é um conceito próximo ao das *Formation Rules*, sendo um caso particular das *Formation Rules* no qual a *Rule* está relacionada a apenas um tipo de *FBE*, enquanto nas *Formation Rules* a *Rule* pode estar relacionada a vários *FBE* (SANTOS, 2017). Uma *FBE Rule* é uma *Rule* que possui escopo local em seu *FBE*, e todas as instâncias deste *FBE* possuem, obrigatoriamente, uma instância da *Rule* em questão (RONSZCKA, 2019).

A título de exemplo considera-se a aplicação do alarme com seu respectivo sensor. Para a execução de vários alarmes seria necessário replicar as *Rules* para cada instância. Porém, com a utilização do conceito de *FBE Rules*, essa replicação pode ser feita de modo automático com a declaração da *Rule* como uma *FBE Rule* do *FBE* do alarme. A Figura 29 ilustra este cenário.

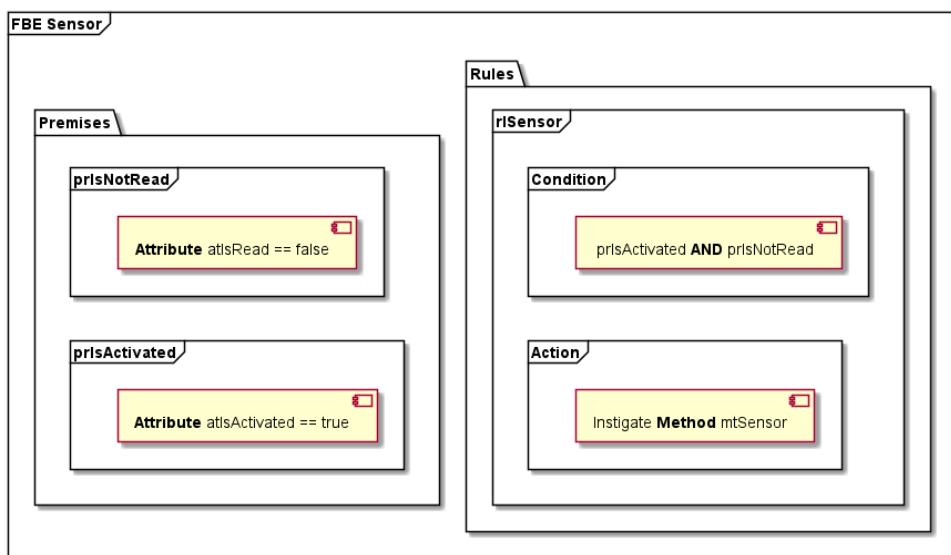


Figura 29 – Exemplo de aplicação de FBE Rules
Fonte: Autoria própria

2.3.10 Agregação entre FBEs

Outro conceito pertinente ao PON é possibilidade de criar agregações entre *FBEs*, permitindo a criação de *FBEs* compostos sem a necessidade de duplicação de estruturas. Esse conceito permite otimizações na redução de código, assim como também facilita a descrição de programas em PON, tanto de forma textual como gráfica (RONSZCKA, 2019). Com isso,

espera-se atingir níveis de organização mais efetivos, melhorando particularmente a escrita e a legibilidade de programas baseados no PON.

Esta forma de estruturação facilita a construção de *Rules* que podem ser compostas internamente aos *FBEs*, sendo o caso das *FBE Rules*. Neste caso a agregação de *FBEs* permite a criação de *Rules* utilizando todos os *FBEs* agregados.

2.3.11 Resumo dos Conceitos de Programação do PON

Nas seções anteriores foram apresentados diversos conceitos de programação do PON. De forma a facilitar o entendimento sobre tais conceitos, a Tabela 4 apresenta um resumo de cada um desses conceitos. Isto é feito de forma a apresentar de forma simplificada e centralizada o conhecimento sobre os conceitos de programação do PON.

É importante ressaltar que apesar de todas as materializações do PON buscarem aplicar todos esses conceitos, raramente isso acontece. Alguns conceitos de programação foram introduzidos por materializações mais recentes do PON, de modo que, naturalmente, as implementações que precedem a introdução de determinado conceito não o materializam. Ainda assim, mesmo em materializações mais recentes, limitações de ordem técnica ou conceitual podem impedir a implementação de todos esses conceitos.

Tabela 4 – Resumo de conceitos de programação do PON

Fonte: Autoria própria

Conceito	Definição
Reatividade das entidades	Entidades são capazes de gerar notificações pontuais de forma espontânea na alteração de estado
Renotificações	Entidades podem gerar notificações de forma forçada, mesmo sem alteração nos seus estados
<i>Keeper</i>	Controle manual de execução das <i>Rules</i> , permitindo execução de uma <i>Rule</i> múltiplas vezes enquanto aprovada
Impertinência	Supressão seletiva de notificações de determinadas entidades, podendo ocorrer de forma estática ou dinâmica
Compartilhamento de entidades	Utilização compartilhada de entidades com conhecimento em comum com outras entidades, reduzindo o número de entidades únicas no sistema
<i>Master Rule</i>	Relação de dependência entre <i>Rules</i> com conhecimento lógico-causal em comum
<i>Formation Rules</i>	Criação de <i>Rules</i> baseadas na representação genérica de uma <i>Rule</i> relativa ao tipo dos <i>FBEs</i>
<i>FBE Rules</i>	Definição de <i>Rules</i> no corpo do <i>FBE</i> , instanciando uma <i>Rule</i> independente para cada instância de dado <i>FBE</i>
Agregação entre <i>FBEs</i>	Criação de <i>FBEs</i> compostos de outros <i>FBEs</i>

2.4 MATERIALIZAÇÕES DO PON EM SOFTWARE

O PON apresenta materializações tanto em software quanto em hardware (LINHARES, 2015; KERSCHBAUMER, 2018; RONSZCKA, 2019; SCHÜTZ, 2019). Entretanto, este trabalho foca nas implementações em software e não hardware. Nas implementações em software, há tanto implementações em *frameworks* como também por meio de linguagem de programação própria, por meio da chamada Tecnologia LingPON. Enquanto a linguagem de programação via Tecnologia LingPON se constitui em estado da arte por ainda ser consideravelmente prototípica, o conjunto de *frameworks* se constituem no estado da técnica por parte deles se encontrar em estado estável de desenvolvimento.

As implementações por meio de *frameworks* existem de forma a oferecer uma interface de programação de aplicativos (API - *Application Programming Interface*) que possibilite o desenvolvimento de aplicações seguindo o modelo do PON, modificando, portanto, a maneira como estas linguagens operam, justamente permitindo e conduzindo-lhes a operarem de forma orientada a notificações. As seções seguintes apresentam cada uma destas materializações em software em maiores detalhes, sendo que a primeira subseção desta presente seção se dedica aos *frameworks* para o PON desenvolvidos em linguagem de programação C++.

2.4.1 *Frameworks* PON C++

A materialização do PON em C++ conta com mais de uma única implementação, sendo que cada uma destas materializações foi construída com base nas materializações anteriores, cada uma com sua proposta de melhoria específica. Estas diferentes materializações são o *Framework* PON C++ Prototípico, *Framework* PON C++ 1.0, *Framework* PON C++ 2.0 e *Framework* PON C++ 3.0. Além dessas materializações, também há o prototípico JuNOC++, desenvolvido por Chierici (2020) com base no trabalho da dissertação de Ronszcka (2012). Estas materializações são exploradas individualmente em maior detalhe nas seções seguintes.

2.4.1.1 *Framework* PON C++ Prototípico

A primeira implementação do PON sobre o formato de *framework*, o chamado *Framework* PON C++ Prototípico, foi proposta por Simão em 2007 (SIMÃO; STADZISZ, 2008; SIMÃO *et al.*, 2012). Esta versão prototípica do *framework* deriva dos esforços de sua dissertação

de mestrado e tese de doutorado no âmbito do metamodelo do Controle Orientado a Notificações (CON), o qual foi aplicado sobre a ferramenta de simulação de sistema de manufatura ANALYTICE II (SIMÃO, 2005; SIMÃO *et al.*, 2009).

A Figura 112 é apresentada com o objetivo de ilustrar uma aplicação desenvolvida com a aplicação do *Framework PON C++ Prototipal*. Neste exemplo de implementação, toma-se uma aplicação intitulada Mira ao Alvo, na qual as entidades miras e as entidades alvos são representadas respectivamente por arqueiros e maçãs, sendo que há uma maçã para cada arqueiro. Isto dito, este cenário é descrito sob a forma de *FBEs* e *Rule* no Código 2.11.

Código 2.11 – FBEs e Rule para cenário do Mira ao Alvo

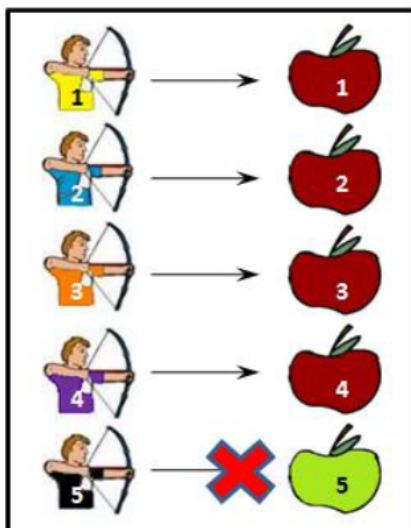


Figura 30 – Cenário do Mira ao Alvo

Fonte: Banaszewski (2009)

```
fbe Archer
    public boolean atStatus = false
    public integer atIdentity = 0
end_fbe
fbe Apple
    public boolean atAppleColor = false
    public integer atIdentity = 0
    public boolean atIsCrossed = false
    private method mtStatusOff
        attribution
            this.atIsCrossed = true
        end_attribution
    end_method
end_fbe
inst
Archer archer
Apple apple
end_inst

rule rlShootApple
    condition
        premise prIdentity
            apple.atIdentity ==
                archer.atIdentity
        end_premise
        and
        premise prColor
            apple.atColor == true
        end_premise
        and
        premise prAppleStatus
            apple.atStatus == true
        end_premise
        and
        premise prArcherStatus
            archer.atStatus == true
        end_premise
        and
        premise prApple IsNotCrossed
            apple.atIsCrossed == false
        end_premise
    end_condition
    action sequential
        instigation sequential
            call apple.mtStatusOff()
        end_instigation
    end_action
end_rule
```

Fonte: Chierici (2020)

Em termos de implementação, no *Framework PON C++ Prototipal*, cada arqueiro e maçã é representado por um objeto notificante na forma de FBE (*i.e.*, um Elemento da Base

de Fatos), os quais interagem de acordo com a avaliação de expressões causais pertinentes na forma de objetos *Rules* (*i.e.*, uma *Rule*) (BANASZEWSKI, 2009). Essa implementação em foco é mostrada no Código 2.12.

Código 2.12 – Exemplo de programa com o framework C++ prototípal

```
//Premises
prAppleColorRead = new AgentePremissa(appleList->at(i)->atAppleColor, True);
prAppleColorRead->conectaPredBooleano(&comparaBooleanos);
prAppleStatusTrue = new AgentePremissa(appleList->at(i)->atAppleStatus, True);
prAppleStatusTrue->conectaPredBooleano(&comparaBooleanos);
prArcherStatusTrue = new AgentePremissa(archerList->at(i)->atArcherStatus, True);
prArcherStatusTrue->conectaPredBooleano(&comparaBooleanos);
//Conditions
AgenteCondicao* cdFireApple;
acFireApple = new AgenteAcao();
acFireApple->conectaAgenteOrdem(appleList->at(i)->mtChangeToGreen);
//Method
mtChangeToGreen = new AgenteMetodoGen<Apple>(&Apple::ChangeToGreen);
//Instigation
aitChangeToGreen = new AgenteOrdem(mtChangeToGreen);
//Rule
AgenteRegra* rlFireApple;
rlFireApple = new AgenteRegra(cdFireApple, acFireApple);
```

Fonte: Adaptado de Banaszewski (2009)

2.4.1.2 Framework PON C++ 1.0

O *Framework PON C++ 1.0* foi implementado por Banaszewski em 2009, com a proposta de facilitar e melhorar a composição de programas em PON. A estrutura de implementação desta versão do *framework* é constituída por dois principais pacotes de classes, o pacote *Core* e o pacote *Application*, conforme ilustrado na Figura 31. O pacote *Core* contém as classes que modelam as entidades do metamodelo do PON, enquanto o pacote *Application* contém as classes utilizadas para a instanciação de uma aplicação em PON, usando as entidades do pacote *Core* (BANASZEWSKI, 2009).

Nesta estrutura, a classe *MyApplication* é um exemplo de implementação que seria definida pelo programador com base na classe abstrata *Application* do *framework*. A classe *Application* representa um modelo de inicialização padrão para aplicações em PON, por meio da construtora da classe, na qual é definida a estratégia de resolução de conflitos, bem como dos métodos *initFacts* e *initRules*, os quais são respectivamente usados para concentrar a instanciação dos *FBEs* e *Rules* (BANASZEWSKI, 2009).

Em resumo, a construção de aplicações utilizando o *Framework PON C++ 1.0* se baseia em torno da especialização da classe *Application*, que contém os métodos necessários para a inicialização das estruturas do PON. Um exemplo de aplicação para o mesmo cenário Mira ao Alvo, já descrito anteriormente na Seção 2.4.1.1, agora com o *framework C++ 1.0*, é mostrado

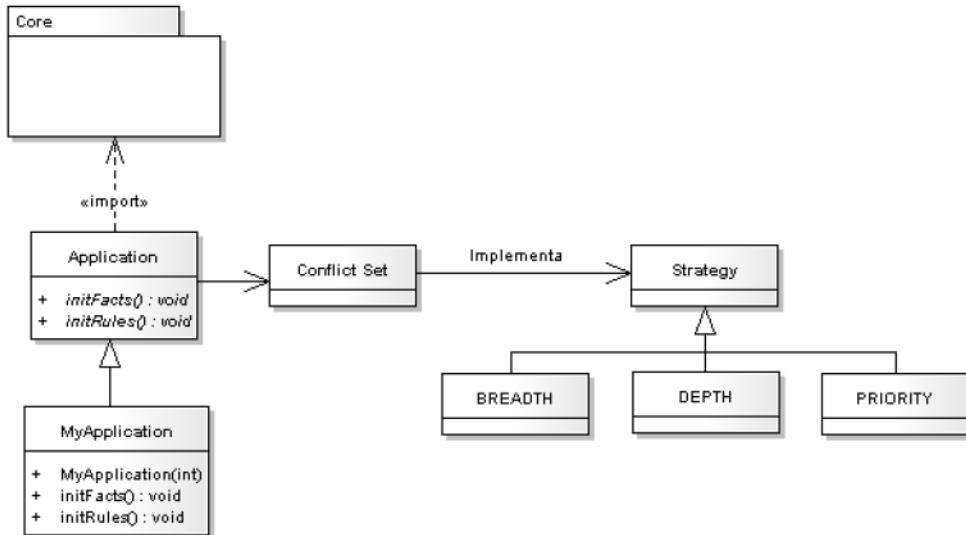


Figura 31 – Estrutura do framework C++ 1.0
Fonte: Banaszewski (2009)

no Código 2.13.

Código 2.13 – Exemplo de programa com o framework C++ 1.0

```

for ( iteratorArcher = archerList->begin(), iteratorApple = appleList->begin();
      iteratorArcher != archerList->end();
      ++iteratorArcher, ++iteratorApple)
{
    Instigation* it = new Instigation((*iteratorApple)->mtStatusOff);
    RuleObject* rlFireApple = new RuleObject("", scheduler, Condition::CONJUNCTION);

    rlFireApple->addPremise((*iteratorApple)->atIdentity,
                           (*iteratorArcher)->atIdentity, Premise::EQUAL, false);
    Boolean::TRUE_NOP, Premise::EQUAL, false);
    rlFireApple->addPremise((*iteratorApple)->atAppleColor,
                           Boolean::TRUE_NOP, Premise::EQUAL, false);
    rlFireApple->addPremise((*iteratorApple)->atAppleStatus,
                           Boolean::TRUE_NOP, Premise::EQUAL, false);
    rlFireApple->addPremise((*iteratorApple)->atAppleIsCrossed,
                           Boolean::FALSE_NOP, Premise::EQUAL, false);
    rlFireApple->addPremise((*iteratorArcher)->atArcherStatus,
                           Boolean::TRUE_NOP, Premise::EQUAL, false);
    rlFireApple->addPremise(gun->atIs Fired,
                           Boolean::TRUE_NOP, Premise::EQUAL, false);
    rlFireApple->addPremise(gun->atIdentityOfBullet,
                           cont, Premise::EQUAL, false);

    rlFireApple->addInstigation(it);
    rlFireApple->end();
    cont++;
}
  
```

Fonte: Adaptado de Banaszewski (2009)

Isto dito, com as melhorias propostas, foi possível obter ganhos de desempenho quando comparado o *Framework PON C++ 1.0* para com o *Framework PON C++ prototípico*. Para avaliar o desempenho destas duas versões foi executada a aplicação do cenário Mira ao Alvo, variando a porcentagem de *Rules* aprovadas em cada iteração, no qual a aplicação com o *Framework PON C++ 1.0* chega a atingir tempos de execução 60% menores que os da aplicação com o *Framework PON C++ prototípico*. Na Figura 32 é mostrado o resultado para a execução de 10.000

iterações, nela “Versão Antiga” se refere ao *framework C++ Prototipal* e “Versão Nova” se refere ao *framework C++ 1.0* (BANASZEWSKI, 2009).

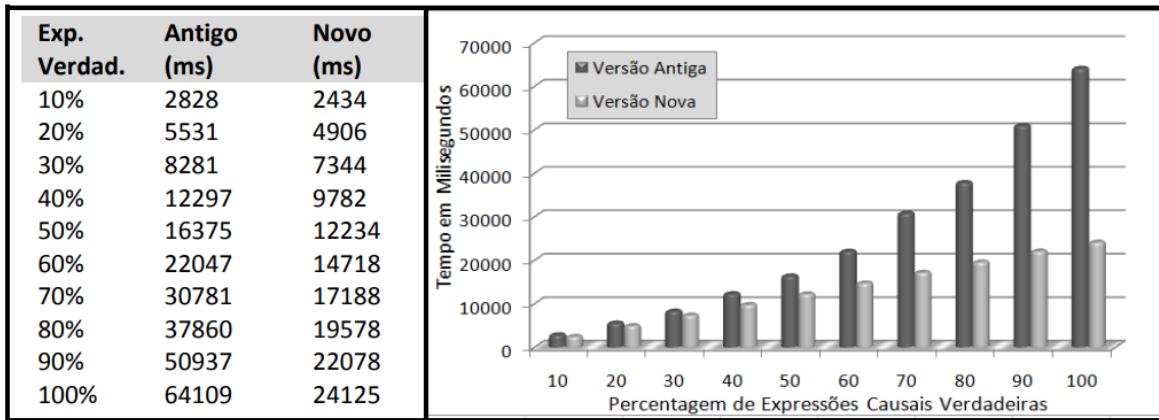


Figura 32 – Comparação do desempenho do *framework C++ 1.0* com o *framework C++ Prototipal*

Fonte: Banaszewski (2009)

2.4.1.3 Framework PON C++ 2.0

Subsequentemente, uma nova versão do *framework* foi introduzida a partir dos esforços de trabalhos de mestrado de Valença e Ronszcka em 2012 (VALENÇA, 2012; RONSZCKA, 2012). A estrutura da versão 2.0, ilustrada na Figura 33, ainda mantém uma estrutura muito similar àquela introduzida na versão 1.0, adotando a mesma estrutura de pacotes (RONSZCKA, 2012). Essa versão é considerada a principal materialização estável para o desenvolvimento de aplicações no PON, por possuir o maior grau de maturidade e estabilidade entre as materializações existentes para implementação do PON em *software* (RONSZCKA, 2019).

Nesta implementação, Ronszcka propôs a utilização de padrões de projeto para o desenvolvimento do *framework* (RONSZCKA, 2012), enquanto Valença efetuou ‘otimizações’ (*i.e.*, aprimoramentos) por meio do uso de estruturas de dados com implementação própria (nomeadamente *NOPVECTOR*, *NOPHASH* e *NOPLIST*), com melhor desempenho computacional que as estruturas da STL. Com as melhorias propostas, foi possível obter ganhos de desempenho quando comparado o Framework PON C++ 2.0 com o Framework PON C++ 1.0 (VALENÇA, 2012).

Ainda, com base nesta versão de *framework*, foi desenvolvida uma aplicação gráfica que possibilita a criação de *FBEs* e *Rules*, chamada *Wizard PON*, a qual é mostrada na Figura 34. Com o auxílio dessa aplicação é possível fazer a construção de um programa em PON em alto nível, de maneira visual. Com essa ferramenta é possível escrever a estrutura do programa em

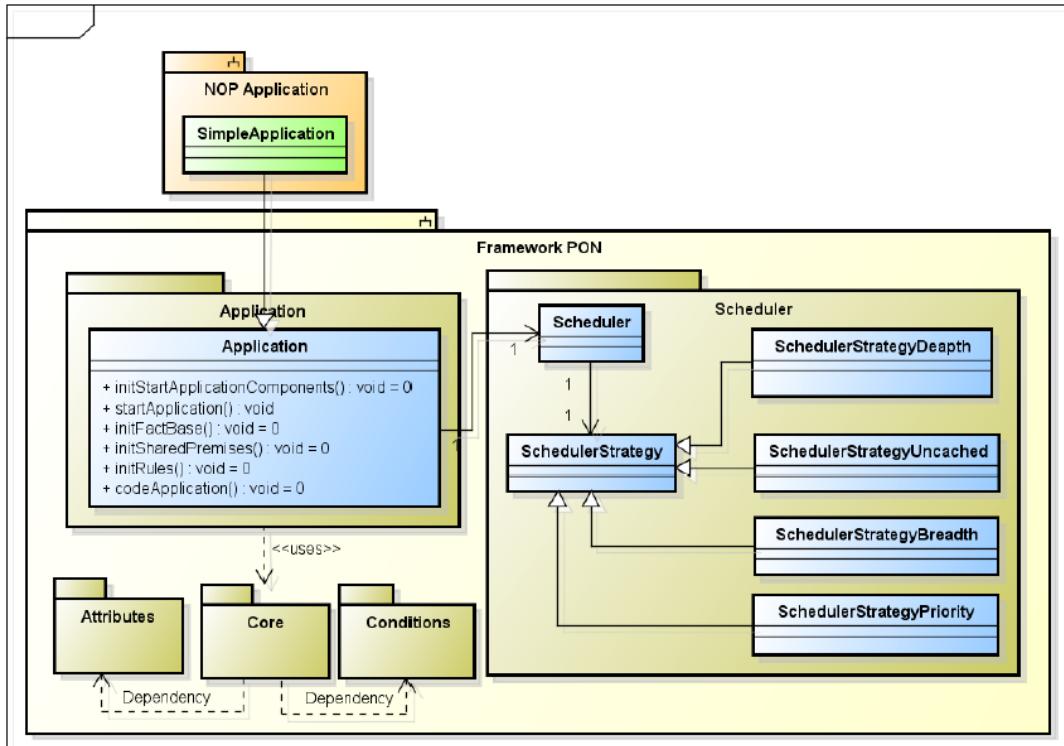


Figura 33 – Estrutura do framework C++ 2.0

Fonte: Valença (2012)

PON que passa então por um processo de geração de código para o *Framework PON C++ 2.0* (VALENÇA, 2012).

Um exemplo de aplicação para o mesmo cenário Mira ao Alvo, já descrito anteriormente na Seção 2.4.1.1, mas agora com o *Framework PON C++ 2.0* é mostrado no Código 2.14. Nota-se a sintaxe bastante similar à do *Framework PON C++ 1.0*, porém com a adição do uso do padrão de projeto *factory*.

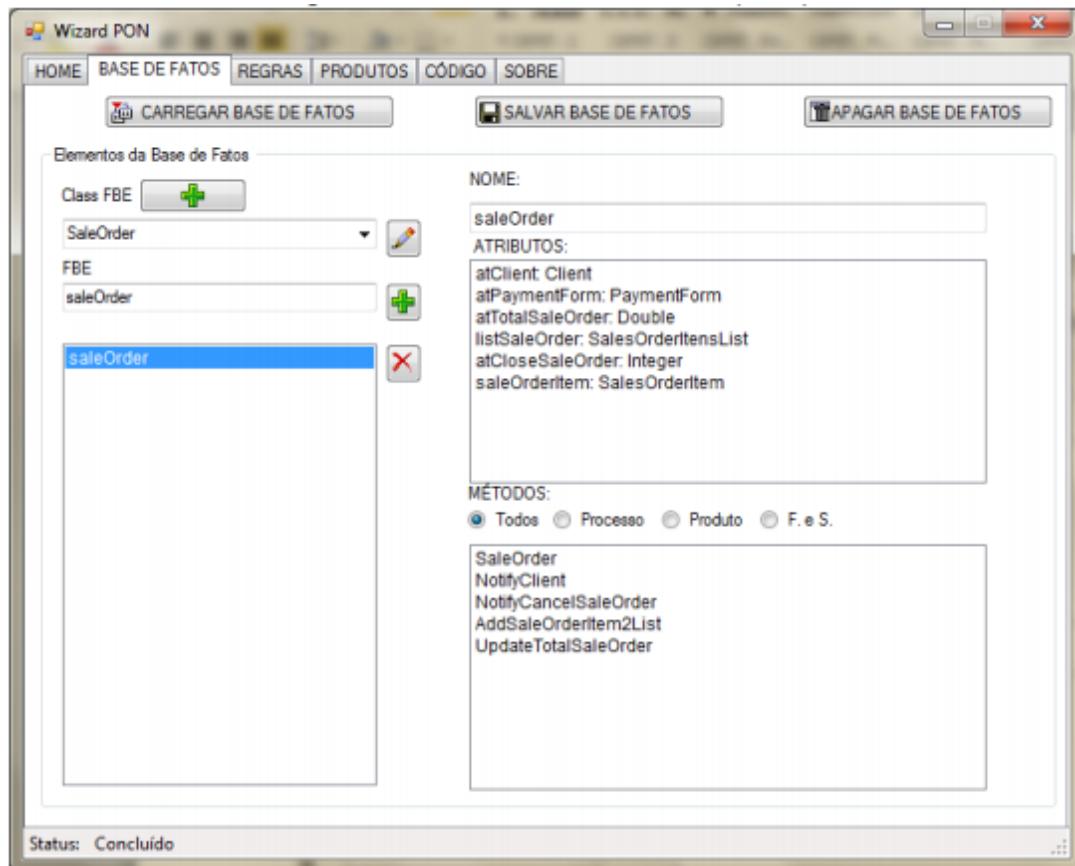


Figura 34 – Wizard PON
Fonte: Valença (2012)

Código 2.14 – Exemplo de programa com o framework C++ 2.0

```
Premise* p = elementsFactory->createPremise(gun->atIsFired, true, Premise::EQUAL, false);
for ( int i = 0; i < appleList->size(); i++ ){
    Apple* appleTmp = appleList->at(i);
    Archer* archerTmp = archerList->at(i)

    RuleObject* rlFireApple = elementsFactory->createRuleObject(
        "rule", scheduler, Condition::CONJUNCTION);
    rlFireApple->addPremise(
        elementsFactory->createPremise(
            appleTmp->atIdentity, archerTmp->atIdentity, Premise::EQUAL, false));
    rlFireApple->addPremise(
        elementsFactory->createPremise(
            appleTmp->atAppleColor, true, Premise::EQUAL, false));
    rlFireApple->addPremise(
        elementsFactory->createPremise(
            appleTmp->atAppleStatus, true, Premise::EQUAL, false));
    rlFireApple->addPremise(
        elementsFactory->createPremise(
            appleTmp->atAppleIsCrossed, false, Premise::EQUAL, false));
    rlFireApple->addPremise(
        elementsFactory->createPremise(
            archerTmp->atArcherStatus, true, Premise::EQUAL, false));
    rlFireApple->addPremise(p);
    rlFireApple->addInstigation(
        elementsFactory->createInstigation(appleTmp->mtStatusOff));
}
```

Fonte: Adaptado de Valença (2012)

A mesma aplicação Mira ao Alvo é utilizada para comparar o desempenho do *Framework PON C++ 2.0* com o *framework C++ 1.0*. Nestes testes, a utilização do *Framework PON C++ 2.0* foi com a estrutura de dados *PONVECTOR* apresentou em média 30% do tempo de processamento utilizado em relação ao *Framework PON C++ 1.0*, o que corresponderia a um ganho de desempenho de cerca três vezes, conforme observado na Figura 35. Nessa figura “*Framework Original*” se refere ao *Framework PON C++ 1.0*.

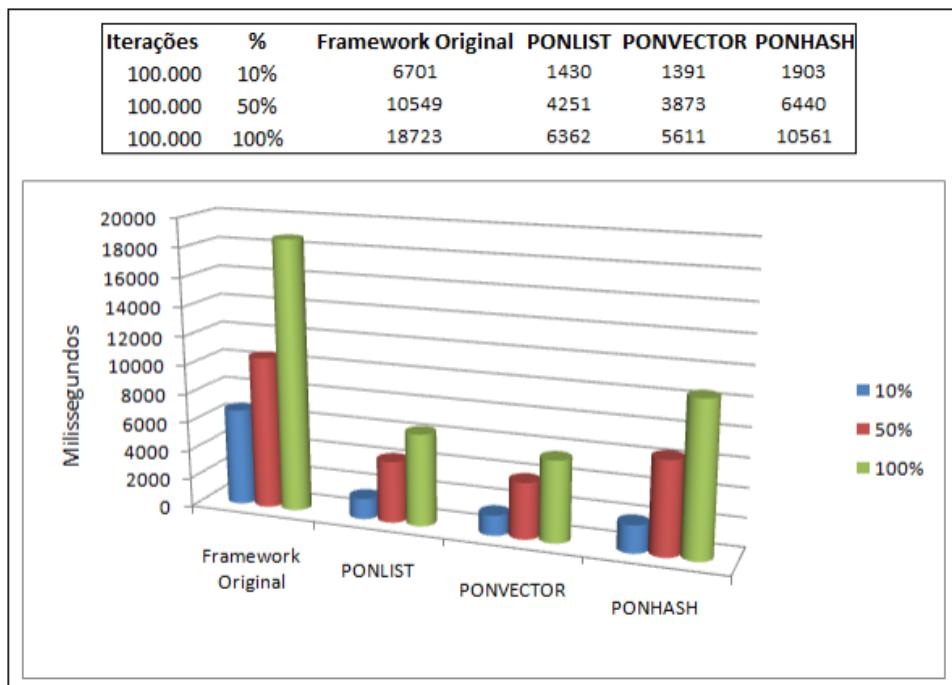


Figura 35 – Comparação do desempenho do *Framework PON C++ 2.0* com o *Framework PON C++ 1.0*
Fonte: Valença (2012)

2.4.1.4 *Framework PON C++ 3.0*

Tanto a versão do *Framework PON C++ 1.0* como o *Framework PON C++ 2.0* consideram apenas a execução em ambientes monoprocessados e monoprocesso/*single thread* em sua concepção. Ou seja, *Frameworks PON C++ 1.0* e *2.0* não consideram execução em ambiente *multithread*, multiprocesso e, menos ainda, *multicore*. Nesse sentido, foi proposto o *Framework PON C++ 3.0*. Esta versão é uma extensão da implementação do *Framework PON C++ 2.0* incluindo a execução de conjuntos de *Rules* por meio de *threads* independentes. Ademais, ele fornece uma maneira de se paralelizar elementos do PON de forma transparente, em nível de *thread* para ambientes *multicore* (BELMONTE, 2012).

A estrutura do *Framework PON C++ 3.0* é apresentada na Figura 36. Nela, os pacotes *Application* e *Core* são os mesmos do *Framework PON C++ 2.0*. Ainda com base nessa figura e

na Figura 33, podem ser destacadas algumas diferenças. Dentre elas, há a criação do *NOPVectorMulticore*, uma especialização do *NOPVECTOR* desenvolvido especialmente para trabalhar com aplicações *multicore*, agindo como um iterador das entidades do PON de forma otimizada, reduzindo o uso do *cache* da aplicação (BELMONTE, 2012; SCHÜTZ *et al.*, 2018). Também há a adição do pacote *Multicore*, cujas entidades são detalhadas abaixo:

- *SynchronizedQueue*: Fila que armazena as entidades do PON a serem executadas, de forma sincronizada, em cada *core* do processador (BELMONTE, 2012; SCHÜTZ *et al.*, 2018).
- *CoreController*: Classe responsável por enfileirar as entidades do PON quando o *CoreControllersManager* registra uma notificação. Também é responsável pela execução paralelizada das *Premises*, *Conditions* e *Methods* em cada *core* (BELMONTE, 2012; SCHÜTZ *et al.*, 2018).
- *CoreControllersManager*: Classe responsável por criar a instância do *CoreController* para cada *core* do processador. Durante a execução responsável por registrar as notificações recebidas pelas entidades do PON (BELMONTE, 2012; SCHÜTZ *et al.*, 2018).
- *Thread*: Classe de execução que controla o uso das *threads*, por meio do uso de mecanismos de exclusão mútua. Para sua implementação é utilizada a biblioteca *pthreads* (BELMONTE, 2012; SCHÜTZ *et al.*, 2018).
- *SleepCondition*: Classe que possui métodos para auxiliar o controle da execução da fila de entidades do PON. Para sua implementação é utilizada a biblioteca *pthreads* (BELMONTE, 2012; SCHÜTZ *et al.*, 2018).

Conforme descrito acima à luz da Figura 36, novas classes foram criadas para suportar o mecanismo *multithread* e usar paralelismo em nível de *thread* quando em um contexto de dois ou mais núcleos ou *cores* (*multicores*), utilizando inclusive a API *pthreads*. Esta API fornece um conjunto de bibliotecas com recursos de controle e sincronização de *threads*, por meio dos conceitos de exclusão mútua (*mutex*) para controlar o acesso e a execução de entidades compartilhadas entre as *threads*.

A classe *CoreControllermanager* é responsável por instanciar cada *FBE* da aplicação em um núcleo específico, de forma que as entidades da cadeia de notificações sejam distribuídas entre os núcleos disponíveis de forma controlada (SCHÜTZ, 2019). A fim de respeitar a execução correta das entidades PON em cada núcleo, a classe *CoreController* gerencia uma fila de entidades

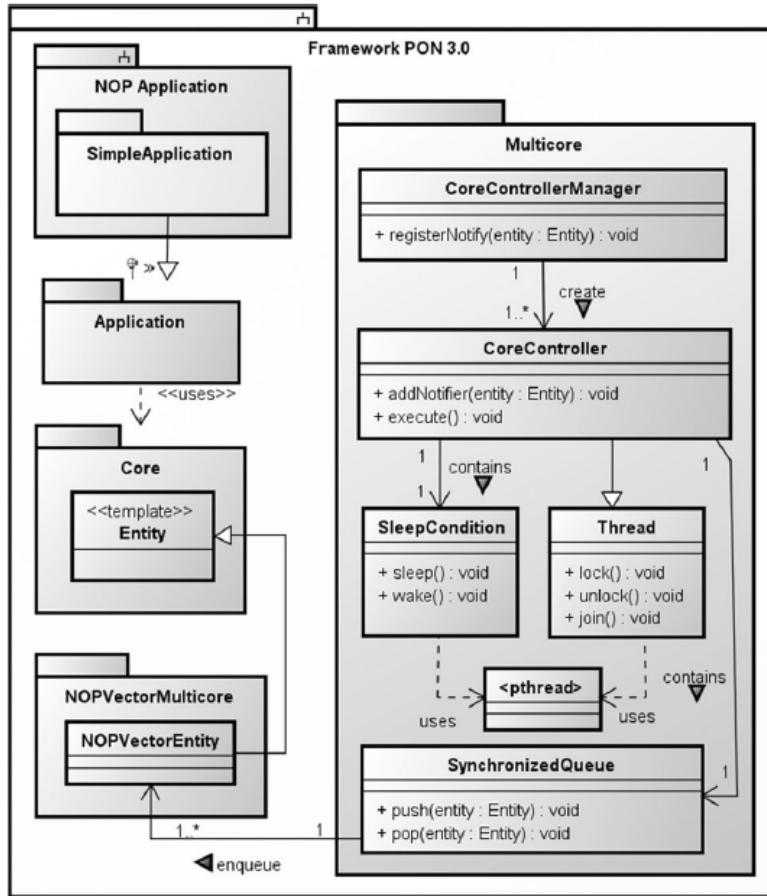


Figura 36 – Estrutura do framework C++ 3.0

Fonte: Schütz et al. (2018)

a serem executadas, a *SynchronizedQueue*. Assim, uma entidade notificada vai para esta fila de execução e, assim que um núcleo do processador esteja disponível, uma entidade é removida da fila e executada, segundo o processo ilustrado na Figura 37

Além disso, durante o desenvolvimento do *Framework PON C++ 3.0* percebeu-se a existência de um problema de *stack overflow* (estouro de pilha), possivelmente presente em todos os *frameworks* em C++ anteriores. O *stack overflow* acontece quando ocorre a chamada de muitas funções em sequência, sem nunca retornar, o que pode ser causado, por exemplo, por uma sequência muito grande de notificações que realimenta o ciclo de notificações indefinidamente. Subsequentemente no *Framework PON C++ 3.0* foi criado um desacoplamento do mecanismo de notificações, onde o *Method* não é instigado diretamente pela entidade notificante, mas sim colocada dentro de uma fila de notificações dentro de uma entidade controladora, o *CoreController*, quebrando assim esse ciclo de execução indefinido.

O processo de balanceamento da carga de trabalho do *software* executado pelo *CoreControllerManager* é realizado de acordo com o método chamado Motor de Balanceamento para

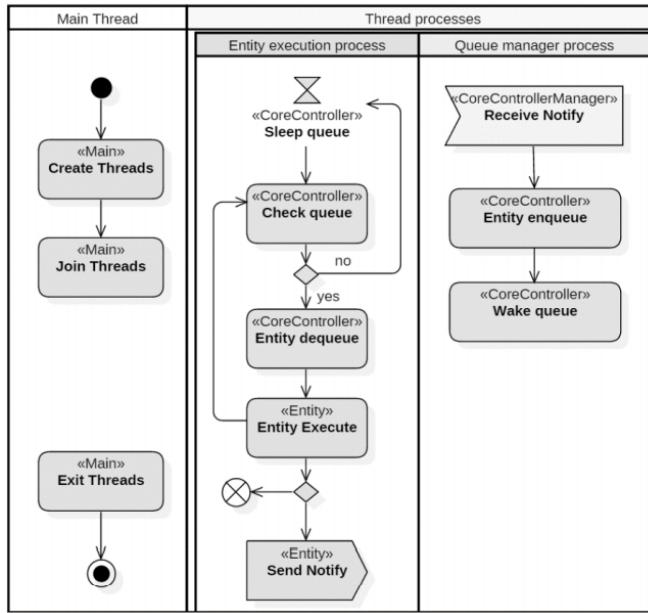


Figura 37 – Diagrama de atividades do controle de entidades do framework C++ 3.0

Fonte: Schütz et al. (2018)

Inferência Orientada a Notificações (*Load balancing engine for Notification-Oriented Inference - LobeNOI*), apresentado na Figura 38. Neste método primeiramente uma etapa de análise e alocação dinâmica verifica o número de núcleos disponíveis do processador e realiza a alocação inicial da aplicação, subsequentemente a etapa de análise e alocação dinâmica monitora a utilização dos núcleos e realiza o balanceamento da carga de trabalho em si por meio da realocação das entidades da aplicação (BELMONTE et al., 2016).

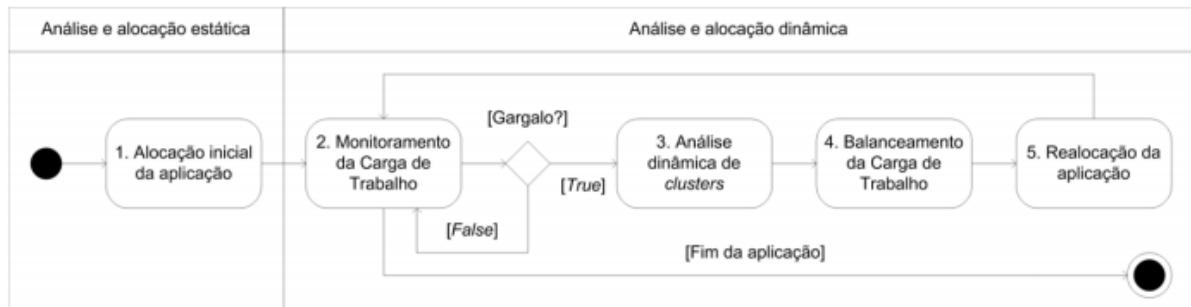


Figura 38 – Visão geral do método LobeNOI

Fonte: Belmonte et al. (2016)

Ainda com base na implementação original do *Framework PON C++ 3.0* foi implementado o NeuroPON em software paralelo (SCHÜTZ et al., 2018), o qual consiste em uma nova arquitetura de treinamento e execução de Redes Neurais Artificiais (RNA) baseados em PON. O NeuroPON acrescenta algumas estruturas novas sobre o *Framework PON C++ 3.0*, de modo a permitir a sua construção sobre este *framework*.

A estrutura das entidades adicionadas pela NeuroPON é mostrada no diagrama da

Figura 39. Essas estruturas modelam os neurônios como *FBEs*, com *Rules* e afins agregadas tratando de sua lógica dita neural, bem como a classe de controle da rede neural em si como uma *NOPApplication* naturalmente com outras *Rules* pertinentes.

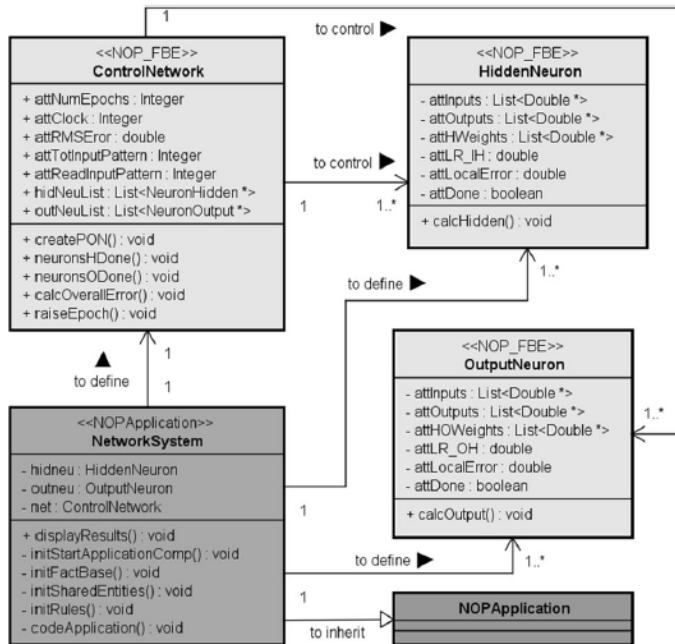


Figura 39 – Estrutura do NeuroPON

Fonte: Schütz et al. (2018)

Conforme descrito acima, o *Framework PON C++ 3.0* possui mecanismos que visam a organização e balanceamento da execução do programa de forma automática, ou seja, sem a intervenção explícita do desenvolvedor, o qual alcançou bons. Entretanto, o *Framework PON C++ 3.0* não alcançou os benefícios de desempenho esperados pelo uso de paralelização em NeuroPON, tendo o tempo de execução consideravelmente superior ao do *Framework PON C++ 3.0*.

Tais problemas de desempenho se deram mais precisamente em uma aplicação desenvolvida com a NeuroPON para o treinamento de uma RNA com *Multiplayer Perception* (MLP) utilizando o método de *Back Propagation* (BP) (SCHÜTZ et al., 2018). Os experimentos realizados com o *Framework PON C++ 3.0* permitem observar a taxa de ocupação dos núcleos, sendo mantido acima de 60% em cada núcleo durante a execução da aplicação, conforme mostrado na Figura 40⁵. Entretanto, isto não aportou um bom desempenho global em função das idiossincrasias da arquitetura do NeuroPON, como a alta conectividade entre os seus constituintes, não

⁵ Experimentos realizados com um processador Core i7-3770 3.5 Ghz, 32 GB de memória RAM DDR3 1357MHz, com o sistema operacional Linux Ubuntu 16.04 LTS 64 bits.

ser bem suportada pela abordagem do *Framework PON C++ 3.0* conforme detalhado na tese de Schütz *et al.* (2018).

No final das contas, no contexto dado, a execução de forma paralelizada trouxe um aumento do tempo de execução do programa em PON. De forma geral, mecanismos de controle de execução paralela (*i.e., mutex, threads*) causam um aumento do tempo de processamento. A utilização de um número pequeno de neurônios nos experimentos faz com que as operações paralelizadas sejam realizadas de forma muito rápida, fazendo com que a aplicação gaste a maior parte do tempo nos mecanismos de controle, de modo que a paralelização não traz melhoria nos tempos de execução (SCHÜTZ *et al.*, 2018). Os resultados, apresentados na Figura 41, mostram que conforme aumenta o número de núcleos utilizados, maior o tempo de execução (SCHÜTZ *et al.*, 2018).



Figura 40 – Taxa de utilização dos núcleos da CPU no treinamento de RNA MLP com método BP
Fonte: Schütz *et al.* (2018)

Os resultados da NeuroPON paralela em *Framework PON C++ 3.0* poderiam não depor contra o *Framework PON C++ 3.0* em si, mas contra a NeuroPON finalmente. Entretanto, experimentos com a NeuroPON em outro *framework* demonstrariam que o problema enfim não seria nela, mas sim no *Framework PON C++ 3.0*. Os resultados a NeuroPON neste outro *framework* serão considerados subsequentemente neste trabalho e se encontram na dissertação de mestrado de Negrini *et al.* (2019a).

2.4.1.5 JuNOC++

Todos os *frameworks* em C++ apresentados até agora foram construídos como evoluções com base nas versões anteriores. Além destes *frameworks* em C++, também há o JuNOC++ (*Just*

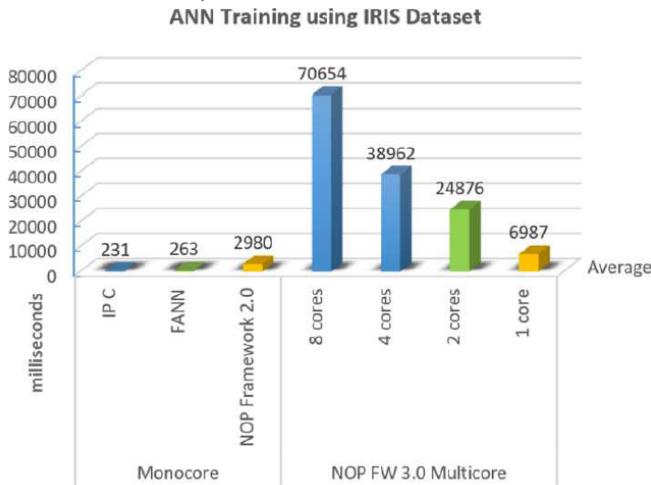


Figura 41 – Tempos de execução (em milissegundos) do treinamento de ANN MLP com método BP
Fonte: Schütz *et al.* (2018)

a *Notification Oriented C++*). Este *framework* se destaca por surgir não como uma evolução dos *frameworks* anteriores, mas sim como uma tentativa de Chierici (2020) de criar um *framework* do PON com base na fundamentação apresentada na dissertação de Banaszewski (2009). Desta forma, Chierici (2020) propõe uma estrutura que deixa de ser mera reprodução dos *frameworks* existentes, mas sim um novo *framework* independente com foco em programação em alto nível, expressividade e desempenho.

Do ponto de vista da estrutura do *framework*, o JuNOC++ utiliza uma estrutura baseada no pacote *Core* do *Framework PON C++ 1.0*, incluindo melhorias principalmente observadas pela utilização de *templates* para a implementação de *Attributes* e *Premises*. Esta estrutura de classes é ilustrada na Figura 42.

O JuNOC++ utiliza recursos avançados, ditos modernos, da linguagem de programação C++ e conceitos de programação genérica em seu desenvolvimento. Os códigos 2.15 e 2.16 apresentam, respectivamente, um *FBE* e sua respectiva implementação com o JuNOC++. No código em C++ com o JuNOC++ podem ser observadas as melhorias alcançadas no sentido de facilitar a programação em alto nível. Com o uso extensivo de *macros* e sobrecarga de operadores a construção das entidades do PON, como a *Rule rlChange* em destaque neste código, pode ser feita de maneira muito similar à sua construção em *LingPON* e natural ao PON.

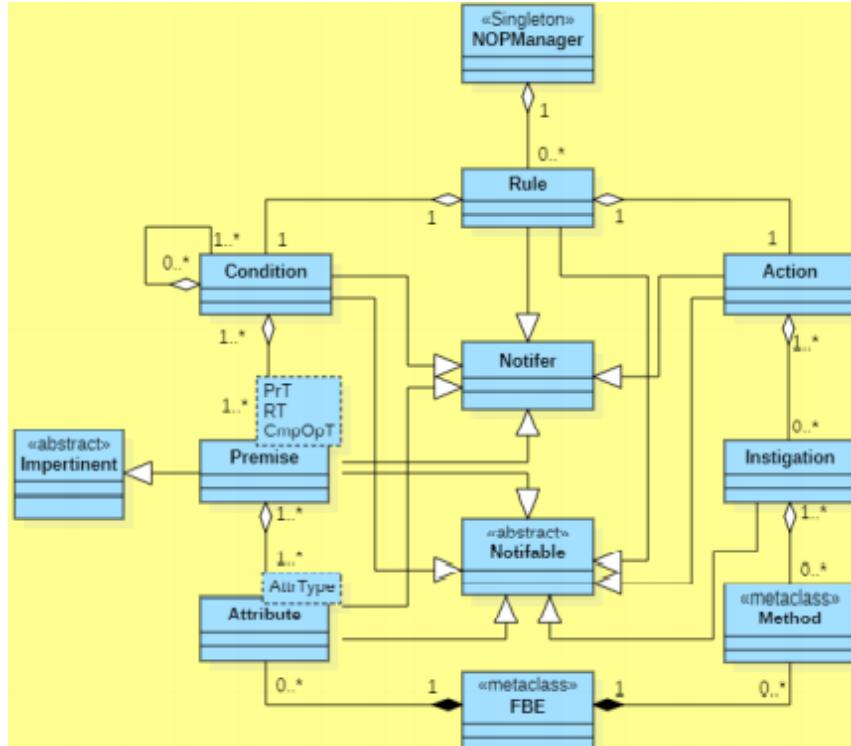


Figura 42 – Diagrama de classes do JuNOC++

Fonte: Chierici (2020)

Código 2.16 – Código em C++ para exemplo no JuNOC++

Código 2.15 – FBE para exemplo no JuNOC++

```
fbe Main
{
    private boolean atStatus = false
    private boolean atStatus2 = false

    private method mtChange()
    {
        this.atStatus=true
    }
    rulerlChange this.atStatus == true
        or this.atStatus2 == true
    {
        call(this.mtChange())
    }
    main
    {
        this.atStatus=true
        this.atStatus2=true
    }
}
```

Fonte: Chierici (2020)

«metaclass» Main

```
class Main
{
private:
    NOP::Attribute<bool> atStatus;
    NOP::Attribute<bool> atStatus2;
    NOP::Rule rlChange;
public:
    Main();
    private:
    void mtChange();
};

Main::Main():
    atStatus{false, "atStatus"},
    atStatus2{false, "atStatus2"},
    rlChange("rlChange")
{
    RULE(rlChange, atStatus == true
        or atStatus2 == true)
        INSTIGATE([&] () {mtChange(); })
    ENDRULE

    atStatus = true;
    atStatus2 = true;
}

void Main::mtChange()
{
    atStatus=true;
}
```

Fonte: Chierici (2020)

Em experimentos realizados, o JuNOC++ inclusive demonstrou desempenho superior aos outros *frameworks* em linguagem de programação C++ considerados, nomeadamente os

*Frameworks PON C++ 2.0 e 4.0*⁶. Destaca-se apenas que a versão do *Framework PON C++ 4.0* neste experimento era uma versão prototípica, que não reflete o desempenho real da sua versão final, que apresenta desempenho superior ao *Framework PON C++ 2.0*, conforme será apresentado no Capítulo 4.

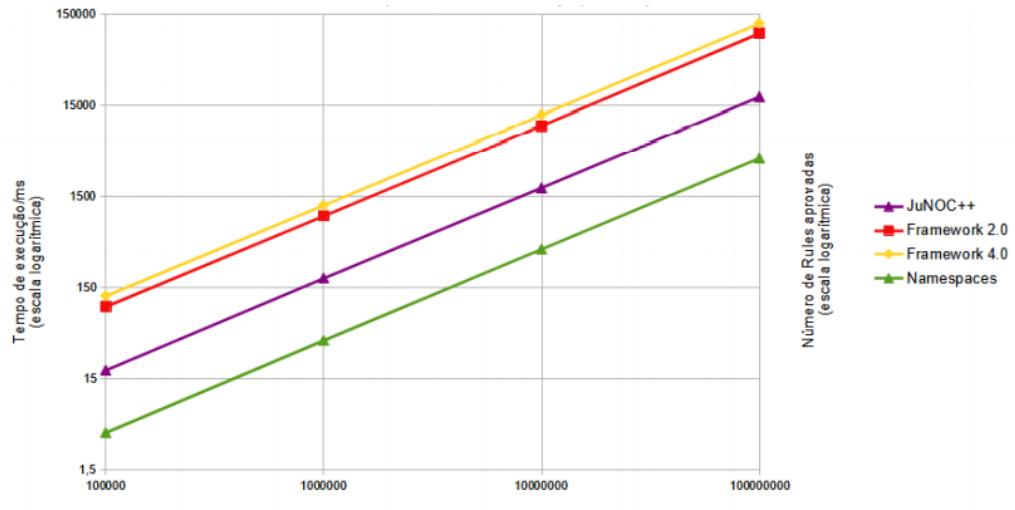


Figura 43 – Gráfico dos resultados de experimento com o JuNOC++
Fonte: Chierici (2020)

O *Framework PON C++ 4.0* em JuNOC++ foram desenvolvidos em paralelo, assim sendo, muitas das melhorias propostas são consideradas em ambos. O JuNOC++ aproveitou principalmente as implementações com *templates* introduzidas pelo *Framework PON C++*, enquanto este buscou inspiração nas melhorias introduzidas pelo JuNOC++ para facilitar a programação em alto-nível. Ainda assim, ambos os *frameworks* possuem propostas diferentes, sendo que o *Framework PON C++ 4.0* se preocupa em disponibilizar uma materialização mais estável, com desenvolvimento orientado a testes, incorporando conceitos de paralelismo, o JuNOC++ é ainda uma materialização de cunho prototípico, explorando a flexibilização da construção das entidades, com foco na programação em alto nível.

2.4.1.6 Implementações realizadas com o *Framework PON C++ 2.0*

Inicialmente, de modo a possibilitar levantar as deficiências do *Framework PON C++ 2.0*, apontadas na Seção 2.9, e servir também como conhecimento de base para a implementação de um novo *framework*, foram desenvolvidas aplicações utilizando o *Framework PON C++ 2.0*. Com o desenvolvimento destas aplicações era objetivado ganhar experiência no desenvolvimento

⁶ Neste experimento também foi considerada uma implementação em *namespaces*, que é um alvo de compilação da LingPON, apresentado neste trabalho na Seção 2.4.7

em aplicações com o PON, em particular com o *Framework PON C++ 2.0*, de forma a facilitar o entendimento dos pontos de melhoria a serem implementados. Essas implementações são descritas com maiores detalhes nas seções seguintes.

2.4.1.6.1 Futebol de Robôs

A primeira aplicação desenvolvida pelo autor deste trabalho utilizando o PON foi no âmbito de Futebol de Robôs. Esta aplicação foi desenvolvida no contexto da disciplina de Tópicos Especiais em Engenharia da Computação: Paradigma Orientado a Notificações, via PPGCA, na UTFPR, durante o terceiro trimestre de 2019 (LIMA *et al.*, 2020). Tal implementação inspirou-se em trabalho prévio desenvolvido por Santos (2017), ainda que esta nova implementação tenha sido a partir de uma versão mais prototípica do trabalho dele por ser a que se encontrou disponível. Em tempo, o trabalho foi conjunto entre cinco discentes da disciplina, sendo que cada qual montou um dado conjunto de *Rules* subsequentemente integradas (LIMA *et al.*, 2020).

Neste âmbito, de forma a aplicar os conhecimentos adquiridos durante a disciplina foi proposto o desenvolvimento de uma aplicação de controle de robôs em *Framework PON C++ 2.0*, em contexto de Futebol de Robôs, para execução em ambiente de simulação. Essa aplicação consiste do controle de duas equipes, compostas cinco robôs cada, executada em um ambiente virtual que simula um campo de futebol. Uma tela da execução deste ambiente de simulação é mostrada na Figura 44, na qual é possível observar o campo e os robôs de ambos os times.

Nesse cenário, a parte da aplicação desenvolvida em *Framework PON C++ 2.0* é responsável por receber e processar comandos de um juiz virtual que envia os comandos da partida (*e.g.*, início, parada, falta etc.), assim como a aplicação desenvolvida em *Framework PON C++ 2.0* é responsável por controlar de forma autônoma todos os robôs de ambas as equipes. Cada equipe é controlada por uma instância independente da aplicação em PON (LIMA *et al.*, 2020). Para o desenvolvimento em PON desta aplicação foi utilizado o *Framework PON C++ 2.0* conforme o que definido em Ronszcka (2012) e Valença (2012).

No escopo deste trabalho foi utilizado um ambiente previamente configurado e funcional, contendo um conjunto básico de *Rules* e *FBEs* já existentes, fruto de uma versão prototípica dos esforços da dissertação de mestrado de Santos (2017), conforme já especificado acima. Baseado nessa estrutura inicial, representada pelo diagrama de classes da Figura 45, foram apenas desenvolvidas novas *Rules*, utilizando os *FBEs* já existentes *RobotPON* e *StrategyPON*. O artigo-relatório contendo todo o desenvolvimento deste projeto é apresentado na íntegra no



Figura 44 – Tela principal do ambiente de simulação do futebol de robôs
Fonte: Lima *et al.* (2020)

Apêndice A, para fins de registro na dissertação.

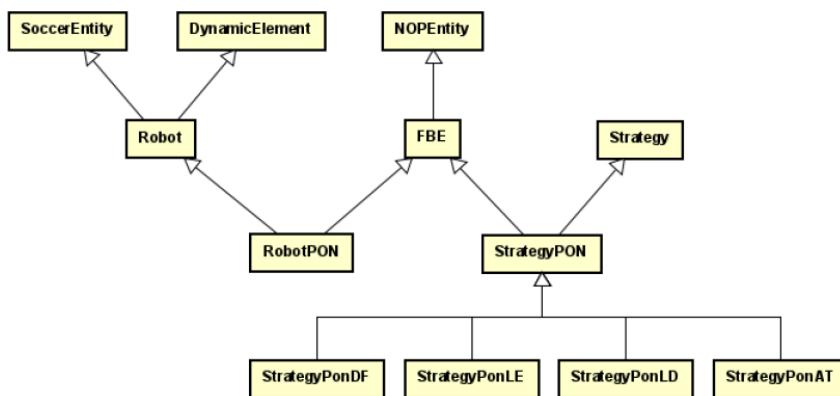


Figura 45 – Diagrama de classes do futebol de robôs
Fonte: Lima *et al.* (2020)

Durante o desenvolvimento do trabalho foram encontradas algumas dificuldades, principalmente no que se refere ao uso do ambiente de desenvolvimento proposto. Isto primeiramente pelo fato do mesmo ser disponibilizado inicialmente na forma de uma máquina virtual que necessitava se configurada e instalada. Ainda, uma vez instalada a máquina e o ambiente como um todo, percebeu-se que não continha todas as *Rules* e, principalmente, não continha *FBEs* com seus *Methods* e *Attributes* de acordo com a dissertação de Santos (2017), concluindo-se subsequentemente que se tratava de um código mais prévio ou prototípico (LIMA *et al.*, 2020).

No contexto acima relatado, especialmente devido ao ambiente conter um número de *FBEs* implementados muito pequeno, não havia recursos para se fazer a criação das novas *Rules* de forma muito abundante. Ainda, perceberam-se outros fatores em algo obstantes, como o fato

de recursos mais avançados do *framework*, como *Master Rule* e *SubConditions*, não possuírem uma construção tão intuitiva quanto seria possível, bem como não haver instruções de utilização adequadas, o que dificultou a implementação das *Rules* que foram criadas.

Em função destas dificuldades postas, também percebeu-se a curva não tão suave (conforme habilidade de cada qual) de aprendizado do *Framework PON C++ 2.0*, ao menos neste exemplo em questão. Não obstante as dificuldades, a criação de conjuntos de *Rules* pelos discentes permitiu bem compreender o PON. Ademais, também foi possível observar propriedades do PON, como os conjuntos de *Rules* trabalhando harmonicamente no final da implementação, isto sem esforços extraordinários de integração em função do nível de desacoplamento entre eles.

2.4.1.6.2 Jogo NOPUnreal

A segunda aplicação, já desenvolvida após certa familiarização com o desenvolvimento em PON foi realizada sob a forma de um jogo. A proposta desta aplicação era realizar a integração do *Framework PON C++ 2.0* com uma API complexa de desenvolvimento de jogos. Para este fim foi escolhida a Unreal Engine. Em tempo, este foi um trabalho desenvolvido no contexto da disciplina de Estudo Especial — Paradigmas de Programação, via CPGEI, na UTFPR, durante o primeiro trimestre de 2020 (NEVES, 2020)⁷. A estrutura básica do jogo consiste em uma nave controlada pelo jogador com o objetivo de neutralizar todos os inimigos. A Figura 46 mostra uma tela do jogo desenvolvido, na qual o jogador é representado pela nave cinza, e os inimigos pelas naves amarelas.

A Unreal Engine foi escolhida por ser o motor gráfico mais utilizado no desenvolvimento de jogos e aplicações gráficas comerciais, chegando a ser utilizada em cerca de 20% dos jogos de computador (NEVES, 2020). O desenvolvimento desta aplicação foi realizado utilizando tanto com o POO como o PON, de modo a permitir a comparação da facilidade de programação utilizando os dois paradigmas. De modo geral o desenvolvimento foi mais fácil e rápido utilizando o PON, chegando a obter um tempo de desenvolvimento até 38.62% menor, porém a verbosidade do *Framework PON C++ 2.0* resultou em um código-fonte com 12.16% mais linhas que a implementação equivalente no POO (NEVES, 2020).

Um dos elementos que dificulta a aplicação do PON neste projeto é a rigidez da estrutura da API da Unreal Engine, sendo implementada utilizando o POO e fazendo uso extensivo de lógica sequencial na sua execução. Neste contexto, o desenvolvedor é obrigado a utilizar as

⁷ Este trabalho também foi apresentado sob a forma de minicurso no SICITE 2020

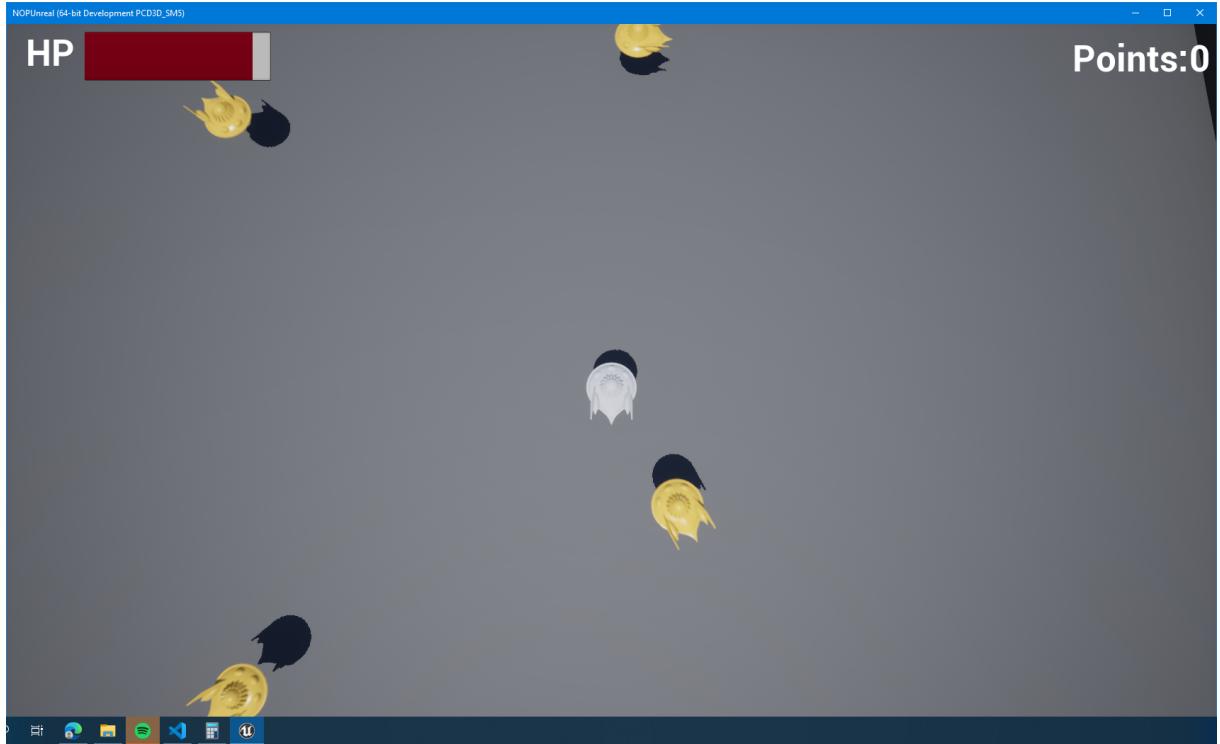


Figura 46 – Screenshot do jogo desenvolvido
Fonte: Neves (2020)

estruturas e classes próprias da Unreal Engine para a sua implementação. Na Figura 47⁸ é mostrado o diagrama de classes do jogo desenvolvido, no qual é importante destacar que as classes implementadas em PON, derivadas da classe *FBE*, também são derivadas das classes próprias de Unreal Engine (*APawn*, *AActor*, *UObject*, *UActorComponent* e *AInfo*). Deste modo pode ser dito que o código implementado é, na verdade, multiparadigma, pois utiliza tanto o PON como o POO na sua implementação.

Nesse contexto de programação multiparadigma, o PON é utilizado para implementar a lógica referente ao comportamento dos inimigos e às regras do jogo, enquanto o POO é utilizado para a implementação dos elementos gráficos da aplicação. O artigo-relatório contendo o desenvolvimento deste projeto, inclusive contendo a especificação completa de todas as *Rules* implementadas, é apresentado na íntegra no Apêndice B.

A implementação deste projeto realçou algumas das deficiências ou imperfeições do Framework PON C++ 2.0. Um dos principais problemas foi a baixa flexibilidade de tipos, que fica bem evidente devido à Unreal Engine utilizar muitos tipos próprios, como estruturas, enumerações e classes (*e.g. FVector, FString* etc.).

⁸ Nos diagramas de classe construídos com a ferramenta PlantUML o símbolo C em um círculo verde identifica classes concretas, já o símbolo A sobre um círculo azul identifica classes abstratas

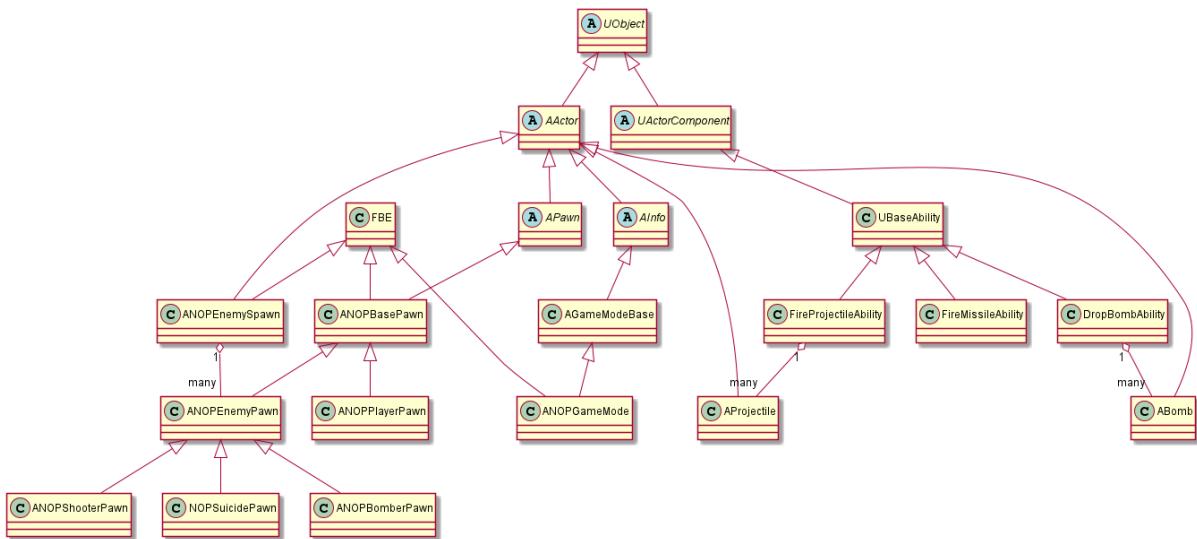


Figura 47 – Diagrama de classes do jogo desenvolvido

Fonte: Neves (2020)

O fato do *Framework PON C++ 2.0* somente permitir o uso de seus tipos pré-definidos (*int, double, bool e string*) faz com que seja necessário converter os tipos próprios da Unreal Engine para a utilização em conjunto com as entidades do PON. Alguns exemplos que melhor ilustram esse problema são apresentados abaixo. No Código 2.17, um *Attribute* do tipo *int* é utilizado para implementar um tipo que utiliza uma enumeração, exigindo que seja realizada uma conversão de tipos.

Código 2.17 – Uso de *static_cast* para converter enumerações

```
INTEGER(this, atStrategym static_cast<int>(ENOPEEnemyStrategy:EFollow));
```

Fonte: Autoria própria

Por sua vez, também o Código 2.18 mostra a estrutura *FVector*, muito utilizada em todo o código para representar a posição no espaço tridimensional dos objetos. Porém, o uso dela em *Premises* e *Attributes* é impossível, pois não é um tipo básico suportado. Assim, faz necessários desmembramentos e afins para seu uso.

Código 2.18 – Estrutura *FVector* da Unreal Engine

```
struct FVector {
    float X;
    float Y;
    float Z;
}
```

Fonte: Autoria própria

Por fim, o Código 2.19 apresenta uma situação na qual é necessário criar *Premises* muito semelhantes, porém, com avaliações opostas. Devido à baixa flexibilidade algorítmica não permitir a criação de *Conditions* com composição de *Premises* utilizando declaração de operações lógicas mais diversas sobre elas, como seria o caso de uma operação de negação, ocorre este tipo de redundância estrutural.

Código 2.19 – Uso de premissas redundantes no PON

```
PREMISE(prIsFarFromTarget, atDistanceToTarget, 800.0f, Premise::GREATEROREQUAL,
        Premise::STANDARD, false);
PREMISE(prIsNOTFarFromTarget, atDistanceToTarget, 800.0f, Premise::SMALLERTHAN,
        Premise::STANDARD, true);
```

Fonte: Autoria própria

Em suma, o desenvolvimento de ambas estas aplicações permitiu principalmente observar as limitações do *Framework PON C++ 2.0*, explorados em detalhes na Seção 2.9. Ainda, essas experiências serviram também como fator motivador para a proposta do *Framework PON C++ 4.0*.

2.4.1.7 Considerações sobre os *frameworks* do PON em C++

De forma geral, as materializações do PON por meio de *frameworks* em linguagem de programação C++ representam uma grande contribuição ao estado da técnica do PON, possibilitando o desenvolvimento de aplicações que permitiram avaliar e comparar o desempenho do PON (BANASZEWSKI, 2009; RONSZCKA, 2012; VALENÇA, 2012). Nas materializações de *frameworks* em C++, foram desenvolvidas a maior parte das aplicações feitas em PON conforme (RONSZCKA, 2012; SANTOS, 2017; RONSZCKA, 2019).

Houve também o desenvolvimento de diversas aplicações como os *frameworks* do PON em C++, nomeadamente a aplicação Mira ao Alvo (BANASZEWSKI, 2009; RONSZCKA, 2012; VALENÇA, 2012), o Futebol de Robôs (SANTOS, 2017; LIMA *et al.*, 2020), o jogo NOPUnreal (NEVES, 2020) e também o NeuroPON (BELMONTE, 2012; BELMONTE *et al.*, 2016; SCHÜTZ, 2019). Todas estas aplicações desenvolvidas são interessantes em pertinentes, demonstrando diversos casos de uso do PON, assim como servindo de *benchmarks*, como o caso da aplicação Mira ao Alvo (que posteriormente evoluí sob nova forma de rede de sensores), frequentemente utilizada para comparações de desempenho entre os *frameworks*.

Apesar disso, essas aplicações têm utilização limitada ao contexto do grupo de pesquisa do PON, de forma que estes *frameworks* carecem de *benchmarks* universalizados, mais conhecidos

dos pela comunidade científica. Salienta-se inclusive que aplicações como o Futebol de Robôs não são particularmente adequadas para fins *benchmark* em si, por ser uma aplicação que não se interessa em tempos de execução.

2.4.2 Frameworks PON Java/C# 1.0

Não obstante, além das materializações em C++, também houve a materialização na forma de *framework* nas linguagens Java e C#, ainda que assaz prototípicas ou ao menos ainda não utilizadas largamente. Esta seção trata de forma unificada destas duas materializações em linguagens diferentes, pois as mesmas são materializações equivalentes principalmente ao *Framework PON C++ 1.0* e sob a forma de adaptações dele para tais linguagens (HENZEN, 2015).

O objetivo destas materializações foi adaptar o *Framework PON C++ 1.0* para Java e C# utilizando técnicas de programação semelhantes, que padronizam e facilitam a manutenção do código. Estas materializações se justificam pelo grande número de programadores que utilizam as linguagens Java e C#, sendo muito utilizadas, principalmente, em aplicações móveis e web (HENZEN, 2015). A linguagem Java também é adotada como principal linguagem no ensino de programação em diversas universidades, o que contribui para o grande número de programadores utilizando a mesma (HENZEN, 2015).

Para a comparação dos resultados entre as materializações de C++, Java e C# foi desenvolvida uma aplicação para controle de um portão eletrônico. A implementação desta aplicação consiste na criação de duas *Rules*, uma para abrir e outra para fechar o portão, com base nos *FBEs* do portão, que encapsula o estado do portão (aberto ou fechado), e controle remoto, que encapsula o estado do botão (pressionado ou não) (HENZEN, 2015). As particularidades das implementações em cada uma das linguagens são apresentadas nas seções seguintes.

2.4.2.1 Framework PON Java 1.0

Algumas alterações foram necessárias para implementar o *framework* em Java, como o fato de a linguagem Java não possuir o conceito herança múltipla, para isto sendo utilizado como alternativa o conceito de *Interface*. Um trecho da aplicação desenvolvida com o *Framework PON Java 1.0* é apresentado no Código 2.20.

Código 2.20 – Exemplo de *Rule* do portão eletrônico em Java

```

Condition cond1R01 = new Condition(Condition.CONJUNCTION);
cond1R01.addPremise(new Premise(remoteControl.atRemoteControlStatus, NBoolean.TRUE_NOP,
    Premise.EQUAL, false));
SubCondition subcond1R01 = new SubCondition(SubCondition.DISJUNCTION, false);
subcond1R01.addPremise(new Premise(gate.atGateStatus, 0, Premise.EQUAL, false));
subcond1R01.addPremise(new Premise(gate.atGateStatus, 5, Premise.EQUAL, false));
cond1R01.addSubCondition(subcond1R01);

Action actR01 = new Action();
actR01.addInstigation(new Instigation(gate.mtOpening));
actR01.addInstigation(instStatusOff);

Rule rule1 = new Rule("Opening gate", scheduler, cond1R01, actR01, false);

```

Fonte: Adaptado de Henzen (2015)

2.4.2.2 Framework PON C# 1.0

Por sua vez, na implementação de *framework* com a linguagem C# foi utilizado conceito de *Delegate* no lugar de ponteiros para funções (HENZEN, 2015). Um trecho da aplicação desenvolvida com o *Framework PON C# 1.0* é apresentado no Código 2.21.

Código 2.21 – Exemplo de *Rule* do portão eletrônico em C#

```

Condition cond1 = new Condition(Condition.CONJUNCTION);
cond1.addPremise(new Premise(remoteControl.bIsPressed,NBoolean.TRUE_NOP,Premise.EQUAL,false));
cond1.addPremise(new Premise(gate.GateStatus,true,Premise.EQUAL,false));

Action action1 = new Action();
action1.addInstigation(new Instigation(remoteControl.bIsPressed,false));
action1.addInstigation(new Instigation(gate.mtOpenGate));

Rule rule1 = new Rule("Open gate", scheduler, cond1, action1, false);

```

Fonte: Adaptado de Henzen (2015)

2.4.2.3 Comparações

Foram realizados testes com 10.000, 100.000, 20.0000 e 500.000 iterações de fechamento do portão⁹, com os resultados mostrados na Figura 48. O desempenho dos *frameworks* em Java e C# foi satisfatório, inclusive superando o desempenho do *Framework PON C++ 1.0* neste cenário, sendo o *framework* Java aquele que apresentou o melhor desempenho. Entretanto, seriam necessários mais experimentos para avaliar como ficariam esses desempenhos, inclusive em relação ao *Framework PON C++ 2.0*.

Isto dito, o desenvolvimento destes *frameworks* demonstrou ser possível a implementação de um *framework* do PON em Java e C#, inspirado no *Framework PON C++ 1.0*, sem haver

⁹ Para este teste foi utilizado um computador modelo Apple Macbook Pro, Core i5 2.5 Ghz, 6 GB de memória RAM, com o sistema operacional Windows 10 Preview.

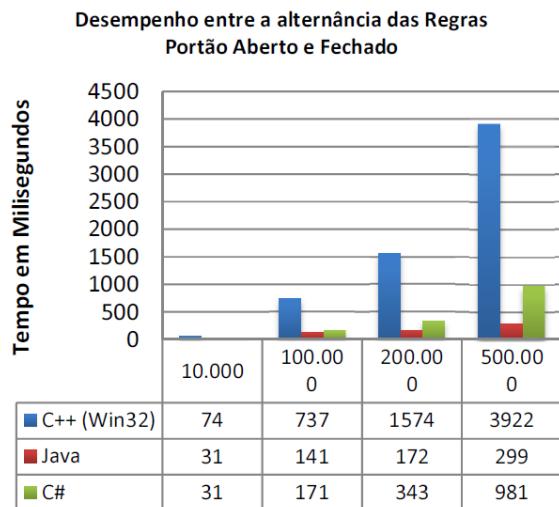


Figura 48 – Comparação de desempenho das aplicações em C++, Java e C#
Fonte: Henzen (2015)

descaracterização das técnicas empregadas na implementação original (HENZEN, 2015). Apesar do aspecto prototípico destes *frameworks*, seu desenvolvimento abriu novos caminhos de pesquisa para o PON explorando tais plataformas, como o subsequente desenvolvimento do *Framework PON C# IoT*.

2.4.3 Framework PON C# IoT

Na materialização *Framework PON C# IoT*, desenvolvida à luz dos esforços de Oliveira (2019) introduzem uma versão com o foco em Internet das Coisas (*IoT - Internet of Things*). Do ponto de vista de implementação esta versão se inspira no *framework PON C++ 2.0* e adapta/evolui o *Framework PON C# 1.0*, com o seu diferencial sendo a proposta da distribuição em rede das entidades do PON, permitindo a aplicação dos conceitos de IoT, por meio de paralelismo, distribuição e a capacidade de reconfiguração das *Rules* em tempo de execução (OLIVEIRA, 2019).

A Figura 49 apresenta o diagrama de classes do *Framework PON C# IoT*. Apesar de apresentar uma estrutura bastante similar ao *Framework PON C++ 2.0* e *C# 1.0* também é possível observar neste diagrama a adição de classes específicas para aplicação no ambiente distribuído, como a classes *SensorFBE* e *NotificationNetwork*.

Este *Framework PON C# IoT* utiliza uma unificação das entidades *FBE* e *Attribute*, chamada de *SensorFBE*, na qual o *FBE* possui apenas um único *Attribute*, relativo a leituras do sensor ou atuador. O *SensorFBE* também possui alguns outros atributos não relativos ao

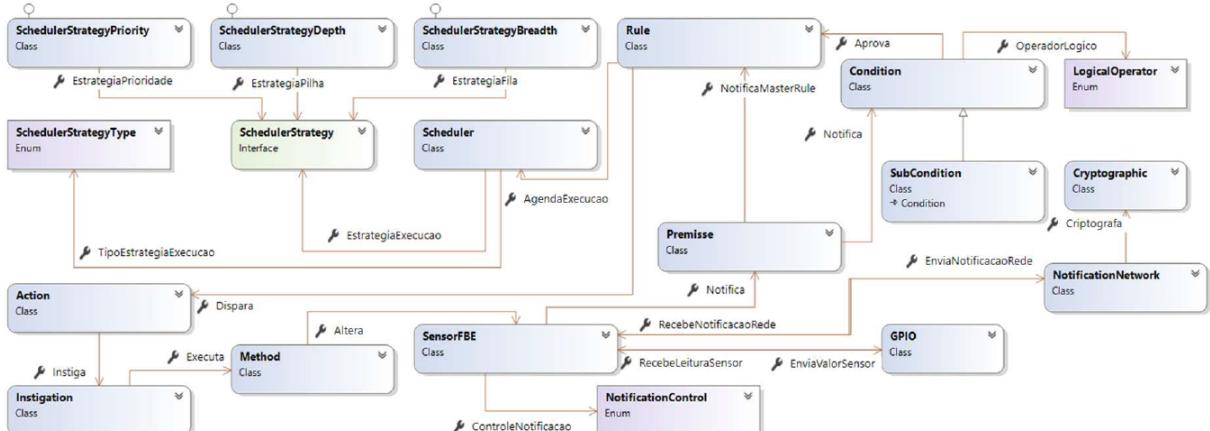


Figura 49 – Diagrama simplificado de classes do Framework PON C# IoT
Fonte: Oliveira (2019)

mecanismo de notificações do PON, como identificador único, nome, tipo, intervalo de leitura, e outras definições relativas ao sensor/atuador¹⁰ (OLIVEIRA, 2019). A entidade *SensorFBE* ainda possui outras variáveis que controlam a forma de notificação das *Premises*, como o controle de notificação, podendo ser *Always*, *On change* ou *Never*. As entidades *SensorFBE* podem ser executadas de forma distribuída, sendo capazes de notificarem *Premises* sendo executadas em outros dispositivos remotos.

O processo de notificação é executado de forma paralela, utilizando a função *Parallel.ForEach* do C#, de modo que cada notificação gerada pelo *SensorFBE* é processada em uma *thread* diferente (OLIVEIRA, 2019). Ainda, nas situações nas quais a entidade a ser notificada pode estar em um ambiente remoto, como em outro dispositivo, a notificação é enviada utilizando o mecanismo que permite o envio para outros clientes, o *IoT.NotificationClient.Notification*, com o uso do método *Send*. Deste modo a notificação é enviada para os outros dispositivos via protocolo TCP/IP (OLIVEIRA, 2019). A Figura 50 ilustra as atividades executadas durante o processo de leitura de um sensor físico utilizando o *Framework PON C# IoT*.

O *Framework PON C# IoT* também faz o uso do conceito de impertinência dinâmica de *Attributes* e *Premises*. Neste mecanismo, sem a interferência do desenvolvedor, a própria *Condition* define a impertinência de suas *Premises*, baseado no valor lógico das outras *Premises* e estas de seus respectivos *Attributes*, baseado na relevância de cada *Premise* na aprovação da *Condition*. Por exemplo, no caso de uma *Condition* com operação lógica de conjunção, as demais *Premises* somente tem suas notificações ativadas após a primeira *Premise* definida como mais relevante possuir valor lógico verdadeiro (OLIVEIRA, 2019).

¹⁰ Este framework foi projetado para trabalhar com uma placa RaspberryPi, portanto apresenta configurações específicas para lidar diretamente com as entradas e saídas desta placa

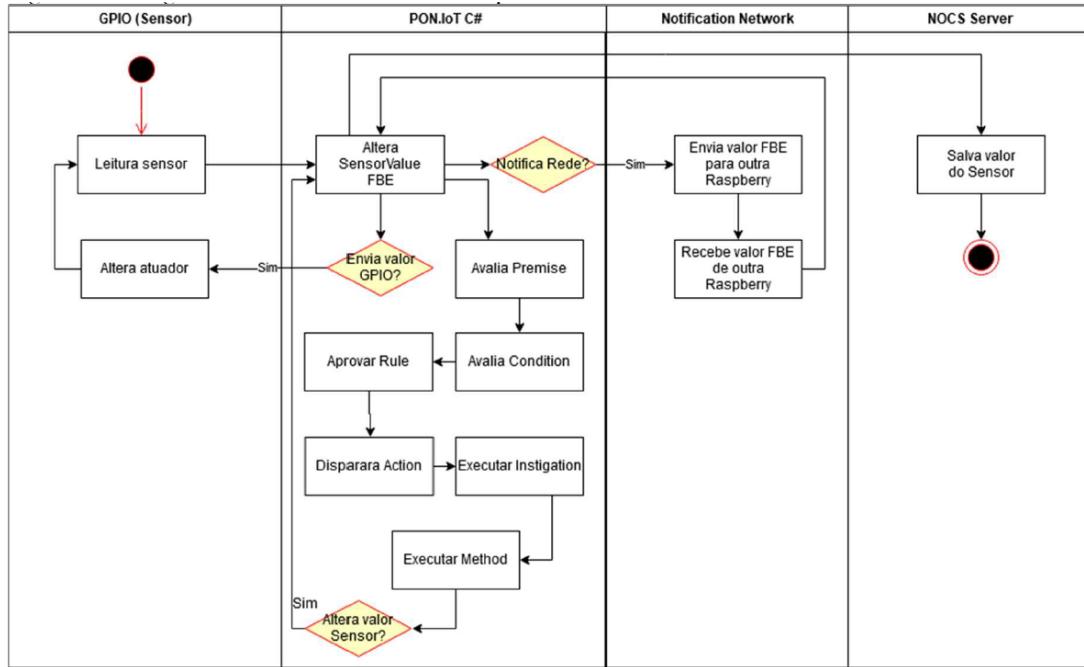


Figura 50 – Diagrama de atividades do PON C# IoT
Fonte: Oliveira (2019)

Esse *framework* foi aplicado para o desenvolvimento do NOCS (*Notification Oriented Care System*), com a utilização do ambiente distribuído mostrado na Figura 51, por meio da conexão dos sensores e atuadores utilizando a entidade *SensorFBE* em um dispositivo RaspberryPi remoto (NOCS Control), que se comunica com um servidor central (NOCS Server), capaz de executar os processos de interface com o usuário e comunicação com outros dispositivos (OLIVEIRA, 2019).

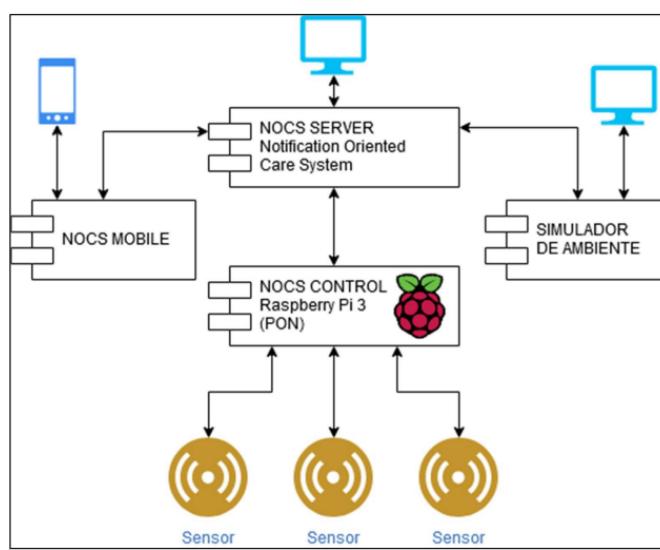


Figura 51 – Diagrama de componentes PON C# IoT
Fonte: Oliveira (2019)

Ainda, o NOCS Server disponibiliza uma interface web na qual os usuários podem

visualizar os estados dos sensores/atuadores, assim como criar *Rules* em tempo de execução da aplicação. Ademais, utilizando a mesma API é possível acessar estes mesmos dados por meio de um aplicativo (NOCS Mobile). Além disso, também é possível a utilização de um simulador de ambiente que simula os sensores e atuadores que estariam normalmente conectados no NOCS Control (OLIVEIRA, 2019). Nesta aplicação as *Rules* eram criadas de forma dinâmica utilizando a interface do NOCS Server, ainda assim o Código 2.22 mostra um exemplo de construção estática de uma *Rule* para controle de temperatura pertinente a esta aplicação.

Código 2.22 – Exemplo de *Rule* com o Framework PON C# IoT

```

SensorFBE sensor1 = new SensorFBE(1, "Thermometer1", "", 0, "", "", 
    DistributedNotification, NotificationControl.OnChange);
SensorFBE sensor2 = new SensorFBE(2, "Thermometer2", "", 0, "", "", 
    DistributedNotification, NotificationControl.OnChange);
SensorFBE sensor3 = new SensorFBE(3, "Thermometer3", "", 0, "", "", 
    DistributedNotification, NotificationControl.OnChange);
SensorFBE sensor4 = new SensorFBE(4, "Air1", "", 0, "", "", 
    DistributedNotificationType.Rest, NotificationControl.OnChange);

Rule rule1 = new Rule(1, "TempControl1", 0);
var subcondition1 = rule1.Condition.AddSubCondition(1);
subcondition1.LogicalOperator = Library.LogicalOperator.Disjunction;
subcondition1.AddPremisse(1, sensor1, "Thermometer1 > 20");
subcondition1.AddPremisse(1, sensor2, "Thermometer2 > 20");
subcondition1.AddPremisse(1, sensor3, "Thermometer3 > 20");
var subcondition2 = rule1.Condition.AddSubCondition(1);
subcondition2.AddPremisse(1, sensor4, "Air1 = 0");
Method method1 = new Method(1, sensor4, 1);
var instigariom1 = rule1.Action.AddInstigationSequential(1, "");
instigariom1.AddMethodSequential(method1);
Method method2 = new Method(1, sensor7, 1);
instigariom1.AddMethodSequential(method2);

```

Fonte: Adaptado de Oliveira (2019)

Na Figura 52 é apresentado um resultado de teste de *stress* do monitoramento de ambientes utilizando RaspberryPi. Nesse teste, coletou-se o número de leituras dos sensores, avaliações de *Premises* e o tempo total de execução, obtendo uma média de 235 notificações de sensores por segundo, e a média de 415 avaliações de *Premises* por segundo, totalizando uma média de 650 execuções por segundo Oliveira (2019). Esses testes permitiram mensurar, neste cenário específico, o número máximo de notificações que o sistema consegue processar por segundo.

2.4.4 Framework PON Elixir/Erlang

Esta materialização chamada *Framework PON Elixir/Erlang* é fruto dos esforços de mestrado de Negrini (2019), sendo proposta com o objetivo de aproveitar os conceitos de desacoplamento e, portanto, potencial paralelização das entidades do PON em conjunto ao modelo de atores da arquitetura Erlang (NEGRINI, 2019).

<i>Ambientes</i>	<i>Leituras</i>	<i>Premises</i>	<i>Tempo (s)</i>
1	341	600	1,47
5	1.705	3.000	7,15
25	8.525	15.000	35,82
50	17.050	30.000	72,87
100	34.100	60.000	144,13

Figura 52 – Resultados de testes do PON C# IoT

Fonte: Oliveira (2019)

Em termos de paradigma, a linguagem Elixir/Erlang segue os princípios do Paradigma Funcional (PF) associado com o Paradigma Orientado a Atores (POA). Nesta materialização é introduzido uma implementação na qual os elementos do PON são modelados por meio de atores. O modelo de atores pode ser definido como uma extensão dos modelos modulares declarativos ou imperativos, com a passagem assíncrona de mensagens entre os agentes computacionais elaborados declarativamente (NEGRINI, 2019).

No *Framework* PON Elixir/Erlang, tanto os elementos da base de fatos (*i.e.*, *FBEs*, *Attributes* e *Methods*), quanto suas condições lógico-causais (*i.e.*, *Rules*, *Conditions*, *Premises*) e ativadores (*i.e.*, *Actions* e *Instigations*) são desmembrados em vários atores, enquanto as notificações são implementadas por meio de mensagens assíncronas, de modo que cada ator carrega uma pequena parte do fluxo de processamento, cunhando assim micro-atores (NEGRINI, 2019). Um detalhamento dessa implementação é apresentado por meio da modelagem em UML apresentada na Figura 53.

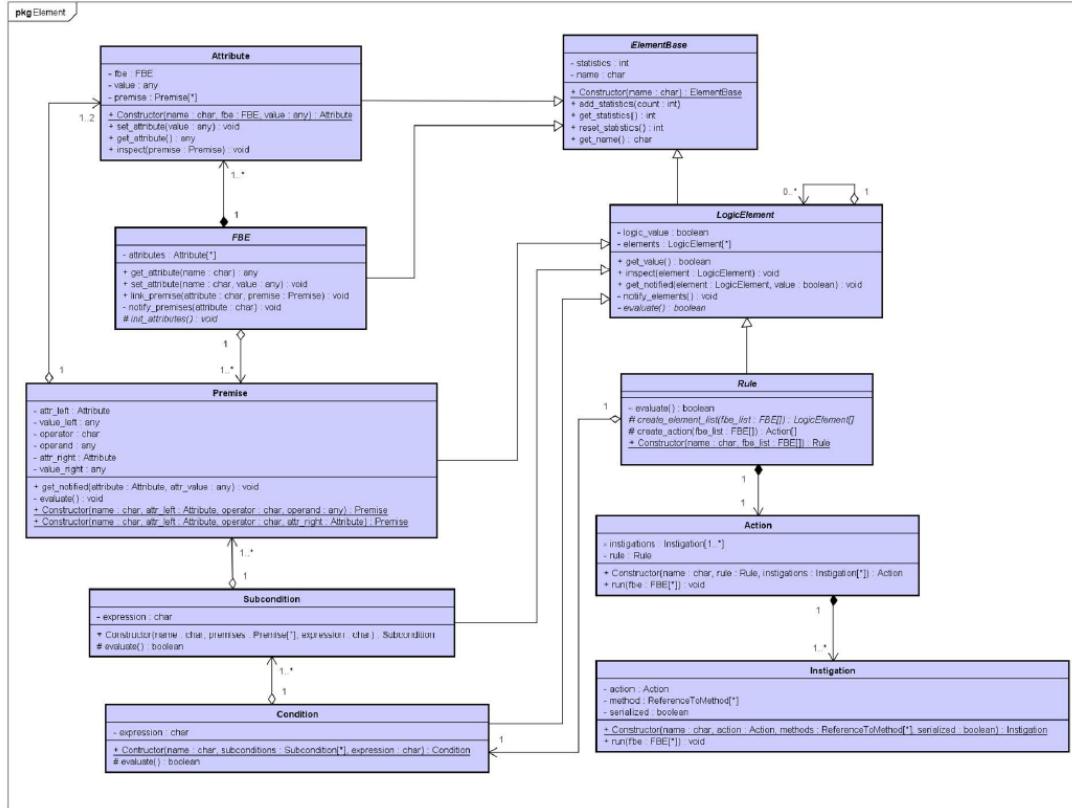


Figura 53 – Modelagem UML dos elementos do PON enquanto micro-atores

Fonte: Negrini (2019)

A utilização do *Framework PON Elixir/Erlang* é exemplificada por meio da implementação de um conjunto de *FBE* e *Rule* bastante simples, conforme descrito no Código 2.23, e implementado com o *framework* no Código 2.24. Neste exemplo é descrito um *FBE* com apenas um *Attribute atValue*, e uma *Rule* que altera o seu valor para 3 quando seu valor é 5, conforme a *Condition* descrita. Esse exemplo ilustra a utilização da especialização de *NOP.Element.FBE* e *NOP.Element.Rule* para a implementação das entidades do PON.

Código 2.23 – Exemplo descrito em *FBE* e *Rule*

```
fbe Dummy
public integer aValue = 0
private method change_value_to_3
    attribution
        this.aValue = 3
    end_attribution
end_method
rule rlExample
    condition
        premise prExample
            this.aValue == 5
        end_premise
    end_condition
    action sequential
        instigation sequential
            call this.change_value_to_3()
        end_instigation
    end_action
end_rule
end_fbe
```

Fonte: Fonte: Autoria própria

Código 2.24 – Exemplo de implementação de *Rule* como especialização de *NOP.Element.Rule*

```
defmodule NOP.Element.FBE_dummy do
  use NOP.Element.FBE

  defp int_attribuges() do
    %{:value => 0}
  end

  def change_value_to_3(fbe) do
    NOP.Service.FBE.set_attribute(fbe, :value, 3)
  end
end

defmodule NOP.Element.Rule_example do
  use NOP.Element.Rule

  defp create_element_list([fbe]) do
    premise = NOP.Service.Premise.create_premise(
      "NOP.element.premise", fbe, :value, :EQ, 5)
    [premise]
  end

  defp create_instigation_list([fbe]) do
    [{NOP.Element.FBE_dummy, :change_value_to_3, [fbe]}]
  end
end
```

Fonte: Adaptado de Negrini (2019)

Para a validação deste *framework* foi escolhida uma aplicação de Controle de Tráfego Automatizado (CTA)¹¹. O objetivo do CTA é simular o tráfego de uma área urbana e aplicar diferentes estratégias de controle automatizado de tráfego, de modo a avaliar o desempenho computacional conforme a estratégia (NEGRINI, 2019). Os testes foram realizados com o objetivo de avaliar a carga nos núcleos do processador em diferentes ambientes, com 2, 4, 8 e 16 núcleos. Esses ambientes são identificados na Figura 54.

As Figuras 55, 56, 57 e 58 exibem a taxa média de ocupação por núcleo e tempo

¹¹ Maiores detalhes sobre esta aplicação podem ser encontrados em (RENAUX *et al.*, 2015)

Apelido	Modelo	Núcleos	Memória
VM02	m5ad.large	2	8GB
VM04	m5ad.xlarge	4	16GB
VM08	m5ad.2xlarge	8	32GB
VM16	m5ad.4xlarge	16	64GB

Figura 54 – Detalhamento dos ambientes do experimento do Framework PON Elixir/Erlang
Fonte: Negrini (2019)

total de execução do experimento para cada um dos ambientes. Estes resultados apresentam uma expressiva redução do tempo de execução conforme são acrescidos núcleos aos ambientes, bem como apresentam que a taxa de ocupação dos núcleos se manteve balanceada durante a simulação, demonstrando que a execução lógico-causal foi distribuída com um bom nível de balanceamento entre os núcleos (NEGRINI, 2019).

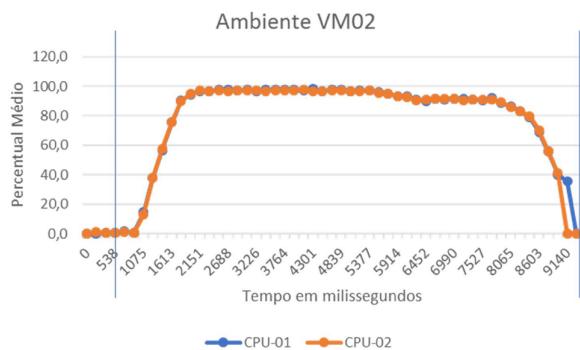


Figura 55 – Taxa média de ocupação por núcleo em ambiente VM02
Fonte: Negrini (2019)

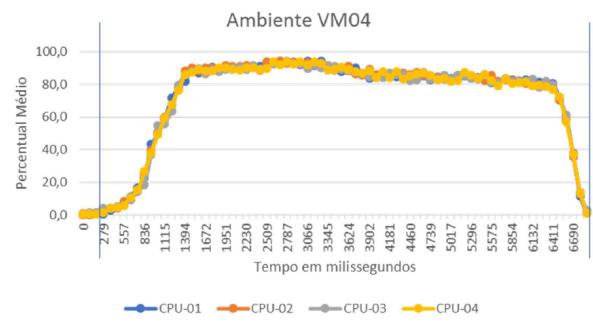


Figura 56 – Taxa média de ocupação por núcleo em ambiente VM02
Fonte: Negrini (2019)

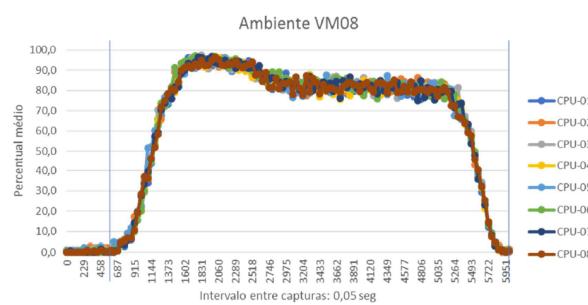


Figura 57 – Taxa média de ocupação por núcleo em ambiente VM08
Fonte: Negrini (2019)

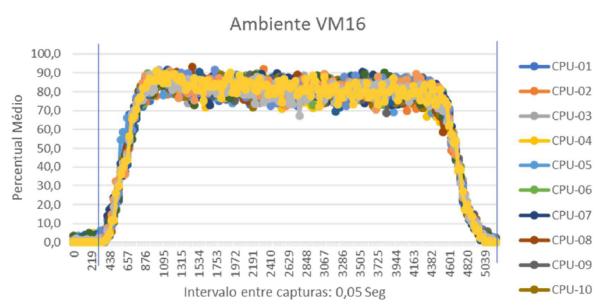


Figura 58 – Taxa média de ocupação por núcleo em ambiente VM16
Fonte: Negrini (2019)

Schütz (2019) também utilizou o Framework PON Elixir/Erlang para a execução da RNA MLP para a função XOR em NeuroPON. Para isto o código foi desenvolvido espelhando-

se no experimento realizado com o *Framework PON C++ 3.0*. A aplicação desenvolvida foi executada em quatro ambientes diferentes, com 1, 2, 4 e 8 *cores* cada respectivamente. Conforme os resultados apresentados na Figura 59, observa-se que a execução no ambiente com *octa core* apresenta uma redução de 94% no tempo de execução quando comparado ao ambiente *mono core*.

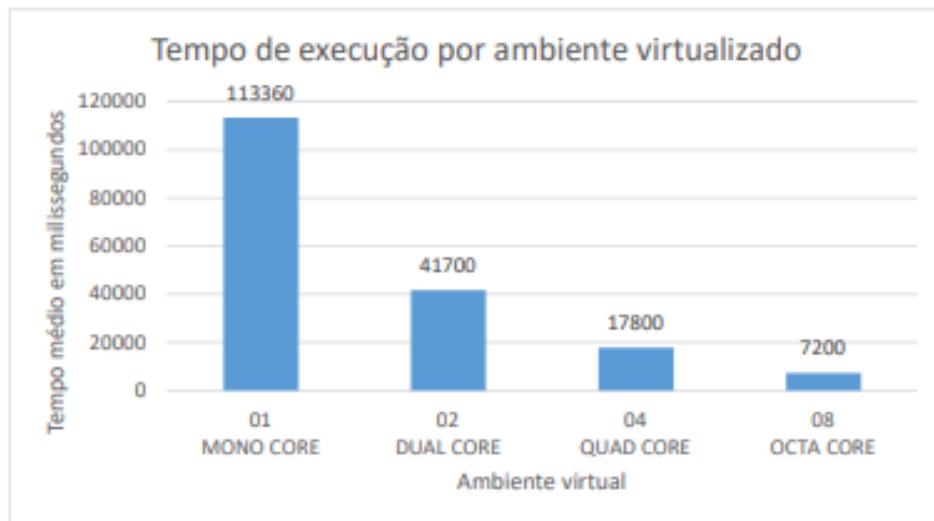


Figura 59 – Tempos médios de execução NeuroPON de uma RNA MLP para a função XOR no Framework PON Elixir/Erlang
Fonte: Schütz (2019)

Isto posto, é possível afirmar que o *Framework PON Elixir/Erlang* é de grande valia para a execução da NeuroPON, obtendo resultados muito superiores (em termos de benefícios da paralelização) aos obtidos com a mesma aplicação no *Framework PON C++ 3.0*. Esse resultado é devido à maneira como o paralelismo intrínseco ao modelo de atores da linguagem de programação Elixir/Erlang e implementado por meio do *Framework PON Elixir/Erlang* é mais eficiente do que o implementado com o *Framework PON C++ 3.0*. Por fim, esse experimento permite melhor perceber a valia de NeuroPON já em termos de paralelismo em software (SCHÜTZ, 2019).

2.4.5 *Framework PON Akka.NET*

Outro *framework* implementado utilizando o modelo de atores, como utilizado no *Framework Elixir/Erlang*, mas de maneira um tanto mais prototípica, é o *Framework PON Akka.NET*, no qual o trabalho das entidades do PON é distribuído em atores. A Figura 60 ilustra a estrutura do modelo de atores em Akka.NET. Os atores criados pela aplicação são criados sobre o endereço */root/user*, enquanto os atores criados automaticamente pelo sistema são criados em

`/root/system`. Todos os atores são acessíveis por seu endereço, mesmo em sistemas distribuídos (MARTINI *et al.*, 2019).

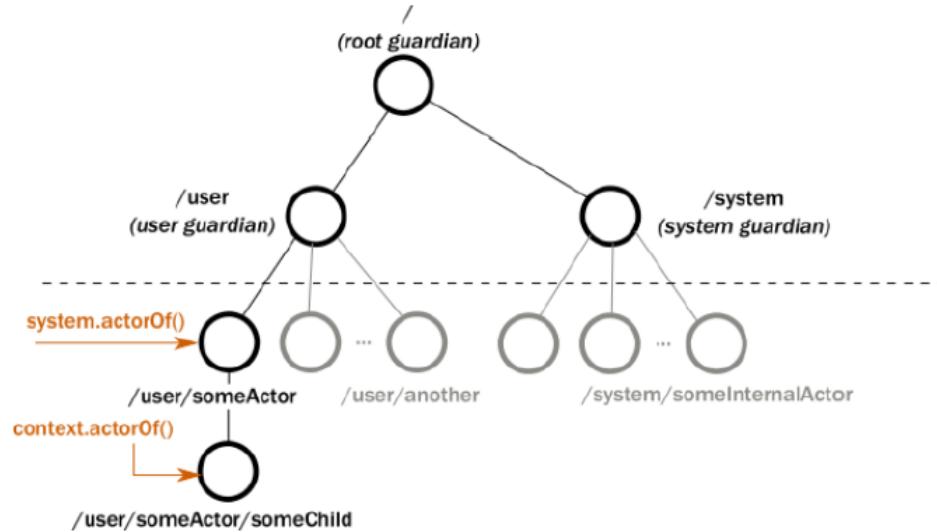


Figura 60 – Estrutura de atores em Akka.NET
Fonte: Martini *et al.* (2019)

Os atores se comunicam entre si por meio de um sistema de mensagens. Cada ator possui uma referência aos atores que precisam receber suas mensagens. As notificações do PON são implementadas por meio deste mecanismo de mensagens (MARTINI *et al.*, 2019). A mesma aplicação do portão eletrônico, descrita em detalhes na Seção 2.4.2, foi desenvolvida com este framework. O diagrama do modelo de atores para esta aplicação é mostrado na Figura 61.

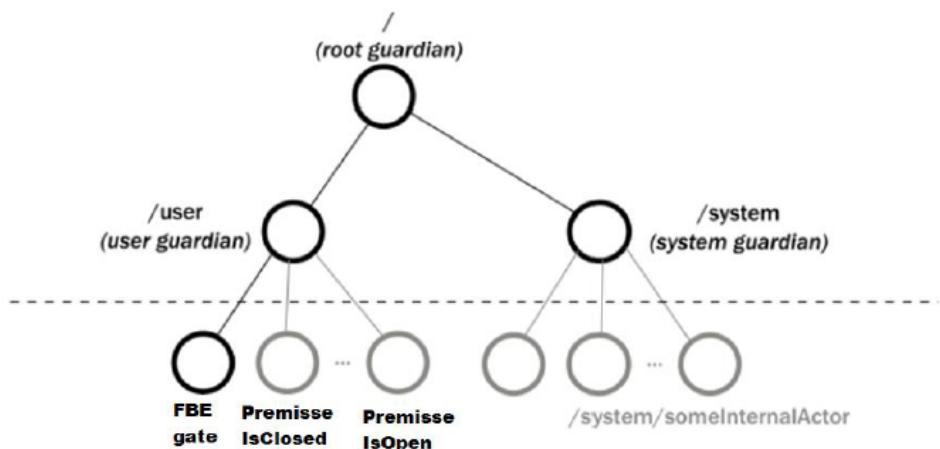


Figura 61 – Modelo de atores na aplicação do portão eletrônico em Akka.NET
Fonte: Martini *et al.* (2019)

Ainda em termos de códigos, em um momento posterior, aquelas entidades criadas, conforme ilustrado no Código 2.25, precisam ser conectadas de forma a passar as referências

para o processo de envio de mensagens do Akka.NET, sendo tal processo ilustrado no Código 2.26.

Código 2.25 – Criação de atores em Akka.NET

```
IActorRef FBEGateActor = NOPActorSystem.ActorOf(Props.Create(() ->
    new FBEGate()), "FBEGateActor");
IActorRef FBERemoteControlActor = NOPActorSystem.ActorOf(Props.Create(() ->
    new FBERemoteControl()), "FBERemoteControlActor");
IActorRef PremisseIsClosedActor = NOPActorSystem.ActorOf(Props.Create(() ->
    new PremisseIsClosed()), "PremisseIsClosedActor");
IActorRef PremisseIsOpenActor = NOPActorSystem.ActorOf(Props.Create(() ->
    new PremisseIsOpen()), "PremisseIsOpenActor");
IActorRef PremiseChangeStateActor = NOPActorSystem.ActorOf(Props.Create(() ->
    new PremiseChangeState()), "PremiseChangeStateActor");
IActorRef ConditionCloseGateActor = NOPActorSystem.ActorOf(Props.Create(() ->
    new ConditionCloseGate()), "ConditionCloseGateActor");
IActorRef ConditionOpenGateActor = NOPActorSystem.ActorOf(Props.Create(() ->
    new ConditionOpenGate()), "ConditionOpenGateActor");
IActorRef ConditionSateChangedActor = NOPActorSystem.ActorOf(Props.Create(() ->
    new ConditionSateChanged()), "ConditionSateChangedActor");
```

Fonte: Adaptado de Martini *et al.* (2019)

Código 2.26 – Conexão entre atores em Akka.NET

```
var FBEGateActorTask1 = FBEGateActor.Ask(
    new ActorReference(ActorRefType.PremisseIsClosedRef, PremisseIsClosedActor));
var FBEGateActorTask2 = FBEGateActor.Ask(
    new ActorReference(ActorRefType.PremisseIsOpenRef, PremisseIsOpenRefActor));
var FBERemoteControlActorTask1 = FBERemoteControlActor.Ask(
    new ActorReference(ActorRefType.PremisseChangeStateRef, PremisseChangeStateActor));
var PremisseIsOpenActorTask1 = PremisseIsOpenActor.Ask(
    new ActorReference(ActorRefType.ConditionCloseGateRef, ConditionCloseGateActor));
var PremiseChangeStateActorTask1 = PremiseChangeStateActor.Ask(
    new ActorReference(ActorRefType.ConditionOpenGateRef, ConditionOpenGateActor));
var PremisseIsOpenActorTask1 = PremisseIsOpenActor.Ask(
    new ActorReference(ActorRefType.ConditionCloseGateRef, ConditionCloseGateActor));
var PremiseChangeStateActorTask3 = PremiseChangeStateActor.Ask(
    new ActorReference(ActorRefType.ConditionOpenGateRef, ConditionOpenGateActor));
var ConditionCloseGateActorTask1 = ConditionCloseGateActor.Ask(
    new ActorReference(ActorRefType.FBEGateRef, FBEGateActor));
var ConditionCloseGateActorTask2 = ConditionCloseGateActor.Ask(
    new ActorReference(ActorRefType.FBERemoteControlRef, FBERemoteControlActor));
var ConditionOpenGateActorTask1 = ConditionOpenGateActor.Ask(
    new ActorReference(ActorRefType.FBEGateRef, FBEGateActor));
var ConditionOpenGateActorTask2 = ConditionOpenGateActor.Ask(
    new ActorReference(ActorRefType.FBERemoteControlRef, FBERemoteControlActor));
var ConditionSateChangedActorTask1 = ConditionSateChangedActor.Ask(
    new ActorReference(ActorRefType.FBERemoteControlRef, FBERemoteControlActor));
```

Fonte: Adaptado de Martini *et al.* (2019)

O desempenho deste *framework* é analisado comparando com a aplicação utilizando o *Framework PON C++ 3.0*. Os resultados da Figura 62 mostram que com menos atores o desempenho é inferior. Problemas de alocação de memória do *Framework PON C++ 3.0* limitaram o teste a um máximo de 7200 atores, não permitindo uma comparação com um maior número de atores, no qual o *Framework Akka.NET* tenderia a apresentar resultados melhores (MARTINI *et al.*, 2019).

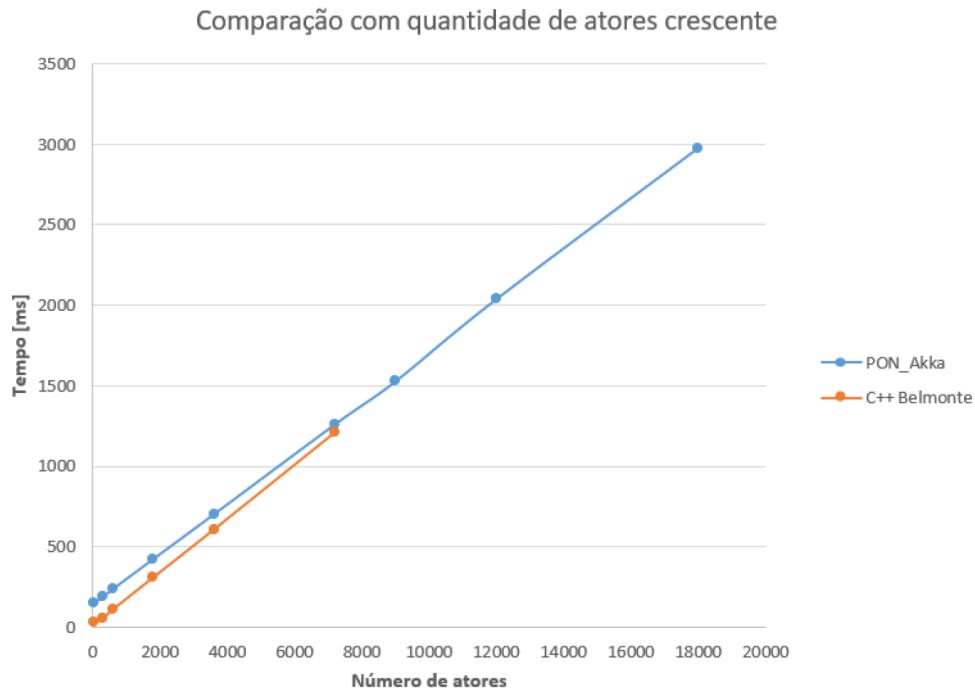


Figura 62 – Comparação entre o Framework PON Akka.NET e C++ 3.0
Fonte: Martini et al. (2019)

2.4.6 Reflexões sobre os *frameworks* do PON

Esta subseção traz um arrazoado sobre os *frameworks* do PON para *software* apresentados nas subseções anteriores. Tal arrazoado analisa estas materializações sobre distintos pontos de vista, nas suas três subseções que seguem,

2.4.6.1 *Frameworks* do PON vis-à-vis sua teoria

O grande número de materializações disponíveis do PON é interessante, pois permite a aplicação do PON em diversos ambientes diferentes. Entretanto, cada uma dessas materializações contempla apenas alguns dos conceitos e propriedades do PON introduzidos em seções prévias deste presente capítulo.

Os conceitos e o potencial de propriedades que são implementados em cada uma destas materializações são apresentados nas tabelas abaixo. A Tabela 5 relaciona quais propriedades elementares alcançam seus potenciais, abordadas na Seção 2.2, são contempladas em cada uma das materializações do PON, do mesmo modo que a Tabela 6 relaciona os conceitos apresentados na Seção 2.3.

Pode ser dito que nenhuma das materializações atinge de forma plena o quesito de programação em alto nível, devido à complexidade do uso dos *frameworks* que requer que o

Tabela 5 – Propriedades elementares contempladas nas materializações do PON
Fonte: Autoria própria

Potencial de propriedade \ Materialização	Fw. C++ Prot.	Fw. C++ 1.0	Fw. C++ 2.0	Fw. C++ 3.0	Fw. Java /C#	Fw. C# IoT	Fw. Elixir	Fw. Akka
Programação em alto nível	~	~	*	~	~	~	~	~
Paralelismo via desacoplamento				~		✓	✓	✓
Distribuição via desacoplamento						✓	*	*
Desempenho via não redundâncias				~		~		

✓ Materializa completamente a propriedade

~ Materializa parcialmente a propriedade

* Materializa a propriedade por meio de ferramenta *wizard* (VALENÇA, 2012)

* A tecnologia de base permite materializar a propriedade, entretanto carece de testes para validação

programador ainda conheça certos detalhes de implementação do *framework* para implementar a aplicação em PON em si. Isso se dá também pelas limitações impostas ao se implementar o PON sobre linguagens com base em outros paradigmas. A programação em alto nível só é atingida de maneira completamente satisfatória por meio da linguagem de programação do PON (a LingPON), ainda que em *framework* seja possível alcançar algo similar conforme observado no prototípico e infundado JuNOC++.

Além dos *frameworks* atuais realmente funcionais/findados não contemplarem apropriadamente desenvolvimento em alto nível, salvo se usarem uma interface *wizard* que limitaria criatividade (ainda que certamente úteis para dados contextos), eles também não contemplam apropriadamente o potencial de bom desempenho, que seria alcançável pelo evitar implícito de redundâncias existentes no PON. De fato, dito mais precisamente e pragmaticamente, nenhuma destas materializações em *frameworks* é capaz de atingir o desempenho esperado à luz do cálculo assintótico do PON, por se basearem em estruturas de dados computacionalmente custosas que acabam mascarando a baixa complexidade assintótica-temporal do PON. Ainda que em *frameworks* não seja possível escapar do uso destas estruturas de dados, possivelmente elas poderiam ser mais bem articuladas, mesmo mais do que no *Framework PON C++ 2.0* que é o de melhor desempenho até então.

Por fim, apesar de as entidades do PON serem paralelizáveis e mesmo distribuíveis, graças ao desacoplamento que existe entre elas por definição, apenas algumas das materializações implementam a potencialidade dessa propriedade. Ainda, quando isto ocorre, não tem conseguido

manter equilíbrio para com questão de performance. Isso ocorre por dois motivos, os quais seriam em algo gerenciáveis, mas finalmente não foram gerenciados. Primeiramente, há a dificuldade de se implementar mecanismos de sincronização entre elementos paralelizáveis em algumas linguagens de programação, além do alto tempo de execução associado a estes mecanismos. Do mesmo modo, a distribuição das entidades é possível, porém também possui um alto custo em tempo de execução devido aos mecanismos de comunicação entre as entidades, como o uso de mensagens assíncronas. Desta forma as materializações com paralelismo e distribuição apresentam desempenho mais baixo.

Além da potencialidade do uso das propriedades elementares do PON, há também os conceitos de programação ou desenvolvimento em PON que foram tratados na Seção 2.3. Isto dito, a Tabela 6 mostra que todas as materializações não implementam um ou mais daqueles conceitos finalmente. Isso pode ser atribuído ao grau de maturidade de cada uma das materializações, assim como pela dificuldade de se implementar determinados conceitos em algumas linguagens de programação. De forma geral o *Framework PON C++ 2.0 e 3.0* são os que atendem a maior parte destes conceitos, também devido ao fato que muitos destes conceitos terem sido introduzidos justamente nos trabalhos que fizeram o desenvolvimento do *Framework PON C++ 2.0* (RONSZCKA, 2012; VALENÇA, 2012).

Tabela 6 – Conceitos do PON contemplados nas materializações do paradigma
Fonte: Autoria própria

Conceito de Programação \ Materialização	Fw. Prot.	Fw. C++ 1.0	Fw. C++ 2.0	Fw. C++ 3.0	Fw. Java / C#	Fw. C# IoT	Fw. Elixir	Fw. Akka
Reatividade das entidades	✓	✓	✓	✓	✓	✓	✓	✓
Compartilhamento de entidades		✓	✓	✓	✓	✓	✓	
Renotificações		✓	✓	✓	✓			
Resolução de conflitos		✓	✓	✓	✓	✓		
<i>Master Rule</i>			✓	✓				
Impertinência estática			✓	✓				
Impertinência dinâmica						✓		
<i>FBE Rules</i>			✓	✓				
<i>FBE Agregador</i>			✓	✓				
<i>Formation Rules</i>								
<i>Keeper</i>			*					

* Implementado sob a forma de uma adaptação no *framework* existente, visto que altera o fluxo tradicional de execução das *Rules* do PON (MUCHALSKI *et al.*, 2012)

Dadas estas reflexões sobre as materializações existentes do PON em termos de frameworks vis-à-vis a teoria do PON, a seção seguinte apresenta reflexões específicas com relação os frameworks em geral enquanto estado da técnica, usando como condutor das reflexões o

Framework PON C++ 2.0, dado que este é a materialização com maior grau de maturidade dentre os *frameworks* do PON.

2.4.6.2 *Frameworks* do PON enquanto estado da técnica

Do ponto de vista de implementação de programas em PON, para a solução de problemas reais, dentre todas as materializações do PON a mais utilizada ainda é o *framework* C++ 2.0, devido a seu maior grau de maturidade e estabilidade entre as materializações desenvolvidas (RONSZCKA *et al.*, 2017). Apesar desta versão de *framework* oferecer todos os recursos necessários para a aplicação dos conceitos e desenvolvimento de programas no PON, ela ainda apresenta alguns problemas que dificultam a sua aplicação no desenvolvimento de um programa no PON. Estes problemas são listados nas seções seguintes, sendo que se replicam ademais nos demais frameworks

Verbosidade

O *framework* C++ 2.0 exige a escrita de uma quantidade muito significativa de código para se desenvolver uma aplicação básica. Isso vem da necessidade de se criar muitos métodos diferentes para se inicializar as diferentes entidades. As construções de objetos são complexas e muitas vezes exigem múltiplas linhas de código.

Essa verbosidade tenta ser mitigada por meio do uso de *macros*, que são fragmentos de código aos quais é dado um nome. Quando esse nome é utilizado, ele é substituído pelo conteúdo da *macro*, expandindo os parâmetros passados na etapa de compilação. Porém, isso apenas tenta esconder a verbosidade, sem de fato solucionar o problema em si.

Esta característica de verbosidade foi herdada pelos demais *frameworks* em geral, exceto pelo todo prototípico JuNOC++.

Baixa flexibilidade de tipos

O *Framework PON C++ 2.0* limita o desenvolvedor ao uso de atributos dos tipos *integer*, *double*, *bool*, *string*. Em boa parte dos casos de uso isso pode ser suficiente, mas principalmente quando é feita a integração com APIs e bibliotecas externas isso pode ser um fator limitador que dificulta o desenvolvimento de código.

Ainda do ponto de vista de implementação do *framework* em si isso leva a uma necessidade de duplicação de código muito grande pela necessidade de se implementar métodos e estruturas de dados especializadas para cada tipo. Isso causa problemas principalmente para eventual manutenção do código e implementação de novas funcionalidades, visto que torna mais trabalhoso precisar replicar alterações idênticas em diversos trechos de código.

Esta característica de baixa flexibilidade foi herdada pelos demais *frameworks* em geral, com exceção daqueles implementados em linguagem com tipagem dinâmica, como o *Framework PON Elixir/Erlang*.

Baixa flexibilidade algorítmica

Além da baixa flexibilidade de tipos, também cabe uma crítica à flexibilidade algorítmica, visto que as estruturas de avaliação lógica das *Conditions* e *Premises* obedecem a uma estrutura rígida com a qual o desenvolvedor não consegue criar avaliações mais complexas. Por exemplo, não se pode agregar diversas *Premises* por meio de uma expressão lógica booleana genérica (*e.g.*, *premise1 and premise2 or (premise3 and not premise5)*), sendo limitado ao uso de *Conditions* utilizando apenas conjunções ou apenas disjunções. Isto leva à necessidade de se criar um número muito maior de entidades, como *Premises* e *Conditions* auxiliares, para se implementar a condição desejada.

Esta característica de baixa flexibilidade algorítmica foi herdada pelos demais *frameworks* em geral, exceto pelo todo prototípico JuNOC++.

Baixa confiabilidade

A baixa confiabilidade é atribuída à falta de testes realizados sob os *frameworks* em si. A aplicação de testes unitários em um *software* é capaz de reduzir significativamente o número de defeitos durante o seu uso (WILLIAMS *et al.*, 2009).

Nos *frameworks* existentes o único método para se testar possível é por meio da construção de aplicações completas, que funcionam como testes de integração. Estes testes são capazes de validar apenas em um contexto limitado do *software* como um todo, não testando cada parte do *framework* individualmente sob a perspectiva de testes unitários. Esse método pode ser útil para validar os conceitos de PON e até mesmo para realizar análises de desempenho, porém é muito menos eficiente para se encontrar problemas.

Esta característica de verbosidade foi herdada pelos demais *frameworks* em geral.

Curva de aprendizado não suave

Todos esses fatores supracitados contribuem para a construção de um ambiente de desenvolvimento pouco amigável ou não tão amigável (conforme o ponto de vista) para a introdução a novos desenvolvedores, restringindo ainda um tanto o desenvolvimento de aplicações no PON a desenvolvedores mais experientes e com conhecimento mais profundo do paradigma ou que passem por treinamento para tal. Tal curva de aprendizado é prejudicial para a popularização do paradigma entre desenvolvedores, sendo que a intenção sempre foi o PON ser intuitivo.

Ademais, este quadro dado também dificulta a manutenção e melhoria dos códigos, visto que dado o contexto acadêmico de sua construção, o autor de uma das versões do *framework* usualmente não contribui mais com seu desenvolvimento após a finalização dos seus estudos, passando essa responsabilidade para outro pesquisador e/ou estudante.

2.4.6.3 Ponderações gerais sobre a pertinência de *Frameworks* do PON

Apesar das imperfeições de cada *framework* em si, bem como no tocante ao aproveitamento das propriedades do PON e mesmo dos próprios conceitos de programação do PON, eles continuam sendo o estado da técnica em PON. Ademais, mesmo antes de ser estado da técnica, enquanto estado da arte, os *frameworks* têm permitido demonstrar vantagens do PON em relação a outros paradigmas.

Neste sentido, os *Frameworks* PON C++ 1.0/2.0, Java e C# permitiram mostrar um bom equilíbrio de programação em mais alto nível que em programação imperativa, sem perda importante de performance como ocorre na declarativa (BANASZEWSKI, 2009; HENZEN, 2015; RONSZCKA *et al.*, 2017). Por sua vez, o *Framework* PON C++ 3.0 mostrou a viabilidade de automaticamente obter balanceamento fino de carga entre núcleos de processamento (BELMONTE, 2012; BELMONTE *et al.*, 2016).

Ainda, os *Frameworks* PON Erlang/Elixir e Akka.net mostraram a viabilidade de implicitamente alcançar melhor paralelismo fino em nível de threads com melhor equilíbrio de carga nos processadores ou núcleos do que suas tecnologias de atores apenas respectivamente sobre paradigmas funcional e imperativo (MARTINI *et al.*, 2019; NEGRINI, 2019). Por fim, o prototípico JuNOC++ mostrou a possibilidade de programação em mais alto nível mesmo em

âmbito de *framework* (CHIERICI, 2020).

Em suma, os *frameworks* ainda são tecnologias importantes para o PON e podem melhorar ao aproveitamento das propriedades e dos conceitos de programação do PON desde que seus benefícios sejam articulados e suas deficiências em algo mitigadas. Dadas estas reflexões sobre a pertinência de *frameworks* PON enquanto estado da técnica, em subseção seguinte apresenta a Tecnologia LingPON enquanto estado da arte. De antemão, a Tecnologia LingPON permite programar em alto nível e, dentre outros, gerar código para a maioria dos *frameworks*, permitindo assim seus usos em altíssimo nível de desenvolvimento.

Por fim e em tempo, este trabalho se interessa nas materializações puramente em *software*, conforme já salientado. Porém, apenas para fins de registro e divulgação, também é pertinente mencionar os esforços das materializações do PON para *hardware* e que se correlacionam em algo com as soluções em *software*, como:

- O PON *Hardware* Digital (PON-HD), que permite a geração de *hardware* digital via VHDL por meio do código PON em considerável alto nível. PON-HD passou por etapas prototípicas a até alcançar um conjunto de componentes em VHDL que se constituem em uma forma de framework para tal (KERSCHBAUMER, 2018; KERSCHBAUMER *et al.*, 2018a; KERSCHBAUMER *et al.*, 2018b).
- O CoPON, um coprocessador em VHDL desenvolvido para acelerar a execução de aplicações desenvolvidas em *Framework* PON C++ 1.0 adaptado para tal. A construção do CoPON inspirou-se em versões prototípicas do PON-HD (PETERS, 2012; PETERS *et al.*, 2012);
- A ArqPON ou NOCA (*Notification Oriented Computer Architecture*), que é uma arquitetura de computação desenvolvida para a execução de *software* segundo o modelo computacional do PON, tendo *assembly* próprio para composição de *software* em si (LINHARES, 2015; LINHARES *et al.*, 2020).
- O Simulador ArqPON (ou NOCASim), um simulador desenvolvido para simular a NOCA, facilitando seu estudo e testes (PORDEUS, 2017; LINHARES *et al.*, 2020).

Em suma, essas materializações em *hardware* apresentam geralmente desempenho mais satisfatório e paralelismo intrínseco, em detrimento da facilidade de programação ou desenvolvimento (RONSZCKA, 2019). Entretanto, isto é resolvido pelo fato da Tecnologia

LingPON permitir programar em alto nível e também gerar código para estas soluções envolvendo em *hardware* do PON, permitindo assim seus usos em altíssimo nível de desenvolvimento, conforme será vista na próxima subseção.

2.4.7 Tecnologia LingPON

Além das materializações por meio de implementação sob a forma de *framework* em outras linguagens de programação para *software*, bem como as supracitadas soluções em *hardware*, o PON também conta com sua própria linguagem de programação, a LingPON. Com a LingPON é possível desenvolver programas diretamente em PON, utilizando a sua sintaxe e linguagem própria. Esse código então passa por um processo de compilação capaz de gerar um código alvo para os *frameworks* disponíveis. Em tempo, LingPON e sua tecnologia de compilação ímpar tem sido chamada de Tecnologia LingPON (RONSZCKA, 2019).

A Tecnologia LingPON já possui diferentes versões: Tecnologia LingPON Prototipal, Tecnologia LingPON 1.X (1.0 e 1.2), Tecnologia LingPON HD 1.0, Tecnologia LingPON 2.0, cada qual com seu sistema de compilação e sua linguagem de programação da tecnologia. No caso da Tecnologia LingPON 2.0, a linguagem de programação LingPON 2.0 também é chamada de NOPL (*Notification Oriented Programming Language*) (RONSZCKA, 2019). Ainda, tal tecnologia já tem mesmo a proposta e protótipo de uma segunda linguagem de programação, a NOPLite (CHIERICI, 2020). Esta última seguiria um padrão mais direto e pontual especialmente voltada para especialistas do PON (RONSZCKA, 2019).

Como exemplo da construção de programas e sintaxe de alto nível em LingPON 2.0 ou NOPL, o Código 2.27 apresenta o modelo de criação de um *FBE* e uma *Rule* para um exemplo do sensor previamente introduzido na Figura 6 da Seção 1.1.3. Maiores detalhes sobre a sintaxe e utilização desta linguagem podem ser consultados em (RONSZCKA, 2019).

Isto posto, para possibilitar o desenvolvimento de cada Tecnologia LingPon, destacando a 1.X e 2.0, foi proposto um novo método para uniformizar o processo de construção de linguagens e compiladores específicos para o PON em plataformas distintas. Para esse método foi dado o nome de MCPON (RONSZCKA, 2019).

Em suma, o MCPON define um conjunto de diretrizes e regras para a construção de uma representação intermediária adequada para programas em PON. Esta forma intermediária é dada por meio de um grafo, o Grafo PON, que permite representar apropriadamente um programa em PON. O MCPON institui um sistema completo para o processo de compilação, usando o

Código 2.27 – Exemplo de construção de entidades na LingPON 2.0

```
fbe Sensor
public boolean atIsRead = false
public boolean atIsActivated = false
private method mtProcess
    attribution
        this.atIsRead = true
        this.atIsActivated = false
    end_attribution
end_method
rule rlSensor
    condition
        premise prIsActivated
            this.prIsActivated == true
        end_premise
        and
        premise prDebug
            this.prIsNotRead == false
        end_premise
    end_condition
    action sequential
        instigation sequential
            call this.mtProcess()
        end_instigation
    end_action
end_rule
end_fbe
```

Fonte: Fonte: Autoria própria

Grafo PON. Nesse âmbito, a Figura 63 ilustra as cinco etapas do método MCPON (RONSZCKA, 2019).

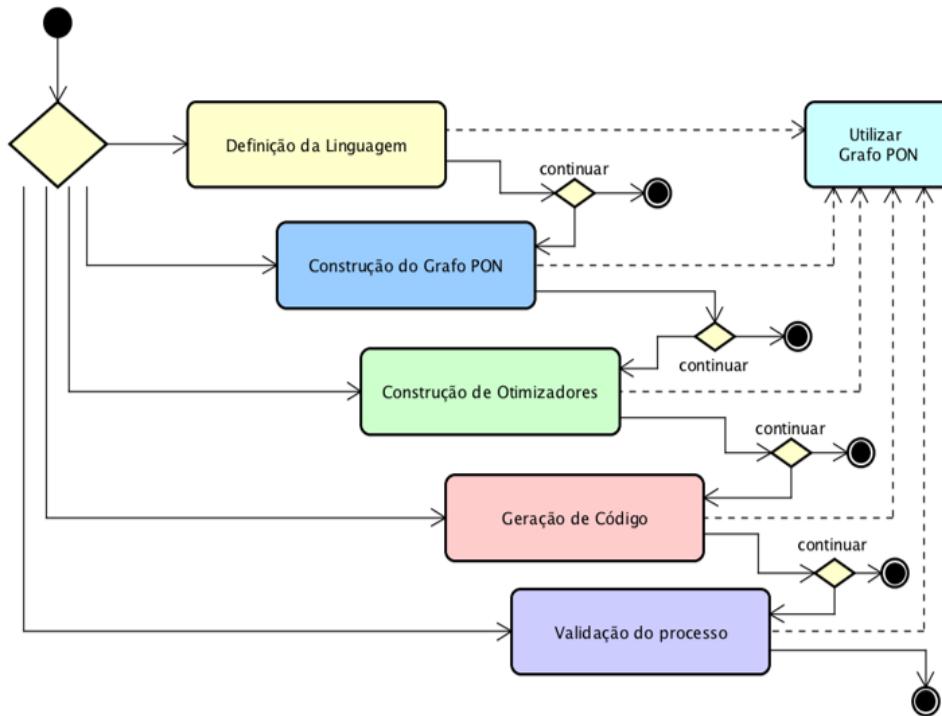


Figura 63 – Método MCPON
Fonte: Ronszcka (2019)

A primeira etapa do MCPON visa construir linguagens particulares para o PON. A segunda etapa visa definir o processo de construção de instâncias do Grafo PON. A terceira etapa

visa a construção de otimizadores. A quarta etapa visa a tradução dos grafos em códigos-alvo tanto em linguagens quanto em plataformas distintas. Por fim, a quinta etapa visa a construção de validadores (RONSZCKA, 2019).

No método MCPON, o Grafo PON é o elemento central, especialmente desenvolvido para a criação de linguagens e compiladores particulares ao PON. Nesse sentido, o Grafo PON serve, então, como uma representação intermediária para o mapeamento completo de programas PON e, principalmente, mantém a essência do PON, a qual é orientada a entidades notificantes desacopladas (RONSZCKA, 2019). De maneira gráfica, uma instância do Grafo PON é demonstrada na Figura 64, ilustrando a relação entre as diversas entidades do PON na estrutura do Grafo PON (NEGRINI, 2019).

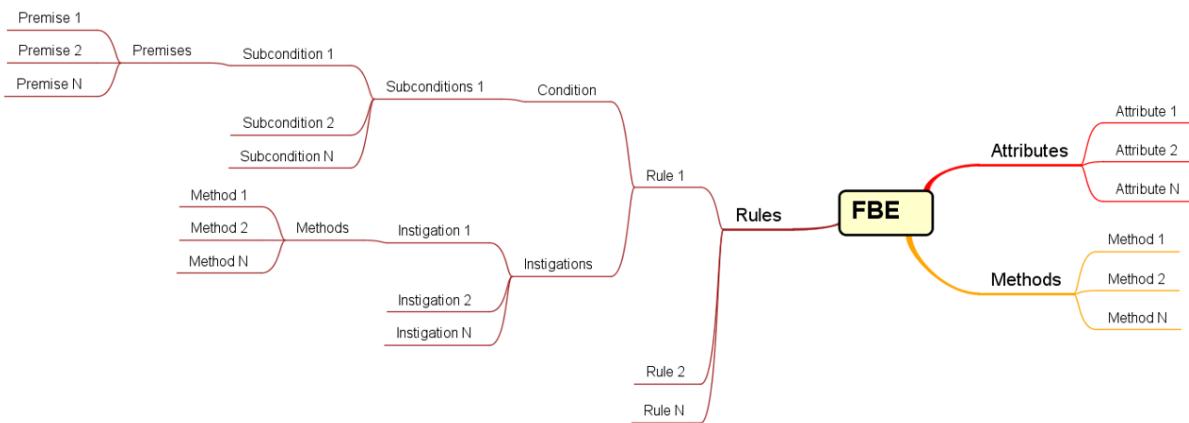


Figura 64 – Estrutura do Grafo PON
Fonte: Negrini (2019)

Essa etapa de compilação intermediária por meio do Grafo PON permite a geração de códigos específicos para diferentes alvos com base em um único código-fonte em LingPON, processo que é ilustrado na Figura 65. Neste âmbito, a Tecnologia LingPON é capaz de gerar código para as implementações de *framework* do PON ou mesmo código notificante específico em linguagens como C e C++.

Esta geração de código notificante específico em C/C++ apresenta excelente resultados de performance, por evitar justamente as sobrecargas de processamento com estruturas de dados dos frameworks, respeitando assim o cálculo assintótico do PON (RONSZCKA *et al.*, 2017; RONSZCKA, 2019; OSHIRO *et al.*, 2021). Ainda, já se começa ter soluções com algum paralelismo em *multicore* (RONSZCKA, 2019; MARTIN *et al.*, 2021).

No âmbito da geração de código notificante específico em C++ com a LingPON 2.0, foram ainda criados os geradores de código LingPON *Static* (Estático) e LingPON *Namespaces*

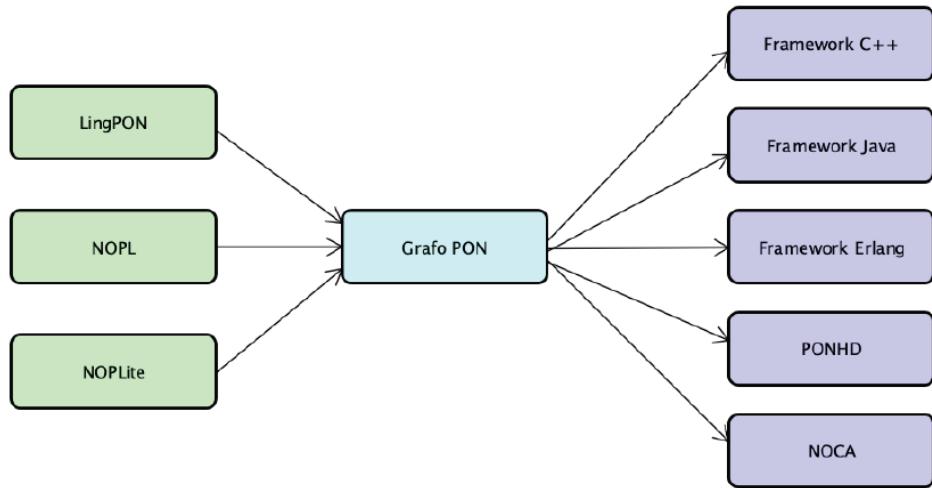


Figura 65 – Sistema de compilação do PON

Fonte: Ronszcka (2019)

(Espaço de Nomes). O gerador de código *LingPON Static* apresenta desempenho ainda melhor que os outros geradores para C++ notificantes, por meio do uso de classes estáticas. Entretanto, essa versão dificultou a integração com outros códigos legados em C++, devido ao código ser orientado a atributos estáticos, que inviabilizam a integração com código usual (SCHÜTZ; RONSZCKA, 2015).

Já o gerador de código *LingPON Namespaces*, tal qual *LingPON Static*, buscava eliminar a sobrecarga causada pelo uso de classes e objetos do POO em C++ específico, mas agora tratando as entidades por meio dos espaços de nomes (*namespaces*) e não código estático. Esta versão *LingPON Namespaces* consegue manter o baixo tempo de processamento da versão estática, ao mesmo tempo que resolve os problemas de integração presentes na mesma justamente (ATHAYDE; NEGRINI, 2016).

Ademais, também é possível gerar código *assembly* para a NOCA e VHDL em PON-HD, conforme ilustrado na Figura 65. No que diz respeito aos *targets* ilustrados na Figura 65, justamente, além destes, ainda é possível gerar código específico para os quase todos os *frameworks* apresentados na Seção 2.4, com exceção do *Framework Akka.NET* e do *Framework PON C++ Prototipal*.

Tudo isto considerado, ainda que a LingPON possibilite o desenvolvimento de *software* em alto nível e diretamente em PON, é difícil fazer a construção de aplicações completas com a LingPON, principalmente devido ao seu estado ainda prototípico vis-à-vis os *frameworks* PON. Neste sentido, aplicações em PON podem depender de acesso a arquivos, protocolos de rede e interfaces de usuário, sendo estas funcionalidades não cobertas pela LingPON. Apesar

da LingPON oferecer algumas ferramentas para realizar a interface com código externo em outras linguagens, seu uso ainda é limitado. Nesses contextos, é mais vantajosa a utilização dos *frameworks* que permitem uma integração mais fácil com outras bibliotecas externas, sendo que o uso pode ser feito associado com a LingPON no tocante a escrita dialógica principal de *FBEs* e *Rules*.

2.5 MÉTODO DE TESTE DE SOFTWARE PARA O PON

Dada que uma das propostas deste trabalho é utilizar o método TDD no desenvolvimento de um *framework* para o PON, é importante destacar os esforços de Kossoski (2015) no desenvolvimento de um método de teste de *software* para o PON. Este método proposto aplica-se tanto nas fases de teste unitário quanto de teste de integração, apresentados nas Seções 2.5.1 e 2.5.2 respectivamente.

2.5.1 Teste unitário em PON

O teste unitário considera a menor unidade ou trecho de código que pode ser testada em uma aplicação. No contexto do PON, foram elaboradas estratégias para o teste de *Premises*, *Conditions*, *Subconditions*, *Rules* e *Methods* dos *FBEs*, enquanto os *Attributes*, *Actions* e *Instigations*, por sua vez, não precisam passar por testes unitários. Do ponto de vista de teste, isso é devido ao fato dos *Attributes* representarem apenas uma declaração de estado, enquanto *Actions* e *Instigations* têm sempre o mesmo comportamento garantido por definição (KOSSOSKI, 2015).

São apresentadas abordagens distintas para o desenvolvimento de testes para cada uma das entidades:

- *Premise*: determinação de classes de equivalência e análise de valores limites que exercitem o operador lógico e os *Attributes* avaliados (KOSSOSKI, 2015).
- *Conditions* e *Subconditions*: são requeridos casos de teste que exercitem os estados das *Premises* e *Subconditions* avaliadas na sua operação lógica (KOSSOSKI, 2015).
- *Rules*: são considerados apenas os estados da *Condition*, que pode estar aprovada ou não aprovada (KOSSOSKI, 2015).
- *Methods* dos *FBEs*: exercitar os parâmetros de entrada do *Method* e configuração de outros *Attributes* ou estados de objetos que serão utilizados (KOSSOSKI, 2015).

Como exemplo, considera-se uma *Premise* *prTest* que avalia se dado *Attribute* *atTest* é menor ou igual a 800. A determinação de classes de equivalência define uma classe de valores válidos e uma classe de valores inválidos para o *Attribute* desta *Premise*, conforme apresentado na Figura 66. Ainda, com o levantamento das classes de equivalência, podem ser planejados os devidos casos de testes para esta *Premise*, conforme apresentados na Tabela 7

					Borda esquerda	Borda direita					
...	796	797	798	799	800	801	802	803	804	...	
Classe de equivalência Valores válidos						Classe de equivalência Valores inválidos					

Figura 66 – Classes de equivalência e análise de valores limite

Fonte: Kossoski (2015)

Tabela 7 – Caso de teste para Premise

Fonte: Adaptado de Kossoski (2015)

Caso de teste	Valor de <i>atTest</i>	Saída esperada ou comportamento esperado
1	799	Aprova a <i>Premise</i>
2	800	Aprova a <i>Premise</i>
3	801	Não aprova a <i>Premise</i>
4	802	Não aprova a <i>Premise</i>

Para cada unidade (*i.e.*, *Premises*, *Conditions*, *Rules*, *Methods*) é necessário o levantamento dessas classes de equivalência que permitem planejar os casos de teste para cada uma delas. A Figura 67 apresenta uma visão expandida da fase de testes unitários, que engloba o planejamento e geração dos casos de testes, execução dos casos de teste e análise dos resultados dos testes unitários (KOSSOSKI, 2015).

2.5.2 Teste de integração em PON

Apenas a verificação das unidades isoladamente não garante o funcionamento adequado do sistema, pois falhas na integração entre eles também podem causar interações que não deveriam ser reproduzidas. Para isso, os testes de integração complementam os testes unitários na verificação do funcionamento do sistema (BINDER, 1999).

Desta forma, os testes de integração em PON podem seguir duas abordagens. A primeira gera casos de testes que exercitem descrições, funcionalidades e comportamentos dos casos de uso definidos para a aplicação ou sistema em PON. Por sua vez a segunda gera casos de testes que exercitem diretamente as entidades do metamodelo do PON. Em todo caso, ambas provocam condições que permitam avaliar os fluxos de notificação da aplicação em PON (KOSSOSKI, 2015).

O processo de criação dos casos de testes para os testes de integração, pode ser muito mais complexo que o dos testes unitários, visto que depende da complexidade e nível de interação entre as entidades do sistema. Dito isto, a estratégia de testes com casos de uso pode ser utilizada para este propósito. A Figura 68 apresenta justamente um diagrama com o fluxo básico e fluxos

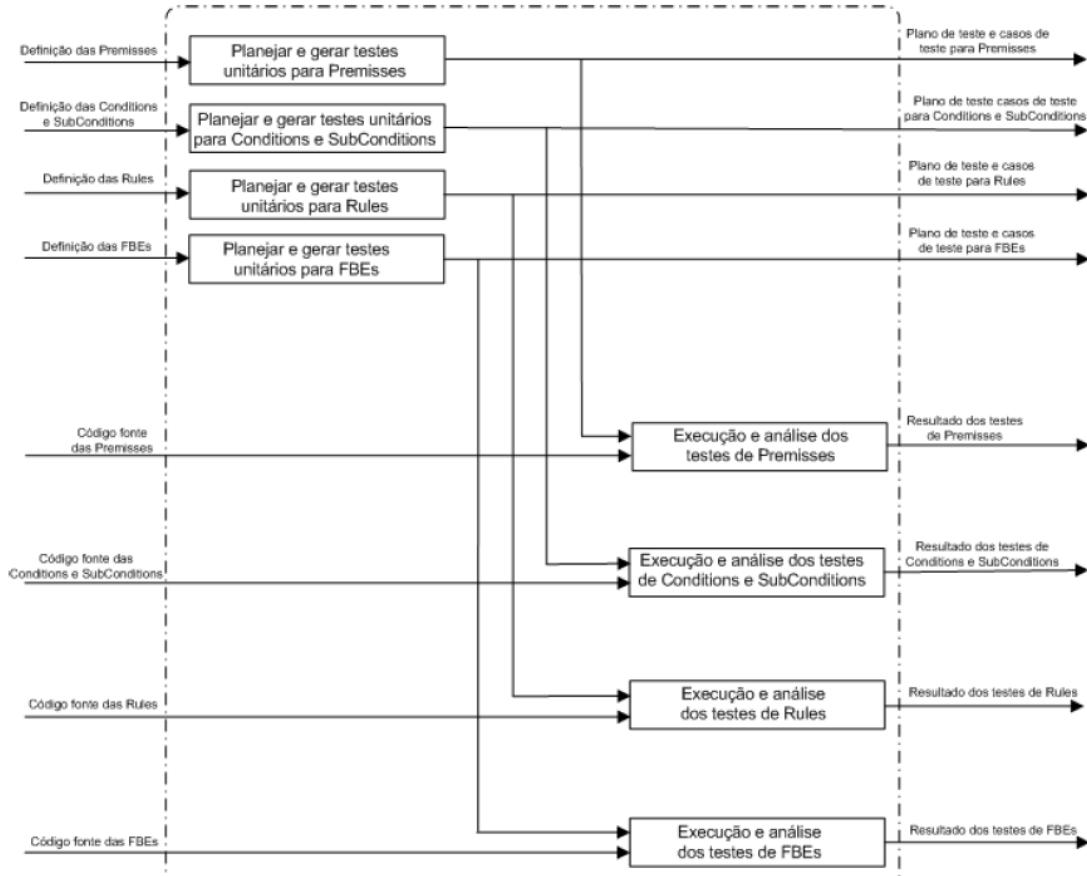


Figura 67 – Fase de testes unitários do PON
Fonte: Kossoski (2015)

alternativos de eventos em um caso de uso.

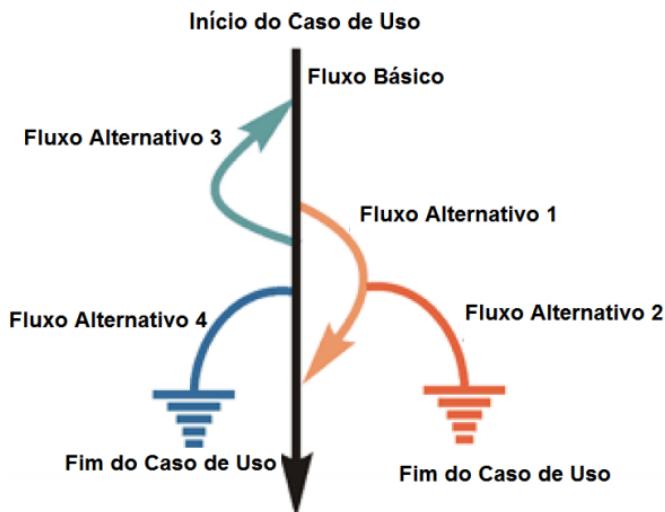


Figura 68 – Fluxo básico e fluxos alternativos de eventos em um caso de uso
Fonte: Kossoski (2015)

Por fim, Kossoski (2015) propõe duas abordagens para o teste de integração utilizando casos de uso: desenvolver estes que exercitem as informações da documentação e comportamento

esperado dos casos de uso; desenvolver testes que exercitem diretamente as entidades que implementam o caso de uso (*i.e.*, *Attributes*, *Premises*, *Conditions* e *Rules*).

A Figura 69 apresenta uma visão expandida da fase de testes de integração, que engloba o planejamento e geração dos casos de testes, execução dos casos de teste e análise dos resultados dos testes de integração (KOSSOSKI, 2015).

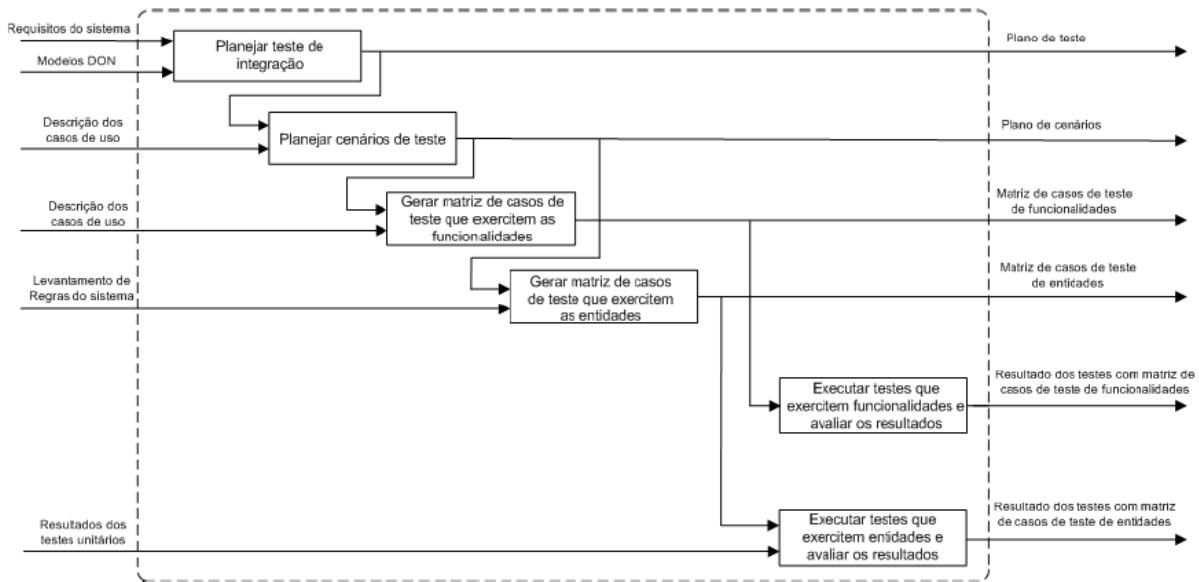


Figura 69 – Fase de testes de integração do PON
Fonte: Kossoski (2015)

Por fim, de modo a concluir a revisão dos assuntos referentes ao PON, a seção seguinte apresenta um breve levantamento dos trabalhos realizados no PON.

2.6 LEVANTAMENTO DOS TRABALHOS REALIZADOS NO PON

Observando a quantidade de trabalhos do PON relatados e citados nas seções anteriores, surge a curiosidade natural de saber em quais anos e qual a natureza destes trabalhos. Assim, nesta seção é feito um levantamento de todos os trabalhos realizados na área do PON enquanto dissertações, teses e publicações de artigo, sendo relatórios técnicos do grupo não aqui contabilizados. Ainda, por se tratar de um trabalho de um grupo específico da UTFPR a busca dos trabalhos é simplificada visto a disponibilidade de um índice contendo a lista de todos os trabalhos já desenvolvidos sob a luz do paradigma orientado a notificações¹².

Os primeiros trabalhos apresentados relativos à base embrionária do PON foram a dissertação de mestrado e tese de doutorado de Simão, defendidas respectivamente em 2001 e 2005

¹² <https://pessoal.dainf.ct.utfpr.edu.br/jeansimao/PON/PON.htm>

(SIMÃO, 2001; SIMÃO, 2005), o que passou no tempo a ser chamado de Controle Orientado a Notificações (CON). A partir de então se generaliza a solução com o que é atualmente chamado Inferência Orientada a Notificações e finalmente como PON (SIMÃO; STADZISZ, 2008; SIMÃO *et al.*, 2009). A partir disso, diversos outros pesquisadores da UTFPR desenvolveram suas dissertações e teses no tema do PON.

Na Figura 70 é apresentado um gráfico que mostra a progressão no número de trabalhos de mestrado e doutorado apresentados, totalizando 16 dissertações de mestrado e 5 teses de doutorado. Além de teses e dissertações, também foi produzido um número significativo de artigos científicos publicados, decorrentes em grande parte dos resultados de pesquisa gerados pelo desenvolvimento das teses e dissertações mencionadas anteriormente. No total já foram publicados 66 trabalhos no tema. Do mesmo modo, na Figura 71 é apresentada a progressão no número de artigos publicados.

Esse levantamento quantitativo dos esforços do PON serve para evidenciar o crescimento deste tema de pesquisa, principalmente por meio da aceitação da comunidade científica, corroborada pelo grande número de artigos publicados. Com base nos trabalhos considerados neste levantamento é feita uma análise mais fina dos esforços referentes à materialização do PON em *software* na Seção 2.4.

2.7 LINGUAGEM DE PROGRAMAÇÃO C++ CONTEMPORÂNEA / C++ MODERNO

Conforme se observou na descrição das materializações do PON, há *frameworks* do PON e alvos de compilação de código notificante específico em Tecnologia LingPON que se apoia na linguagem de programação C++. Isto assim se dá em função do conjunto de características dessa, como seu popular em várias plataformas, ser multi-paradigma, bom equilíbrio entre baixo e alto nível, dentre outros.

Neste sentido, a linguagem de programação C++ já é extremamente conhecida e aplicada na indústria a mais de 30 anos. Portanto, assim como outras linguagens e tecnologias, é natural que ela apresente mudanças e evoluções ao longo dos anos à luz da maturidade e massa crítica em seu uso. Isto posto, principalmente nas revisões do padrão ISO C++ em 2011, 2014 e 2017 e agora também mais recentemente em 2020, um novo e rico conjunto de funcionalidades foi adicionado à linguagem. Ainda assim, manteve-se compatibilidade completa com as versões anteriores (STROUSTRUP, 2020).

Neste âmbito de mudanças, o gráfico da Figura 72 ilustra a evolução da linguagem

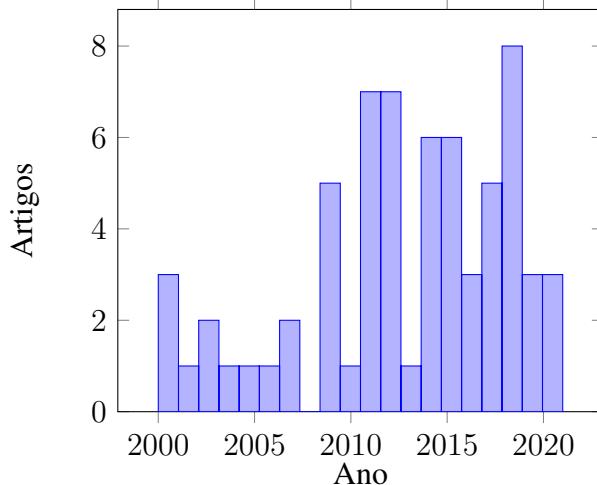


Figura 70 – Gráfico com contagem de artigos publicados a cada ano

Fonte: Autoria própria

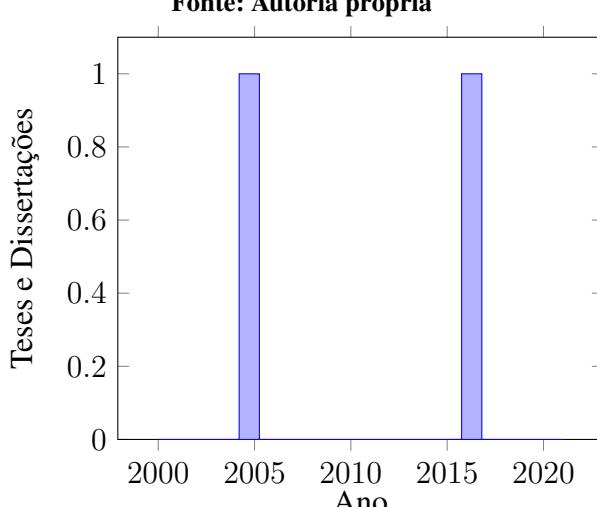


Figura 71 – Gráfico com contagem de teses e dissertações publicadas a cada ano

Fonte: Autoria própria

por meio do número de páginas do documento que compõem a especificação do padrão ISO C++ em suas diferentes versões (ISO/IEC, 2017; FELDMAN, 2019). Esse número de páginas é naturalmente um reflexo das funcionalidades que são adicionadas a cada versão.

Neste trabalho, no quadro da linguagem de programação C++ contemporânea, refere-se como C++ moderno a linguagem C++ com os recursos adicionados no padrão ISO C++ 11 em diante. Dentre essas novas funcionalidades, algumas delas são mais detalhadas nessa subseção, visto que elas são utilizadas como objetivo de melhor atender os objetivos descritos na Seção 1.5, os quais envolvem a elaboração do *Framework PON C++ 4.0*.



Figura 72 – Número de páginas nas diferentes versões do padrão ISO C++

Fonte: Feldman (2019)

2.7.1 Smart Pointers

Um dos principais recursos da linguagem C++ é o uso de ponteiros, que possibilitam a referência a objetos por meio do seu endereço de memória. Entretanto, apesar de ser uma ferramenta que provê grande versatilidade à linguagem, ponteiros também podem ser muito “perigosos” quando utilizados incorretamente, causando acessos inválidos de memória ou vazamentos de memória, podendo resultar em falhas na execução do software (TURNER, 2021).

Essas falhas podem ser desde uma falha total da aplicação, resultando em seu fechamento imediato, ou falhas mais sutis. Estas permitiriam que a aplicação continue sua execução, porém com variáveis apresentando valores inválidos, que por sua vez podem comprometer o funcionamento correto do programa (MEYERS, 2015).

Neste âmbito, uma usual fonte de erros, particularmente para desenvolvedores inexperientes, é gerenciamento de memória manual de ponteiros justamente, visando resolver o problema da alocação e gerenciamento de memória manual imposto pelo uso de ponteiros, o padrão C++ introduziu o conceito de *smart pointers*. Desde então os ponteiros tradicionais são chamados de *raw pointers* (MEYERS, 2015).

A diferença na utilização entre os tipos de *raw pointers* e *smart pointers* é demonstrada nos Códigos 2.28 e 2.29 respectivamente. O principal diferencial é que o uso tradicional de *raw pointers* requer a alocação e desalocação da memória de forma explícita, com o uso das funções *new* e *delete*, enquanto ao utilizar os *smart pointers* isso é feito de forma transparente ao desenvolvedor, pois a alocação é feita na própria inicialização do objeto em *make_unique*, e a desalocação é feita automaticamente ao final do escopo do ponteiro.

Código 2.28 – Uso de raw pointers

```
Object* foo = new Object();
foo->bar();
delete foo;
```

Fonte: Autoria própria**Código 2.29 – Uso de smart pointers**

```
std::unique_ptr<Object> foo =
    std::make_unique<Object>();
foo->bar();
```

Fonte: Autoria própria

Os *smart pointers* funcionam de forma que não é necessário declarar explicitamente a alocação e desalocação de memória, como era feito utilizando as funções *new* e *delete* com os *raw pointers*. Os *smart pointers* são implementados por meio de classes que contam com uma estrutura de controle adicional que é capaz de chamar o destrutor da estrutura quando o objeto sai do escopo, desalocando a memória automaticamente.

Existem 3 principais tipos de *smart pointers* em C++: *unique_ptr*, *shared_ptr* e *weak_ptr*. O *unique_ptr* é um ponteiro único, que só pode ser movido (*i.e.*, transferindo a propriedade da memória para outro ponteiro, e invalidando o ponteiro original), mas não pode ser copiado. Os ponteiros compartilhados *shared_ptr* e *weak_ptr* possuem um contador do número de referências e podem ser tanto movidos como copiados. No caso do *shared_ptr* isso garante que cada instância copiada do ponteiro incrementa o contador de referências, de modo que a memória só é desalocada quando todas as referências deixam de existir. O *weak_ptr* funciona de forma similar ao *shared_ptr* com a diferença que ele não incrementa o número de referências, não garantindo a desalocação da memória do objeto. O *weak_ptr* em si não permite acesso direto à memória, sendo necessário o uso da função *lock()*, que retorna um *shared_ptr* que será válido caso ainda exista alguma referência válida ao objeto.

Esse gerenciamento de memória parte do conceito de propriedade de objetos, no qual o ponteiro que possui a propriedade do objeto é responsável por desalocar a memória no final do seu escopo. Por isso ponteiros do tipo *weak_ptr* não tem propriedade do objeto, enquanto o *unique_ptr* não pode ser copiado, pois possui propriedade exclusiva, e o *shared_ptr* possui propriedade compartilhada.

No contexto do framework PON os *smart pointers* podem ser utilizados para facilitar o desacoplamento das entidades, ao facilitar a criação das entidades de forma não centralizada, não mais necessitando de uma classe responsável pelo gerenciamento da memória manual dos ponteiros, como era utilizado no Framework PON C++ 2.0. A manutenção de referências aos ponteiros entre as estruturas garante a validade de todas as estruturas envolvidas na cadeia de notificações, assim como a subsequente desalocação da memória quando apropriado, de modo a evitar vazamentos de memória que poderiam ocorrer.

Nesse caso, as entidades do PON podem ser utilizadas como *shared pointers* (e.g. `textitstd::shared_ptr<Attribute>`, `std::shard_ptr<Premise>` etc.). Entretanto, para a lógica do mecanismo de notificações, no qual se faz necessário o armazenamento de listas de entidades a serem notificadas, seria mais pertinente o uso de *weak pointers* (e.g.. `std::vector<std::weak_ptr<Premise>>`), pois não seria necessária a criação de novas referências, permitindo a desalocação da memória quando uma entidade deixa de existir.

2.7.2 Expressões *lambda*

Outra adição do padrão C++11 foram as expressões *lambda*. As expressões *lambda* são um objeto de uma função sem nome, capaz de capturar variáveis em seu escopo. As expressões *lambda* permitem declarar uma função *inline* rapidamente (MEYERS, 2015). Elas podem ser utilizadas localmente ou passadas como parâmetro, com a facilidade de não precisar declarar uma função, pois geralmente representam funções que não são utilizadas em outros lugares do código.

Uma expressão *lambda* pode capturar variáveis locais e globais, sendo armazenadas no objeto. Essas capturas podem ocorrer por valor ou por referência, sendo que quando é capturada uma variável por referência é preciso apenas ter cuidado no que diz respeito ao escopo das variáveis. Isto porque a expressão *lambda* pode ser passada como parâmetro, sendo que pode ocorrer o caso da função ser chamada em um ponto no qual a variável que foi capturada por referência já não existe mais.

A cláusula de captura da expressão *lambda* é dada pelo conteúdo entre colchetes `[]`. A captura por valor é dada pela ausência do operador `&` e pode ser auxiliada pelo operador `=`, enquanto a captura de valor por referência é especificada com o uso do operador `&`. Abaixo seguem alguns exemplos de possibilidades de uso destes operadores:

- `[ε]:` captura a variável *epsilon* por referência.
- `[epsilon]:` captura a variável *epsilon* por valor.
- `[&]:` captura todas as variáveis utilizadas no corpo da expressão *lambda* por referência.
- `[=]:` captura todas as variáveis utilizadas no corpo da expressão *lambda* por valor.
- `[&, epsilon]:` captura todas as variáveis utilizadas no corpo da expressão *lambda* por referência e *epsilon* por valor.

- [=, &epsilon]: captura todas as variáveis utilizadas no corpo da expressão lambda por valor e *epsilon* por referência.

O uso de expressão *lambda* é exemplificado no Código 2.30, no qual são declaradas três expressões *lambdas* similares, porém uma recebendo a variável como parâmetro, outra capturando como valor e a última capturando por referência. Nota-se inclusive que em *f_referencia*, devido à captura por referência, o valor da variável *a* é alterado;

Código 2.30 – Caso de uso de expressões *lambda*

```
int a = 1;

auto f[](int a) { printf("%d\n", ++a); }
f(a); // prints "2"
printf("%d\n", a); // prints "1"

auto f_valor=[]() { printf("%d\n", ++a); }
f_valor(); // prints "2"
printf("%d\n", a); // prints "1"

auto f_referencia[&]() { printf("%d\n", ++a); }
f_referencia(); // prints "2"
printf("%d\n", a); // prints "2"
```

Fonte: Autoria própria

No contexto do *framework PON* essas funções são extremamente úteis, pois permitem maior flexibilidade algorítmica na expressão das *Conditions*, permitindo elaborar expressões mais complexas entre *Premises* ao invés de utilizar apenas disjunções e conjunções. Isso se torna possível aliado ao uso de *smart_pointers* e do mecanismo de captura das expressões *lambda*, de modo que a *Condition* poderia utilizar em sua composição uma expressão nos moldes de *[&](pr1 && pr2 || (!pr3 && pr4))*. Além disso, as expressões *lambda* podem prover maior facilidade na declaração de *Methods*, permitindo utilizar qualquer função declarada por meio de uma expressão *lambda*.

2.7.3 *Templates*

Templates são a principal ferramenta para o desenvolvimento de código que opera sobre tipos genéricos em C++. As declarações de classes e funções podem ser feitas por meio de *templates* que realizam a parametrização de um modelo assaz genérico, o qual pode ser aplicado a um ou mais tipos, de tal maneira que o compilador consegue gerar automaticamente uma definição em particular para a utilização naqueles tipos.

O uso de *templates* permite ao desenvolvedor escrever o código apenas uma vez, deixando a cargo do compilador gerar todas as variações necessárias aos tipos aplicados. O uso

correto de *templates* pode reduzir significativamente ou até mesmo eliminar a quantidade de código repetido, que seria necessário para a implementação específica de funções e algoritmos para diversos tipos. Um exemplo disso é mostrado no Código 2.31. Nesse caso uma mesma função é utilizada para realizar uma avaliação aritmética sobre dois tipos diferentes, *int* e *long*.

Código 2.31 – Uso de *templates*

```
template<typename T>
int GetMax(T a, T b)
{
    T result;
    result = (a>b) ? a:b;
    return result;
}

int main()
{
    int i=5, j=6, k;
    long l=10, m=50, n;
    k = GetMax<int>(i,j);
    n = GetMax<long>(l,m);
    return 0;
}
```

Fonte: Autoria própria

A utilização de *templates* facilita o desenvolvimento de bibliotecas de código, pois em muitos casos o desenvolvedor não tem conhecimento de todos os tipos que podem ser utilizados com suas funções. Nesse caso, os *templates* permitem que o código seja implementado de forma genérica no sentido de deixar o tipo como um parâmetro, de modo que os tipos serão especificados apenas durante a sua utilização. Essa técnica também é conhecida por programação genérica (DEHNERT; STEPANOV, 1998).

A técnica de *templates* pode ser aplicada ao *framework* do PON de forma a prover suporte para *Attributes* com tipos genéricos por meio da implementação de uma classe *Attribute* utilizando *templates*, o que remove uma das limitações do *Framework PON C++ 2.0* que era o uso de tipos fixos, assim como permite reduzir a quantidade de código repetido necessário para a implementação específica para cada tipo de *Attribute*. Como as operações sobre os *Attributes* realizadas nas *Premises* são geralmente comparações simples, sendo definidas por padrão para os tipos básicos (*int*, *float*, *string* etc.), isso já cobre todos os cenários que o *Framework PON C++ 2.0* suporta, e para tipos definidos pelo usuário (classes e *structs*) bastaria que o mesmo declarasse os operadores de comparação. Nesse sentido, o Código 2.32 apresenta uma declaração simplificada de uma classe *Attribute*, na qual o tipo *T* é utilizado como *template* para a variável *value* que armazena o valor do *Attribute*.

Código 2.32 – Attribute com template

```
template<typename T>
class Attribute {
    T value;
};

template class Attribute<int>;
template class Attribute<float>;
template class Attribute<std::string>;
```

Fonte: Autoria própria

2.7.3.1 Variadic templates e Fold expressions

O padrão C++11 adicionou novos recursos que permite um uso mais avançado de *templates*, os pacotes de parâmetros, por meio de *variadic templates*. Com esses recursos, é possível declarar funções que aceitam um número variável de argumentos *templates*, que é o dito *variadic template*. Ou seja, uma função com pacote de parâmetros pode aceitar um número variável de parâmetros de qualquer tipo. Quando uma função que utiliza pacotes de parâmetros é instanciada, o compilador é responsável por expandir os parâmetros na mesma ordem em que foram declarados (MEYERS, 2015).

Os *variadic templates* são também frequentemente utilizados em conjunto com as *fold expressions*, que são um recurso adicionado no padrão C++17. Até C++11, para trabalhar com os pacotes de parâmetros ainda era necessário implementar uma função para cada caso, porém isso é contornado com as *fold expressions*. A *fold expression* é nada mais que um novo método para expandir os parâmetros de um *variadic template*, agora podendo instanciar várias chamadas de funções e operadores sob os parâmetros expandidos.

Um exemplo clássico de aplicação é a função de soma utilizando *fold expression* mostrada no Código 2.33. Nesse exemplo é mostrado o pacote de parâmetros, indicado por “...”. A expansão também se encarrega de expandir o operador “+” para cada um dos parâmetros. Desta forma uma expansão de *fold_sum(a, b, c)* resultaria em *return (a + b +c);*.

Código 2.33 – Uso de fold expressions

```
template<typename... T>
auto fold_sum(T... s) {
    return (... + s);
}
```

Fonte: Autoria própria

Para o *framework* do PON o benefício que as *fold expressions* podem trazer é ao facilitar a construção das estruturas mais complexas como das *Conditions* e *Instigations*, pois elas podem

conter um número variável de parâmetros e com isso é possível criar funções genéricas que cobrem todos os casos de utilização e facilitam a declaração destas estruturas, tornando possível inicializar estruturas complexas com uma única linha de código. Uma suposta inicialização de uma *Action*, por exemplo, poderia ser dada conforme o Código 2.34, de modo que sua inicialização poderia ser feita com um número variável de *Instigations*.

Código 2.34 – Action com *fold expressions*

```
class Action {
    template<typename... T>
    Action(T... instigations);
}

Action ac1(in1);
Action ac2(in1, in2, in3, in4);
```

Fonte: Autoria própria

2.7.3.2 Expressões constantes

As expressões constantes em C++ são expressões compostas por valores ou funções que podem ser avaliadas em tempo de compilação. O identificador *constexpr* foi adicionado no C++11 para especificar funções e variáveis que compõe expressões constantes.

Em sua forma mais simples, *constexpr* pode ser aplicado a uma variável que tenha sua inicialização imediata em sua construção. Funções definidas como *constexpr* são capazes de ter seu valor de retorno avaliado já durante a compilação, ao invés de ser avaliada durante a execução do programa. Uma aplicação comum para funções *constexpr* é a realização de cálculos matemáticos, inclusive com o uso de recursividade, como, por exemplo, para a função de cálculo factorial, mostrada no Código 2.35 (SILISTEANU, 2017).

Código 2.35 – Aplicação de função *constexpr*

```
constexpr int factorial(int n) {
    return n <= 1 ? 1 : (n * factorial(n - 1));
}

constexpr int i = factorial(10);
```

Fonte: Adaptado de Silisteau (2017)

Desde o C++17 as expressões do tipo *if* também permitem o uso de *constexpr*. Isso permite que o compilador otimize o código ao resolver a expressão durante a compilação, de modo a não avaliar a expressão condicional durante a execução do programa (LIPPMAN *et al.*, 2013). Entretanto, para isto ser possível, a expressão contida no *if* deve ser passível de ser

avaliada em tempo de compilação, seja isto por meio de outras funções *constexpr* ou até mesmo com o uso de *templates* (DEANE; TURNER, 2017).

Além do ponto de vista de otimizações do compilador, o uso de *constexpr* permite também escrever código genérico de forma simplificada, pois possibilita a descartar o pedaço de código que não atendam a expressão determinada, como no caso da função apresentada no Código 2.36. Neste exemplo não é possível compilar a linha *return *t;* caso o tipo do template *T* não seja um ponteiro.

Código 2.36 – Aplicação de função *constexpr*

```
template <typename T>
auto get_value(T t) {
    if constexpr (std::is_pointer_v<T>) {
        return *t;
    }
    else {
        return t;
    }
}

int i = 123;
int a = get_value(i);
int b = get_value(&i);
```

Fonte: Autoria própria

No contexto do *framework* do PON, as expressões constantes podem ser utilizadas de modo a melhorar significativamente o desempenho ao reduzir ou até mesmo eliminar o número de operações condicionais (*if*) durante a execução ao substituir as mesmas por *if constexpr* onde possível. Pode ser considerado como exemplo uma função hipotética do mecanismo de notificações *notify*, na qual qualquer avaliação lógica desnecessária pode representar uma significativa perda de desempenho devido a ser chamada com muita frequência. Nesse caso podem ser utilizadas as expressões constantes para resolver estas avaliações na etapa de compilação, como ilustrado no Código 2.37, onde a função *notify* hipotética possui duas lógicas possíveis para o tratamento das notificações.

2.7.4 Programação concorrente

No contexto de programação concorrente, C++11 e C++17 apresentaram uma base para a criação de aplicações concorrentes, visando possibilitar a execução de programas em ambientes *multithread* de forma consideravelmente mais descomplicada e eficiente do que as manipulações “burocráticas” baseadas em bibliotecas externas das versões anteriores do C++.

Código 2.37 – Método *Notify* da classe *Observable* no Framework PON C++ 4.0

```

using NOPFlag = int;
static constexpr NOPFlag Default = 0b0000;
static constexpr NOPFlag NoNotify = 0b0001;
static constexpr NOPFlag ReNotify = 0b0010;
static constexpr NOPFlag Parallel = 0b0100;

template <NOPFlag Flag>
void Notify() {
    if constexpr (0 < (ReNotify & Flag))
    {
        // Não notifica
    }
    else if constexpr (0 < (ReNotify & Flag))
    {
        // Realiza processo de renotificação
    }
    else if constexpr (0 < (Parallel & Flag))
    {
        // Realiza processo de notificação paralelizado
    }
    else // Default
    {
        // Realiza processo de notificação normal
    }
}

```

Fonte: Autoria própria

2.7.4.1 Tarefas assíncronas

Com o C++11 foi introduzindo o conceito de tarefas (*tasks*). Uma *task* é um simples trabalho que é iniciado e é gerenciado automaticamente pelo sistema (GRIMM, 2017). A estrutura mais básica disponível para execução de operações assíncronas em C++ é a *std::thread*, a qual permite que múltiplas linhas de execução sejam executadas de forma concorrente.

Uma *std::thread* inicia assim que é construída, ficando a cargo do sistema operacional escalarizar sua execução. O programa que inicia a *std::thread* deve aguardar o final de sua execução, podendo obter resultados por meio do retorno de valores ou do acesso a regiões de memória compartilhada.

Além disso, existe a função *std::async*, com ela é possível iniciar uma tarefa em uma *thread*. Ao chamar a função *std::async* é passado como parâmetro uma função, por meio de ponteiros de função ou expressões *lambda*. Nesse ponto é inicializada de forma transparente uma *thread* responsável pela execução da função, e retornando um objeto do tipo *std::future*, que pode, por meio do método *get()*, ser utilizado para obter o valor de retorno da função inicializada com *std::async*. O sistema vai escolher quando a tarefa vai ser executada, mas o objeto *std::future* criado pode ser acessado a qualquer momento para obter o retorno do processo executado e, caso o processo ainda não tenha sido executado, o obriga a ser executado e retornar o valor imediatamente.

O exemplo apresentado no Código 2.38 mostra o uso de `std::async` para a realização de um cálculo matemático para verificar se o número é primo. Essa operação representa um cálculo que, para números grandes, pode ser bastante lento. Desta forma o cálculo pode ser executado por uma *thread* separada enquanto a aplicação fica livre para realizar outras operações mais importantes. Idealmente, quando a aplicação precisar acessar o valor calculado pela função, sua operação já terá sido finalizada.

Código 2.38 – Uso de `std::async`

```
bool is_prime (int x) {
    for (int i=2; i<x; ++i) {
        if (x%i==0) {
            return false;
        }
    }
    return true;
}

int main () {
    std::future<bool> fut = std::async(is_prime, 313222313);
    /* Realiza outras operações importantes */
    if (fut.get()) // aguarda is_prime finalizar execução
    {
        std::cout << "It is prime!\n";
    }
    return 0;
}
```

Fonte: Autoria própria

No contexto do *framework* PON a função `std::async` pode ser implementada para a execução de *Methods* de maneira assíncrona, como exemplificado no Código 2.39. Nesse caso, o *Method* possui uma variável estática *future*, que é utilizada para controlar a execução do *Method* e fazer com que uma nova instigação do *Method* garanta que a execução anterior seja finalizada, ao chamar o método `future.get()`.

Código 2.39 – Execução de métodos assíncronos

```
void async_method(std::function<void()> method)
{
    static std::future<void> future;
    if (future.valid())
    {
        future.get();
    }
    future = std::async(std::launch::async, [mt = std::move(method)]() { mt(); });
}
```

Fonte: Autoria própria

2.7.4.2 Políticas de execução

Já no C++17 foram introduzidos os algoritmos paralelos da *Standard Template Library* (STL). Isso significa que a maior parte dos algoritmos da STL pode ser executado de forma sequencial, paralela ou vetorizada (GRIMM, 2017). Entretanto, nem todos os algoritmos apresentam ganhos de desempenho quando comparados com a implementação sequencial (O’NEAL, 2018).

A invocação destes algoritmos de forma paralela é feita através das políticas de execução (*execution policies*), que permitem especificar a maneira como o desenvolvedor deseja executar dado algoritmo, com três opções possíveis:

- Sequencial (*std::seq*): execução de forma tradicional sequencial sem paralelização.
- Paralela (*std::par*): execução de forma paralela, que permite a execução em múltiplas *threads* em paralelo.
- Vetorizada (*std::par_unseq*): execução de forma paralela, similar a *std::par*, porém permitindo entrelaçar a execução das múltiplas *threads* de forma não sequencial, dita vetorizada (FILIPEK, 2018).

Em suma, *std::seq* executa algoritmos sequencialmente, *std::par* executa com paralelismo e *std::par_unseq* executa com paralelismo e utilizando instruções vetorizada, como SSE e AVX¹³ (FILIPEK, 2018). Um exemplo para a aplicação de um algoritmo de ordenação *std::sort* é apresentado no Código 2.40.

Código 2.40 – Uso de políticas de execução

```
std::vector<int> v = { 1, 2, 3 };
std::sort(std::execution::seq, v.begin(), v.end());
std::sort(std::execution::par, v.begin(), v.end());
std::sort(std::execution::par_unseq, v.begin(), v.end());
```

Fonte: Autoria própria

Enquanto os algoritmos da STL já apresentam implementações paralelizáveis, como *std::sort*, o desenvolvedor também pode desenvolver seus próprios algoritmos e recorrer às políticas de execução por meio do uso de expressões *lambda* combinados com algoritmos de aplicação de funções como *std::for_each*, que aplica dada função sobre todos os elementos da

¹³ SSE e AVX fazem parte do conjunto de instruções SIMD suportado em processadores modernos que permite a processamento paralelizado em múltiplos núcleos de uma CPU (JEONG *et al.*, 2012)

sequência desejada. No Código 2.41 é demonstrado o potencial uso para a paralelização do mecanismo de notificações, considerando o processo de notificação das *Premises* de um *Attribute*. Esse mecanismo seria similar ao *Parallel.ForEach* do C# utilizado no *Framework PON C# IoT*.

Código 2.41 – Uso de políticas de execução no PON

```
// Conjunto hipotético de premissas a serem notificadas por um Attribute
std::vector<Premise> premises{pr1, pr2, ..., prn};

std::for_each(std::execution::par_unseq,
    premises.begin(), premises.end(),
    [] (Premise& premise) {
        premise.notify();
    }
);
```

Fonte: Autoria própria

No C++20 ainda não é possível especificar como essa paralelização será realizada, como especificar em quantas ou em quais *threads* o código deve ser executado, deixando os compiladores responsáveis por determinar o melhor modo de execução. Como exemplo, no compilador da Microsoft (MSVC), isto é feito por meio de uma *thread pool*¹⁴, que faz proveito dos mecanismos de paralelização do Windows, não disponíveis na STL (O’NEAL, 2018).

2.7.4.3 Mecanismos de sincronização

Ainda no que diz respeito à execução de aplicações concorrentes é sempre importante prestar atenção ao acesso de variáveis compartilhadas. A proteção do acesso por múltiplos processos concorrentes pode ser feita por meio do uso de mecanismos de exclusão mútua, já presentes nas versões C++11, como *std::mutex* (BOCCARA, 2019).

Um *mutex* é utilizado para impedir que duas ou mais *threads* acessem simultaneamente determinado trecho de código ou dados que deve ser exclusivo para uma dada *thread* em um dado momento. Assim, cada *thread* executando pode bloquear o *mutex* impedindo a execução de outras *threads* que utilizem o mesmo *mutex* enquanto a mesma não liberar o recurso (BOCCARA, 2019).

No contexto do PON, o mecanismo *std::thread* pode ser utilizado, por exemplo, para a execução de *Methods* de maneira assíncrona, enquanto os algoritmos com política de execução paralela podem ser utilizados na materialização do mecanismo de notificação em si, utilizando *std::mutex* como mecanismo para sincronizar a execução. Nesses casos, potencialmente haveria

¹⁴ *Thread pools*, no contexto do Windows, fazem parte de um subsistema responsável por criar e gerenciar as *threads* para um número arbitrário de tarefas requeridas por aplicações e serviços (TEIXEIRA, 2012).

o acesso concorrente a recursos, como ao tentar atribuir o valor de dado *Attribute*, nesses casos o `std::mutex` pode ser utilizado para controlar o acesso a este recurso, como é mostrado no Código 2.42.

Código 2.42 – Uso de *mutex* nos *Attributes*

```
void Attribute<T>::SetValue(T value)
{
    static std::mutex mutex_;
    mutex_.lock();
    value_ = value;
    // Realiza outras operações do processo de notificações
    mutex_.unlock();
}
```

Fonte: Autoria própria

Em que pese suas vantagens, esses recursos apresentados de C++ moderno potencialmente podem trazer ganhos tanto do ponto de vista como de desempenho como de usabilidade do *framework*. Entretanto, o uso de tais recursos pode também contribuir a um aumento da complexidade do código do *framework*. Nesse sentido, é importante a existência de mecanismos que permitam testar o *framework*, com o propósito de validar que todos os recursos utilizados se comportam conforme o esperado. Deste modo, as seções seguintes discorrem sobre o método TDD, particularmente explorando os *frameworks* de testes em linguagem de programação C++.

2.8 TDD E FRAMEWORKS DE TESTE

Ao passo em que a Seção 1.1.5 introduz os principais conceitos do TDD, inclusive elucidando o seu método definido à luz da bibliografia de referência (AMBLER, 2006), a seção atual apresenta uma revisão estado da técnica do TDD, particularmente no que diz respeito à aplicação do TDD na linguagem de programação C++, que se da por meio do uso de *frameworks* de testes.

Oportunamente, os testes unitários, de integração ou desempenho por si só não precisariam necessariamente de um *framework* específico para serem executados, como o Google Test, sendo que bastaria a criação de código que execute o teste e seja capaz de indicar o sucesso ou falha. Porém, este processo de escrever e executar os testes pode ser trabalhoso devido à grande quantidade de código dito *boilerplate*¹⁵ para se conseguir implementar a funcionalidade desejada pelo desenvolvedor e necessária para se executar um único teste. Neste quadro, seria necessário

¹⁵ Em termos de jargão, o código dito *boilerplate* se refere a código que deve ser repetido em diversos lugares, sem modificações

criar pelo menos uma função principal pertinente, assim como laços de repetição que iterem chamando todos os testes.

Para contornar esse problema aqui relatado, no estado da técnica de desenvolvimento e testes de *software* são utilizados os *frameworks* de teste justamente, havendo *frameworks* para tal aplicáveis a distintas linguagens de programação. Neste âmbito, existem vários *frameworks* de testes disponíveis em C++, como Catch2, QTest e Google Test. Para este projeto foi escolhido o Google Test por, além de automatizar os testes unitários e de integração, possuir também integração para com um *framework* de *benchmark* compatível, o Google Benchmark, facilitando o desenvolvimento dos testes de desempenho. Em tempo, do ponto de vista de teste de software, um *benchmark* é um teste com o objetivo de avaliar o desempenho de execução de determinada tarefa a luz de um cenário conhecido e de resultados conhecidos (NAMBIAR *et al.*, 2009).

Com a utilização dos *frameworks* de teste são atendidos requisitos importantes para a aplicação efetiva do TDD, que os testes devem ser fáceis de se desenvolver, e também fáceis de se executar (GOOGLE, 2020b). O uso dos *frameworks* permite que o desenvolvedor não precise desenvolver o código *boilerplate* necessário para a execução dos testes, facilitando o desenvolvimento, assim como torna simples e rápida a execução dos testes, geralmente por meio da execução de um único executável que execute e avalie todos os testes de forma automatizada.

A facilidade provida pelo uso de *frameworks* na aplicação do vem ao encontro da proposta de testes no PON de Kossoski (2015), na qual a execução dos testes era feita de forma manual, dificultando a aplicação do método. Neste contexto, observa-se que nenhum outro *framework* existente do PON faz a aplicação do método de testes proposto. Ainda que o método tenha sido proposto com foco no teste de aplicações desenvolvidas em PON, e não dos *frameworks* do PON em si, muitos dos conceitos apresentados também são relevantes ao teste de *frameworks* do PON.

Para a implementação dos testes do novo *framework* do PON foram escolhidos dois *frameworks* de teste para fins de TDD, o Google Test e Google Benchmark. Em suma, o Google Test é utilizado para a execução de testes unitários e de integração, com o objetivo de garantir o funcionamento correto de todas as unidades de execução do *framework*, como os *Attributes*, *Premises*, *Conditions* e *Rules*, inspirando-se no que foi definido por Kossoski (2015). Por sua vez, o Google Benchmark é utilizado para realizar testes de desempenho, permitindo de forma fácil e confiável que seja avaliado o desempenho do *framework* em casos determinados. A execução destes testes durante o desenvolvimento auxilia o desenvolvedor ao prover garantias e

demonstrações do funcionamento das implementações conforme o esperado.

2.8.1 Google Test

O Google Test é um dentre os diversos *frameworks* criado para facilitar o desenvolvimento de testes unitários em C++. Ele foi criado pela equipe de tecnologia do Google, sendo desenvolvido para ser multiplataforma e suportar todo tipo de teste, não apenas testes unitários, o que naturalmente inclui os testes de integração.

O *framework* Google teste é construído em cima dos seguintes princípios (GOOGLE, 2020b):

- Testes devem ser independentes e repetíveis.
- Testes devem ser organizados e refletir a estrutura do código testado.
- Testes devem ser portáveis e reutilizáveis.
- Quando os testes falham eles devem prover o máximo de informação possível sobre o problema.
- O desenvolvedor dos testes deve poder se preocupar apenas com a lógica do teste, sem precisar se preocupar em como eles serão executados.
- O teste deve ser capaz de ser executado de forma rápida, com baixo tempo de processamento.

Durante os testes o desenvolvedor deve inserir asserções (*i.e.*, assertivas ou afirmações), as quais são responsáveis por parametrizar a validação do estado das variáveis do código em testes. Existem dois tipos de asserções no Google Test, EXPECT e ASSERT. Ambas realizam a mesma avaliação lógica, porém ASSERT gera uma falha fatal que aborta o teste atual, enquanto um EXPECT que falha permite que o resto do teste seja executado. As funções de asserções podem ser executadas sobre qualquer variável que aceite os operadores de comparação.

Nesse contexto, pode ser tomado como exemplo um caso de teste de uma *Premise* simples, como no Código 2.43. Nesse exemplo a *Premise* *pr1*, que avalia se *at1 == 1*, é testada como uma unidade, ou seja, interessa ao teste apenas o estado da *Premise*. Desta forma, a *Premise* inicialmente não deve estar aprovada, então com a alteração do valor de *at1* para 1, a mesma

deve ser aprovada. Nesse exemplo são utilizadas as asserções básicas do tipo EXPECT para validar o resultado do teste.

Código 2.43 – Caso de teste com Google Test

```
TEST(Premise, Simple)
{
    NOP::SharedAttribute<int> atl = NOP::BuildAttribute<int>(-1);
    NOP::SharedPremise prl = NOP::BuildPremise<int>(atl, 1, NOP::Equal());

    EXPECT_FALSE(prl->Approved());
    atl->SetValue(1);
    EXPECT_TRUE(prl->Approved());
}
```

Fonte: Autoria própria

2.8.2 Google Benchmark

No que diz respeito ao Google Benchmark ele é muito similar ao que é aplicado no Google Test, inclusive o Google Benchmark é construído com base na fundação do Google Test. O Google Benchmark também é utilizado para se escrever testes, porém ao invés de testes unitários como no Google Test, ele realiza testes de desempenho.

O Google Benchmark possui um mecanismo que realiza o teste de uma função quantas vezes for necessário para se avaliar com maior confiabilidade o seu tempo de execução, pois podem ocorrer variações entre cada execução. Portanto, o *framework* Google Benchmark possui esse mecanismo que executa a mesma função até que sua temporização seja estável ou após uma passagem de tempo muito longa (*timeout*).

É possível realizar o *benchmark* de qualquer função, porém é necessário declarar a função utilizando as variáveis de controle do Google Benchmark. Isto justamente para que o mesmo consiga realizar esse processo de avaliação da temporização enquanto controla automaticamente o número de execuções.

Como mostrado no Código 2.44, a criação do teste é bem simples, sendo que basta declarar qualquer função que receba *benchmark::State&* como parâmetro e esta função a ser testada deve ser colocada no laço de repetição *for(auto _ : state)*, qualquer código fora desse laço não é temporizado. Este laço é precisamente o responsável por gerenciar o número de iterações que serão executadas automaticamente. Neste exemplo é criada uma função de *benchmark* parametrizada para uma aplicação hipotética de sensores em PON, com 2 argumentos que permitem controlar o número de sensores criados e o número de *Rules* aprovadas.

Código 2.44 – Caso de teste com Google Benchmark

```

static void BM_Sensor(benchmark::State& state)
{
    // Este código não é temporizado
    std::vector<std::shared_ptr<NOPSensor>> sensors;

    // state.range(0) é o primeiro argumento
    for (int i = 0; i < state.range(0); i++)
    {
        sensors.push_back(std::make_shared<NOPSensor>());
    }

    // Este código é temporizado
    for (auto _ : state)
    {
        // state.range(1) é o segundo argumento
        for (int i = 0; i < state.range(1); i++)
        {
            sensors[i]->Activate();
        }
    }
}

BENCHMARK(BM_Sensor)->Args({100000, 100000}); // Registra benchmark com parâmetros
BENCHMARK_MAIN(); // Executa todos os benchmarks registrados mod

```

Fonte: Adaptado de Google (2020b)

O resultado proveniente da execução de um teste com o Google Benchmark é exemplificado na Figura 73, na qual são apresentadas 4 colunas, as quais são explicadas abaixo:

- *Benchmark*: Nome do teste, acompanhado dos parâmetros passados
- *Time*: Tempo total gasto para cada iteração
- CPU: Tempo de CPU gasto para cada iteração
- *Iterations*: Número total de iterações realizadas

Benchmark	Time(ns)	CPU(ns)	Iterations
<hr/>			
BM_SetInsert/1024/1	28928	29349	23853
BM_SetInsert/1024/8	32065	32913	21375
BM_SetInsert/1024/10	33157	33648	21431

Figura 73 – Resultado de teste com Google Benchmark

Fonte: Google (2020b)

2.9 REFLEXÕES SOBRE OS PROBLEMAS EM ABERTO

O PON é um paradigma relativamente novo, por isso é natural se esperar que as suas materializações ainda não tenham atingido um nível de maturidade pleno necessário para o desenvolvimento de um projeto de *software* completo, particularmente do ponto de vista dito

industrial. Neste âmbito, enquanto a Tecnologia LingPON encontra-se em um estado bem mais prototípico tendendo ao estado da arte, os *frameworks* em PON estão em estado menos prototípico e mais estáveis tendendo conjuntamente ao chamado estado da técnica.

Assim, as materializações do PON em *framework* para *software*, em especial o *Framework* PON C++ 2.0 enquanto efetivo estado da técnica em PON, ainda são a principal ferramenta utilizada para o desenvolvimento de aplicações em PON. Apesar das versões de *framework* entregarem toda a funcionalidade necessária para o desenvolvimento de aplicações no PON, eles ainda apresentam alguns problemas, inclusive de usabilidade, conforme foi relatado caso a caso e também de maneira geral no decorrer deste capítulo.

Neste âmbito, ainda que o *Framework* PON C++ 2.0 seja considerado a materialização mais estável dentre os *frameworks* existentes, a Seção 2.4.6.2 destaca os problemas de verbosidade, baixa flexibilidade de tipos, baixa flexibilidade algorítmica, baixa confiabilidade e curva de aprendizado íngreme presentes no *framework*. Apesar da implementação do *Framework* PON C++ 2.0, apresentar melhorias significativas com relação a versões anteriores, ela continua demandando uma quantidade muito significativa de código *boilerplate*.

Ainda no que diz respeito ao paralelismo, o esforço apresentado pelo *Framework* PON C++ 3.0, introduziu a implementação de paralelismo em nível de *threads* nos *frameworks* em C++, entretanto com limitações e instabilidades que inviabilizam o seu uso, conforme relatado na Seção 2.4.1.4 (MARTINI *et al.*, 2019). Além disso, apesar de haver outros dois *frameworks* que tratem bem de paralelismo em nível de *threads*, nomeadamente o *Framework* PON Erlang/Elixir e o *Framework* Akka descritos na Seção 2.4, eles pecam em termos de desempenho de processamento por não usarem uma linguagem de programação performante como o C++. Nesse âmbito, um novo *framework* PON em C++ deveria explorar a propriedade de desacoplamento para fins de paralelismo implícito em nível de *thread*, utilizando para tal recursos ditos modernos da linguagem.

Neste sentido, o desenvolvimento de uma nova versão de *framework* que seja capaz de tratar os problemas mencionados acima é necessário para facilitar e viabilizar o desenvolvimento de aplicações mais complexas em PON, tornando assim o seu estado da técnica mais efetivo e apropriado a este termo de um ponto de vista mais sistêmico. Em suma, já se está no momento de alcançar uma materialização do PON madura e profissionalizada a ponto de permitir aplicações cada vez mais com caráter industrial.

Neste âmbito, a aplicação dos conceitos de programação genérica e de técnicas de

desenvolvimento de *software* ditas modernas em C++ deve permitir o desenvolvimento de uma versão do *framework* com significativamente menos linhas de código escritas, podendo influenciar em questões de desempenho, por exemplo, o que é uma propriedade elementar do PON. Isto facilitaria a execução de testes e a manutenção futura do código do *framework*, de modo que se torne significativamente mais fácil expandir e adicionar novos recursos, sem necessitar de refatoração de código para cada nova implementação.

Além disso, a aplicação de programação genérica permite melhorar usabilidade do *framework* facilitando a integração com outras bibliotecas e APIs, assim como reduzindo a dificuldade de utilização e redução da curva de aprendizado. Aliado a isto, a forma de escrita de *FBE* e *Rules* pode e deve ser sinergicamente melhorada para permitir programação em mais alto nível, o que se constitui em outra propriedade elementar do PON em si. Com isso tudo, espera-se futuramente observar um aumento no volume e complexidade de aplicações desenvolvidas com PON.

Por fim, a aplicação do TDD, com o auxílio de *frameworks* de teste, pode e deve ser utilizada para garantir que o funcionamento do *framework* corresponde com o esperado, com isso garantindo em algo que não existirão problemas durante a utilização do *framework* e mesmo suas aplicações, bem como tratando o problema da baixa confiabilidade já levantado. É derradeiramente pertinente e importante ressaltar ser virtual e naturalmente impossível se testar completamente um programa, como *framework* para o PON, porém quanto melhor o conjunto de testes desenvolvidos maior é a garantia de que o mesmo não apresente problemas durante sua execução (NAIK; TRIPATHY, 2018).

3 FRAMEWORK PON C++ 4.0

Esse capítulo apresenta os esforços realizados no desenvolvimento do *Framework PON C++ 4.0*. Primeiramente, na Seção 3.1, é apresentada de forma geral a estrutura do *framework* em si, enquanto as seções seguintes apresentam maiores detalhes de implementação de cada uma das entidades do PON no *framework*. A Seção 3.9 também apresenta alguns conceitos introduzidos com o objetivo de facilitar o desenvolvimento com o *Framework PON C++ 4.0*. Ainda, a Seção 3.11 apresenta os testes desenvolvidos à luz do método TDD. Por fim, a Seção 3.12 reflete sobre os resultados apresentados pelo desenvolvimento do *Framework PON C++ 4.0*.

3.1 ESTRUTURA

A proposta do *Framework PON C++ 4.0* é solucionar de maneira suficiente os problemas detalhados ao longo do Capítulo 2, como aqueles reportados na Seção 2.9, de acordo com os objetivos da Seção 1.5. Para alcançar esses objetivos, propõe-se arquitetar este novo *framework*, nomeadamente *Framework PON C++ 4.0*, utilizando ferramentas de desenvolvimento de C++ dito moderno, como *smart pointers*, *variadic templates*, *fold expressions* e expressões *lambda*, fazendo a aplicação dos conceitos de programação genérica e utilizando o método de desenvolvimento orientado a testes.

Em resumo, a Tabela 8 apresenta os principais objetivos a serem atingidos com o desenvolvimento do *Framework PON C++ 4.0*, à luz dos objetivos da dissertação apresentados na Seção 1.5. Na tabela são apresentados os problemas presentes nos *frameworks* anteriores e, de maneira sucinta, como o desenvolvimento do *Framework PON C++ 4.0* pretende resolvê-los.

Tabela 8 – Problemas a serem endereçados pelo *Framework PON C++ 4.0*
Fonte: Autoria própria

Objetivo	Proposta
Adicionar a flexibilidade de tipos	<i>Attribute</i> com tipo <i>template</i>
Adicionar flexibilidade algorítmica	<i>Condition</i> com expressões <i>lambda</i>
Reducir a verbosidade da utilização	<i>Builders</i> com <i>variadic templates</i>
Permitir execução com paralelismo	Notificações com políticas de execução

Em tempo, naturalmente trechos de código do proposto *Framework PON C++ 4.0* serão apresentados ao longo deste capítulo, de forma a explicar avanços e melhorias nele. Entretanto, para fins de simplicidade, os trechos de código em sua maior parte omitem alguns detalhes de implementação, apresentando apenas os trechos necessários para a compreensão dos conceitos

de forma adequada¹.

De forma a facilitar o entendimento dos conceitos e técnicas aplicados no *Framework PON C++ 4.0* e apresentados ao longo deste capítulo, será utilizado o exemplo da aplicação de sensores, que também já foi introduzido na Figura 6 da Seção 1.1.3. A representação deste *FBE* e *Rule* em NOPL e sua implementação com o *Framework PON C++ 4.0* são apresentados, no Código 3.1. Nestes exemplos destaca-se que, salvo as peculiaridades da linguagem de programação C++ (como inicialização das variáveis), ambos os exemplos apresentam estruturas bastante similares².

Isto dito, o já bem supra salientado precedente *Framework PON C++ 2.0* foi construído com base no C++98, que contém um conjunto de recursos muito limitado quando comparado às versões mais novas disponíveis, como C++20. De maneira geral, a despeito da implementação em si, o modelo/projeto de estrutura de classes em UML em si do pacote *Core* do *Framework PON C++ 2.0*, apresentado na Figura 74, segue pertinente. Assim, a implementação *Framework PON C++ 4.0* naturalmente apresenta uma modelagem inspirada no modelo de estrutura de classes similar ao do *Framework PON C++ 2.0*, porém naturalmente faz algumas simplificações, visando aumentar a facilidade de uso do código decorrente.

O diagrama de classes para modelagem do *Framework PON C++ 4.0*, por sua vez, é apresentado na Figura 75. Nesse contexto, podem ser destacadas as principais diferenças entre as duas versões, sendo a principal diferença a eliminação do pacote *Application*, detalhado anteriormente na Figura 33. Isto é viabilizado ao utilizar elementos mais desacoplados que não precisam de estruturas gerenciadoras acoplantes.

O *Framework PON C++ 4.0*, ao adotar uma estrutura mais genérica, elimina a necessidade da implementação de classes especializadas e potencialmente redundantes (*e.g.*, *MethodDerived*, *MethodPointer*, *RuleObject*, *RuleDerived* e *SubCondition*). No lugar da *SubCondition*, de modo a alcançar a mesma funcionalidade sem se tornar necessária uma especialização da classe, foi implementada uma agregação entre as próprias *Conditions*.

¹ O código completo do *Framework PON C++ 4.0* atualizado também pode ser consultado no servidor de artefatos do PON, uma vez que se tenha acesso a ele (via login e senha) em <https://nop.dainf.ct.utfpr.edu.br/nop-implementations/frameworks/nop-framework-cpp-4>.

² Existe compilador de NOPL para *Framework PON C++ 4.0* desenvolvido por Skora (2020).

Código 3.1 – Implementação do FBE Sensor em Framework PON C++ 4.0

```

/*
fbe Sensor
public boolean atIsRead = false
public boolean atIsActivated = false
private method mtProcess
    attribution
        this.atIsRead = true
        this.atIsActivated = false
    end_attribution
end_method
rule rlSensor
    condition
        premise prIsActivated
        this.atIsActivated == true
    end_premise
    and
        premise prIsNotRead
        this.atIsRead == false
    end_premise
end_condition
action sequential
    instigation sequential
        call this.mtProcess()
    end_instigation
end_action
end_rule
end_fbe
*/
struct NOPSensor: NOP::FBE {
    // Essa é a parte relativa a definição do FBE
    // Aqui se declaram e inicializam os Attributes
    NOP::SharedAttribute<bool> atIsRead{NOP::BuildAttribute(false)};
    NOP::SharedAttribute<bool> atIsActivated{NOP::BuildAttribute(false)};
    // Aqui se define o Method utilizando uma expressão lambda
    NOP::Method mtProcess{[&]()
        atIsRead->SetValue(true);
        atIsActivated->SetValue(false);
    };
    // Essa é a parte relativa a definição da Rule agregada ao FBE
    // com a definição das respectivas Premises, Condition, Action
    // e Instigation que a compõe
    NOP::SharedRule rlSensor{NOP::BuildRule(
        NOP::BuildCondition<NOP::Conjunction>(
            NOP::BuildPremise(atIsActivated, true, NOP::Equal()),
            NOP::BuildPremise(atIsRead, false, NOP::Equal())
        ),
        NOP::BuildAction(
            NOP::BuildInstigation(mtProcess)
        )
    );
};

```

Fonte: Autoria própria

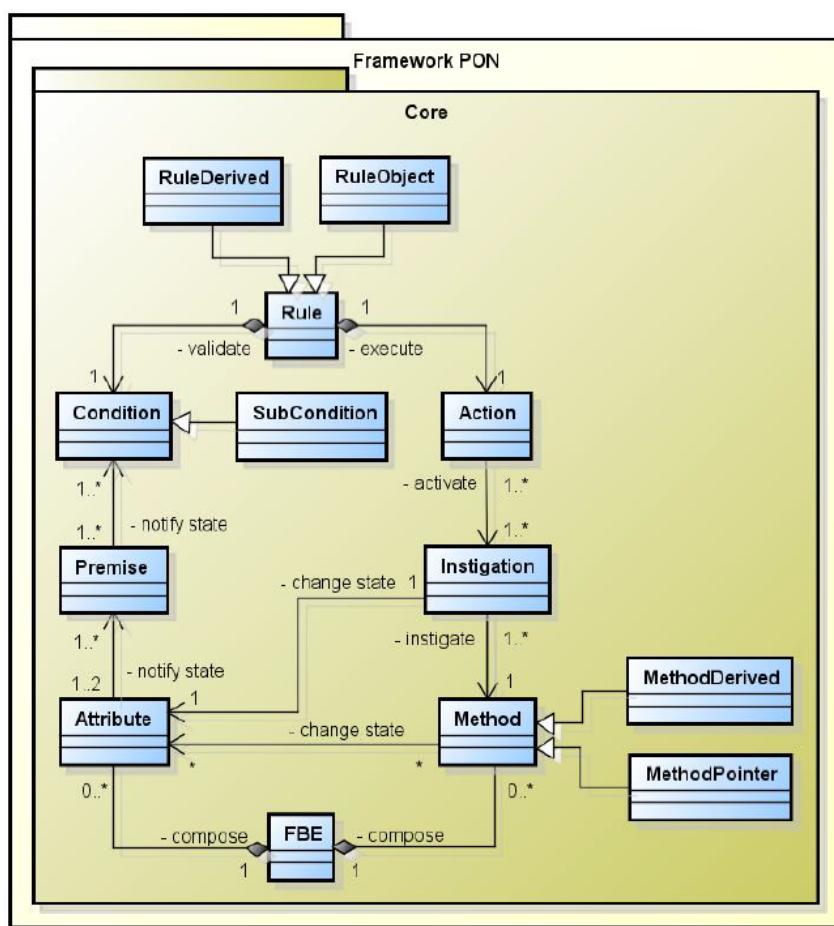


Figura 74 – Diagrama de classes do pacote *Core* do Framework PON C++ 2.0
 Fonte: Ronszcka (2012)

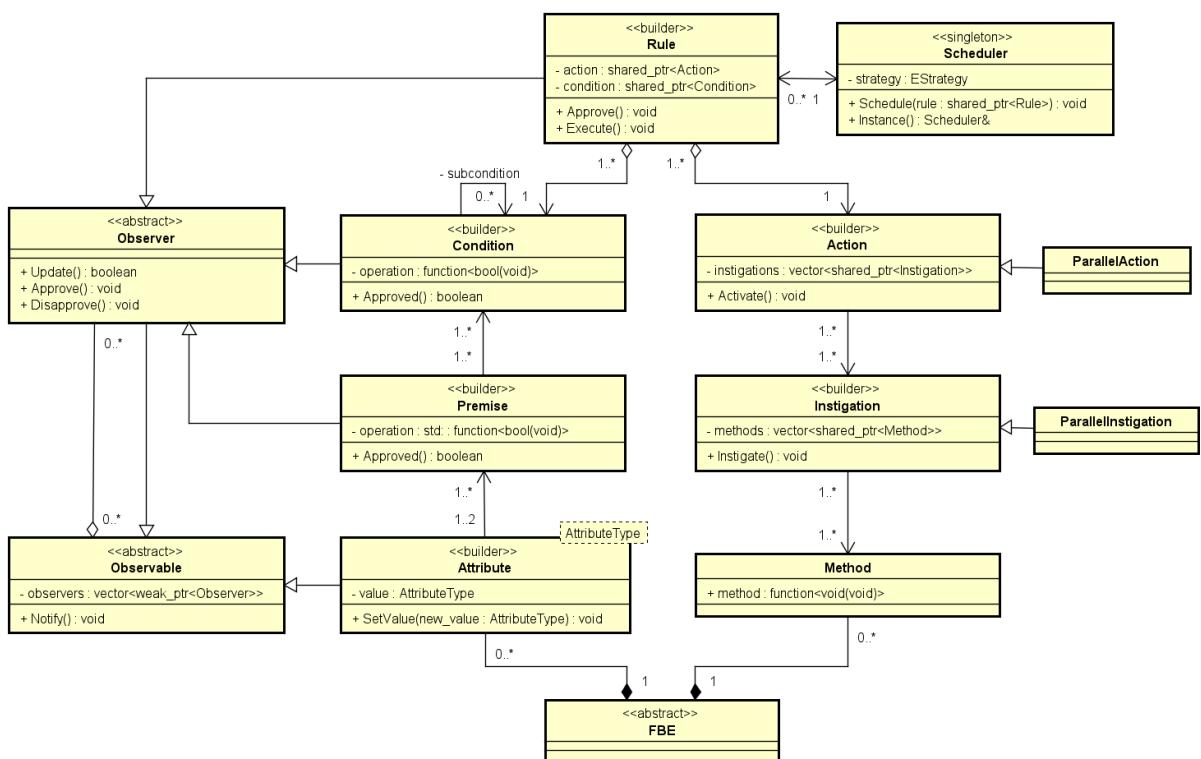


Figura 75 – Diagrama de classes do Framework PON C++ 4.0
Fonte: Autoria própria

A implementação também contempla a utilização de um escalonador de *Rules*, que pode ser utilizado para controlar o fluxo de execução das *Rules* aprovadas. Esse escalonador é materializado pela classe *Scheduler*, que usa o padrão de projeto *Singleton*, implementando as estratégias de escalonamento de *Rules* equivalentes às propostas por Banaszewski (2009) (*Breadth*, *Depth*, *Priority*, *NoOne*, detalhadas na Seção 2.3.6) e reaplicadas por Valença (2012) e Ronszcka (2012).

A classe *Premise*, apesar de se associar a *Attributes* com tipos *template*, não utiliza *template*. Isso se torna possível por essa associação ser realizada por meio da utilização de expressões *lambda*, que permitem abstrair o tipo dos *Attributes* sobre o qual a *Premise* opera, interessando a essa apenas o tipo de retorno da expressão de comparação dos *Attributes*, que no caso será sempre um tipo *bool*.

Por fim, do ponto de vista de padrão de projetos, aproveitam-se os conceitos introduzidos por Ronszcka (2012), sendo que no *Framework PON C++ 4.0* são utilizados os padrões *Observer*, *Iterator*, *Builder* e *Singleton*. A Tabela 9 sumariza onde cada padrão é aplicado no *framework*, enquanto detalhes sobre os padrões de projeto em si são discutidos ao longo do capítulo.

Dada esta introdução sobre a estrutura do *Framework PON C++ 4.0*, as seções seguintes se encarregam de detalhar a implementação de cada um de seus componentes, inclusive da implementação de cada uma das entidades do PON. Utiliza-se neste capítulo uma abordagem *bottom-up*, ou seja, parte-se da explicação do cerne do *framework*, dado pelo mecanismo de notificações, seguido das entidades principais, e por fim as abstrações e facilitadores de desenvolvimento.

Tabela 9 – Padrões de projeto aplicados no *Framework PON C++ 4.0*
Fonte: Autoria própria

Padrão de projeto	Aplicação
<i>Observer</i>	Aplicado para a implementação do mecanismo de notificações
<i>Iterator</i>	Utilizado por meio dos <i>iterators</i> da STL na estrutura de dados <i>std::vector</i>
<i>Builder</i>	Aplicado para a implementação dos construtores das entidades do PON
<i>Singleton</i>	Aplicado para a implementação do escalonador de <i>Rules</i>

3.2 MECANISMO DE NOTIFICAÇÕES

O mecanismo de notificações faz parte do cerne do *framework*, sendo responsável por possibilitar as interações entre as entidades do PON, que interagem entre si por meio de notificações pontuais. No *Framework PON C++ 4.0*, do mesmo modo que no *Framework PON*

C++ 2.0, optou-se por utilizar o padrão de projeto *Observer* para tal. O padrão de projeto *Observer* define uma relação de um para muitos entre objetos. Nesse padrão, na mudança de estado de um objeto, todos os objetos interessados nesta mudança são notificados (GAMMA *et al.*, 1995; RONSZCKA, 2012).

A implementação do padrão de projeto *Observer* no *Framework PON C++ 4.0* é dada por meio das classes *Observer* e *Observable*. Estas classes são utilizadas para materializar o mecanismo de notificações do PON, e implementadas de modo a facilitar a reutilização de código e remover a duplicidade de código no *framework*. Tanto a classe *Observer* como *Observable* são classes abstratas, servindo de modo a abstrair o processo de notificações das classes *Attribute*, *Premise*, *Condition* e *Rule*. Isto foi preservado a partir do *Framework PON C++ 2.0*, ajustando para o contexto do *Framework PON C++ 4.0*.

Em detalhe no Código 3.2, a classe *Observer* possui um método *Update* que permite a instância de classes derivadas notificar algum estado pertinente. O método *Update* é chamado por instância da classe *Observable* quando ela deseja notificar. Ainda na classe *Observer* há o método virtual *Approve* que pode ser especializado pelas classes para realizar tratamentos específicos em caso de aprovação, como o caso da *Rule* que deve executar a sua *Action*, enquanto para outras entidades (*i.e.*, *Attribute*, *Premise*, *Condition*) ele não precisa ser especializado.

Já a classe *Observable*, em destaque no Código 3.3, possui apenas 2 métodos, *Attach* que permite a instâncias adicionar um novo *Observer* a sua lista de entidades a serem notificadas, e *Notify* que permite a tais instâncias realizar as notificações de fato. Para o armazenamento das entidades é declarada a estrutura genérica *NOPContainer*, que pode ser especializada de modo a utilizar qualquer estrutura de dado sem necessitar de alterações no código. A classe *Observer* também deriva de *std::shared_from_this*, que é uma classe da STL utilizada para permitir de forma simples acessar um *shared_ptr* em funções membros da própria classe, de forma análoga ao uso de *raw pointers* que são acessíveis por meio do ponteiro *this*.

Código 3.2 – Implementação da classe *Observer* no Framework PON C++ 4.0

```
class Observer : public Observable
{
public:
    virtual bool Approved() = 0;
    template <NOPFlag Flag = Default>
    bool Update();
    virtual void Approve();
    virtual void Disapprove();

protected:
    bool approved_{false};
};
```

Fonte: Autoria própria**Código 3.3 – Implementação da classe *Observable* no Framework PON C++ 4.0**

```
class Observable
{
public:
    void Attach(const std::shared_ptr<class Observer>& observer);
    template <NOPFlag Flag = Default>
    void Notify();

private:
    NOPContainer<std::weak_ptr<class Observer>> observers_;
};
```

Fonte: Autoria própria

No *Framework PON C++ 4.0* a estrutura *NOPContainer* utiliza *std::vector* da STL, não sendo implementada uma estrutura de dados própria, simplesmente definindo o template conforme apresentado no Código 3.4. No *Framework PON C++ 4.0* optou-se pela utilização apenas do tipo *std::vector* ao invés da criação de estruturas de dados otimizadas, como *NOPLIST*, *NOPVECTOR* e *NOPHASH* do *Framework PON C++ 2.0*, pois *std::vector* pode ser facilmente utilizada com a aplicação de algoritmos paralelizados do C++.

Código 3.4 – Definição de *NOPContainer* no Framework PON C++ 4.0

```
template <typename T>
using NOPContainer = std::vector<T>;
```

Fonte: Autoria própria

Ainda assim, *NOPContainer* pode ser implementado como qualquer estrutura de dados compatível com as interfaces de uso da STL com *std::vector*, com a capacidade de inserção e remoção de elementos, assim como a navegação por iteradores. Entretanto, tal implementação adiciona significante grau de complexidade, e potencialmente vulnerabilidades no contexto da aplicação paralelizada. Nesse sentido, há uma troca de potencial ganho de desempenho por maior garantia de estabilidade.

O processo de notificação em si acontece por meio da chamada do método *Notify*

nas instâncias, que utiliza *iterators* para navegar entre todos os elementos do *NOPContainer*, realizando a chamada do método *Update* da respectiva instância de *Observer*. Por meio do uso de *templates* e expressões constantes, os diferentes métodos de notificação podem ser implementados nessa mesma função sem acarretar custos suplementares de desempenho em tempo de execução. Esse mecanismo é mostrado no Código 3.5. Nesse código é interessante observar o uso extensivo de expressões constantes, o que traz grandes benefícios em redução de tempo de execução ao eliminar a execução de instruções condicionais, pelo fato de ser um método muito utilizado, por meio da utilização de *if constexpr* com parâmetros de *template*.

Código 3.5 – Método *Notify* da classe *Observable* no Framework PON C++ 4.0

```
template <NOPFlag Flag>
inline void Observable::Notify()
{
    // Define expressão lambda para notificar entidades
    auto notify_entities = [] (auto& observer)
    {
        bool ret{false};
        if (const auto& observer_ptr = observer.lock(); observer_ptr)
        {
            ret = observer_ptr->template Update<Flag>();
        }
        return ret;
    };

    if constexpr (0 < (NoNotify & Flag))
    {
        // Não notifica
    }
    else if constexpr (0 < (Parallel & Flag))
    {
        if constexpr (0 < (Exclusive & Flag))
        {
            // Notifica todas as entidades com execução de forma paralelizada
            // até o primeiro update retornar TRUE quando a entidade é aprovada
            const bool ret =
                std::none_of(std::execution::par_unseq, observers_.begin(),
                            observers_.end(), notify_entities);
        }
        else
        {
            // Notifica todas as entidades com execução
            // de forma paralelizada
            std::for_each(std::execution::par_unseq, observers_.begin(),
                         observers_.end(), notify_entities);
        }
    }
    else if constexpr (0 < (Exclusive & Flag))
    {
        // Notifica todas as entidades de forma sequencial até
        // o primeiro update retornar TRUE quando a entidade é aprovada
        const bool ret =
            std::none_of(observers_.begin(), observers_.end(), notify_entities);
    }
    else
    {
        // Caso padrão
        // Notifica todas as entidades de forma sequencial
        std::for_each(observers_.begin(), observers_.end(), notify_entities);
    }
}
```

Fonte: Autoria própria

O método *Update* utiliza o parâmetro *Parallel* para determinar a execução do mecanismo

de notificações de forma paralelizada. Com isso, todas as notificações geradas por determinada entidade são realizadas de forma paralelizada. Cabe aqui detalhar que a paralelização é alcançada por meio da aplicação de políticas de execução, evidentes pelo uso de `std::execution::par_unseq`. Ainda, é importante reforçar que com o uso da política de execução, o balanceamento de carga fica a cargo da implementação do compilador (O'NEAL, 2018).

Também pode ser observado no Código 3.2 que a classe *Observer* é derivada da classe *Observable*. Isto é feito de modo a permitir a chamada do método *Notify* com templates, conforme mostrado no Código 3.6. Desta forma, todo *Observer* também é um *Observable*. Neste caso específico foi necessário *Observer* derivar da classe *Observable*, ao invés de serem classes distintas derivadas de uma classe base, pois em C++ não é permitida a declaração de métodos virtuais com *templates*.

Código 3.6 – Detalhes de implementação do método *Update* na classe *Observer* do Framework PON C++ 4.0

```
template <NOPFlag Flag>
bool Observer::Update()
{
    const bool new_state = Approved();
    const bool changed = new_state != approved_;

    if (new_state)
    {
        Approve();
        LOG(this, "Approved");
    }
    else
    {
        Disapprove();
        LOG(this, "Disapproved");
    }

    if constexpr (0 < (ReNotify & Flag))
    {
        Notify<Flag>();
    }
    else if (changed)
    {
        Notify<Flag>();
    }
    return approved_;
}
```

Fonte: Autoria própria

Por fim, cabe destacar também que, conforme pode ser observado no diagrama de classes da Figura 75, as classes *Action*, *Instigation* e *Method* não herdam diretamente das classes *Observer* ou *Observable*, devido à menor complexidade da notificação entre estas entidades, que permite uma implementação mais simples. Este mecanismo é mais bem detalhado na Seção 3.7.

3.3 ATTRIBUTE

A implementação da classe *Attribute* deriva apenas da classe *Observable*. Essa classe é implementada por meio do uso de *templates*, de modo a permitir armazenar valores (no atributo *value_*). Esses detalhes de implementação são mostrados no trecho de Código 3.7. A função *SetValue*, por meio do uso de *templates*, além de atribuir o valor do *Attribute*, recebe um parâmetro que permite controlar o tipo de notificação gerada, conforme apresentado no Código 3.5.

Código 3.7 – Detalhes de implementação do *Attribute* no Framework PON C++ 4.0

```
template <typename T>
class Attribute final : public Observable
{
public:
    Attribute() = default;
    explicit Attribute(const T value);

    template <NOPFlag Flag = Default>
    void SetValue(T value);
    T value_;
};
```

Fonte: Autoria própria

Desta forma, o tipo do *template* especificado em *T* pode ser qualquer tipo especificado pelo desenvolvedor, seja este um dos tipos básicos (*e.g.*, *int*, *bool*, *float*, *string*, etc) ou até mesmo classes definidas pelo desenvolvedor. Nesse sentido, a utilização deste *template* contribui de forma a possibilitar a flexibilidade de tipos no *framework*.

Como exemplo de construção de *Attributes* são utilizados os *Attributes* da aplicação do sensor *atIsRead* e *atIsActivated*, conforme apresentado no Código 3.8. O código equivalente em NOPL é apresentado como referência em comentários no código. A utilização da função *BuildAttribute* é explicada em maiores detalhes na Seção 3.9.2.

Código 3.8 – Criação de *Attributes* no Framework PON C++ 4.0

```
// public boolean atIsRead = false
NOP::SharedAttribute<bool> atIsRead{NOP::BuildAttribute(false)};

// public boolean atIsActivated = false
NOP::SharedAttribute<bool> atIsActivated{NOP::BuildAttribute(false)};
```

Fonte: Autoria própria

3.4 PREMISE

A classe *Premise* deriva da classe *Observer*, pois ela precisa ser notificada pelos *Attributes* e notificar as *Conditions*. No Código 3.9 pode ser observado que operação realizada pela *Premise* é definida na sua construção, e é armazenada no atributo *operation_*, que utiliza o tipo *std::function<bool(T, T)>*. A utilização do tipo *std::function<bool(T, T)>* permite armazenar qualquer função que receba dois parâmetros e retorne um tipo *bool*, que enfim possa ser utilizada como operação aplicada para determinar o estado da *Premise*.

Outro detalhe importante é a sobrecarga do operador *bool()* da *Premise*, que permite a composição de operações booleanas entre *Premises* de forma extremamente fácil, o que será mais explorado na implementação da *Condition*.

Código 3.9 – Detalhes de implementação da *Premise* no Framework PON C++ 4.0

```
class Premise final : public Observer
{
public:
    Premise(std::function<bool()> op);
    bool Approved() override;

    // Overload operator for boolean logic
    explicit operator bool() const { return approved_; }

private:
    std::function<bool()> operation_;
};
```

Fonte: Autoria própria

Apesar de a classe *Attribute* ser implementada com *templates*, foi possível eliminar essa dependência de *templates* na classe *Premise* ao abstrair a informação dos tipos dos *Attributes* com a utilização de expressões *lambda*, de forma que a classe *Premise* não precisa armazenar a informação do tipo do *template* dos seus *Attributes*.

Como exemplo de construção de *Premises* são utilizados as *Premises* da aplicação do sensor *prIsActivated* e *prIsNotRead*, conforme apresentado no Código 3.10. O código equivalente em NOPL é apresentado como referência em comentários no código. A utilização da função *BuildPremise* é explicada em maiores detalhes na Seção 3.9.2.

3.5 CONDITION

A implementação da *Condition* é de certa forma muito parecida com a da *Premise*. No trecho de Código 3.11, podem ser observadas as semelhanças com a *Premise*, porém alguns detalhes podem ser destacados, como o atributo *std::function<bool(void)> operation_*, que nesse

Código 3.10 – Criação de *Premises* no *Framework PON C++ 4.0*

```

/* Baseado no trecho em NOPL:
premise prIsActivated
    this.atIsActivated == true
end_premise
*/
NOP::SharedPremise prIsActivated{NOP::BuildPremise(atIsActivated, true, NOP::Equal())};

/*
premise prIsNotRead
    this.atIsRead == false
end_premise
*/
NOP::SharedPremise prIsNotRead{NOP::BuildPremise(atIsRead, false, NOP::Equal())};

```

Fonte: Autoria própria

caso permite a passagem de qualquer função que retorne um valor *bool* para a avaliação do estado da *Condition*. Esse recurso aparentemente simples provê grande flexibilidade algorítmica quando utilizado em conjunto com expressões *lambda*, que se aproveitando do operador *bool()* sobrecarregado (tanto de *Premises* como *Conditions*), permite a composição de *Conditions* baseada em expressões booleanas de forma simples.

Código 3.11 – Detalhes de implementação da *Condition* no *Framework PON C++ 4.0*

```

class Condition final : public Observer
{
    public:
        Condition() = default;
        explicit Condition(std::function<bool(void)> operation);
        bool Approved() override;

        // Overload operator for boolean logic
        explicit operator bool() const { return approved_; }

    private:
        std::function<bool(void)> operation_;
};

```

Fonte: Autoria própria

Como exemplo de construção de *Condition* é utilizada a *Condition* da aplicação do sensor, conforme apresentado no Código 3.12. O código equivalente em NOPL é apresentado como referência em comentários no código. A utilização da função *BuildCondition* é explicada em maiores detalhes na Seção 3.9.2.

Outro ponto interessante nessa implementação de *Condition* de forma genérica é que ela pode ser usada também da mesma forma que uma *SubCondition* do *Framework PON C++ 2.0*, sem a necessidade de uso de uma classe específica para isso, podendo inclusive ser utilizada na composição da operação da mesma forma que as *Premises*. De forma similar, a *Rule* pode ser utilizada como *Master Rule*, pois a *Condition* genérica pode ser composta por *Rules*. Esta utilização de *Master Rule* é demonstrada no Código 3.13, no qual a *Condition cn2* é composta

Código 3.12 – Criação de *Conditions* no Framework PON C++ 4.0

```

/*
condition
    premise prIsActivated
        this.atIsActivated == true
    end_premise
    and
    premise prIsNotRead
        this.atIsRead == false
    end_premise
end_condition
*/
NOP::SharedCondition cnRule = NOP::BuildCondition<NOP::Conjunction>(
    NOP::BuildPremise(atIsActivated, true, NOP::Equal()),
    NOP::BuildPremise(atIsRead, false, NOP::Equal())
);

```

Fonte: Autoria própria

pela *Premise* *pr2* e uma *Master Rule* *masterRule1*.

Código 3.13 – Utilização de *Master Rule* em *Conditions* no Framework PON C++ 4.0

```

/*
rule masterRule1
    condition cn1
        pr1
    end_condition
    ac1
end_rule
/*
NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Single>(pr1);
NOP::SharedRule masterRule1 = NOP::BuildRule(cn1, ac1)

/*
rule normalRule1
    condition cn2
        masterRule1
        and
        pr2
    end_condition
    ac2
end_rule
*/
NOP::SharedCondition cn2 = NOP::BuildCondition<NOP::Conjunction>(masterRule1, pr2);
NOP::SharedCondition normalRule1 = NOP::BuildRule(cn2, ac2);

```

Fonte: Autoria própria

Por fim, também é possível a construção de *Conditions* com composição de *Premises*, introduzindo o conceito de *Condition* flexível, que pode ser definido como a capacidade de se criar *Conditions* baseadas em expressões *booleanas* entre *Premises*. Tome-se o exemplo com 3 *Premises* (*pr1*, *pr2* e *pr3*), aplicadas em uma *Condition* *cnFlex* de acordo com a expressão booleana *pr1 and pr2 or (not pr1 and pr3)*, de modo que a operação da *Condition* pode ser declarada utilizando uma expressão *lambda*, conforme apresentado no Código 3.14.

Código 3.14 – Condition flexível no Framework PON C++ 4.0

```
NOP::SharedCondition cnFlex = NOP::BuildCondition(CONDITION(
    *pr1 && *pr2 || (!*pr1 && *pr3), // Declaração da expressão booleana
    pr1, pr2, pr3)); // Declaração explícita das Premises notificantes
```

Fonte: Autoria própria

3.6 RULE

A classe *Rule* também deriva da classe *Observer* e, de forma similar à *Condition*, possui sobrecarga do operador *bool()*, o que permite que também seja utilizada para a composição de uma *Condition*, tendo funcionalidade equivalente ao conceito dado *Master Rule* do Framework PON C++ 2.0 (RONSZCKA, 2012). A estrutura da *Rule* é apresentada no Código 3.15.

Código 3.15 – Detalhe de implementação da classe Rule no Framework PON C++ 4.0

```
class Rule final : public Observer, public std::enable_shared_from_this<Rule>
{
public:
    explicit Rule(const int priority = 0);
    bool Approved() override;
    void SetCondition(std::shared_ptr<Condition> condition);
    void SetAction(std::shared_ptr<Action> action);
    void Execute() const;
    void Approve() override;

    // Overload operator for boolean logic
    explicit operator bool() const { return approved_; }

    int priority_;

private:
    std::shared_ptr<Condition> condition_;
    std::shared_ptr<Action> action_;
    std::recursive_mutex mutex_;
};
```

Fonte: Autoria própria

A *Rule* também apresenta o membro *priority_*, sendo utilizada a aplicação do mecanismo escalonador. O método *Approve*, apresentado no Código 3.16, é implementado de forma a permitir a utilização com ou sem o escalonador de *Rules*. Essa implementação é interessante, pois permite a execução com comportamento equivalente à estratégia *NO_ONE* do Framework PON C++ 2.0, porém sem exigir a utilização do escalonador que aumenta o tempo de execução das aplicações.

A classe *Rule* também deriva da classe *std::shared_from_this*, que disponibiliza o método *shared_from_this()*, utilizado para permitir acessar um *shared_ptr* da classe em seus métodos, o que facilita a passagem desse parâmetro para o escalonador.

Como exemplo de construção de *Rule* é utilizada a *Rule* da aplicação do sensor *rl-Sensor*, conforme apresentado no Código 3.17. O código equivalente em NOPL é apresentado como referência em comentários no código. A utilização das funções *BuildRule*, *BuildAction* e

Código 3.16 – Detalhe de implementação do método *Approve* da classe *Rule* no Framework PON C++ 4.0

```
inline void Rule::Approve()
{
    Observer::Approve();
    if constexpr (USE_SCHEDULER)
    {
        Scheduler::Instance().Schedule(shared_from_this());
    }
    else
    {
        Execute();
    }
}
```

Fonte: Autoria própria

BuildInstigation são explicadas em maiores detalhes na Seção 3.9.2.

Código 3.17 – Criação de *Rules* no Framework PON C++ 4.0

```
/*
rule rlSensor
    condition
        premise prIsActivated
            this.atIsActivated == true
        end_premise
        and
        premise prIsNotRead
            this.atIsRead == false
        end_premise
    end_condition
    action sequential
        instigation sequential
            call this.mtProcess()
        end_instigation
    end_action
end_rule
*/
```



```
NOP::SharedRule rlSensor{NOP::BuildRule(
    NOP::BuildCondition<NOP::Conjunction>(
        NOP::BuildPremise(atIsActivated, true, NOP::Equal()),
        NOP::BuildPremise(atIsRead, false, NOP::Equal())
    ),
    NOP::BuildAction(
        NOP::BuildInstigation(mtProcess)
    )
);
};
```

Fonte: Autoria própria

3.7 ACTION, INSTIGATION E METHOD

As entidades *Action* e *Instigation* são implementadas de maneira bastante simples. A *Action* guarda os ponteiros para as *Instigations*, utilizando a estrutura definida por *NOPContainer*, enquanto a *Instigation* guarda referências para os *Methods*. A *Action* é ativada quando a sua *Rule* é aprovada, instigando as *Instigations* em sua lista. A *Instigation*, por sua vez, quando instigada executa os *Methods* na sua lista. A implementação das classes *Action* e *Instigation*, assim

como suas classes derivadas paralelizadas *ParallelAction* e *ParallelInstigation* são mostradas, respectivamente, no Código 3.18 e Código 3.19.

Código 3.18 – Implementação da classe *Action* no Framework PON C++ 4.0

```
class Action
{
public:
    void AddInstigation(std::shared_ptr<Instigation> instigation);
    virtual void Activate();

protected:
    NOPContainer<std::shared_ptr<Instigation>> instigations_;
};

class ParallelAction : public Action
{
public:
    void Activate() override;
};
```

Fonte: Autoria própria

Código 3.19 – Implementação da classe *Instigation* no Framework PON C++ 4.0

```
class Instigation
{
public:
    void AddMethod(Method method);
    virtual void Instigate();

protected:
    NOPContainer<Method> methods_;
};

class ParallelInstigation final : public Instigation
{
    void Instigate() override;
};
```

Fonte: Autoria própria

O *Method*, ao seu turno, é implementado por meio de *std::function*, assim como as operações da *Premise* e *Condition*, conforme mostrado no código 3.20. Neste ponto ele apresenta a mesma vantagem de permitir o uso de expressões *lambda* para prover grande flexibilidade ao *Method*, sendo possível criar de forma fácil *Methods* que realizam operações complexas, como fazer múltiplas atribuições de valores a *Attributes* e variáveis ou chamada de outras funções e métodos.

Como exemplo de construção de *Method* é utilizado o *Method* da aplicação do sensor *mtProcess*, que realiza a atribuição de valores a dois *Attributes*, conforme apresentado no Código 3.21. O código equivalente em NOPL é apresentado como referência em comentários no código. O uso do *Method* como *std::function<void(void)>* permite ser utilizado com qualquer função que não tenha parâmetros nem retorne nenhum valor.

Código 3.20 – Definição de Method no Framework PON C++ 4.0

```
using Method = std::function<void(void)>;
```

Fonte: Autoria própria

Código 3.21 – Uso de Method no Framework PON C++ 4.0

```
/*
private method mtProcess
    attribution
        this.atIsRead = true
        this.atIsActivated = false
    end_attribution
end_method
*/
NOP::Method mtProcess{ [&] () (
    atIsRead->SetValue(true);
    atIsActivated->SetValue(false);
)};
```

Fonte: Autoria própria

As implementações paralelizadas diferem apenas no sentido que recorrem às políticas de execução paralelas (*std::execution::par_unseq*), para a iteração entre as entidades a serem notificadas. Isso é feito de forma similar tanto nas classes *Instigation* e *ParallelInstigation*, mostrado no Código 3.22, como nas classes *Action* e *ParallelAction*, mostrado no código 3.23.

Código 3.22 – Detalhe de implementação de paralelização da *Instigation* no Framework PON C++ 4.0

```
void Instigation::Instigate()
{
    // Executa todos os métodos de forma sequencial
    for (const auto& method : methods_)
    {
        method();
    }
}

void ParallelInstigation::Instigate()
{
    // Executa todos os métodos com execução paralelizada
    std::for_each(std::execution::par_unseq,
        methods_.begin(), methods_.end(),
        [] (auto& method) { method(); });
}
```

Fonte: Autoria própria

Cabe aqui detalhar que a paralelização é alcançada por meio da aplicação de políticas de execução, evidentes pelo uso de *std::execution::par_unseq*. O *loop* de iteração entre as entidades, quando executado com esta política de execução, é realizado de forma paralela, ao contrário do *for* tradicional que executa de forma sequencial.

Código 3.23 – Detalhe de implementação de paralelização da Action no Framework PON C++ 4.0

```

void Action::Activate()
{
    // Instiga todas as intigações de forma sequencial
    for (const auto& instigation : instigations_)
    {
        instigation->Instigate();
    }
}

void ParallelAction::Activate()
{
    // Instiga todas as intigações com execução paralelizada
    std::for_each(std::execution::par_unseq,
                 instigations_.begin(), instigations_.end(),
                 [] (auto& instigation) {
                     instigation->Instigate();
                 });
}

```

Fonte: Autoria própria

3.8 ESCALONADOR

O escalonador é implementado de forma a funcionar como mecanismo de resolução de conflitos nesta implementação do PON. A implementação do escalonador no *Framework PON C++ 4.0* é dada por meio da classe *Scheduler*, de forma análoga à classe *SingletonScheduler* do *Framework PON C++ 2.0*. Do mesmo modo, é utilizado o padrão *singleton*, que garante a existência de uma única entidade da classe em cada instância de execução da aplicação. Esta estrutura é apresentada no Código 3.24.

As *Rules* são enfileiradas para execução ao serem aprovadas, por meio da chamada do método *Schedule* do escalonador, passando como parâmetro um ponteiro para a *Rule*, sendo inserido na fila de execução. O escalonador apresenta quatro diferentes estratégias de resolução de conflitos, que se baseiam na ordem na qual as *Rules* são inseridas na fila.

Código 3.24 – Implementação da classe *Scheduler* no Framework PON C++ 4.0

```

enum EStrategy
{
    None,
    FIFO,
    LIFO,
    Priority
};

class Scheduler
{
    public:
        explicit Scheduler(const EStrategy strategy = FIFO);
        ~Scheduler();
        Scheduler(Scheduler&) = delete;
        Scheduler& operator=(Scheduler) = delete;

        [[nodiscard]] static Scheduler& Instance();
        void Schedule(const std::shared_ptr<class Rule>& rule);
        void Run();
        void FinishAll();
        void SetStrategy(EStrategy strategy, bool ignore_conflict = false);

    private:
        std::thread scheduler_thread_;
        EStrategy strategy_{FIFO};
        bool ignore_conflict_{false};

        std::deque<std::weak_ptr<class Rule>> rules_;

        std::mutex queue_mutex_;
        std::atomic_bool finished_{false};
        std::atomic_bool executing_{false};
};


```

Fonte: Autoria própria

A maneira como o escalonador executa as *Rules* em cada estratégia é dada da seguinte forma:

- *None*: executa as *Rules* na mesma ordem em que foram inseridas, do mesmo modo que a estratégia *FIFO*, porém ignora os conflitos.
- *FIFO (First-In-First-Out)*: executa as *Rules* na mesma ordem em que foram inseridas.
- *LIFO (Last-In-First-Out)*: executa as *Rules* na ordem inversa que foram inseridas, ou seja, executa primeiro a *Rule* que foi inserida por último.
- *Priority*: executa as *Rules* de acordo com sua prioridade, dada pelo membro *priority_-* presente na classe *Rule*.

Nesse contexto, o mecanismo de escalonamento e execução das *Rules* é mostrado no Código 3.25. Neste código é possível observar a diferença no comportamento para as diferentes estratégias de escalonamento.

Código 3.25 – Método *Run* do *Scheduler* no Framework PON C++ 4.0

```

void Scheduler::Run()
{
    while (!finished_)
    {
        std::weak_ptr<Rule> current_rule;
        {
            std::lock_guard<std::mutex> lock(queue_mutex_);
            if (!rules_.empty())
            {
                std::deque<std::weak_ptr<Rule>>::iterator current_rule_it;
                if (FIFO == strategy_)
                {
                    current_rule_it = rules_.begin();
                }
                else if (LIFO == strategy_)
                {
                    current_rule_it = --rules_.end();
                }
                else if (Priority == strategy_)
                {
                    current_rule_it = std::max_element(
                        rules_.begin(), rules_.end(), [&] (auto r11, auto r12) {
                            auto lock1 = r11.lock();
                            auto lock2 = r12.lock();
                            return lock1->priority_ < lock2->priority_;
                        });
                }
                else
                {
                    current_rule_it = rules_.begin();
                }
                current_rule = *current_rule_it;
                rules_.erase(current_rule_it);
                executing_ = true;
            }
        }

        if (const auto& rule_lock = current_rule.lock(); rule_lock)
        {
            if (rule_lock->Approved() || ignore_conflict_ || (None == strategy_))
            {
                LOG(rule_lock.get(), "Executed by scheduler");
                rule_lock->Execute();
            }
            else
            {
                LOG(rule_lock.get(), "Discarded by scheduler due to conflict");
            }
            executing_ = false;
        }
    }
}

```

Fonte: Autoria própria

Também, de modo a desacoplar a execução do escalonador do resto do mecanismo de notificações, todo o processo de escalonamento das *Rules* é executado em uma *thread* separada, declarada como *scheduler_thread_*. Esta *thread* é inicializada no momento em que a instância do singleton *Scheduler* é criada, conforme mostrado no Código 3.26.

Código 3.26 – Inicialização da *thread* do *Scheduler* no Framework PON C++ 4.0

```

Scheduler::Scheduler(const EStrategy strategy) : strategy_{strategy}
{
    scheduler_thread_ = std::thread([&]() { Run(); });
}

```

Fonte: Autoria própria

Ainda, de modo a realizar a efetiva resolução de conflito, mesmo em ambientes com execução *multithread*, apenas uma *Rule* é executada de cada vez, pois dessa forma é possível verificar se a *Rule* ainda está aprovada no momento da sua execução. A verificação da aprovação da *Rule* é feita por meio do método *Approved*, que atualiza o estado de aprovação da *Rule*. É possível ignorar essa resolução de conflito ao utilizar a estratégia *None*. Esse comportamento é mostrado no Código 3.27.

O escalonador apresenta uma única *thread* de execução, não sendo criadas threads exclusivas para a execução das *Rules* pelo próprio escalonador. Nesse sentido, o paralelismo em ambientes *multithread* é alcançado por meio do uso das entidades parallelizadas (*i.e.*, *ParallelAction* e *ParallelInstigation*), assim como da declaração de *Methods* assíncronos. A criação de *Methods* assíncronos é detalhada na Seção 3.9.1.

Código 3.27 – Detalhe do mecanismo de resolução de conflito da classe *Scheduler* no Framework PON C++ 4.0

```
void Scheduler::Run()
{
    ...
    if (const auto& rule_lock = current_rule.lock(); rule_lock)
    {
        // Verifica se a Rule ainda está aprovada
        // Se a Rule não está mais aprovada significa
        // que houve um conflito
        if (rule_lock->Approved() || ignore_conflict_ || (None == strategy_))
        {
            LOG(rule_lock.get(), "Executed by scheduler");
            rule_lock->Execute();
        }
        else
        {
            LOG(rule_lock.get(), "Discarded by scheduler due to conflict");
        }
    }
    ...
}
```

Fonte: Autoria própria

3.9 FACILITADORES DE DESENVOLVIMENTO

Com o objetivo de abstrair conhecimentos específicos de implementação e detalhes técnicos complexo do Framework PON C++ 4.0, de modo a facilitar o uso e reduzir a curva de aprendizado são adotadas algumas estratégias, como o uso de funções *builder* e a definição de *macros*, sendo detalhadas nas seções seguintes.

3.9.1 Abstrações

Um artifício utilizado para reduzir a complexidade exposta ao desenvolvedor, no sentido de abstrair o uso de *smart pointer*, é a definição de tipos especializados para os *smart pointer* de cada entidade, conforme mostrado no Código 3.28, de modo que o desenvolvedor não precisa conhecer a sintaxe dos *smart pointers* para utilizá-los.

Código 3.28 – Abstração de *smart pointers* do Framework PON C++ 4.0

```
template <typename T>
using SharedAttribute = std::shared_ptr<Attribute<T>>;
using SharedPremise = std::shared_ptr<Premise>;
using SharedCondition = std::shared_ptr<Condition>;
using SharedRule = std::shared_ptr<Rule>;
using SharedInstigation = std::shared_ptr<Instigation>;
using SharedAction = std::shared_ptr<Action>;
```

Fonte: Autoria própria

Outra abstração utilizada é por meio do uso de *macros*, neste caso para esconder o uso de expressões *lambda*, que são um conceito bastante novo e podem confundir desenvolvedores inexperientes. No Código 3.29 é mostrada a macro que define um modelo padrão de expressão *lambda* para ser utilizado na construção de *Condition*. Neste código é criada a mesma *Condition* do exemplo de sensores, porém utilizando este formato de construção mais flexível.

Código 3.29 – Uso de Macros para abstrair o uso de expressões *lambda* no Framework PON C++ 4.0

```
#define CONDITION(expression) [&] () { return bool(expression); }

/*
condition
premise prIsActivated
    this.atIsActivated == true
end_premise
and
premise prIsNotRead
    this.atIsRead == false
end_premise
end_condition
*/
NOP::SharedPremise prIsActivated{NOP::BuildPremise(atIsActivated, true, NOP::Equal())};
NOP::SharedPremise prIsNotRead{NOP::BuildPremise(atIsRead, false, NOP::Equal())};

NOP::SharedCondition cnRule = NOP::BuildCondition(CONDITION(prIsActivated && prIsNotRead);
                                                    prIsActivated, prIsNotRead);
```

Fonte: Autoria própria

De forma similar o Código 3.30 apresenta o uso das *macros* para a criação de *Methods*. Em especial, uma macro é definida para a criação de *Methods* com execução assíncrona. O *Method* possui a estrutura genérica, de modo que aceita qualquer função, porém essas *macros* são disponibilizadas de forma a facilitar a utilização nos casos de uso comuns previstos. No código é

exemplificado seu uso para criação do *Method* *mtProcess* do exemplo de sensores, tanto na sua execução sequencial, como na execução assíncrona paralelizada, dada por *mtProcessAsync*.

Código 3.30 – Uso de Macros para criação de Methods no Framework PON C++ 4.0

```
#define METHOD(expression) [&]() { expression }
#define ASYNC_METHOD(expression) METHOD(run_async_method(METHOD(expression));)

/*
private method mtProcess
    attribution
        this.atIsRead = true
        this.atIsActivated = false
    end_attribution
end_method
*/
NOP::Method mtProcess(METHOD(
    atIsRead->SetValue(true);
    atIsActivated->SetValue(false);
));
NOP::Method mtProcessAsync(METHOD(
    atIsRead->SetValue(true);atIsActivated->SetValue(false);
));
```

Fonte: Autoria própria

Isto dito, o mecanismo de execução assíncrona pode ser implementado com a utilização de *threads*. Conforme mostrado no Código 3.30, a macro *ASYNC_METHOD* cria e executa o *Method* em uma *thread* isolada. Essa construção permite a declaração de código paralelizado de forma simples. Além disso, essa construção é apenas uma possibilidade de execução paralelizada, porque devido à flexibilidade do uso de expressões *lambda*, o desenvolvedor é livre para utilizar outros modos de execução paralela conforme sua necessidade.

3.9.2 Builders de entidades compartilhadas

A inicialização de objetos das classes *Attribute*, *Premise*, *Condition*, *Rule*, *Action* e *Instigation* utiliza as chamadas funções *builder*. O *builder* é um padrão de projeto, que neste contexto é aplicado de forma a abstrair a construção dos objetos desejados, por meio de uma função que recebe parâmetros e retorna um *shared_ptr* para o objeto instanciado. O padrão *builder* é materializado por meio de uma função.

O padrão de projeto *Builder* no *Framework PON C++ 4.0* desempenha uma função semelhante ao padrão *Factory* aplicado no *Framework PON C++ 2.0*, cumprindo o propósito de facilitar a criação de entidades. A principal diferença sendo que o padrão *Builder* não requer a implementação de classes *Factory* complexas, e com a utilização de recursos avançados de C++ moderno, como *variadic templates*, pode ser implementado como funções simples.

Os códigos 3.31 a 3.36 mostram os *builders* de cada uma dessas classes mencionadas.

Nos *builders* pode ser destacado o uso das funções *std::make_shared*, sendo utilizada para criar a instância de *shared_ptr*, e a função *std::move*, sendo utilizada para transferir de forma otimizada as instâncias de *shared_ptr* sem o incremento dos contadores de referências internos.

O Código 3.31 é utilizado para a construção de *Attributes*, recebendo como parâmetro apenas o valor inicial do *Attribute*, como em *NOP::BuildAttribute<bool>(false)*.

Código 3.31 – *Builder de Attribute do Framework PON C++ 4.0*

```
template <typename T>
auto BuildAttribute(const T value)
{
    return std::make_shared<Attribute<T>>(value);
}
```

Fonte: Autoria própria

O Código 3.32 apresenta os dois *builders* utilizados para a construção de *Premises*, recebendo como parâmetro um par de *Attributes*, ou também um *Attribute* e um valor, assim como a operação da *Premise*, como em *NOP::BuildPremise<bool>(at1, true, NOP::Equal())*.

Código 3.32 – *Builder de Premise do Framework PON C++ 4.0*

```
template <typename T, typename Func>
auto BuildPremise(std::shared_ptr<Attribute<T>> lhs,
                  std::shared_ptr<Attribute<T>> rhs, Func op)
{
    auto premise = std::make_shared<Premise>(
        [=]() { return op(lhs->GetValue(), rhs->GetValue()); });
    lhs->Attach(premise);
    rhs->Attach(premise);
    premise->Update();
    return premise;
}

template <typename T, typename Func>
auto BuildPremise(std::shared_ptr<Attribute<T>> lhs, T rhs, Func op)
{
    auto premise =
        std::make_shared<Premise>([=]() { return op(lhs->GetValue(), rhs); });
    lhs->Attach(premise);
    premise->Update();
    return premise;
}
```

Fonte: Autoria própria

O Código 3.33 apresenta os *builders* utilizados para a construção de *Conditions*. O primeiro *builder* recebe como parâmetro a expressão lógica (na forma de uma expressão *lambda*) composta por *Premises* e *Conditions*, assim como os respectivos ponteiros dessas entidades, como em *NOP::BuildCondition(CONDITION(pr1 && pr2), pr1, pr2)*. Em comparação, o segundo *builder* apresenta uma construção simplificada para a construção de *Condition* mais simples, como o caso de única *Premise*, conjunção ou disjunção, utilizando um parâmetro de *template* na construtora para isso, como em *NOP::BuildCondition<NOP::Conjunction>(pr1, pr2)*.

Código 3.33 – Builder de Condition do Framework PON C++ 4.0

```

template <typename... Args>
auto BuildCondition(std::function<bool(void)> operation, Args... args)
{
    auto condition = std::make_shared<NOP::Condition>(operation);
    auto attach = [&](auto element) { element->Attach(condition); };
    (attach(args), ...);
    condition->Update();
    return condition;
}

template <EConditionType type, typename... Args>
auto BuildCondition(Args... args)
{
    std::shared_ptr<Condition> condition=nullptr;
    if constexpr (Single == type)
    {
        // Explicit operator bool
        condition = std::make_shared<Condition>(
            [=] () { return (*args && true), ...; });
    }
    else if constexpr (Conjunction == type)
    {
        condition =
            std::make_shared<Condition>([=] () { return (*args && ...); });
    }
    else if constexpr (Disjunction == type)
    {
        condition =
            std::make_shared<Condition>([=] () { return (*args || ...); });
    }
    auto attach = [&](auto element) { element->Attach(condition); };
    (attach(args), ...);
    return condition;
}

```

Fonte: Autoria própria

O Código 3.34 é utilizado para a construção de *Instigations*, recebendo como parâmetros um número qualquer de *Methods*, como em *NOP::BuildInstigation(mt1, mt2, mt3)*. Destaca-se ainda que é possível utilizar um parâmetro de *template NOP::Parallel* para realizar a construção de uma *ParallelInstigation* da mesma forma, como em *NOP::BuildInstigation<NOP::Parallel>(mt1, mt2, mt3)*.

Código 3.34 – Builder de Instigation do Framework PON C++ 4.0

```

template <NOPFlag Flag = Default, typename... Args>
auto BuildInstigation(Args... methods)
{
    std::shared_ptr<Instigation> instigation;
    if constexpr (NOP::Parallel == Flag)
    {
        instigation = std::make_shared<ParallelInstigation>();
    }
    else
    {
        instigation = std::make_shared<Instigation>();
    }
    (instigation->AddMethod(methods), ...);
    return instigation;
}

```

Fonte: Autoria própria

O Código 3.35 é utilizado para a construção de *Actions*, recebendo como parâmetros um número qualquer de *Instigations*, como em *NOP::BuildAction(in1, in2, in3)*. Da mesma forma que no *builder* de *Instigation*, o *template NOP::Parallel* pode ser utilizado para construir

uma *ParallelAction* como em *NOP::BuildAction<NOP::Parallel>(in1, in2, in3)*.

Código 3.35 – *Builder de Action do Framework PON C++ 4.0*

```
template <NOPFlag Flag = Default, typename... Args>
auto BuildAction(Args... instigation)
{
    std::shared_ptr<Action> action{};
    if constexpr (NOP::Parallel == Flag)
    {
        action = std::make_shared<ParallelAction>();
    }
    else
    {
        action = std::make_shared<Action>();
    }
    (action->AddInstigation(instigation), ...);
    return action;
}

template <NOPFlag Flag = Default, typename... Args>
auto BuildActionNamed(const std::string_view name, DebugEntity* fbe,
                      Args... instigation)
{
    auto action = BuildAction<Flag>(instigation...);
    SET_DEBUG_PROPERTIES(action, name, fbe);
    return action;
}
```

Fonte: Autoria própria

O Código 3.36 é utilizado para a construção de *Rules* e é bastante simples, pois recebe um número fixo de parâmetros, apenas uma *Condition* e uma *Action* que compõem a *Rule*, como em *NOP::BuildRule(cn1, ac1)*.

Código 3.36 – *Builder de Rule do Framework PON C++ 4.0*

```
inline std::shared_ptr<Rule> BuildRule(std::shared_ptr<Condition> condition,
                                         std::shared_ptr<Action> action,
                                         const int priority = 0)
{
    auto rule = std::make_shared<NOP::Rule>(priority);
    condition->Attach(rule);
    rule->SetAction(std::move(action));
    rule->SetCondition(std::move(condition));
    if (rule->Approved())
    {
        rule->Execute();
    }
    return rule;
}
```

Fonte: Autoria própria

Com relação a estas implementações de *builders* apresentadas podem ser destacadas as implementações dos Códigos 3.32, 3.33, 3.35 e *fold expressions*, de modo a definir uma interface comum que, utilizando *variadic templates*, constrói objetos complexos e diversos, pois estes *builders* permitem a construção de objetos com um número de parâmetros genérico, de acordo com a necessidade do desenvolvedor.

Ainda aproveitando do conceito de entidades compartilhadas, todos os *builders* constroem objetos do tipo *shared_ptr*, permitindo o fácil compartilhamento das entidades ao longo do

código, aproveitando as vantagens que o uso de *smart pointers* oferece, provendo gerenciamento de memória de forma automática e segura, até mesmo em ambientes *multithread*, pois os *shared pointers* utilizam contadores atômicos para o controle do número de referências, garantindo o sincronismo entre as *threads*.

3.9.3 Logger

Uma das maiores dificuldades apresentadas no desenvolvimento de aplicações em PON é justamente a depuração de código, devido ao fluxo de execução do PON que difere daquele de uma aplicação tradicional em C++, que acontece de forma sequencial. Em função disso, as ferramentas de depuração existentes nas *IDEs* de C++ não atendem as necessidades da depuração de um programa em PON (RONSZCKA, 2012).

Desta forma, outro recurso adicionado de forma a facilitar o desenvolvimento de aplicações como *framework* foi um mecanismo de *log*, similar ao já presente no *Framework PON C++ 2.0*. Por *log* se entende um registro na forma textual de eventos do código. O mecanismo de *log* do *Framework PON C++ 4.0* permite que o desenvolvedor observe de forma sequencial os eventos acionados pela cadeia de notificações do PON sob a forma de texto. Na Figura 76 é apresentado o diagrama das classes utilizadas para o desenvolvimento desse mecanismo.

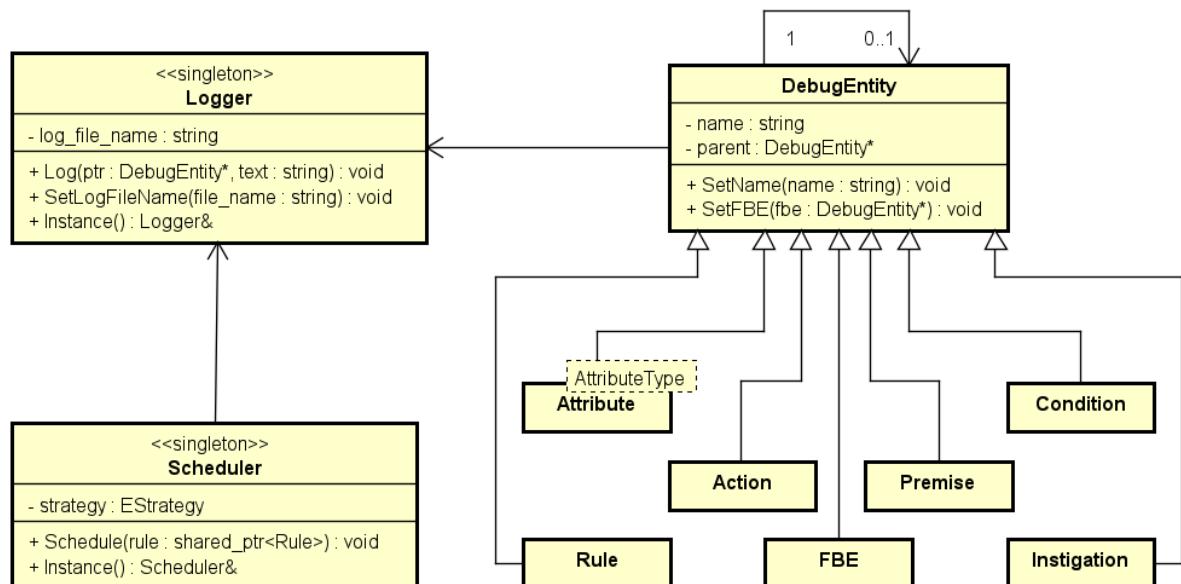


Figura 76 – Diagrama de classes de logs do Framework PON C++ 4.0
Fonte: Autoria própria

Como a linguagem C++, ao contrário de linguagens como *Python* e *JavaScript*, ainda

não suporta mecanismos de reflexão de código, que permitiriam acessar os nomes das variáveis, é necessária a criação de entidades para armazenar estas propriedades. A classe *DebugEntity* é utilizada para este fim, contendo o nome e um ponteiro para o *FBE*, caso exista, que contém a entidade. Todas as classes das entidades do PON derivam dessa classe *DebugEntity*.

A classe *Logger* é a entidade responsável pelos registros em si, realizando operações de escrita nos arquivos. Esta classe é implementada como um *singleton*, facilitando o seu acesso por todas as outras entidades do código. Além das entidades do tipo *DebugEntity*, a classe *Scheduler* também usa a classe *Logger* para registrar a execução das *Rules*.

Para permitir a utilização desse mecanismo de *log* com nomes legíveis para o desenvolvedor foram criados uma série de *builders*. Esses *builders* seguem a mesma estrutura dos *builders* já apresentados na Seção 3.9.2, com a adição de parâmetros para o nome e ponteiro para o *FBE*, conforme mostrado no Código 3.37 no *builder* da *Condition*.

Código 3.37 – *Builder* com propriedades de *debug* do Framework PON C++ 4.0

```
template <EConditionType type, typename... Args>
auto BuildConditionNamed(const std::string_view name, DebugEntity* fbe, Args... args)
{
    auto condition = BuildCondition<EConditionType>(args...);
    SET_DEBUG_PROPERTIES(condition, name, fbe);
    return condition;
}
```

Fonte: Autoria própria

De modo a exemplificar esta utilização, a estrutura *Test* e sua respectiva implementação com o *Framework PON C++ 4.0* são apresentadas no Código 3.38, na qual um *FBE* simples é inicializado nomeando os seus membros, assim como nomeando o próprio *FBE*. Um trecho de *log* resultante da utilização dessa estrutura é apresentado na Tabela 10. Nesse trecho podem ser observados os registros de mudança de valor dos *Attributes*, nas linhas 1 e 3, até a aprovação e execução da *Rule* pelo escalonador, nas linhas 7 e 8.

Código 3.38 – Exemplo de FBE com nomes do Framework PON C++ 4.0

```

/*
fbe Test
public integer at1 = -1
public integer at2 = -2
public integer atExecutionCounter = 0
private method mt
    attribution
        this.atExecutionCounter = this.atExecutionCounter + 1
    end_attribution
end_method
rule rlTest
    condition
        premise pr1
            this.at1 == this.at2
        end_premise
    end_condition
    action sequential
        instigation sequential
            call this.mt()
        end_instigation
    end_action
end_rule
end_fbe
*/
struct Test : NOP::FBE
{
    explicit Test(const std::string_view name) : FBE(name) {}
    // Essa é a parte relativa a definição do FBE
    // Aqui se declaram e inicializam os Attributes
    NOP::SharedAttribute<int> at1 =
        NOP::BuildAttributeNamed("at1", this, -1);
    NOP::SharedAttribute<int> at2 =
        NOP::BuildAttributeNamed("at2", this, -2);
    NOP::SharedAttribute<int> atExecutionCounter =
        NOP::BuildAttribute(0);
    // Aqui se define o Method utilizando uma expressão lambda
    NOP::Method mt = METHOD(
        atExecutionCounter.SetValue(executionCounter.GetValue() + 1);
    );

    // Essa é a parte relativa a definição da Rule agregada ao FBE
    // com a definição das respectivas Premise, Condition, Action
    // e Instigation que a compõe
    NOP::SharedPremise pr1 =
        NOP::BuildPremiseNamed("pr1", this, at1, at2, NOP::Equal());
    NOP::SharedCondition cn1 =
        NOP::BuildConditionNamed("cn1", this, CONDITION(*pr1), pr1);
    NOP::SharedInstigation in1 =
        NOP::BuildInstigationNamed("in1", this, mt);
    NOP::SharedAction ac1 =
        NOP::BuildActionNamed("ac1", this, in1);

    NOP::SharedRule rl1 = NOP::BuildRuleNamed("rl1", this, cn1, ac1);
};


```

Fonte: Autoria própria

Como esse mecanismo de *log* pode ser computacionalmente bastante custoso, aumentando os tempos de execução devido ao tempo gasto escrevendo em arquivos, assim como aumentando o consumo de memória das entidades para armazenar os nomes das mesmas, o Framework PON C++ 4.0 é construído de forma a possibilitar habilitar ou desabilitar a utilização dos *logs*. Isso é feito por meio de uma opção de compilação *LIBNOP_LOG_ENABLE* que pode ser configurada pelo desenvolvedor de acordo com a sua necessidade. Esta e outras opções de compilação são detalhadas na seção seguinte.

```

1 [2021-03-27 18:40:21.494] [NOP] [info] TestFBE - at1: Value changed to 1
2 [2021-03-27 18:40:21.494] [NOP] [info] TestFBE - pr1: Disapproved
3 [2021-03-27 18:40:21.494] [NOP] [info] TestFBE - at2: Value changed to 1
4 [2021-03-27 18:40:21.494] [NOP] [info] TestFBE - pr1: Approved
5 [2021-03-27 18:40:21.494] [NOP] [info] TestFBE - cn1: Approved
6 [2021-03-27 18:40:21.494] [NOP] [info] TestFBE - rl1: Scheduled
7 [2021-03-27 18:40:21.494] [NOP] [info] TestFBE - rl1: Approved
8 [2021-03-27 18:40:21.494] [NOP] [info] TestFBE - rl1: Executed by scheduler
9 [2021-03-27 18:40:21.494] [NOP] [info] TestFBE - ac1: Activated
10 [2021-03-27 18:40:21.494] [NOP] [info] TestFBE - in1: Instigated

```

Tabela 10 – Exemplo de logs do Framework PON C++ 4.0**Fonte:** Autoria própria

Ainda, para a implementação deste mecanismo é utilizada a biblioteca `spdlog`³. Esta é uma biblioteca de alto desempenho para *logs* em linguagem de programação C++. Além de facilitar o desenvolvimento, o uso dessa biblioteca garante o funcionamento correto do mecanismo de *logs* mesmo para a execução com paralelismo, sem ocorrer sobreposição indevida das linhas de *log*, pois a biblioteca apresenta mecanismos de controle que gerenciam a escrita dos *logs*.

3.10 OPÇÕES DE COMPILAÇÃO

De modo a permitir ao desenvolvedor obter o melhor desempenho possível com o *Framework PON C++ 4.0*, alguns recursos podem ser desabilitados. Esses recursos são o mecanismo de *logs* e o escalonador. Ambos esses recursos, apesar de úteis, aumentam os tempos de execução do *software* em *Framework PON C++ 4.0*, sendo ambos desabilitados por padrão. Dito isto, é importante ressaltar que a execução com o escalonador desabilitado apresenta o mesmo comportamento da estratégia *NO_ONE* do *Framework PON C++ 2.0*.

O controle dessas opções pode ser feito por meio de definições no código. O controle por meio de definições de código é simples, e pode ser feito inserindo `#define LINBOP_SCHEDULED_ENABLE` e/ou `#define LINBOP_LOGGER_ENABLE` antes de incluir os arquivos do *framework*, conforme exemplificado no Código 3.39.

Ademais, um guia em inglês com instruções detalhadas de uso do *Framework PON C++ 4.0*, desenvolvido com o propósito de facilitar o uso por novos desenvolvedores, é disponibilizado no Apêndice E. Esse manual apresenta exemplos de uso das entidades do PON, assim como instruções de compilação⁴.

³ Código fonte disponível em <https://github.com/gabime/spdlog>

⁴ Além do manual também foi realizado um treinamento virtual de utilização do *Framework PON C++ 4.0* que foi gravado e será posteriormente disponibilizado na forma de vídeo

Código 3.39 – Uso de defines do Framework PON C++ 4.0

```
#define LIBNOP_SCHEDULER_ENABLE
#define LINBOP_LOGGER_ENABLE
#include "libnop/framework.h"
```

Fonte: Autoria própria

3.11 TESTES UNITÁRIOS E DE INTEGRAÇÃO

A partir da implementação de todas as funcionalidades do *Framework PON C++ 4.0*, que foram apresentadas ao longo deste capítulo, foi construído um conjunto de testes capaz de validar as funcionalidades do *framework*, como a execução correta do mecanismo de notificações, bem como a aprovação de *Premises*, *Conditions* e *Rules* conforme o esperado em dados contextos.

Foram desenvolvidos 38 testes englobando todas as funcionalidades do *framework*. A cada modificação feita no código, os testes são novamente executados para se garantir que nenhuma funcionalidade que já havia sido previamente implementada tenha sido degradada. No Código 3.40, mais precisamente, é apresentado um desses testes contemplando todas as unidades do *framework* sob a forma de um teste de integração. O conjunto completo de testes do *Framework PON C++ 4.0* está disponível para consulta no Apêndice D.

Código 3.40 – Caso de teste do Framework PON C++ 4.0

```
TEST(Complete, Basic)
{
    // Criação dos Attributes
    NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(-1);
    NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(1);

    // Criação da Premise
    NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());

    // Criação da Condition
    NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Single>(pr1);

    // Contador utilizado para verificar execução do método
    std::atomic<int> executionCounter{0};

    // Criação da Action com Instigation e Method criadas de forma conjunta
    NOP::SharedAction acl = NOP::BuildAction(
        NOP::BuildInstigation(
            METHOD(executionCounter++)));
}

// Criação da Rule
NOP::SharedRule r11 = NOP::BuildRule(cn1, acl);
// Valida estados iniciais das entidades
EXPECT_FALSE(pr1->Approved());
EXPECT_FALSE(cn1->Approved());
EXPECT_FALSE(r11->Approved());

// Altera valor do Attribute para causar aprovação da Rule
at1->SetValue(1);
// Valida novos estados das entidades
EXPECT_TRUE(pr1->Approved());
EXPECT_TRUE(cn1->Approved());
EXPECT_TRUE(r11->Approved());
// Valida execução do método por meio do contador
```

```

    } EXPECT_EQ(executionCounter, 1);
}

```

Fonte: Autoria própria

Da mesma forma os Códigos 3.41 e 3.42 mostram, respectivamente os casos de teste para a validação de uma *Premise* e uma *Condition*. Esses são testes de unidades, mais elementares que o teste do Código 3.40, que permitem uma maior granularidade no nível dos testes, facilitando encontrar os erros em caso de falha nos testes, pois se torna possível observar em qual unidade elementar do *framework* ocorrem problemas.

Código 3.41 – Caso de teste para *Premise* do *Framework PON C++ 4.0*

```

/*
public integer at1 = -1
public integer at2 = -2

premise prIsActivated
    this.at1 == this.at2
end_premise
*/
TEST(Premise, Simple)
{
    // Inicialização das unidades
    NOP::SharedAttribute<int> at1 = NOP::BuildAttribute<int>(-1);
    NOP::SharedAttribute<int> at2 = NOP::BuildAttribute<int>(-2);
    NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());

    // Validação dos testes
    EXPECT_FALSE(pr1->Approved());
    at1->SetValue(1);
    EXPECT_FALSE(pr1->Approved());
    at2->SetValue(1);
    EXPECT_TRUE(pr1->Approved());
}

```

Fonte: Autoria própria

Código 3.42 – Caso de teste para *Condition* do Framework PON C++ 4.0

```

/*
public integer at1 = -1
public integer at2 = -2

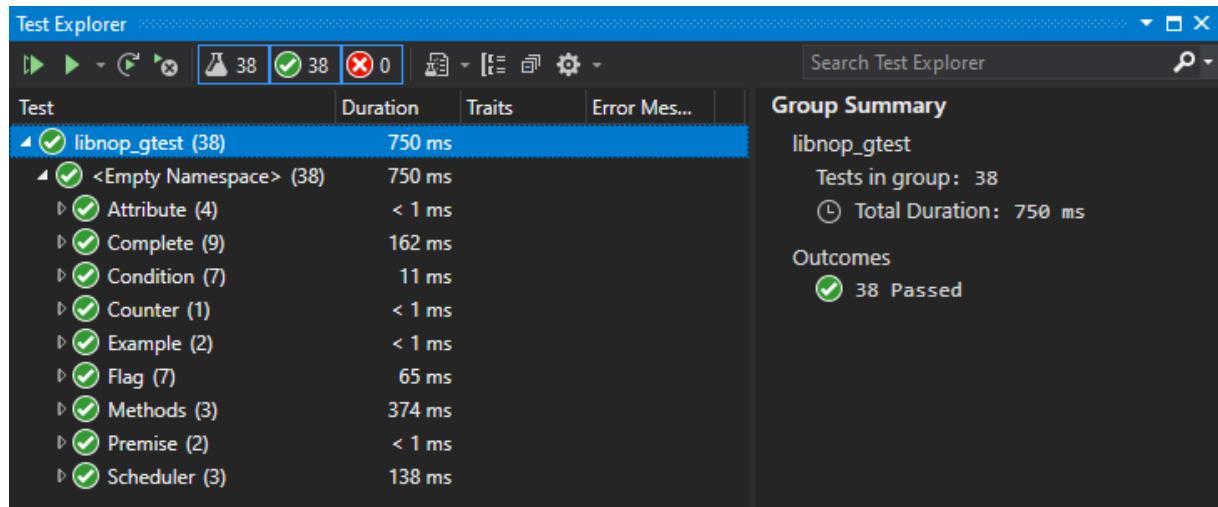
condition
    premise prIsActivated
        this.at1 == this.at2
    end_premise
end_condition
*/
TEST(Condition, Conjunction)
{
    // Inicialização das unidades
    NOP::SharedAttribute<int> at1 = NOP::BuildAttribute<int>(-1);
    NOP::SharedAttribute<int> at2 = NOP::BuildAttribute<int>(-1);
    NOP::SharedPremise pr1 = NOP::BuildPremise(at1, 1, NOP::Equal());
    NOP::SharedPremise pr2 = NOP::BuildPremise(at2, 2, NOP::Equal());
    NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Conjunction>(pr1, pr2);

    // Validação dos testes
    EXPECT_FALSE(cn1->Approved());
    at1->SetValue(1);
    EXPECT_FALSE(cn1->Approved());
    at2->SetValue(2);
    EXPECT_TRUE(cn1->Approved());
}

```

Fonte: Autoria própria

A Figura 77 mostra o relatório gerado com a execução de todos os testes utilizando o framework Google Test. Esse relatório foi obtido por meio da integração da ferramenta Visual Studio 2019, que detecta automaticamente os casos de testes escritos com o Google Test e consegue listar os mesmos nesta interface gráfica. Nesta figura é possível observar que todos os testes foram aprovados e que a execução levou menos de 1 segundo.

**Figura 77 – Relatório dos testes do Framework PON C++ 4.0****Fonte:** Autoria própria

3.12 REFLEXÕES SOBRE O FRAMEWORK PON C++ 4.0

Dado o desenvolvimento do *Framework PON C++ 4.0*, a Tabela 11 reapresenta os objetivos propostos para o desenvolvimento do mesmo, ressaltando que todos foram atingidos em plenitude, conforme apresentado ao longo de todo este capítulo. Ademais, nesta seção são feitas reflexões sobre como esses objetivos foram alcançados.

Tabela 11 – Objetivos atingidos pelo Framework PON C++ 4.0
Fonte: Autoria própria

Objetivo	Proposta	Atingido
Adicionar a flexibilidade de tipos	<i>Attribute</i> com tipo <i>template</i>	Sim
Adicionar flexibilidade algorítmica	<i>Condition</i> com expressões <i>lambda</i>	Sim
Reducir a verbosidade da utilização	<i>Builders</i> com <i>variadic templates</i>	Sim
Permitir execução com paralelismo	Notificações com políticas de execução	Sim

A aplicação da programação genérica, por meio do uso de *templates* permitiu adicionar a flexibilidade de tipos aos *Attributes*, possibilitando o uso de *Attributes* com tipos definidos pelo desenvolvedor. Ainda, tal flexibilização permitiu simplificar o código do *framework*, quando comparado ao *Framework PON C++ 2.0*, por eliminar as instâncias de código repetido que seriam necessárias para implementar as especializações de cada tipo.

A flexibilidade algorítmica, por sua vez, é alcançada com a criação do conceito de *Condition* flexível, possibilitado pelo uso de expressões *lambda* para a definição da operação booleana entre as *Premises*. Naturalmente, como esse é um conceito novo introduzido pelo *Framework PON C++ 4.0*, ainda não é possível de ser representado com a NOPL.

Entretanto, a adição do conceito de *Condition* flexível, apesar de facilitar a programação em alto nível, dificulta a implementação do conceito de impertinência. A implementação do conceito de impertinência depende da entidade *Condition* ser capaz de avaliar o número de *Premises* que devam mudar de estado para poder ser aprovada, de modo a ser capaz de remover a impertinência quando necessário. A adição da flexibilidade algorítmica, implementada através de expressões *lambda* no *Framework PON C++ 4.0*, abstrai o número de premissas e a operação realizada entre elas, de modo que a classe *Condition* se torna incapaz de avaliar a impertinência das entidades.

Deste modo, o conceito de impertinência fica de fora do escopo da implementação do *Framework PON C++ 4.0* neste trabalho, porém ela pode ser implementada em trabalhos futuros. A implementação de impertinência no *Framework PON C++ 4.0* pode ser feita com a implementação de classes que realizem o tratamento de *Conditions* apenas com disjunções ou

conjunções, não permitindo a expressão de *Condition* com flexibilidade algorítmica. Essas novas classes naturalmente herdariam da classe *Condition* e implementariam a lógica específica para o tratamento de impertinência.

A redução da verbosidade da utilização é dada principalmente com a implementação dos *Builder*, que utilizando os recursos avançados da linguagem de programação C++ dita moderna, como *variadic templates* e *fold expression*, permitiu a criação de interfaces bastante simples de utilizar, mas que também apresentam a flexibilidade necessária para o uso. Desta forma, como fica evidente ao longo dos diversos códigos apresentados neste capítulo, *Framework PON C++ 4.0* permite uma estrutura de declaração de suas entidades de forma muito similar a estrutura da NOPL, contribuindo assim com a facilitação da programação em alto nível, de modo similar ao almejado pelo prototípico JuNOC++.

A questão do paralelismo é tratada de forma bastante transparente, visto que com a adição de recursos como as políticas de execução foi possível a implementação de código altamente paralelizável sem ser necessária a adição de mecanismos complexos de controle. A complexidade referente ao paralelismo é tratada justamente por esses recursos da linguagem, não sendo necessário reimplementar no *framework*.

O paralelismo é, de fato, implementado de duas principais maneiras, utilizando as notificações paralelizadas por meio das políticas de execução em si, e também por meio do uso de *Methods* paralelizáveis. O *framework* apresenta uma maneira padrão de se criar *Methods* paralelizáveis (pelo uso da macro *ASYNC_METHOD*), porém o desenvolvedor também é livre para implementar seus próprios mecanismos de paralelização caso seja necessário, devido à flexibilidade dada pelo uso das expressões *lambda*.

Além disso, o conjunto de testes unitários e de integração desenvolvidos à luz do método TDD se prova essencial para avaliar o funcionamento das entidades desenvolvidas no *Framework PON C++ 4.0*. Entretanto, a aplicação diverge um pouco do método proposto por Kossoski (2015), devido ao método ser proposto para o teste de aplicações, e não de frameworks. Dessa forma, não há a existência de casos de uso em nível de aplicação, como utilizado no método de Kossoski (2015) para levantamento dos casos de teste. Ainda assim, se aproveitam os conceitos de testes comuns às entidades do PON, como as classes de equivalência para testes de *Premises*. Apesar disso, o *Framework PON C++ 4.0* também evolui sobre o método proposto ao aplicar o uso de *frameworks* de teste para automatizar o processo de testes.

Por fim, também é importante abordar a questão do desempenho, pois a construção do

Framework PON C++ 4.0 utiliza diversos recursos da STL em sua composição, como *std::vector*, *std::functions*, *std::unique_ptr* e *std::shared_ptr*. A utilização destas estruturas pode incorrer em um aumento do tempo de execução de programas devido ao custo computacional da utilização destas estruturas.

Esse custo de desempenho poderia ficar aparente principalmente quando comparado ao *Framework PON C++ 2.0*, que não utiliza estas entidades computacionais potencialmente custosas, ao optar por uma implementação utilizando estruturas de dados dedicadas, justamente com o objetivo de se obter um melhor desempenho com um menor tempo de execução (VALENÇA, 2012).

Ainda assim, o uso eficiente dos recursos da STL possibilita uma melhor estruturação algorítmica do código, o que por sua vez pode melhorar o desempenho ao diminuir os tempos de execução. Essa melhor estruturação algorítmica possibilita um melhor aproveitamento do recurso de compartilhamento de entidades, reduzindo o número de entidades como *Premises* e *Conditions* necessárias, o que por sua vez também reduz número de notificações geradas no sistema.

De modo a ilustrar os benefícios dessa melhor estruturação algorítmica com o *Framework PON C++ 4.0*, o Código 3.43 mostra como é complicada a construção da *Condition mainCondition* que faz a combinação de diversas *Premises* e exigindo o uso de duas *SubConditions*, enquanto o Código 3.44 simplifica essa mesma construção por meio do uso das *Conditions* com flexibilidade algorítmica. Essa melhoria permite reduzir o número de entidades necessárias para a construção do código, passando de quatro *Premises* e três *Conditions* para apenas três *Premises* e uma única *Condition*, com isso reduzindo o número de notificações necessárias para aprovar uma *Rule* e melhorando a expressividade do código.

Código 3.43 – Construção complexa de *Condition* com o *Framework PON C++ 4.0*

```
/*
condition mainCondition
    subcondition cn1
        premise prIsNotAt1 at1 == false
        and
        premise prIsAt2 at2 == true
    end_subcondition
    or
    subcondition cn2
        premise prIsAt1 at1 == true
        and
        premise prIsAt3 at3 == true
    end_subcondition
end_condition
*/
NOP::SharedAttribute<bool> at1 = NOP::BuildAttribute(false);
NOP::SharedAttribute<bool> at2 = NOP::BuildAttribute(false);
```

```

NOP::SharedAttribute<bool> at3 = NOP::BuildAttribute(false);
NOP::SharedPremise prIsAt1 = NOP::BuildPremise<bool>(at1, true, NOP::Equal());
NOP::SharedPremise prIsNotAt1 = NOP::BuildPremise<bool>(at1, false, NOP::Equal());
NOP::SharedPremise prIsAt2 = NOP::BuildPremise<bool>(at2, true, NOP::Equal());
NOP::SharedPremise prIsAt3 = NOP::BuildPremise<bool>(at3, true, NOP::Equal());
NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Conjunction>(prIsNotAt1, prIsAt2);
NOP::SharedCondition cn2 = NOP::BuildCondition<NOP::Conjunction>(prIsAt1, prIsAt3);
NOP::SharedCondition mainCondition = NOP::BuildCondition<NOP::Disjunction>(cn1, cn2);

```

Fonte: Autoria própria

Código 3.44 – Construção simplificada de *Condition* com o *Framework PON C++ 4.0*

```

NOP::SharedAttribute<bool> at1 = NOP::BuildAttribute(false);
NOP::SharedAttribute<bool> at2 = NOP::BuildAttribute(false);
NOP::SharedAttribute<bool> at3 = NOP::BuildAttribute(false);
NOP::SharedPremise prIsAt1 = NOP::BuildPremise<bool>(at1, true, NOP::Equal());
NOP::SharedPremise prIsAt2 = NOP::BuildPremise<bool>(at2, true, NOP::Equal());
NOP::SharedPremise prIsAt3 = NOP::BuildPremise<bool>(at3, true, NOP::Equal());
NOP::SharedCondition mainCondition = NOP::BuildCondition(CONDITION(
    (!*prIsAt1 && *prIsAt2) || (*prIsAt1 && *prIsAt3) ),
    prIsAt1, prIsAt2, prIsAt3);

```

Fonte: Autoria própria

Observando os resultados atingidos pelo desenvolvimento do *Framework PON C++ 4.0*, apresentados neste Capítulo, é interessante analisar de que forma este novo *framework* avança sobre as propriedades elementares e conceitos comparados aos outros *frameworks* disponíveis. Estes conceitos foram anteriormente introduzidos na Seção 2.3. A propriedade de distribuição do PON ainda não foi contemplada na implementação atual do *Framework PON C++ 4.0*, porém trabalhos futuros pretendem contemplar esta propriedade em futuras melhorias no *framework*. A Tabela 12 resume estes resultados.

Tabela 12 – Propriedades elementares contempladas nas materializações do PON

Fonte: Autoria própria

Potencial de propriedade \ Materialização	Fw. C++ Prot.	Fw. C++ 1.0	Fw. C++ 2.0	Fw. C++ 3.0	Fw. C++ 4.0	Fw. Java /C#	Fw. C# IoT	Fw. Eli- xir	Fw. Akka
Programação em alto nível	~	~	*	~	✓	~	~	~	~
Paralelismo via desacoplamento				~	✓		✓	✓	✓
Distribuição via desacoplamento					○		✓	*	*
Desempenho via não redundâncias				~		✓	~		

✓ Materializa completamente a propriedade

~ Materializa parcialmente a propriedade

* Materializa a propriedade por meio de ferramenta *wizard* (VALENÇA, 2012)

* A tecnologia de base permite a propriedade, entretanto carece de testes para validação

○ Trabalho subsequente proposto de Pelegrine Figueiredo detalhado na Seção 5.2.3

Pelo fato do *Framework PON C++ 4.0* se tratar de uma evolução sobre o *Framework*

PON C++ 2.0, era de se esperar que todos os conceitos de programação materializados no *Framework PON C++ 2.0* também se encontrassem no *Framework PON C++ 4.0*. A tabela 13 apresenta os conceitos de programação materializados pelos diferentes *frameworks*, destacando o *Framework PON C++ 4.0*. Em suma, no que diz respeito aos conceitos atingidos é possível notar que o *Framework PON C++ 4.0* apresenta os conceitos que o *Framework PON C++ 2.0* materializava, com a exceção da impertinência estática e *Keeper*, porém também materializa o conceito de *Condition* flexível.

Devido ao aumento da flexibilidade dado pela adição do conceito de *Condition* flexível é dificultada a implementação do conceito de impertinência, conforme discutido na Seção 3.1. Ainda, um dos objetivos da adição do conceito de impertinência é justamente melhorar o desempenho das aplicações. Entretanto, espera-se que, apesar de não contar com a materialização desse conceito, reduza os tempos de execução das aplicações em PON ao permitir uma melhor eficiência na composição e compartilhamento das entidades e também ao reduzir o número de entidades totais do código, como demonstrado nos Códigos 3.43 e 3.44. Ainda, o conceito *Keeper* também não é materializado, visto que sua implementação altera o fluxo tradicional de execução das *Rules* do PON (MUCHALSKI *et al.*, 2012), podendo ser facilmente implementado com as mesmas adaptações que foram necessárias no *Framework PON C++ 2.0*. Por fim, o conceito de *Formation Rules* também não é implementado por nenhum *framework* sendo materializado apenas por meio da tecnologia LingPON, visto que tal nível de abstração é difícil de ser implementado nas linguagens de programação dos *frameworks*, pois exige uma etapa de compilação para interpretar a expressão destas *Rules*.

Os resultados obtidos por meio da utilização do *Framework PON C++ 4.0* para o desenvolvimento de aplicações são detalhados no capítulo seguinte. Nesses resultados é inclusive discutida a efetividade das implementações de paralelização no *Framework PON C++ 4.0*.

Tabela 13 – Conceitos do PON contemplados nas materializações do paradigma - com adição do Framework PON C++ 4.0

Fonte: Autoria própria

Conceito de Programação \ Materialização	Fw. Prot.	Fw. C++ 1.0	Fw. C++ 2.0	Fw. C++ 3.0	Fw. C++ 4.0	Fw. Java /C#	Fw. C# IoT	Fw. Elixir	Fw. Akka
Reatividade das entidades	✓	✓	✓	✓	✓	✓	✓	✓	✓
Compartilhamento de entidades		✓	✓	✓	✓	✓	✓	✓	
Renotificações		✓	✓	✓	✓	✓			
Resolução de conflitos		✓	✓	✓	✓	✓	✓		
<i>Master Rule</i>			✓	✓	✓				
Impertinência estática			✓	✓					
Impertinência dinâmica							✓		
<i>FBE Rules</i>				✓	✓	✓			
<i>FBE Agregador</i>				✓	✓	✓			
<i>Formation Rules</i>									
<i>Keeper</i>			*						
<i>Condition</i> flexível					✓				

* Implementado sob a forma de uma adaptação no *framework* existente, visto que altera o fluxo tradicional de execução das *Rules* do PON (MUCHALSKI *et al.*, 2012)

4 EXPERIMENTOS COM O *FRAMEWORK PON C++ 4.0* E SEUS RESULTADOS

O desenvolvimento do *Framework PON C++ 4.0* foi proposto com o objetivo de introduzir recursos que facilitem o desenvolvimento de aplicações em PON, assim como utilizar recursos modernos da linguagem C++ que permitiriam ganhos de desempenho quando comparado ao *Framework PON C++ 2.0*, particularmente por ser até então o de melhor desempenho. Desta forma se torna necessária a realização de testes que validem o *Framework PON C++ 4.0* do ponto de vista funcional, assim como do ponto de vista de desempenho.

Os resultados de todos os testes realizados em linguagem de programação C++¹ foram obtidos por meio do *framework Google Benchmark*, de modo a permitir a reprodutibilidade dos testes e resultados de forma simples. Além disso, o *Google Benchmark* garante a confiabilidade dos tempos de execução apresentados, pois ele gerencia o número de iterações necessárias de forma dinâmica para obter um resultado estatisticamente estável (GOOGLE, 2020a).

Além das aplicações desenvolvidas para os testes de desempenho, na Seção 4.2 é apresentada uma nova da implementação do jogo *NOPUnreal*. Esse jogo já foi introduzido na Seção 2.4.1.6. Entretanto, nesta seção ele é reimplementado com o *Framework PON C++ 4.0*, o que permite a comparação em termos de facilidade de desenvolvimento e verbosidade com a implementação anterior desenvolvida com o *Framework PON C++ 2.0*.

Após apresentadas estas aplicações, a Seção 4.3 apresenta a pesquisa de opinião relativa ao uso do *Framework PON C++ 4.0* realizada com desenvolvedores do grupo de pesquisa do PON apenas com o intuito de se obter algum *feedback* ou alimentação externa. Por fim, na Seção 4.4, é realizada uma reflexão sobre os resultados apresentados neste capítulo.

4.1 TESTES DE DESEMPENHOS

Nas seções seguintes são apresentadas as aplicações desenvolvidas com o propósito de avaliar o desempenho do *Framework PON C++ 4.0*. A Seção 4.1.1 apresenta a aplicação de sensores, a Seção 4.1.2 apresenta o algoritmo *Bitonic Sort*, a Seção 4.1.3 apresenta o algoritmo *Random Forest* e, por fim, a Seção 4.1.4 apresenta a aplicação de controle de semáforos.

Cada um destes *benchmarks* apresenta seu próprio propósito. O objetivo da aplicação

¹ Para permitir os testes com o *Google Benchmark*, aplicações na linguagem de programação C também são compiladas em C++

de sensores tem como objetivo permitir a comparação do desempenho do novo *Framework PON C++ 4.0* com o estável *Framework PON C++ 2.0*, assim como comparar o desempenho de ambos os *frameworks* com POO. A aplicação de controle de tráfego automatizado tem o propósito de permitir avaliar o comportamento do paralelismo introduzidas pelo *Framework PON C++ 4.0*, de modo que esta aplicação apresenta comparações com o *Framework PON C++ Elixir/Erlang*, pelo fato deste ser um *framework* que faz amplo uso das capacidades de paralelismo da linguagem de programação Elixir. Por fim, ambos os *benchmarks* universalizados por meio dos algoritmos *Bitonic Sort* e *Random Forest* tem como principal objetivo permitir a comparação do desempenho do *Framework PON C++ 4.0* com o PP, assim como avaliar os benefícios da utilização de paralelismo para a execução destes algoritmos.

4.1.1 Aplicação de sensores

A chamada aplicação de sensores materializa uma solução para uma rede de sensores simulados, na qual cada sensor possui um estado (ativado ou desativado) que pode ser observado. Uma *Rule* determina o comportamento do sensor, quando o sensor é ativado esta *Rule* é aprovada, reiniciando os estados de ativação e leitura do sensor.

A representação desta aplicação sob a forma de um *FBE* e uma *Rule* foi previamente apresentada na Figura 6 da Seção 1.1.3. O Código 4.1 em LingPON, poderia ser utilizado para gerar código nos *frameworks*, conforme feito em (SKORA, 2020). Porém, neste presente trabalho, por questões de possibilitar uma otimização mais fina do código, as implementações com o *Framework PON C++ 2.0* e *4.0* foram escritas manualmente, servindo o código em LingPON de guia tão somente. O Código 4.2 apresenta a estrutura utilizada para implementar o sensor com o *Framework PON C++ 4.0*, utilizando um total de apenas 13 linhas.

Código 4.1 – FBE Sensor em LingPON

```
fbe Sensor
public boolean atIsRead = false
public boolean atIsActivated = false
private method mtProcess
    attribution
        this.atIsRead = true
        this.atIsActivated = false
    end_attribution
end_method
rule rlSensor
    condition
        premise prIsActivated
            this.atIsActivated == true
        end_premise
        and
        premise prIsNotRead
            this.atIsRead == false
        end_premise
    end_condition
    action sequential
        instigation sequential
            call this.mtProcess()
        end_instigation
    end_action
end_rule
end_fbe
```

Fonte: Fonte: Autoria própria**Código 4.2 – Código da estrutura do sensor com o Framework PON C++ 4.0**

```
struct NOPSensor{
    NOP::SharedAttribute<bool> atIsRead{ NOP::BuildAttribute(false) };
    NOP::SharedAttribute<bool> atIsActivated{ NOP::BuildAttribute(false) };
    NOP::SharedPremise prIsActivated{ NOP::BuildPremise<bool>(atIsActivated, true,
        NOP::Equal()) };
    NOP::SharedPremise prIsNotRead{ NOP::BuildPremise<bool>(atIsRead, false, NOP::Equal()) };
    NOP::SharedRule rlSensor{ NOP::BuildRule(
        NOP::BuildCondition<NOP::Conjunction>(prIsActivated, prIsNotRead),
        NOP::BuildAction(NOP::BuildInstigation([&]() { this->Read(); this->Deactivate(); })))
    };
    void Read() const { atIsRead->SetValue(true); }
    void Activate() const { atIsActivated->SetValue(true); atIsRead->SetValue(false); }
    void Deactivate() const { atIsActivated->SetValue(false); }
};
```

Fonte: Autoria própria

O Código 4.3 apresenta a estrutura utilizada para implementar o sensor com o *Framework PON C++ 2.0*. Comparando o Código 4.2 com o Código 4.3 é interessante observar que a implementação com o *Framework PON C++ 4.0* é mais simples que com a implementação com o *Framework PON C++ 2.0*, utilizando muito menos linhas de código. Por sua vez, o Código 4.4 apresenta a estrutura utilizada para implementar o sensor com o POO em C++, utilizando um total de apenas 8 linhas.

Código 4.3 – Código da estrutura do sensor com o Framework PON C++ 2.0

```
struct NOP2Sensor : public FBE
{
    Boolean* atIsActivated, atIsRead;
    Premise* prIsActivated, prIsNotRead;
```

```

Instigation* inSensor;
RuleObject* rlSensor;
Method* mtSensor;
NOP2Sensor() {
    BOOLEAN(this, atIsActivated, false);
    BOOLEAN(this, atIsRead, false);
    mtSensor = new MethodPointer<NOP2Sensor>(this, &NOP2Sensor::ProcessSensor);
}
void ProcessSensor() { atIsRead->setValue(true); atIsActivated->setValue(false); }

class SensorApp : public NOPApplication {
public:
    SensorApp(int count) : NOPApplication() {
        SingletonFactory::changeStructure(SingletonFactory::NOPVECTOR);
        SingletonScheduler::changeScheduler(SchedulerStrategy::NO_ONE);
        for (int i = 0; i < count; i++) {
            sensors.push_back(std::make_shared<NOP2Sensor>());
        }
        for (auto& sensor : sensors) {
            PREMISE(sensor->prIsActivated, sensor->atIsActivated, true,
                    Premise::EQUAL, Premise::STANDARD, false);
            PREMISE(sensor->prIsNotRead, sensor->atIsRead, false,
                    Premise::EQUAL, Premise::STANDARD, false);
            INSTIGATION_NAMED(sensor->inSensor, sensor->mtSensor);
            RULE(sensor->rlSensor, SingletonScheduler::getInstance(), Condition::CONJUNCTION);
            sensor->rlSensor->addPremise(sensor->prIsActivated);
            sensor->rlSensor->addPremise(sensor->prIsNotRead);
            sensor->rlSensor->addInstigation(sensor->inSensor);
            sensor->rlSensor->end();
        }
    }
    std::vector<std::shared_ptr<NOP2Sensor>> sensors;
};

Fonte: Autoria própria

```

Para testar esta aplicação são instanciados um total de 100.000 sensores com uma taxa de aprovação que varia para cada iteração de forma que a cada iteração apenas uma fração dos sensores é ativada². O gráfico da Figura 78 mostra o resultado dos testes, apresentando o tempo de execução com relação à taxa de aprovação.

Código 4.4 – Código da estrutura do sensor em POO

```

struct OOPSensor {
    inline static std::atomic<int> counter{ 0 };
    bool isRead{ false };
    bool isActivated{ false };
    void Read() { isRead = true; }
    void Activate() { isActivated = true; isRead = false; }
    void Deactivate() { isActivated = false; }
};

Fonte: Autoria própria

```

No gráfico da Figura 78 é possível observar como ambos os *frameworks* apresentam um comportamento similar, com desempenho superior ao POO para taxas de aprovação mais baixa e com desempenho inferior para taxas de aprovação mais alta, sendo que o tempo de execução cresce de maneira linear com a taxa de aprovação das regras, conforme esperado pela complexidade linear do PON. As aplicações utilizando o PON ainda apresentam tempos de execução superiores ao POO em determinados casos devido ao custo computacional em tempo

² Teste executado em um computador com processador Ryzen 5 3600 a 3.6GHz e 16 GB de RAM DDR4 a 3000MHz (dual channel) e sistema operacional Windows 10 64 bits.

de execução adicionado pela utilização das estruturas de dados utilizadas para a materialização do mecanismo de notificações. Isto fica mais evidente quanto mais sensores são ativados, amenizando assim o efeito das redundâncias estruturais e temporais do POO/PI. A Figura 79 detaca os tempos de execução com o *Framework PON C++ 4.0* relativos aos do *Framework PON C++ 2.0*, reforçando como o mesmo apresenta desempenho superior em todos os cenários.

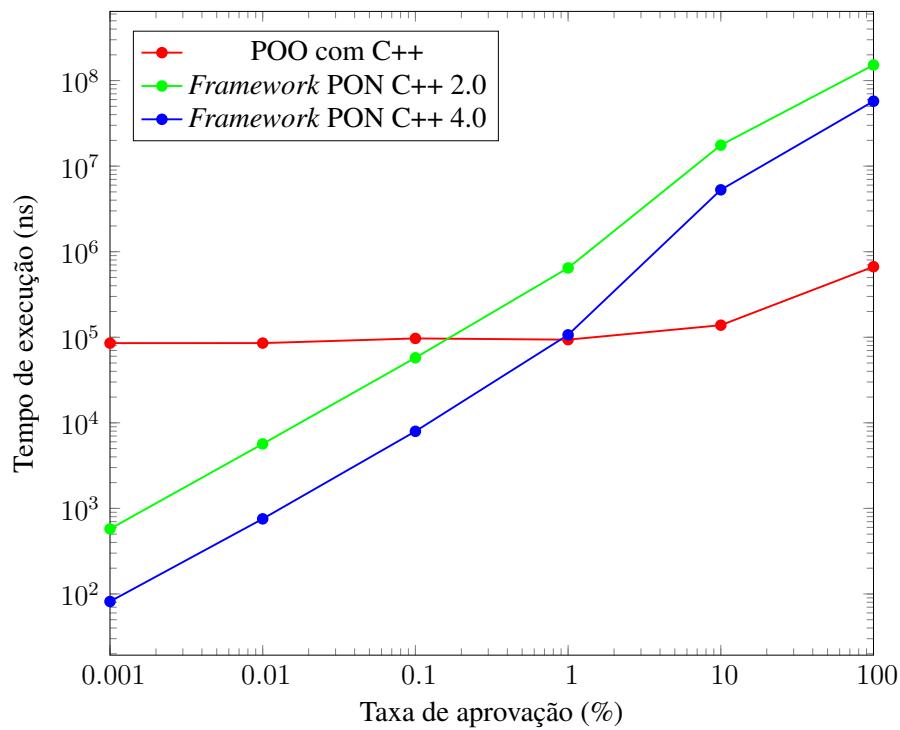


Figura 78 – Testes de desempenho da aplicação do sensor
Fonte: Autoria própria

Outra análise realizada neste mesmo cenário da aplicação de sensores foi o do consumo de memória das aplicações. Para este teste a aplicação instancia uma quantidade de 10.000 sensores, sem realizar a aprovação de nenhuma *Rule*. Essa instanciação dos sensores se repete ao longo de pelo menos 10 segundos, permitindo traçar o perfil de consumo de memória da aplicação, utilizando as ferramentas de análise do Visual Studio 2019.

O consumo de memória da aplicação em POO com C++ é mostrado na Figura 80. Naturalmente, devido à natureza da aplicação ser bastante simples, como não há uso de nenhum *framework*, atinge um máximo de 2 MB durante o teste. A baixa taxa de amostragem combinada ao rápido tempo de execução desta aplicação não permite observar as curvas no consumo de memória sendo alocada e desalocada neste cenário. É interessante saber o consumo de memória da aplicação em POO, pois esta serve como referência para comparar o consumo dos *frameworks*.

O consumo de memória da aplicação com o *Framework PON C++ 2.0* é mostrado na Figura 81, é interessante observar a curva crescente que demonstra que a memória não está sendo

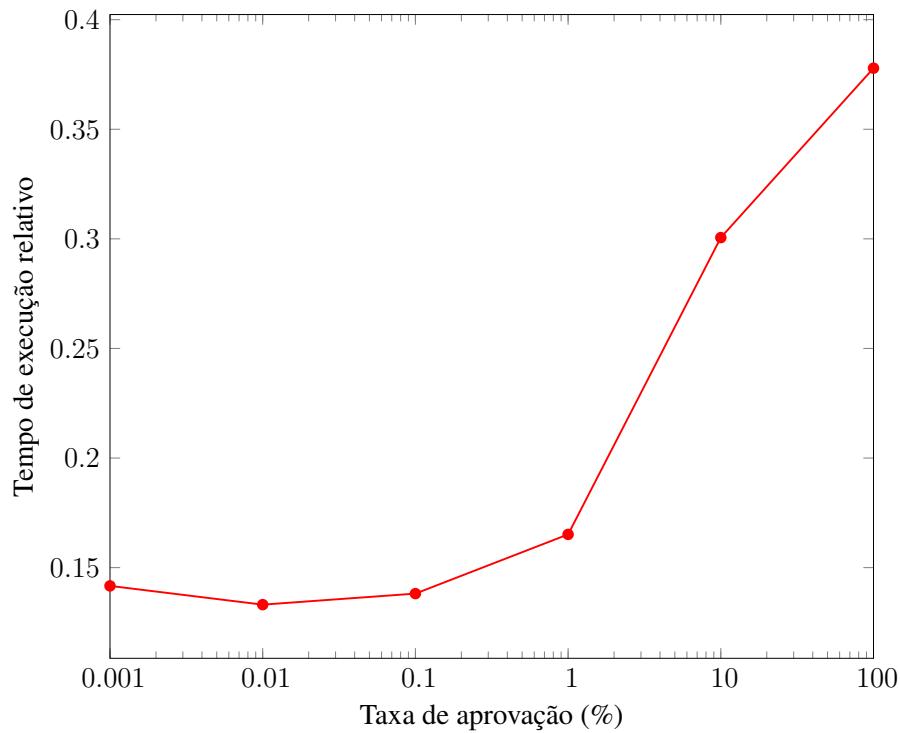


Figura 79 – Tempos de execução da aplicação do sensor com o *Framework PON C++ 4.0* relativo ao *Framework PON C++ 2.0*

Fonte: Autoria própria

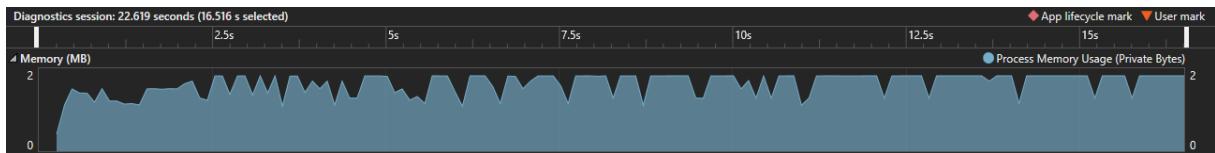


Figura 80 – Consumo de memória POO em C++

Fonte: Autoria própria

corretamente desalocada, fazendo com que o consumo de memória apenas aumente ao longo do tempo, até o ponto em que a memória do sistema se esgote e a aplicação falhe. Um teste isolado realizando a instanciação das entidades uma única vez apresentou um consumo de 69 MB de RAM. Xavier (2014) também já havia encontrado problemas no mecanismo de alocação de memória do *Framework PON C++ 2.0*, evidenciados neste experimento.

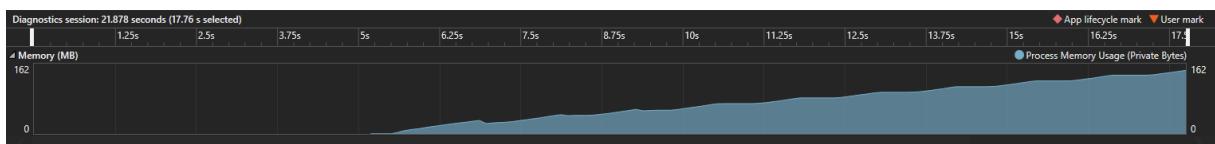


Figura 81 – Consumo de memória *Framework PON C++ 2.0*

Fonte: Autoria própria

O consumo de memória da aplicação com o *Framework PON C++ 4.0* é mostrado na Figura 82, na qual pode ser observado um comportamento esperado da memória sendo alocada

para as entidades até atingir cerca de 24 MB, e então sendo apropriadamente desalocada antes da nova iteração, demonstrando a estabilidade do gerenciamento de memória, sendo realizado com a utilização dos *smart pointers*.

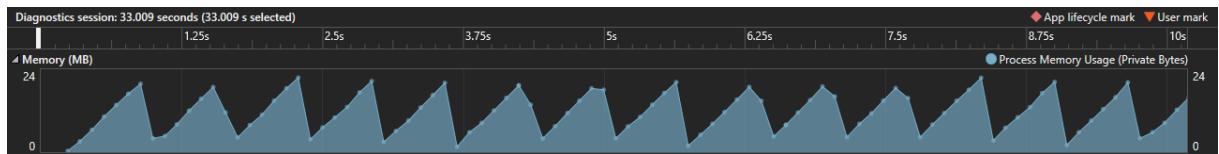


Figura 82 – Consumo de memória Framework PON C++ 4.0

Fonte: Autoria própria

Desta forma, fica explícito que o *Framework PON C++ 4.0* apresenta ganhos de desempenho significativos com relação ao *Framework PON C++ 2.0*. No cenário da aplicação de sensores, quando comparado ao *Framework PON C++ 2.0*, o *Framework PON C++ 4.0* reduziu em 62% o tempo de execução no pior caso, e em 85% no melhor caso. Além da redução nos tempos de execução, também reduziu em quase dois terços o consumo de memória e não apresentou o problema de vazamento de memória apresentado no gráfico da Figura 81.

Uma aplicação de cunho similar, porém com contexto ainda mais restrito ao grupo de pesquisa, a aplicação mira ao alvo. Esta aplicação apresenta nível de complexidade e lógica similares à aplicação de sensores e é apresentada como curiosidade no Apêndice F.

4.1.2 Aplicação *Bitonic Sort*

O *Bitonic Sort* é um algoritmo de ordenação originalmente proposto por Batcher (1968). Esta seção discorre sobre os detalhes do algoritmo, assim como sua implementação em PON, apresentando comparações com a implementação tradicional do algoritmo em linguagem de programação C.

Uma sequência é dita como *bitonic* caso sua primeira parte seja crescente e a segunda parte decrescente, de forma que para uma sequência $[0 \dots n - 1]$ a mesma é *bitonic* caso exista um índice i nos limites $0 \leq i \leq n - 1$ de modo que $x_0 \leq x_1 \leq \dots \leq x_i$ e $x_i \geq x_{i+1} \geq \dots \geq x_{n-1}$. Como exemplo, a sequência $[5, 6, 7, 8, 4, 3, 2, 1]$ é *bitonic*, pois pode ser dividida em duas sequências $[5, 6, 7, 8]$ e $[1, 2, 3, 4]$, que são, respectivamente, crescentes e decrescentes.

Desta forma, uma sequência *bitonic* pode ser ordenada por um conjunto de comparadores que operam em pares de valores da sequência, no contexto da chamada ordenação *bitonic*. O algoritmo de ordenação *Bitonic Sort* pode ser dividido em duas etapas, sendo que primeiramente

a sequência é transformada em uma sequência *bitonic* (etapa 1) e subsequentemente essa sequência *bitonic* é ordenada de forma crescente (etapa 2). Estas duas etapas são ilustradas na Figura 83. Os comparadores dessas etapas podem ser executados de forma paralela permitindo melhor desempenho do algoritmo em ambientes multiprocessados.

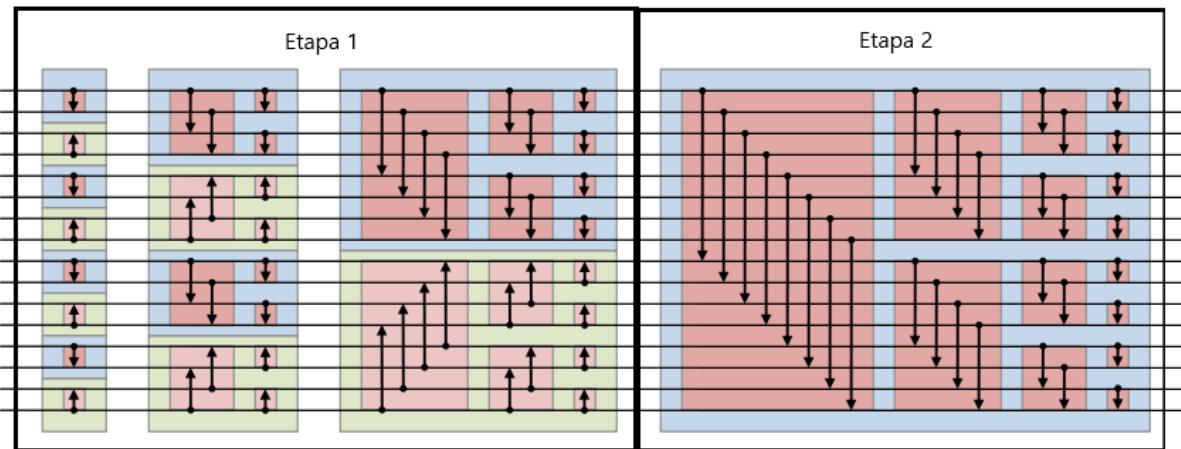


Figura 83 – Processo de ordenação com Bitonic Sort

Fonte: Adaptado de Mullapudi (2014)

O algoritmo do *Bitonic Sort* pode ser dividido em vários estágios, dado o fato que as comparações são realizadas entre dois elementos e os mesmos são comparados apenas uma vez dentro de cada estágio, de tal modo que as comparações dentro de cada estágio podem ser realizadas de forma paralela. A existência desses estágios paralelizáveis favorece uma implementação em PON devido ao seu paralelismo intrínseco. Deste modo, os comparadores do *Bitonic Sort* podem ser representados por meio de *Rules* (PORDEUS, 2020; PORDEUS *et al.*, 2021).

Em uma interpretação para a implementação em PON os diferentes estágios podem ser representados por *Attributes* independentes, de modo que a ordenação ocorre movendo os valores do estágio anterior para o próximo. Assim, cada comparador é materializado como um *FBE* com duas *Rules*, conforme ilustrado na Figura 84, sendo que uma *Rule* opera no caso da comparação ser verdadeira e a outra no caso da comparação ser falsa. Isto visto que pela divisão em etapas, é necessário mover os valores da etapa anterior para a próxima, invertendo os valores apenas quando a comparação é verdadeira. Os métodos *mtMove* e *mtSwap* são responsáveis por mover os valores de *at1* e *at2* do *FBE* do comparador atual para os *Attributes* dos comparadores das etapas seguintes.

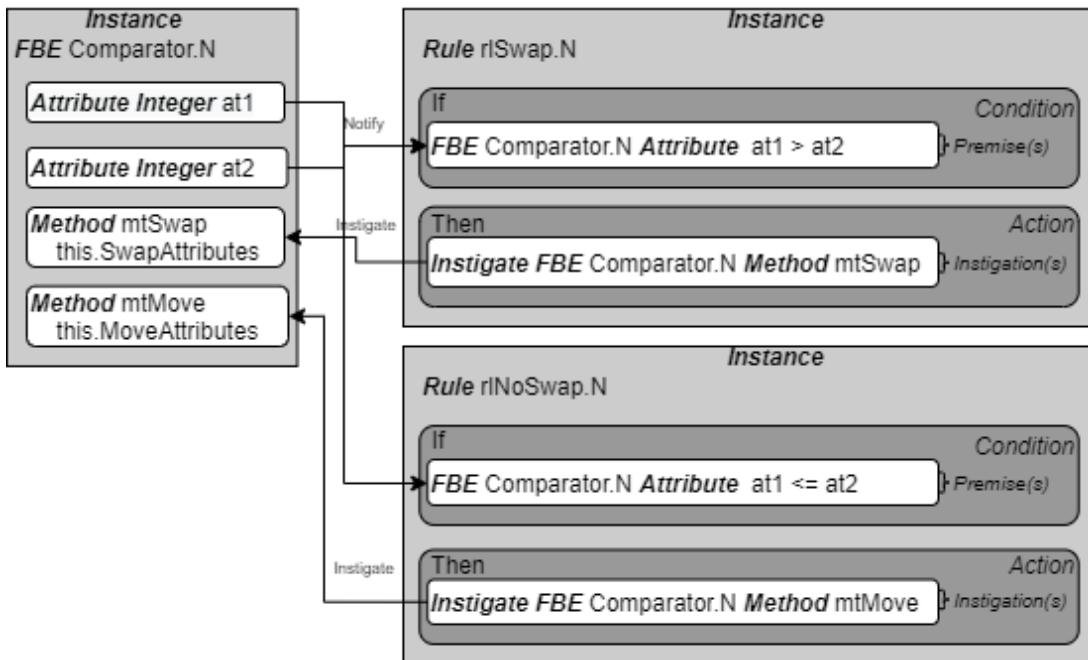


Figura 84 – Rules para comparador do Bitonic Sort em PON
Fonte: Autoria própria

Na estrutura do ordenador implementado com o *Framework PON C++ 4.0*, todas as entidades do PON são criadas de forma dinâmica na construtora da *struct*, baseando-se no número de elementos, sendo passado como parâmetro. A função *Sort*, por sua vez, atribuí os valores aos *Attributes* pertinentes ao primeiro estágio de comparação, conforme mostrado no Código 4.5, e a ordenação é realizada pela interação das entidades do PON já declaradas na construção do objeto, ao final da operação retornando o vetor *out* que armazena os resultados da ordenação. Devido à extensão dos códigos, essa seção traz apenas trechos das implementações, com o código completo disponível para referência no Apêndice G.

Código 4.5 – Trecho de código da estrutura NOPBitonicSorter

```
std::vector<T> NOPBitonicSorter::Sort(const std::vector<T>& input) {
    for (auto i = 0; i < input.size(); i++) {
        elements[1][i] -> SetValue(input[i]);
    }
    return out;
}
```

Fonte: Autoria própria

Essa implementação apresenta elementos completamente independentes e paralelizáveis, de modo que qualquer alteração no estado de um *Attribute* do estágio de entrada do ordenador é refletida na saída do mesmo. Entretanto, esta implementação em PON apresenta desempenho ruim, devido ao fato de ser necessário mover os dados entre os diferentes estágios. A operação de mover os dados necessita da aprovação de uma *Rule* para cada comparação para

sua execução. Além disso, também pode ocorrer execução de *Rules* desnecessárias decorrentes da ordem de avaliação dos comparadores, porque os valores nos estágios intermediários são substituídos pelos estágios anteriores à medida que as *Rules* são aprovadas, causando a aprovação de regras desnecessárias durante o processo. Dados estes problemas, se torna necessária uma nova interpretação deste problema em PON por meio da proposta de uma solução mais eficiente.

Esta solução mais eficiente proposta se aproveita da característica da divisão em estágios do algoritmo, desta vez não utilizando *Attributes* para os estágios intermediários, mas sim uma *Premise* adicional aos comparadores que controla sua execução apenas durante o seu estágio. Desta forma também é possível reduzir o número de *Rules* pela metade, utilizando apenas uma *Rule* por comparador. Esta interpretação é representada na Figura 85.

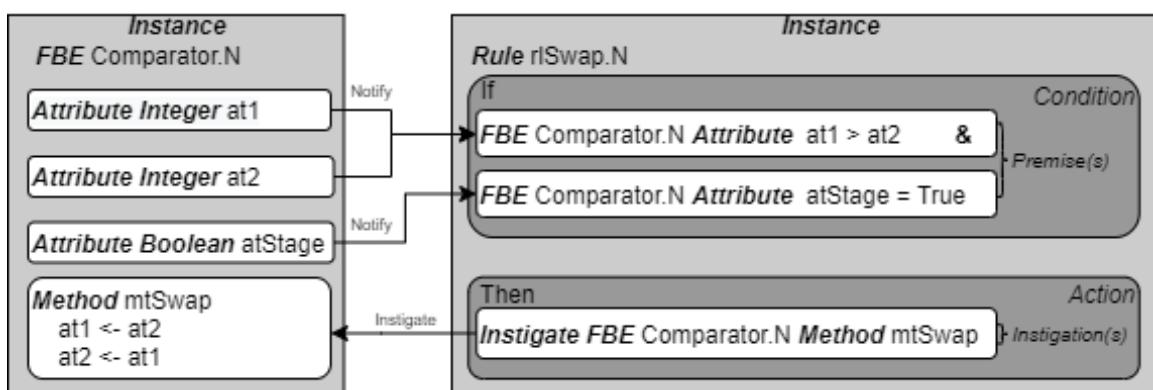


Figura 85 – Rules para implementação mais eficiente do comparador do Bitonic Sort em PON
Fonte: Autoria própria

O Código 4.6 mostra em detalhe o corpo da função de ordenação desta implementação, onde os valores dos *Attributes* são atribuídos de acordo com o vetor de entrada, porém a ordenação é realizada por meio da mudança do estado dos *Attributes* de controle de cada estágio para *true*.

Código 4.6 – Trecho de código da estrutura NOPBitonicSorterStages

```
template<typename T>
std::vector<T> NOPBitonicSorterStages::Sort (std::vector<T>& input)
{
    for (size_t i = 0; i < input.size(); i++)
    {
        elements[i]->SetValue(input[i], NOP::NoNotify);
    }
    for (const auto& stage : stages)
    {
        stage->SetValue(true);
        stage->SetValue(false);
    }
    for (auto i = 0; i < elements.size(); i++)
    {
        out[i] = elements[i]->GetValue();
    }
    return out;
}
```

Fonte: Autoria própria

A Figura 86 mostra como a implementação com a divisão em estágios apresenta tempos de execução significativamente menores que a implementação original em PON, sendo que para o caso da ordenação de 64 elementos a implementação em estágios chega a ser 20.000 vezes mais rápida.

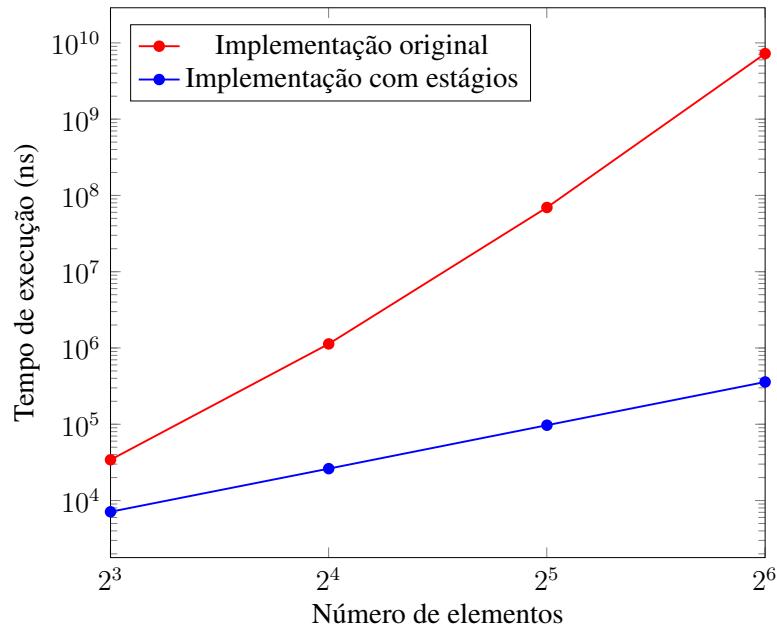


Figura 86 – Testes de desempenho da aplicação do Bitonic Sort com diferentes implementações em PON
Fonte: Autoria própria

Além disso, o número reduzido de *Rules* utilizadas faz com que a implementação em estágio tenha o consumo de memória cerca de 50% menor. Utilizando as ferramentas de análise do Visual Studio 2019 foi possível traçar o perfil do consumo de memória das aplicações. Neste teste, para o caso do ordenador de 1024 elementos, o total de memória alocado pela aplicação chega a 74 MB na implementação original e 35 MB na implementação em estágios. Esse alto consumo de memória é decorrente do grande número de elementos do PON utilizados na construção do programa, sendo que para o ordenador de 1024 elementos são necessários 28.160 comparadores. Os gráficos do consumo de memória para a implementação original e a implementação em estágios são apresentados nas Figuras 87 e 88 respectivamente.

Além das implementações em PON, foi escolhida uma implementação em C do algoritmo, disponível na íntegra no Apêndice H(PITSIANIS, 2008), de modo a servir como base de comparação para o desempenho da implementação em PON. Nos testes foi avaliado o tempo de execução para realizar a ordenação de 32, 64, 128, 256, 512, 1024, 2048 e 4096 elementos³. Na avaliação do tempo de execução da implementação em PON não é considerado o tempo gasto na

³ Teste executado em um computador com processador Ryzen 5 3600 a 3.6GHz e 16 GB de RAM DDR4 a 3000MHz (dual channel) e sistema operacional Windows 10 64 bits.



Figura 87 – Consumo de memória para a aplicação do algoritmo *Bitonic Sort* com o Framework PON C++ 4.0 na implementação original

Fonte: Autoria própria

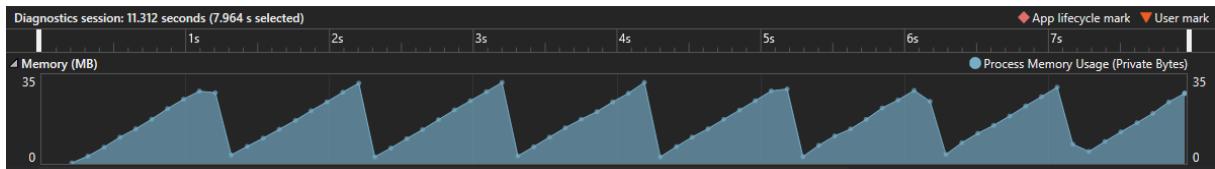


Figura 88 – Consumo de memória para a aplicação do algoritmo *Bitonic Sort* com o Framework PON C++ 4.0 na implementação em estágios

Fonte: Autoria própria

inicialização da estrutura em si, mas sim da execução da função *Sort*. Os resultados são exibidos no gráfico da Figura 89. Para esta comparação foi utilizada a implementação em estágios com o PON.

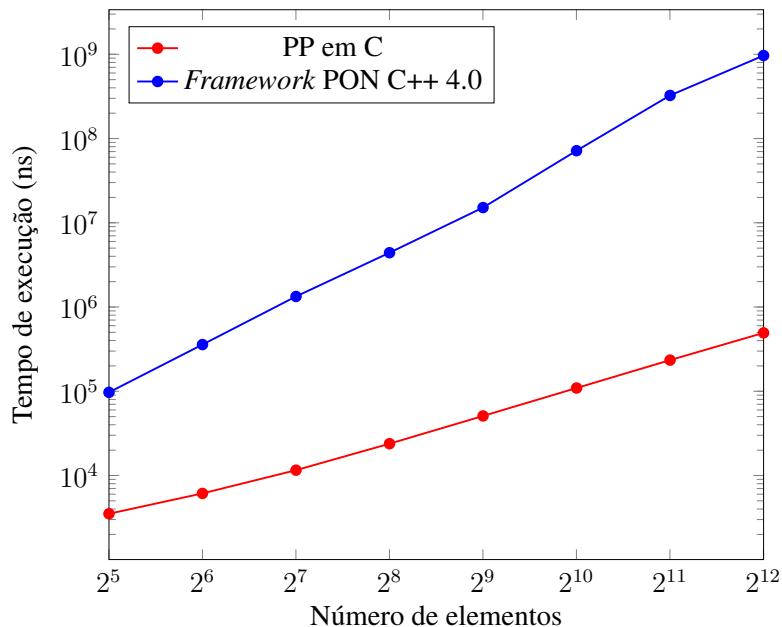


Figura 89 – Testes de desempenho da aplicação do algoritmo *Bitonic Sort*
Fonte: Autoria própria

O tempo de execução do processo de ordenação com o PON foi superior ao da implementação em C utilizando o PP e, quanto maior o número de elementos, maior a diferença entre os tempos de execução, chegando a uma diferença de 2000 vezes no caso de ordenação de 4096 elementos. O problema da implementação em PON se dá devido ao elevado número de comparadores, que por sua vez representam um número elevado de entidades do PON instanciadas para a construção do comparador, cada uma com um significativo custo de memória associado, além do

elevado número de notificações necessárias para a realização do processo de ordenação.

De forma a aproveitar a natureza paralelizável do algoritmo, também é possível paralelizar a implementação em PON com o *Framework PON C++ 4.0*, fazendo usos do mecanismo de notificações paralelas, de forma que cada notificação pode ser processada em uma *thread* separada, potencialmente reduzindo os tempos de execução para aplicações com elevado número de entidades a serem notificadas.

Esta implementação é bastante simples com o *Framework PON C++ 4.0*, pois basta utilizar a função *SetValue* com um parâmetro de template *<NOP::Parallel>* para que as notificações sejam realizadas de forma paralela. O Código 4.7 mostra a função de ordenação de forma paralela, similar ao já apresentado no Código 4.6.

Código 4.7 – Bitonic Sort paralelizado com o *Framework PON C++ 4.0*

```
template<typename T>
std::vector<T> NOPBitonicSorterStages::Sort (std::vector<T>& input)
{
    for (size_t i = 0; i < input.size(); i++)
    {
        elements[i]->SetValue(input[i], NOP::NoNotify);
    }

    for (const auto& stage : stages)
    {
        stage->SetValue<NOP::Parallel> (true);
        stage->SetValue<NOP::Parallel> (false);
    }

    for (auto i = 0; i < elements.size(); i++)
    {
        out[i] = elements[i]->GetValue();
    }
}

return out;
}
```

Fonte: Autoria própria

No gráfico da Figura 90 é possível observar que a paralelização oferece ganhos de desempenho para a ordenação de números elevados de elementos, enquanto para números menores de elementos o tempo de execução aumentou. Por sua vez, na Figura 91, são comparados os tempos de execução da aplicação com paralelização relativos aos da execução sequencial. Para a ordenação de 4096 elementos, a execução de forma paralelizada teve seu tempo de execução reduzido em cerca de 50%.

Além de observar os tempos de execução, a fim de verificar os efeitos do paralelismo nessa aplicação, é possível observar a utilização de CPU durante a execução, também utilizando as ferramentas de análise do Visual Studio 2019, da mesma forma que o consumo de memória foi observado nas outras aplicações.

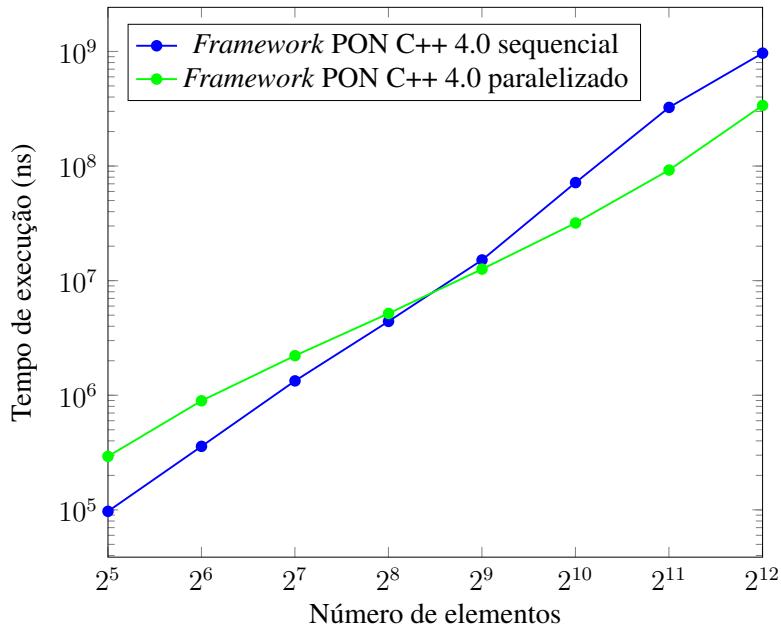


Figura 90 – Comparação da paralelização na aplicação do algoritmo Bitonic Sort
Fonte: Autoria própria

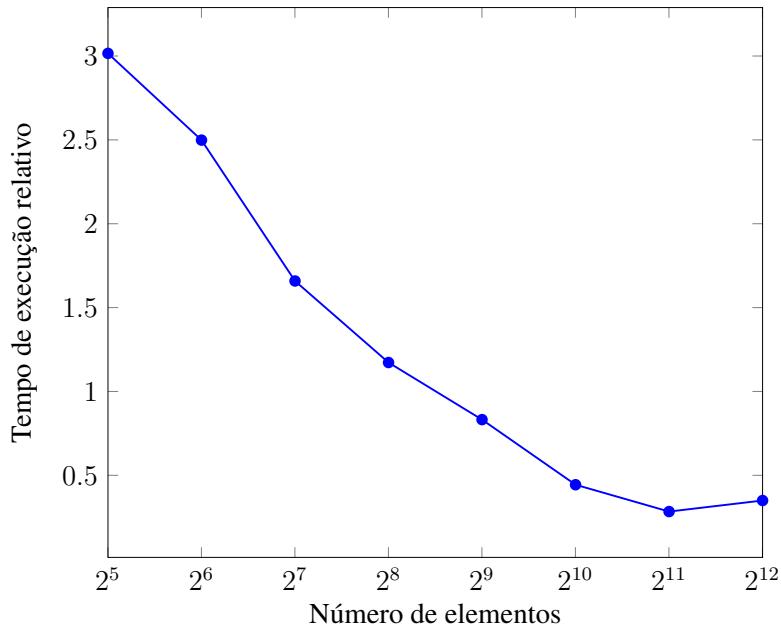


Figura 91 – Tempos de execução do algoritmo Bitonic Sort com o Framework PON C++ 4.0 paralelizado relativo ao sequencial
Fonte: Autoria própria

Ao avaliar o desempenho da aplicação para ordenar 4096 elementos durante a execução sequencial, com o gráfico da utilização de CPU na Figura 92, a utilização de CPU não passa de 8%. Isso se dá pelo fato do ambiente de testes possuir 12 núcleos, de modo que uma aplicação sem paralelismo executando de forma sequencial somente consegue utilizar um dos núcleos, equivalente a 8.33% do processamento disponível.

Já a implementação paralelizada consegue fazer uma utilização muito melhor da CPU.

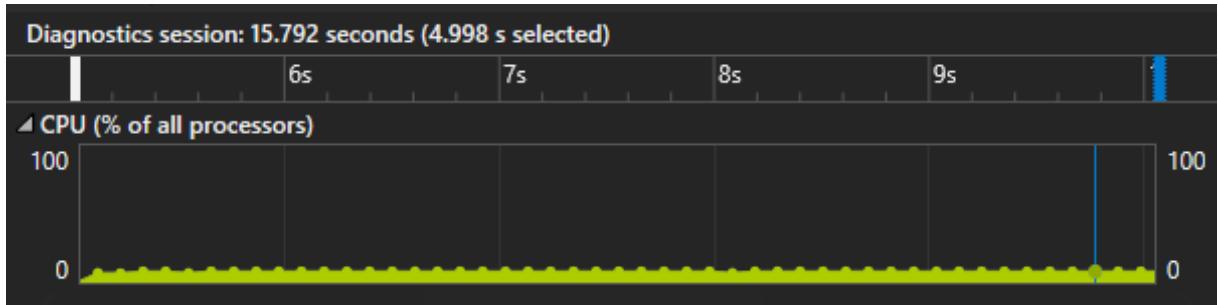


Figura 92 – Utilização de CPU durante execução do algoritmo *Bitonic Sort* com o Framework PON C++ 4.0 sequencial

Fonte: Autoria própria

Como ilustrado na 93, a utilização de CPU varia, atingindo quase 100% em alguns momentos. Ou seja, essa implementação utiliza de melhor forma os recursos disponíveis, com isso atingindo um desempenho melhor que a implementação sequencial.

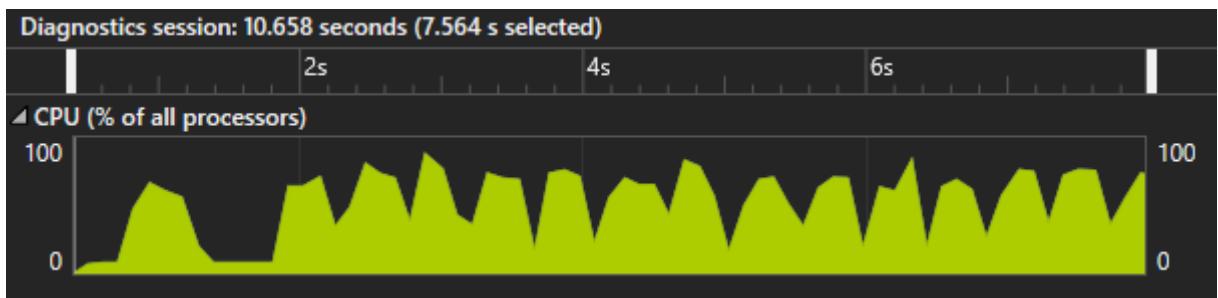


Figura 93 – Utilização de CPU durante execução do algoritmo *Bitonic Sort* com o Framework PON C++ 4.0 paralelizado

Fonte: Autoria própria

4.1.3 Aplicação *Random Forest*

O *Random Forest* é um algoritmo popular de aprendizado de máquina, utilizado em muitas aplicações para classificação e regressão. O algoritmo consiste em um conjunto de árvores de decisão, treinadas individualmente e combinadas para obter um resultado com menor erro de classificação/regressão. Para esta aplicação é comparado o desempenho da implementação em PON com implementações em linguagem de programação *Python* e *C*. Em tempo, *Python* é linguisticamente mais alto-nível que a linguagem *C* justamente.

Isto dito, no *Random Forest*, cada árvore é avaliada separadamente, viabilizando a execução de maneira paralela. Ao final da execução de todas as árvores os resultados são combinados para todo o conjunto (CRIMINISI *et al.*, 2011). A Figura 94 ilustra a estrutura das árvores de decisão (*decision trees*) independentes do *Random Forest*. No contexto do PON, a

implementação do *Random Forest* é interessante devido ao fato deste algoritmo ser construído essencialmente por meio de expressões lógicas *if-else*. Deste modo, o PON pode permitir eliminar as redundâncias temporais e estruturais, além de também permitir a paralelização da execução das árvores (PORDEUS, 2020).

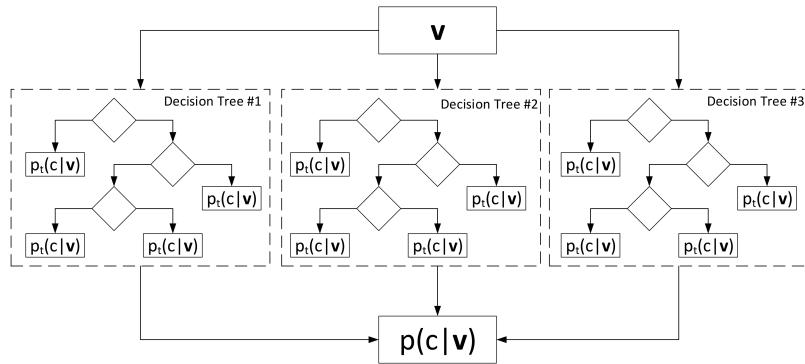


Figura 94 – Árvores de decisão do algoritmo *Random Forest*
Fonte: Pordeus (2020)

Uma implementação realizada originalmente por Pordeus (2020)⁴ possibilita a geração de código em PON com base nas árvores de decisão geradas com o auxílio da biblioteca scikit-learn⁵ na linguagem de programação Python. Esta implementação era capaz de gerar código específico para o PON em LingPON e PON-HD, assim como para o PP em linguagem de programação C. Esta implementação foi modificada para gerar código específico para o PON utilizando o Framework PON C++ 4.0, com base na implementação para geração de código em LingPON já existente. A implementação em proposta neste trabalho contempla apenas a utilização do algoritmo para a classificação de dados, sendo que o treinamento e geração das árvores de classificação são realizados por meio do uso da biblioteca scikit-learn.

No Código 4.8 é apresentado como exemplo uma das *Rules* geradas por esta implementação. Esta *Rule* é composta por diversas *Premises* que são capazes de avaliar o valor de *Attributes*, que representam os dados de entrada da classificação, com relação aos valores atribuídos durante o processo de treinamento das árvores. Ainda, devido a extensão dos códigos gerados aqui são disponibilizados apenas trechos, com o código-fonte completo para o caso de uma árvore disponível no Apêndice I.

Código 4.8 – *Rule* do algoritmo *Random Forest* para o Framework PON C++ 4.0

⁴ Implementação disponível em <https://github.com/leonardopordeus/RP>

⁵ Maiores detalhes sobre a biblioteca podem ser encontrados em <https://scikit-learn.org/stable/>

```

/*
rule rlTree_0_5
premise prTree_0_5_1
    this.attr3 > 75
end_premise
and
premise prTree_0_5_2
    this.attr2 <= 495
end_premise
and
premise prTree_0_5_3
    this.attr3 <= 165
end_premise
and
premise prTriggerTree0
    this.trigger_tree_0 == true
end_premise
end_condition
action sequential
    instigation sequential
        call this.mt_count_versicolor()
        call this.mtTrigger1()
        call this.mtRstTrigger0()
    end_instigation
end_action
end_rule
*/
NOP::SharedRule rlTree_0_5 = NOP::BuildRule(
    NOP::BuildCondition<NOP::Conjunction>(
        NOP::BuildPremise(attr3, 75, NOP::Greater()),
        NOP::BuildPremise(attr2, 495, NOP::LessEqual()),
        NOP::BuildPremise(attr3, 165, NOP::LessEqual()),
        NOP::BuildPremise(trigger_tree_0, true, NOP::Equal())
    ),
    NOP::BuildAction(
        NOP::BuildInstigation(mt_count_versicolor, mtTrigger1, mtRstTrigger0)
    )
);

```

Fonte: Autoria própria

O algoritmo *Random Forest* possibilita a utilização de um número variável de árvores de decisão. Dessa forma, com a implementação supracitada, também é possível gerar código para diferentes números de árvores. Nos testes foram utilizados os valores de 1, 10, 20, 50 e 100 árvores. Na Tabela 14 são relacionados os números de elementos necessários para a construção de cada versão do classificador com diferentes números de árvores.

Tabela 14 – Número de elementos em relação ao número de árvores
Fonte: Autoria própria

Número de árvores	1	10	20	50	100
Attributes	7	34	64	154	304
Premises	8	39	56	80	93
Conditions	9	90	173	437	839
Rules	9	90	173	437	839
Actions	6	60	120	300	600
Instigations	6	60	120	300	600
Methods	6	60	120	300	600

Esse elevado número de entidades e, por sua vez, de linhas de código faz com que o tempo de compilação dessa aplicação seja bastante elevado. No computador utilizado nos testes, com processador Ryzen 5 3600 e 16 GB de RAM DDR4, o tempo de compilação da aplicação de testes chegou a ser superior a 3 minutos, pois somente o arquivo contendo o código-fonte

definindo a estrutura das árvores contém mais de 32.000 linhas de código.

A Figura 95 exibe o resultado dos testes de tempo de execução das diferentes implementações do algoritmo *Random Forest*. As implementações em PP (C) e PON (C++) utilizam o código gerado, enquanto a aplicação em linguagem de programação Python utiliza a função disponível na própria biblioteca, *model.predict(data)*. O desempenho deste algoritmo em PON, com o uso do *Framework PON C++ 4.0*, foi muito inferior ao desempenho da implementação em linguagem de programação C. Ainda assim, os tempos de execução foram menores que os da aplicação em linguagem de programação Python.

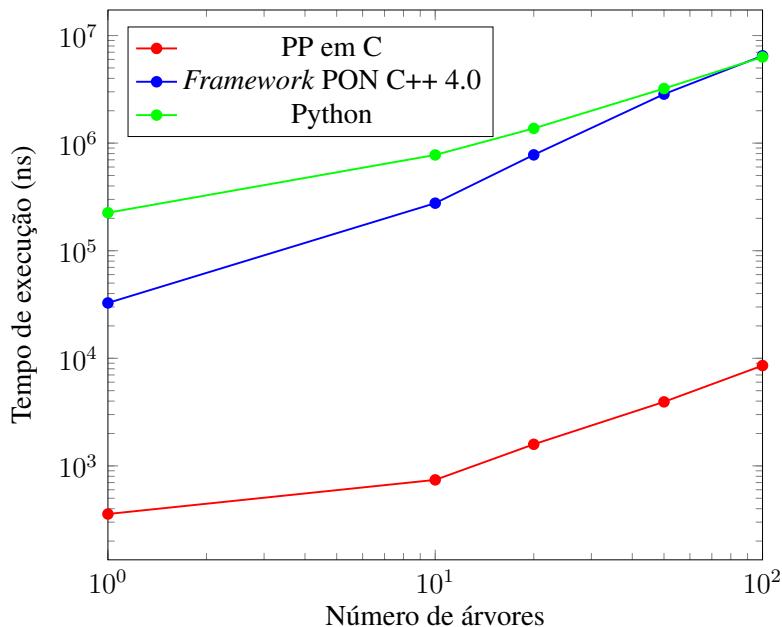


Figura 95 – Testes de desempenho da aplicação do algoritmo *Random Forest*
Fonte: Autoria própria

Além disso, ao contrário da aplicação *Bitonic Sort*, a aplicação de paralelismo, usando a função *SetValue<NOP::Parallel>*, não resultou em redução nos tempos de execução. Conforme pode ser observado na Figura 96 a implementação paralelizada aumentou os tempos de execução quando comparada com a implementação sequencial. Isso pode ser justificado pelo fato da implementação utilizar diversos *Attributes* e *Premises* de gatilho, que, na prática, forçam a execução de certas etapas a ocorrer de forma sequencial, limitando os benefícios da paralelização. Ainda, conforme o gráfico da Figura 97 que ilustra o tempo de execução da implementação paralelizada relativo ao da implementação sequencial, a implementação paralelizada apresenta tempos de execução pelo menos cinco vezes mais lentos.

Para avaliar a utilização de CPU foi utilizada a implementação do algoritmo utilizando 100 árvores. Do mesmo modo que a aplicação *Bitonic Sort*, a utilização de CPU para a imple-

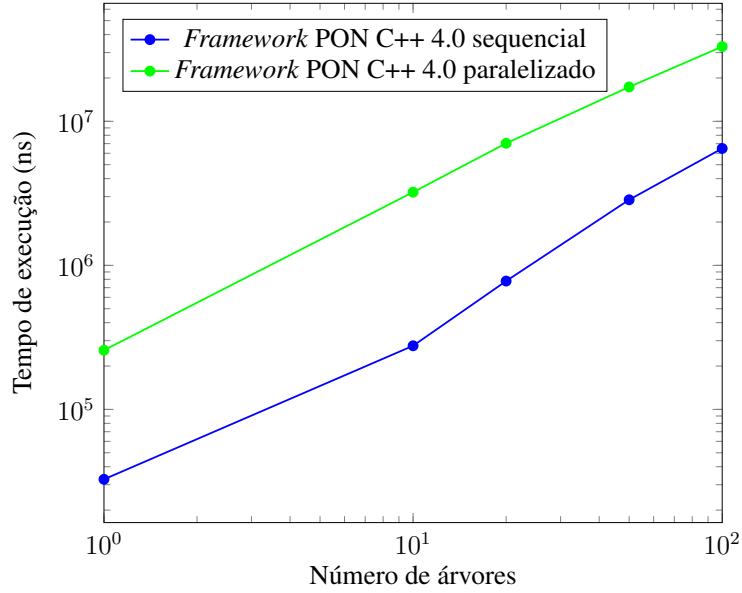


Figura 96 – Comparação da paralelização na aplicação do algoritmo *Random Forest*
Fonte: Autoria própria

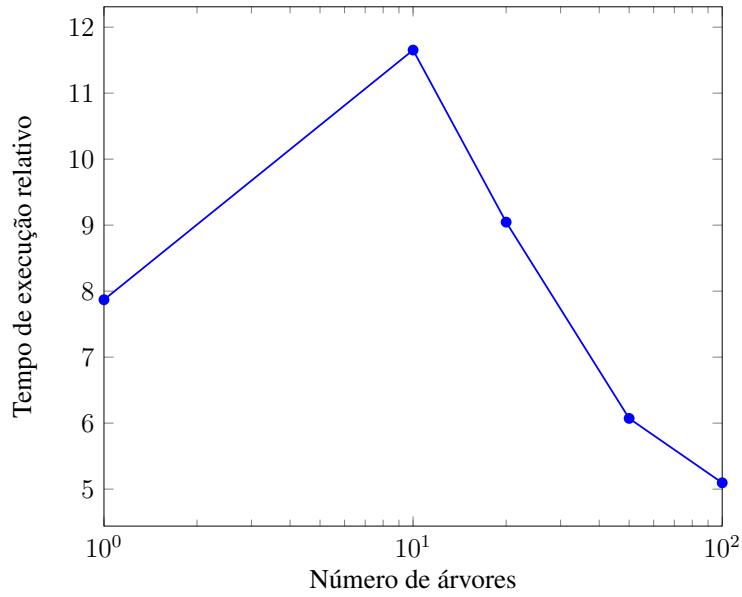


Figura 97 – Tempos de execução do algoritmo *Random Forest* com o Framework PON C++ 4.0 paralelizado relativo ao sequencial
Fonte: Autoria própria

mentação sequencial do *Random Forest* também fica limitada a 8%. Diferente do observado na aplicação *Bitonic Sort*, a utilização de CPU para a implementação sequencial do *Random Forest* não utiliza de forma eficiente todos os núcleos da CPU. De acordo com a Figura 99, o consumo de CPU fica próximo de apenas 50%. Isto é refletido no resultado dos tempos de execução que não apresentam redução na execução paralelizada.

Em suma, foi concluído que nesta implementação do algoritmo *Random Forest* em *Framework PON C++ 4.0*, a utilização de *Attributes* e *Premises* de gatilho limitam a paralelização. Essa limitação na paralelização faz com que não seja possível utilizar de forma equilibrada

todos os núcleos da CPU, de modo que o desempenho não apresenta benefícios com a execução paralelizada. De fato, ocorre justamente o contrário, pois o custo de execução dos mecanismos de paralelização sobrepassa os benefícios de desempenho da mesma, aumentando o tempo de execução total da aplicação.

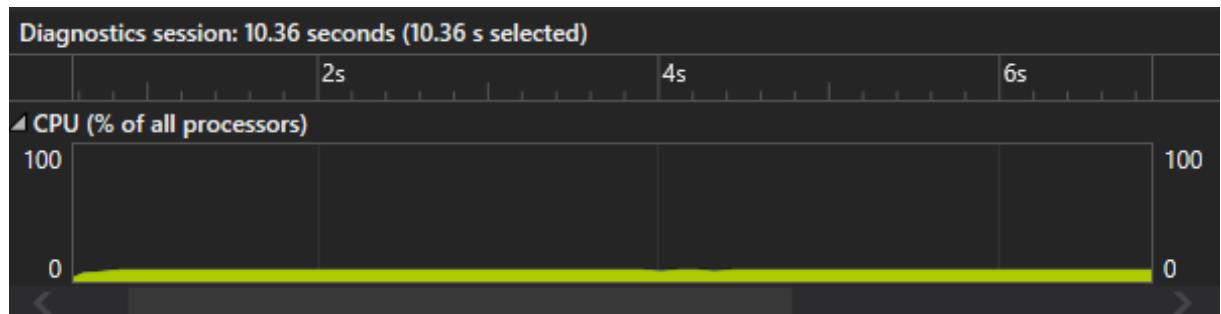


Figura 98 – Utilização de CPU durante execução do algoritmo *Random Forest* com o *Framework PON C++ 4.0* sequencial
Fonte: Autoria própria

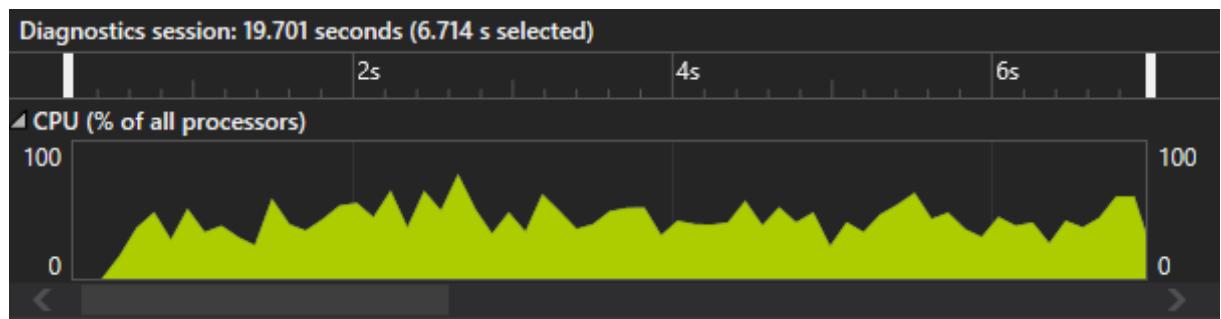


Figura 99 – Utilização de CPU durante execução do algoritmo *Random Forest* com o *Framework PON C++ 4.0* paralelizado
Fonte: Autoria própria

4.1.4 Aplicação de Controle de Tráfego Automatizado (CTA)

A aplicação de Controle de Tráfego Automatizado (CTA) é mais um caso de estudo frequentemente utilizado para avaliar o desempenho do PON. Esta aplicação foi escolhida por permitir avaliar o *Framework PON C++ 4.0* do ponto de vista de implementação do paralelismo, por meio da comparação com outro *framework* que materializa esta propriedade, o *Framework PON Elixir/Erlang* (NEGRINI, 2019).

O ambiente de simulação desta aplicação proposto por RENAUX *et al.* (2015) é composto por uma matriz 10x10, com um total de 100 interseções. As linhas e colunas da matriz

representam ruas enquanto as interseções representam os cruzamentos com semáforos. Uma representação gráfica deste ambiente é apresentada na Figura 100.

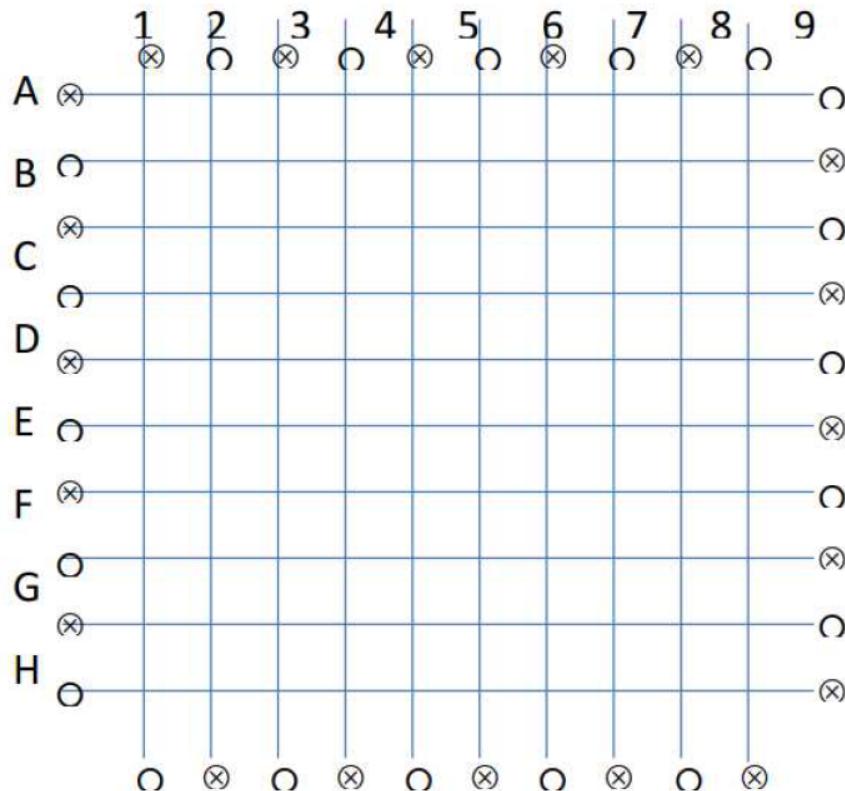


Figura 100 – Ambiente de simulação

Fonte: RENAUD et al. (2015)

Neste experimento são consideradas duas estratégias que regem o comportamento dos semáforos, a estratégia de controle independente e estratégia de controle baseado em congestionamento facilitado (CBCF). Na estratégia de controle independente cada semáforo possui tempos fixos para cada estado (NEGRINI, 2019). O diagrama de estados com a temporização para esta estratégia é apresentado na Figura 101. Neste diagrama as cores representam o estado em si (verde, amarelo e vermelho), enquanto os círculos H representam o estado do semáforo da via horizontal do cruzamento, e o círculo V representa o estado do semáforo da via vertical, de acordo com a disposição apresentada na Figura 100.

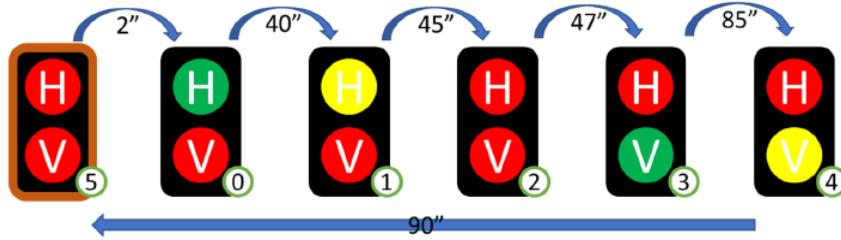


Figura 101 – Estados do CTA com estratégia independente

Fonte: Negrini (2019)

Na estratégia de controle CBCF, diferente da estratégia independente, um sensor é utilizado para detectar a porcentagem de veículos parados, sendo que se o sensor detecta que a porcentagem de veículos parados está acima de 60% e o tempo total do semáforo vermelho é menor que 30 segundos, ele tem seu tempo ajustado para 30 segundos. O diagrama de estados para esta estratégia é ilustrado na Figura 102.

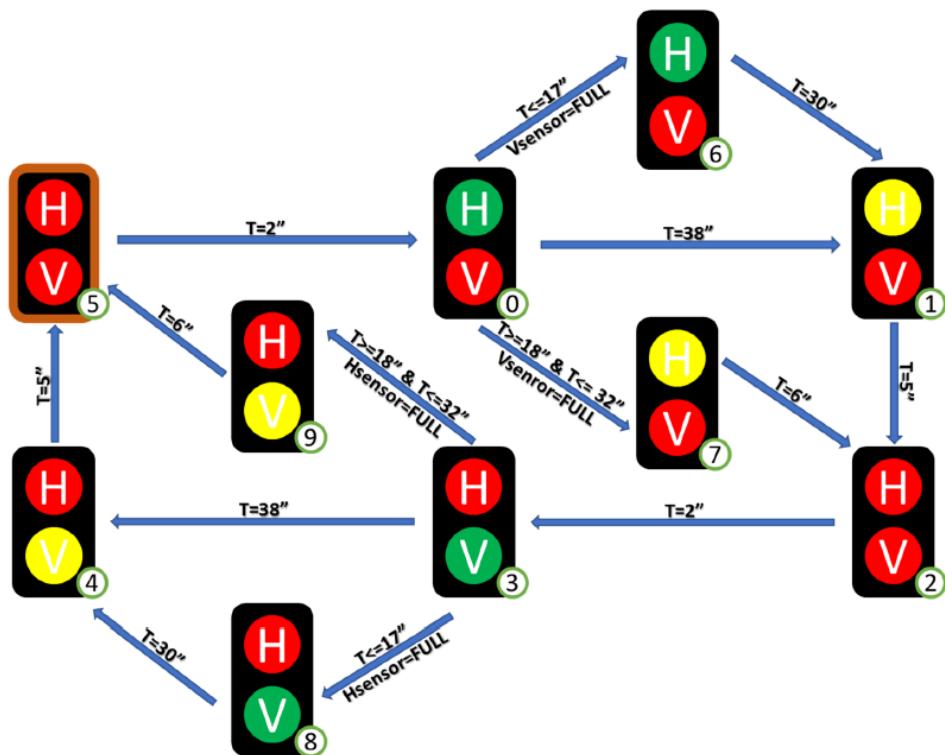


Figura 102 – Estados do CTA com estratégia CBCF

Fonte: Negrini (2019)

Para a execução deste experimento, foi utilizado código em LingPON tanto para a aplicação o *Framework PON C++ 4.0* como para o *Framework PON Elixir/Erlang*. Devido às limitações atuais no compilador LingPON para *Framework PON C++ 4.0*, o código foi modificado apenas de forma a utilizar o mecanismo de notificações paralelas (com *SetValue<NOP::Parallel>*). O Código 4.9 apresenta um trecho deste código em LingPON, enquanto o Código 4.10 apresenta

um trecho do código gerado em *Framework PON C++ 4.0*, contendo um *Method* e uma *Rule*, para o *FBE* do semáforo na estratégia independente. Os códigos-fonte completos são disponibilizados no Apêndice J.

Código 4.9 – Trecho do FBE para o CTA na estratégia independente em LingPON

```
fbe Semaphore_CTA
    private integer atSemaphoreState = 5
    public integer atSeconds = 0
    private method mtResetTimer
        assignment
            this.atSeconds = 0
        end_assignment
    end_method
    private method mtHorizontalTrafficLightGREEN
        assignment
            this.atSemaphoreState = 0
        end_assignment
    end_method
    ...
    rule rlHorizontalTrafficLightGreen
        condition
            premise prSeconds
                this.atSeconds == 2
            end_premise
            and
            premise prSemaphoreState
                this.atSemaphoreState == 5
            end_premise
        end_condition
        action sequential
            instigation parallel
                call this.mtHorizontalTrafficLightGREEN()
            end_instigation
        end_action
    end_rule
    ...
end_fbe
```

Fonte: Autoria própria

Código 4.10 – Trecho do FBE para o CTA na estratégia independente em *Framework PON C++ 4.0*

```
SemaphoreCTA::SemaphoreCTA()
: atSeconds{NOP::BuildAttribute<int>(0)},
  atSemaphoreState{NOP::BuildAttribute<int>(5)},
  prSeconds{NOP::BuildPremise<>(atSeconds, 2, NOP::Equal())},
  prSemaphoreState{
    NOP::BuildPremise<int>(atSemaphoreState, 5, NOP::Equal())},
  ...
  rlHorizontalTrafficLightGreen = NOP::BuildRule(
    NOP::BuildCondition(CONDITION(*prSeconds && *prSemaphoreState),
                         prSeconds, prSemaphoreState),
    NOP::BuildAction(NOP::BuildInstigation<NOP::Parallel>(
      METHOD(mtHorizontalTrafficLightGREEN())))
  );
  ...
}

void SemaphoreCTA::mtHorizontalTrafficLightGREEN()
{
  atSemaphoreState->SetValue<NOP::Parallel>(0);
}
```

Fonte: Autoria própria

O experimento utilizado para a avaliação de desempenho consiste em iterar quatro mil vezes em uma matriz 10x10 de semáforos, sendo que cada iteração incrementa o tempo do

Attribute atSeconds de cada semáforo. Desta forma, o experimento permite avaliar o tempo de execução da lógica, sem considerar o tempo de espera real entre os estados do semáforo, que existiriam em uma situação prática, mas que não são relevantes do ponto de vista de desempenho.

Para fins de comparação com o *Framework PON Elixir/Erlang* são considerados os resultados apresentados por Negrini (2019) utilizando o ambiente VM16, que utiliza uma instância de ambiente virtualizado na nuvem da Amazon com processadores AMD EPYC série 7000 com 16 núcleos a uma frequência de *clock* de 2,5 GHz em todos os núcleos (NEGRINI, 2019), enquanto os experimentos com o *Framework PON C++ 4.0 Framework* utilizaram um processador Ryzen 5 3600 com 6 núcleos e 12 *threads* a 3,6 GHz.

O consumo de memória da aplicação durante a execução com a estratégia de controle independente atingiu um máximo de 2,7 MB, enquanto para a estratégia de controle CBCF o consumo de memória checou a 4,8 MB, devido ao maior número de *Rules* utilizadas neste método de controle. O consumo de memória durante a execução das duas estratégias é mostrado nas Figuras 103 e 104. A aplicação com o *Framework PON Elixir/Erlang* consumiu, respectivamente, cerca de 800 MB e 700 MB nas estratégias independente e CBCF.

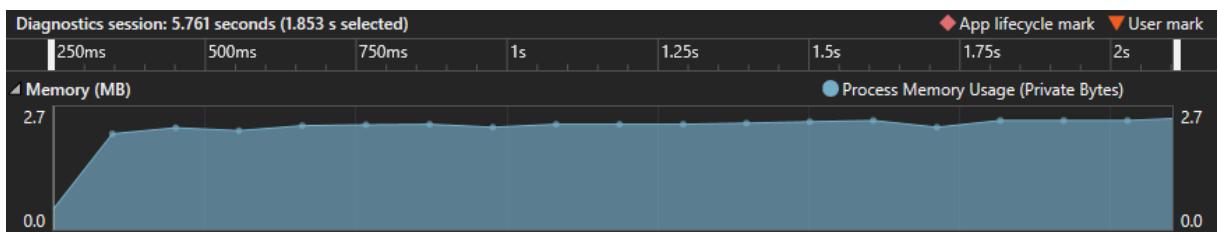


Figura 103 – Consumo de memória para a aplicação de semáforo com estratégia independente com o Framework PON C++ 4.0

Fonte: Autoria própria

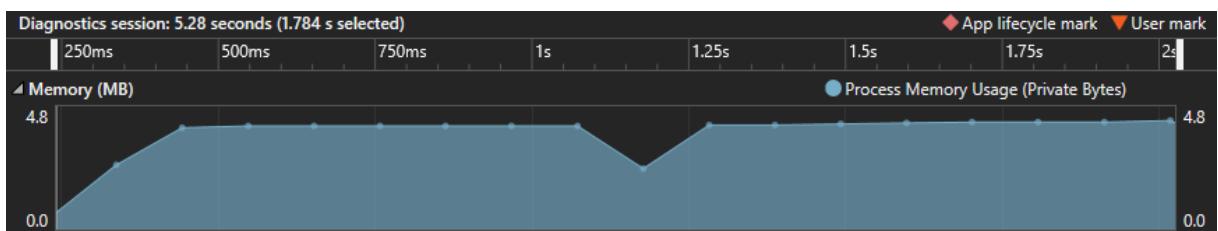


Figura 104 – Consumo de memória para a aplicação de semáforo com estratégia CBCF com o Framework PON C++ 4.0

Fonte: Autoria própria

De modo a observar a paralelização o uso de CPU também foi observado o uso de CPU durante a execução das duas estratégias. Como pode ser observado nos gráficos das Figuras 105 e 106, o uso de CPU oscila entre cerca de 40% e 70% em ambos os cenários.

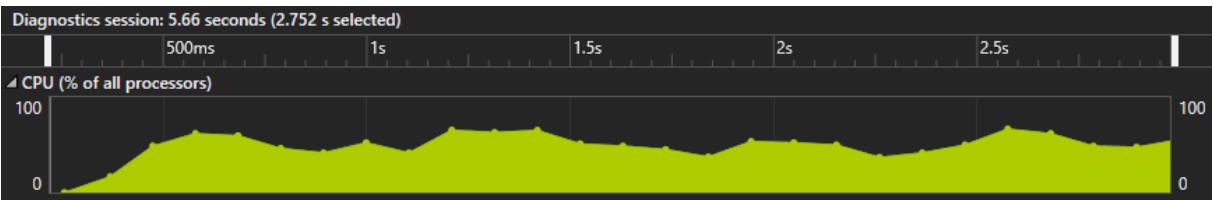


Figura 105 – Uso de CPU durante execução do Semáforo com estratégia independente com o Framework PON C++ 4.0

Fonte: Autoria própria

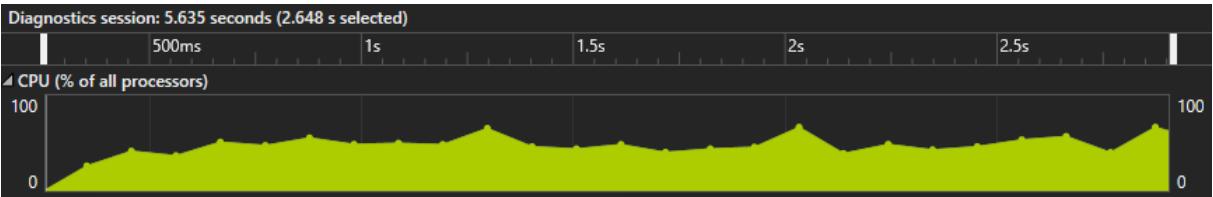


Figura 106 – Uso de CPU durante execução de semáforo com estratégia CBCF com o Framework PON C++ 4.0

Fonte: Autoria própria

O tempo de execução das estratégias de controle independente e CTBF foram de 336 ms e 434 ms respectivamente, utilizando o *Framework PON C++ 4.0*. No caso dos testes com o *Framework PON Elixir/Erlang* os tempos de execução foram de 4.703 ms e 8.525 ms. Esses resultados são comparados no gráfico da Figura 107. A aplicação com o *Framework PON C++ 4.0* apresenta tempos de execução uma ordem de grandeza inferiores aos tempos de execução da mesma aplicação com o *Framework PON Elixir/Erlang*, mesmo considerando a execução da aplicação em Elixir/Erlang em um ambiente com poder de processamento muito superior, além de apresentar consumo de memória mais de cem vezes menor.

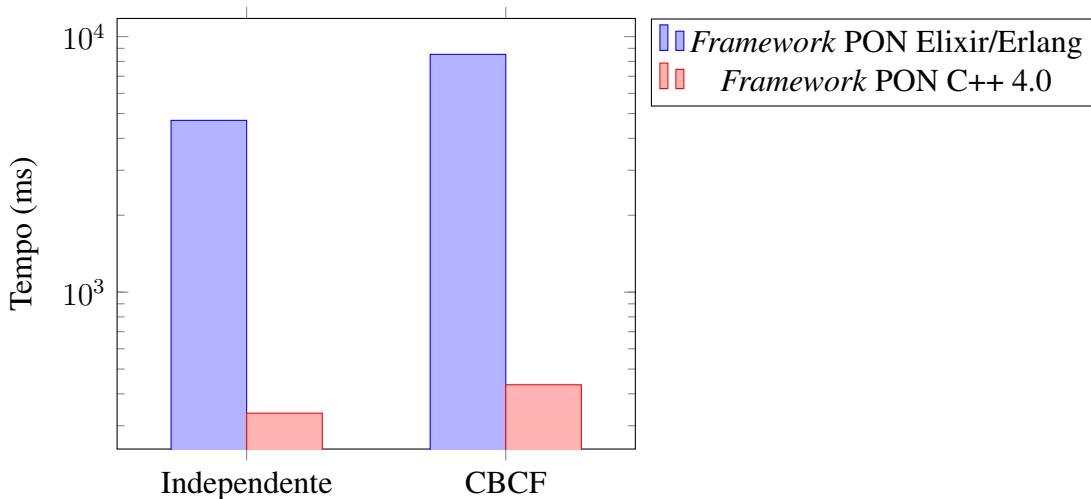


Figura 107 – Tempo de execução da aplicação de semáforo com o Framework PON C++ 4.0 Framework PON Elixir/Erlang

Fonte: Autoria própria

O desempenho superior do *Framework PON C++ 4.0* era esperado, visto que Elixir

é uma linguagem de programação de mais alto nível, utilizando o PF, com maior grau de abstração, que por sua vez é associada a custos computacionais elevados quando comparado a uma linguagem como C++. O objetivo desta comparação não é apenas observar como aplicações em C++ possuem desempenho superior a aplicações com Elixir/Erlang, pois ambas as linguagens de programação têm propostas diferentes, mas sim demonstrar no contexto do PON como o *Framework PON C++ 4.0* permite o desenvolvimento de aplicações que usam paralelismo ao mesmo tempo que consegue entregar um alto desempenho com baixos tempos de execução e baixo consumo de memória, o que não é possível com outros *frameworks* do PON.

4.2 JOGO NOPUNREAL COM *FRAMEWORK PON C++ 4.0*

O jogo apresentado na Seção 2.4.1.6.2, desenvolvido com o *Framework PON C++ 2.0*, foi reimplementado com o *Framework PON C++ 4.0* de forma a permitir comparar a verbosidade do código nas duas implementações. A implementação é feita em parte diretamente no POO, utilizando as bibliotecas da Unreal Engine principalmente para os cálculos matemáticos e implementações de nível gráfico da *engine* em si, enquanto a parte do jogo relativa ao PON é dada pela lógica de controle automatizado dos inimigos. No total, o jogo contém 17 *Rules* em sua composição.

A implementação com o *Framework PON C++ 4.0* utiliza como base a implementação original com o *Framework PON C++ 2.0*, utilizando a mesma estrutura de classes, apenas modificando o código referente às *Rules*. Desta forma o diagrama de classes, apresentado na Figura 108, é equivalente ao da Figura 46. Como ambos os *frameworks* apresentam funcionalidades equivalentes o processo de implementação com o *Framework PON C++ 4.0* foi bastante simples, bastando converter o formato de código das estruturas do PON para o modelo do novo *framework*. Esse processo levou apenas uma tarde de trabalho. Como resultado, as funcionalidades do jogo foram mantidas.

Foi comparada a diferença na quantidade de código necessária para implementar o jogo utilizando o POO em C++, o *Framework PON C++ 2.0* e o *Framework PON C++ 4.0*. Utilizando a ferramenta *cloc*⁶, foi avaliado o número de linhas contidas nos códigos que compõem cada uma destas implementações. Conforme apresentado na Tabela 15, a implementação com o *Framework PON C++ 4.0* conseguiu reduzir em 14.75% o número de linhas de código quando comparado com a implementação com o *Framework PON C++ 2.0*, e possui apenas 7.6% mais linhas de

⁶ Ferramenta disponível em <https://github.com/AlDanial/cloc>

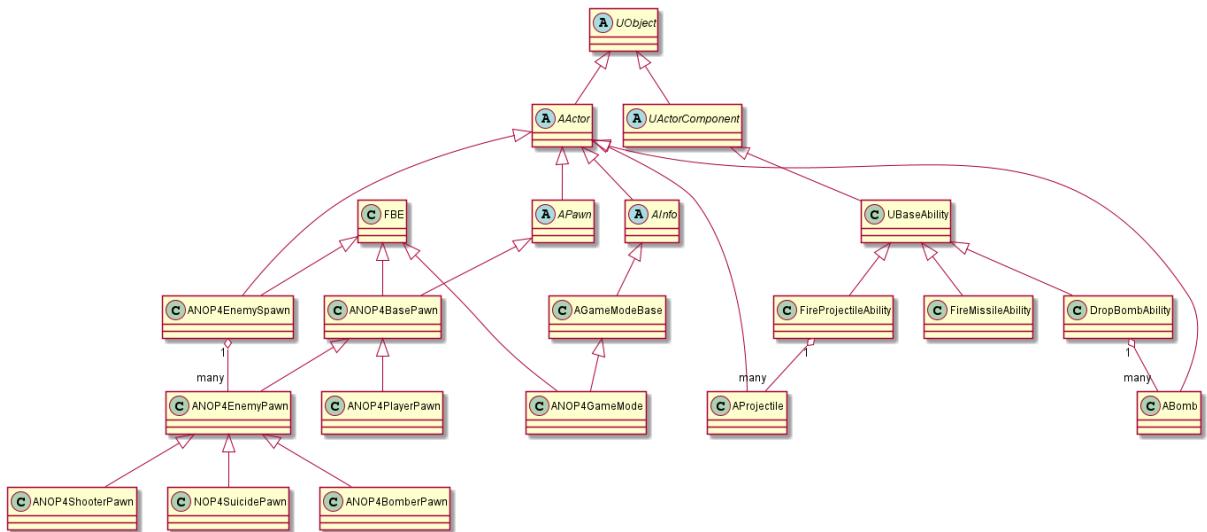


Figura 108 – Diagrama de classes do jogo desenvolvido com o *Framework PON C++ 4.0*
Fonte: Autoria própria

código que a implementação com o POO em C++. A implementação com o *Framework PON C++ 4.0* permitiu converter diretamente o código com o *Framework PON C++ 2.0* ao mesmo tempo que simplificou o código reduzindo o número de linhas de código totais.

Tabela 15 – Linhas de código para a composição do jogo NOPUnreal
Fonte: Autoria própria

Versão	Linhas de código
POO em C++	1407
<i>Framework PON C++ 2.0</i>	1776
<i>Framework PON C++ 4.0</i>	1514

4.3 PESQUISA DE OPINIÃO DE DESENVOLVEDORES

De forma a permitir avaliar os resultados referentes à facilidade de uso de *framework*, que é algo que não há como avaliar de forma objetiva, apenas subjetiva, foi montado um questionário com diversas questões referentes ao uso dos *frameworks* do PON. O questionário completo está disponível no Apêndice K. O resultado desta pesquisa é apresentado meramente a fim de curiosidade, por contar com uma amostra muito pequena sem significância estatística devido ao pequeno número de desenvolvedores que já utilizaram o *Framework PON C++ 4.0*. O questionário contou com apenas três respostas⁷, sendo que todas estas três pessoas possuem experiência utilizando tanto o *Framework PON C++ 4.0* como o *Framework PON C++ 2.0*.

⁷ Respostas dos alunos do grupo de pesquisa do PON Lucas Skora, Gustavo Chierici e Lucas Mamann.

Foi perguntado aos desenvolvedores, numa escala de 0 a 5, o quanto consideravam que o *Framework PON C++ 4.0* apresenta melhorias sobre o *Framework PON C++ 2.0*. Como pode ser observado na Figura 109, é consenso entre os desenvolvedores que o *Framework PON C++ 4.0* apresenta melhorias significativas sobre o *Framework PON C++ 2.0*.

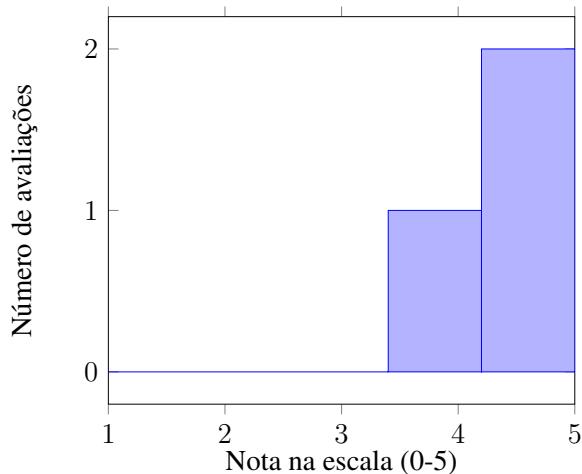


Figura 109 – Resultado da pesquisa de avaliação da melhoria do *Framework PON C++ 4.0* sobre o *Framework PON C++ 2.0*

Fonte: Autoria própria

Em maiores detalhes, foi requisitado aos desenvolvedores avaliar ambos os *frameworks* de acordo com diversos critérios, como facilidade de aprendizado, facilidade de uso, verbosidade, versatilidade, desempenho e aderência aos fundamentos do PON. Para cada critério o usuário pode responder com fraco, moderado, satisfatório, muito bom ou excelente. Para facilitar a visualização dos resultados as respostas foram convertidas para uma escala de 1 a 5, onde 1 representa fraco, e 5 excelente, e o resultado considera a média aritmética das respostas. Conforme apresentado no gráfico da Figura 110, o *Framework PON C++ 4.0* é considerado superior em todos os critérios avaliados.

4.4 REFLEXÕES SOBRE OS RESULTADOS OBTIDOS

Por meio dos testes unitários realizados, é possível afirmar que o *Framework PON C++ 4.0* consegue atender de forma satisfatória as funcionalidades necessárias para a construção de aplicações em PON. Considerando as implementações atuais, é possível comparar a diferença na quantidade de código necessária para implementar o *Framework PON C++ 2.0* e o *Framework PON C++ 4.0*. Utilizando a ferramenta *cloc*, foi avaliado o número de linhas contidas nos códigos que compõem o *Framework PON C++ 2.0* e o *Framework PON C++ 4.0*. Nessa comparação são consideradas apenas as linhas de código, desconsiderando linhas em branco e comentários.

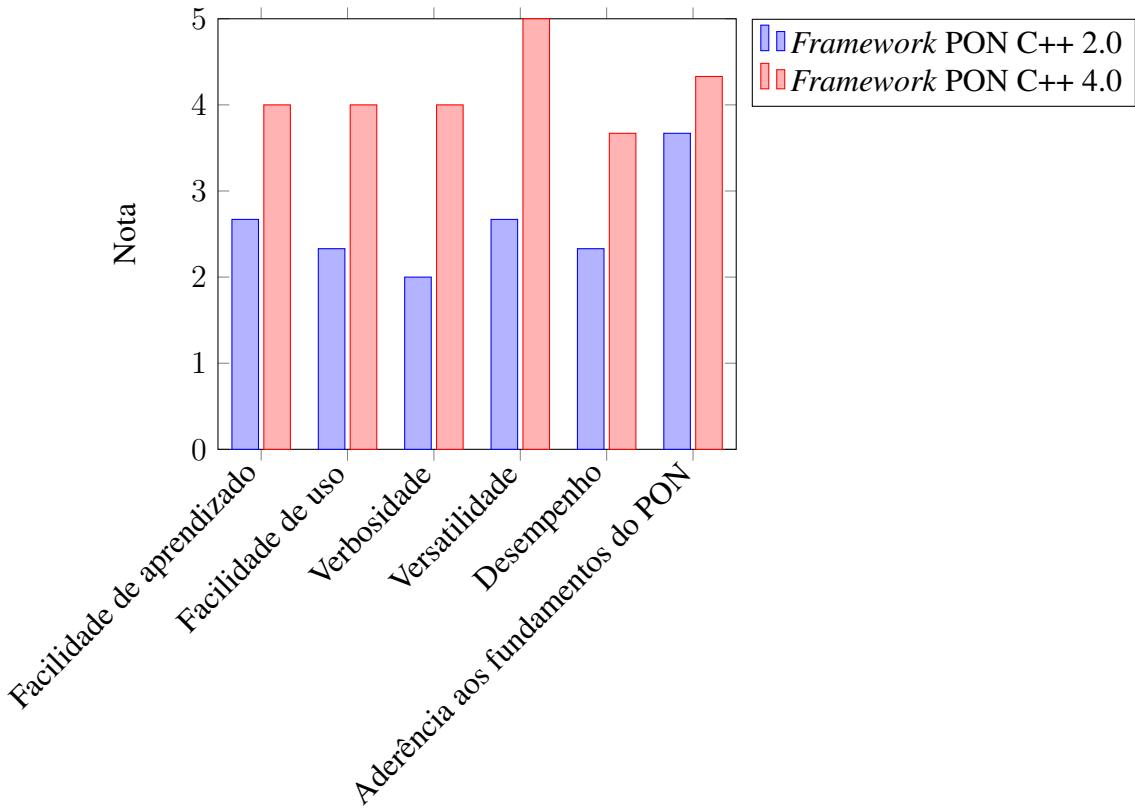


Figura 110 – Resultado da pesquisa de avaliação da melhoria do *Framework PON C++ 4.0* sobre o *Framework PON C++ 2.0*
Fonte: Autoria própria

Tabela 16 – Linhas de código para a composição do *framework*
Fonte: Autoria própria

Versão	Linhas de código
<i>Framework PON C++ 2.0</i>	6587
<i>Framework PON C++ 4.0</i>	970

A grande diferença no número de linhas de código se dá principalmente pela estrutura genérica do *Framework PON C++ 4.0* que permite eliminar a repetição de código que era presente no *Framework PON C++ 2.0*. Essa redução no número de linhas de código é importante, pois isso facilita a manutenção do código por outros desenvolvedores, pois reduz o tamanho da base de código com a qual um novo desenvolvedor precisa se habituar ao trabalhar com o código, assim como torna mais simples eventuais alterações e melhorias na estrutura do *framework*.

Dentre os objetivos da implementação do *Framework PON C++ 4.0* estão aumentar facilidade de uso e reduzir da curva de aprendizado, entretanto esses critérios são bastante subjetivos. Ademais, com o desenvolvimento do jogo NOPUnreal foi possível constatar de forma objetiva a simplificação do código por meio da redução do número de linhas de código utilizadas na implementação do mesmo com o *Framework PON C++ 4.0* quando comparado com a implementação com o *Framework PON C++ 2.0*.

A adição de novos recursos da linguagem de programação C++ ao *Framework PON C++ 4.0*, como o uso de *smart pointers*, e o uso de expressões *lambda*, além de facilitar o uso e desenvolvimento, não trouxe prejuízos no desempenho uma vez comparado com o *Framework PON C++ 2.0* quando aplicado para o desenvolvimento da aplicação de sensores, sendo que o *Framework PON C++ 4.0* apresentou tempos de execução menores que o *Framework PON C++ 2.0* neste caso. Entretanto, essa melhoria de desempenho ainda não foi suficiente a ponto de superar o desempenho da aplicação no POO em C++ para este cenário.

No entanto, a utilização desses recursos mais avançados do C++ no *Framework PON C++ 4.0* causa um aumento na complexidade do código, principalmente quando comparado com o *Framework PON C++ 2.0*, devido à utilização de construções mais novas e com sintaxe rebuscada (como expressões *lambdas*, *fold expressions*). Ainda assim são apresentadas soluções que permitem abstrair essas construções mais complexas de modo suficiente para facilitar a aplicação por desenvolvedores com conhecimento menos avançado dessas sintaxes.

O desenvolvimento da aplicação *Bitonic Sort* demonstra o potencial do PON para o desenvolvimento de algoritmos paralelizáveis, apesar dos elevados tempos de execução e consumo de memória, visto que parte destes custos computacionais podem ser atribuídos à implementação do *Framework PON C++ 4.0*.

Além disso, ao se avaliar o desempenho da versão paralelizada da aplicação *Bitonic Sort* pode se observar que a aplicação da paralelização no mecanismo de notificações pode resultar em ganhos de desempenho referentes ao tempo de execução em determinados cenários. Porém, a paralelização também pode causar degradação no desempenho, como no caso da aplicação *Random Forest*, dependendo da estrutura da aplicação, devido ao natural custo computacional atrelado ao processo de execução de *threads* em C++.

Nem toda aplicação que pode ser paralelizada consegue obter ganhos de desempenho como, por exemplo, a implementação do algoritmo *reverse* da STL paralelizado, que no compilador para C++ da Microsoft (MSVC) teve desempenho 1.6 vezes mais lento que a implementação sequencial. Isso não significa que estes algoritmos não devam ser paralelizados, mas sim que o *hardware* atual para o qual o código é compilado não traz ganho de desempenho (O'NEAL, 2018).

Por fim, principalmente durante o desenvolvimento das aplicações para os algoritmos *Bitonic Sort* e *Random Forest*, foi possível constatar a maneira como o *Framework PON C++ 4.0* facilita o desenvolvimento de aplicações em PON, pois o desenvolvimento destas mesmas

aplicações com o *Framework PON C++ 2.0* não foi realizado justamente por ser extremamente complicada. Por exemplo, na implementação do algoritmo para o *Bitonic Sort*, as *Rules* são criadas de forma dinâmica durante a inicialização, o que só foi possível realizar de forma simples devido aos facilitadores de desenvolvimento, por meio dos *Builder* e gerenciamento de memória com *smart pointers*, introduzidos pelo *Framework PON C++ 4.0*.

O *Framework PON C++ 4.0* avança sobre o que já havia sido apresentado no *Framework PON C++ 2.0* no que diz respeito à programação em alto nível ao disponibilizar interfaces de programação mais fáceis de utilizar, dessa forma contribuindo para a programação em alto nível. O *Framework PON C++ 4.0* também é o primeiro *framework* na linguagem C++ a contemplar de forma satisfatória a propriedade de paralelismo do PON, ao usar recursos modernos da linguagem C++ que permitem o desenvolvimento de aplicações *multithread* de forma bastante simples. Os benefícios da aplicação do paralelismo no *framework* são particularmente destacados nos resultados da aplicação *Bitonic Sort* apresentados na Seção 4.1.2, enquanto na Seção 4.1.3, com a aplicação do algoritmo *Random Forest*, demonstra que existem casos em que a paralelização pode não trazer melhorias ao desempenho. Em suma, os benefícios da paralelização dependem das características da aplicação, sendo que o *Framework PON C++ 4.0* disponibiliza as ferramentas necessárias que permitem ao desenvolvedor facilmente utilizar a paralelização.

Além disso, no tocante à propriedade do paralelismo, o desenvolvimento da aplicação de controle de semáforos permitiu comparar o *Framework PON C++ 4.0* com o *Framework PON Elixir/Erlang*. Com este experimento, fica demonstrada a capacidade de paralelismo do *Framework PON C++ 4.0*, cujos níveis de utilização de CPU atingiram níveis altos, comparáveis aos do *Framework PON Elixir/Erlang*, ao mesmo tempo que mantém tempos de execução mais baixos, característicos das implementações de *framework* do PON em C++. O *Framework PON C++ 4.0* consegue materializar propriedades de desempenho e paralelismo do PON simultaneamente, algo que não foi atingido por nenhum dos outros *frameworks* existentes.

No que diz respeito ao desempenho, como era esperado, o *Framework PON C++ 4.0* não consegue apresentar os ganhos esperados quando comparado a aplicações no PI devido ao alto custo computacional das estruturas utilizadas na composição do *framework* em si. Entretanto, quando comparado com as outras materializações em C++, é possível constatar que o *Framework PON C++ 4.0* apresenta uma melhora no desempenho das aplicações, reduzindo tanto os tempos de execução como consumo de memória.

5 CONCLUSÕES E TRABALHOS SUBSEQUENTES

Este capítulo apresenta na Seção 5.1 as conclusões obtidas por meio do desenvolvimento desta dissertação, à luz dos objetivos apresentados no Capítulo 1, mais especificamente nas seções 1.4 e 1.5. Tais objetivos foram desenvolvidos ao longo desta pesquisa de mestrado e seus resultados foram apresentados nos capítulos 3 e 4. Por fim, são apontados pontos levantados como perspectiva para trabalhos futuros, apresentados na Seção 5.2.

5.1 CONCLUSÃO

Este trabalho teve como principal objetivo o desenvolvimento do *Framework PON C++ 4.0*, orientado à programação genérica e utilização recursos avançados de desenvolvimento em linguagem C++, introduzidos nos padrões mais recentes da linguagem, como C++17 e C++20. Isto permitiu conceber uma versão de *framework* mais eficiente e fácil de se utilizar, no sentido de ser menos “burocrático” e mais flexível, que os *frameworks* em linguagem C++ já existentes.

Em suma, a principal contribuição deste trabalho para o PON é a disponibilização de um novo *framework* para o desenvolvimento de aplicações em PON, o *Framework PON C++ 4.0*, que apresenta melhorias sobre as materializações já existentes em linguagem de programação C++. O *Framework PON C++ 4.0* apresenta uma interface de programação que permite a representação do conhecimento em mais alto nível, apresentando alto desempenho em termos de tempo de execução e permitindo a execução com paralelismo de forma estável, de modo que ainda não era possível nas materializações anteriores.

Como parte dos objetivos, as principais melhorias apresentadas pelo *Framework PON C++ 4.0*, com relação aos *frameworks* do PON em C++ existentes, foram o suporte a execução *multithread/multicore* estável, adição de flexibilidade de tipos aos *Attributes* e flexibilidade algorítmica às *Conditions*, além da redução da verbosidade da utilização do *framework*. O desenvolvimento do *Framework PON C++ 4.0* também foi relatado sob a forma de um artigo no ICIST 2021 (NEVES *et al.*, 2021).

O *Framework PON C++ 4.0* se inspira na estrutura e implementação do *Framework PON C++ 2.0*, no que tange as suas vantagens, permitindo aproveitar a organização em padrões de projeto, como os padrões *Observer*, *Iterator* e *Singleton* que também estão presentes nesta implementação, mas agora de forma mais pragmática no novo *framework*, além do padrão

Builder que foi adicionado no lugar do padrão *Factory*. Isto permitiu, dentre outros, redução de quantidade de código em si no *Framework PON C++ 4.0* vis-à-vis o *Framework PON C++ 2.0*.

Neste âmbito, pode ser constatada, por meio da análise do código-fonte que contém a implementação do *Framework PON C++ 4.0*, quando comparado com a versão do *Framework PON C++ 2.0*, uma considerável redução do volume de código utilizado para a implementação do *framework* em si. Essa redução no volume de código foi possível por meio da eliminação de estruturas de código redundantes, devido à aplicação da programação genérica. Por exemplo, no caso da implementação de *Attributes* específicos para cada tipo, a implementação com a utilização de *templates* remove completamente a duplicidade de código, o que também facilita a manutenção de código futuramente.

Além disso, a aplicação do método de desenvolvimento orientado a testes (TDD) permite garantir que o *Framework PON C++ 4.0* possua o comportamento correto nos casos de uso previstos para cada uma das entidades do PON. Os testes disponíveis, além de garantir o funcionamento do *framework* em seu estado de desenvolvimento atual, servem como garantia que desenvolvimentos futuros, como aqueles que serão propostos na Seção 5.2, não prejudiquem nenhuma das funcionalidades já existentes. Ademais, o conjunto de testes pode naturalmente ser expandido, conforme o desejo de grupos de desenvolvedores.

Ainda, o trabalho desenvolveu aplicações de *benchmark* para o *Framework PON C++ 4.0*. As aplicações desenvolvidas foram a aplicação de sensores, aplicação de controle de semáforo, algoritmo *Random Forest* e algoritmo *Bitonic Sort*. Tanto a aplicação de sensores como o controle de semáforo são *benchmarks* amplamente utilizados no grupo de pesquisas do PON, enquanto os algoritmos *Random Forest* e *Bitonic Sort* são ditos *benchmarks* universalizados, por se tratar de algoritmos bem estabelecidos e amplamente utilizados em uma variedade de linguagens de programação e sistemas. Ainda, também houve o desenvolvimento de um jogo, enquanto exemplo mais universal de aplicação.

A aplicação de sensores permite principalmente comparar o desempenho do *Framework PON C++ 4.0* com o *Framework PON C++ 2.0*, demonstrando ganhos significativos de desempenho sobre o mesmo, reduzindo tanto o tempo de execução como consumo de memória das aplicações. Por sua vez, a aplicação de controle de semáforo permitiu comparar a aplicação do paralelismo, demonstrando a capacidade do *Framework PON C++ 4.0* de atingir alta utilização dos núcleos disponíveis do processador, em níveis comparáveis aos apresentados pelo *Framework PON C++ Elixir/Erlang*.

Com os *benchmarks* universalizados desenvolvidos, por meio dos algoritmos *Random Forest* e *Bitonic Sort*, foi possível demonstrar a viabilidade da utilização do PON, mais especificamente com o *Framework PON C++ 4.0*, para o desenvolvimento de aplicações bem estabelecidas, mesmo que o desempenho ainda seja inferior ao das implementações com a linguagem C no PP, o que absolutamente natural em função do caráter arquétipal de *framework* e deve ser resolvido quando a Tecnologia LingPON estiver mais próxima de estado da técnica do que estado da arte. Em todo caso, ainda nestes *benchmarks* também foi demonstrada a capacidade de execução *multithread* do *Framework PON C++ 4.0*.

Neste âmbito, espera-se que estas implementações motivem trabalhos subsequentes do PON, incluindo com a emergente Tecnologia LingPON, a adotarem o uso destes *benchmarks* universalizados, visto que este trabalho demonstra sua viabilidade para tais implementações. Na verdade, já se observa no grupo de pesquisa a tendência a uso de benchmarks universalizados, por influência deste trabalho, bem como dos trabalhos de Pordeus (PORDEUS, 2020; PORDEUS *et al.*, 2020; PORDEUS *et al.*, 2021), com *benchmarks* em PON em *hardware* que influenciaram este presente trabalho neste âmbito.

Isto dito, além das aplicações de *benchmark* universalizados e também do grupo desenvolvidas com o propósito de avaliar o *Framework PON C++ 4.0* do ponto de vista de desempenho, também houve o desenvolvimento de um jogo com *NOPUnreal*. Este jogo já havia sido implementado pelo próprio autor deste trabalho dissertação, conforme Neves (2020), sendo então reimplementado com o *Framework PON C++ 4.0*. Esta aplicação serve inclusive como demonstração do uso do *Framework PON C++ 4.0* para o desenvolvimento de uma aplicação complexa, com um grande número de *Rules* em sua composição.

O desenvolvimento dessa aplicação de jogo com *NOPUnreal* permitiu também demonstrar em algo a redução da verbosidade do *Framework PON C++ 4.0* quando comparado com o *Framework PON C++ 2.0*, que pode ser constatada pela redução de linhas de código desta nova implementação do jogo. A implementação com o *Framework PON C++ 4.0* resultou em um total de 1514 linhas de código, o que significa 258 linhas de código a menos que a mesma implementação com o *Framework PON C++ 2.0*. Ainda que subjetivo, este resultado aliado aos demais, permitem estabelecer a redução de verbosidade enfim com o aqui proposto *Framework PON C++ 4.0*.

No que diz respeito aos avanços promovidos pelo *Framework PON C++ 4.0* no sentido de facilitar a programação, do ponto de vista de desacoplamento, a implementação com *smart*

pointers contribui ao facilitar o gerenciamento de memória. Isto porque possibilita a criação de elementos de forma dinâmica em qualquer lugar do código e permite também o compartilhamento de forma simples em diversos pontos do código, até mesmo em *threads* diferentes. Assim, este recurso foi fundamental como construto para o *Framework PON C++ 4.0* como um todo.

Em geral, o uso de *smart pointers* tem como principal benefício o gerenciamento de memória de forma transparente ao desenvolvedor e, com isto, existe a garantia de que não haverá vazamento de memória, evitando muitos problemas que podem ocorrer em tempo de execução, como ocorre em versões anteriores, particularmente do *Framework PON C++ 2.0*, conforme observado na Seção 4.1. Esses benefícios também se aplicam em ambientes *multithread*, pois os *shared pointers* utilizam contadores atômicos para o controle do número de referências, minimizando os problemas decorrentes ao acesso de recursos compartilhados entre as *threads*, como aqueles observados no *Framework PON C++ 3.0*.

A complexidade adicionada ao *framework* decorrente da aplicação de conceitos de programação genérica e uso dos recursos avançados da linguagem C++ é transparente ao desenvolvedor do PON, que fará a utilização do *Framework PON C++ 4.0*, devido à existência de abstrações, como as interfaces de construção de entidades por meio de *builders*. Desta forma, o desenvolvedor do PON que deseja utilizar o *Framework PON C++ 4.0* necessita conhecer apenas as interfaces genérica e de uso simples oferecidas pelo *framework*. A facilidade de programação, que é um critério subjetivo, foi corroborada por meio da opinião de desenvolvedores do grupo de pesquisa do PON, conforme apresentado na Seção 4.3, enquanto pode ser observada *in loco* nos exemplos que foram apresentados ao longo da dissertação.

Com a facilitação da programação em PON apresentada pelo *Framework PON C++ 4.0*, no sentido de desenvolvimento desburocratizado e em alto nível, espera-se obter maior engajamento dos desenvolvedores no uso do PON para o desenvolvimento de suas aplicações, principalmente no contexto das disciplinas de PON na UTFPR, de forma a ampliar o número de aplicações desenvolvidas em PON, ajudando a corroborar os aspectos teóricos do PON e demonstrando sua pertinência em aplicações cada vez mais práticas e realísticas. Neste âmbito, a redução na verbosidade da utilização do *framework* contribui principalmente na redução do número de linhas de código utilizadas no desenvolvimento destas aplicações com o *Framework PON C++ 4.0* e também tem como benefício simplificar as aplicações e potencialmente reduzir o tempo de desenvolvimento das mesmas.

Ademais, é importante observar como o *Framework PON C++ 4.0* se apresenta como

uma evolução sobre as materializações existentes em *frameworks* com C++, materializando a propriedade de paralelização que não era atingida pelo *Framework PON C++ 2.0* e, finalmente, não efetivamente nem mesmo em *Framework PON C++ 3.0*, enquanto também apresenta as mesmas funcionalidades e conceitos de programação disponíveis no *Framework PON C++ 2.0/3.0*, com exceção do conceito de impertinência e *Keeper*, devido aos motivos apresentados na Seção 2.4.6.

Em suma, dadas todas as evoluções apresentadas no estado da técnica desenvolvidas neste trabalho e no estado da arte com comparações via *benchmarks* fidedignos, pode-se dizer que as demandas de desenvolvimento de *software* apresentadas como motivação para o desenvolvimento deste trabalho, já apresentadas na Seção 1.2, de desenvolvimento em alto nível, com alto desempenho e paralelismo são atendidas pelo *Framework PON C++ 4.0* enquanto arquétipo de paradigma de programação/desenvolvimento emergente. Ainda, no que diz respeito ao atendimento dessas demandas, o *Framework PON C++ 4.0* se mostra superior aos demais *frameworks* do PON disponíveis. Por fim, pode ser dito que a maior contribuição deste trabalho do ponto de vista técnico é ainda a disponibilização, sob a forma do *Framework PON C++ 4.0*, de uma ferramenta com caráter profissional que facilita o desenvolvimento de aplicações sob a luz do Paradigma Orientado a Notificações e que foi devidamente demonstrando no âmbito de uma pesquisa de mestrado.

5.2 TRABALHOS SUBSEQUENTES

O presente trabalho apresenta o *Framework PON C++ 4.0* como uma versão mais madura e estável de *framework* a ser utilizada tanto como base para o desenvolvimento de aplicações em PON como para a evolução da tecnologia de *frameworks* em si. Espera-se que o *Framework PON C++ 4.0* tenha ampla utilização no desenvolvimento dos trabalhos futuros do grupo de pesquisa do PON.

Durante o desenvolvimento deste trabalho foram vislumbradas diversas possibilidades para o desenvolvimento de outros trabalhos pertinentes, mas que fogem do escopo desta dissertação, de forma que são deixados como propostas para trabalhos futuros. Estas propostas são detalhadas nas seções seguintes.

5.2.1 Desenvolvimento de aplicações com o *Framework PON C++ 4.0*

Ainda neste trabalho foram demonstrados diversos casos de uso com o *Framework PON C++ 4.0*, por meio da aplicação de sensores, aplicação de controle de semáforos, algoritmo *Bitonic Sort*, algoritmo *Random Forest* e do jogo *NOPUnreal*. Essas aplicações, apesar de demonstrarem o potencial de utilização do *Framework PON C++ 4.0* em diversos contextos, inclusive e particularmente do ponto de vista de pesquisa, ainda tem escopo limitado do ponto de vista industrial, sendo consideradas aplicações simples do ponto de vista de desenvolvimento de *software*.

Como uma das motivações para o desenvolvimento do *Framework PON C++ 4.0* era justamente possibilitar o desenvolvimento de aplicações mais complexas, devido à facilitação do uso do *framework* em si, se torna pertinente o desenvolvimento de tais aplicações. Isto de forma tal a consolidar não somente o *Framework PON C++ 4.0* como também a aplicação do PON ao desenvolvimento de *software* no contexto industrial ou similar, além do contexto de *benchmarks* de pesquisa, *benchmarks* estes que justamente serviram de viabilizadores para ensejar tais aplicações com caracteres industriais. Nesse sentido, já está em andamento trabalho de Abu Ahmed Babu que visa aplicar o *Framework PON C++ 4.0* em sistemas sencientes ou afins, possivelmente uma *smart-house*, com olhar sobre computação verde.

Neste sentido destaca-se a possibilidade do desenvolvimento de uma nova versão do futebol de robôs, anteriormente implementado por Santos (2017) e também por Lima *et al.* (2020) com o *Framework PON C++ 2.0*, que agora pode ser implementado com o *Framework PON C++ 4.0* como viabilizador para aplicações industriais devido a suas complexidades e escalas. Ainda neste escopo, também poderia ser desenvolvida a NeuroPON conforme Schütz (2019) e PONFuzzy conforme (MELO, 2016) e suas aplicações em *Framework PON C++ 4.0*.

Por fim, Skora (2021) também propõe a utilização do PON, por meio do *Framework PON C++ 4.0* para o desenvolvimento de um analisador léxico-sintático. Ainda, esta aplicação apresenta um caráter preliminar, necessitando ser evoluída.

5.2.2 Compilador LingPON com alvo *Framework PON C++ 4.0*

Ainda durante o próprio desenvolvimento desta dissertação, foi desenvolvido um novo alvo de compilação da LingPON para o *Framework PON C++ 4.0*, implementado por Skora (2020). A estrutura similar à do *Framework PON C++ 2.0*, porém ainda mais genérica e fácil de

utilizar, facilita o desenvolvimento de um novo alvo de compilação para a LingPON.

Ademais, existem esforços direcionados no sentido de melhorar a LingPON, permitindo a criação dinâmica de entidades (OSHIRO *et al.*, 2021). Os *frameworks* do PON atuais ainda não dão suporte a esta criação dinâmica de entidades, que seria viabilizada com a compilação para o *Framework PON C++ 4.0*, assim como a possível adoção do conceito de *Condition* com flexibilidade algorítmica na LingPON.

5.2.3 Framework PON C++ 4.0 Distribuído

Diversos estudos afirmam e mesmo demonstram que o PON pode ser facilmente utilizado em sistemas distribuídos (BANASZEWSKI, 2009; TALAU, 2016; BARRETTO *et al.*, 2018; OLIVEIRA *et al.*, 2018; OLIVEIRA, 2019; NEGRINI, 2019; MARTINI *et al.*, 2019), de forma que um desenvolvimento pertinente seria a implementação da propriedade de distribuição no *Framework PON C++ 4.0*. Conforme mostrado na Figura 111, a propriedade de distribuição no PON permite a execução das entidades do PON, como *Premises* e *Attributes*, de forma transparente ao sistema em ambientes distintos.

Figura 111 – Comunicação distribuída com o PON

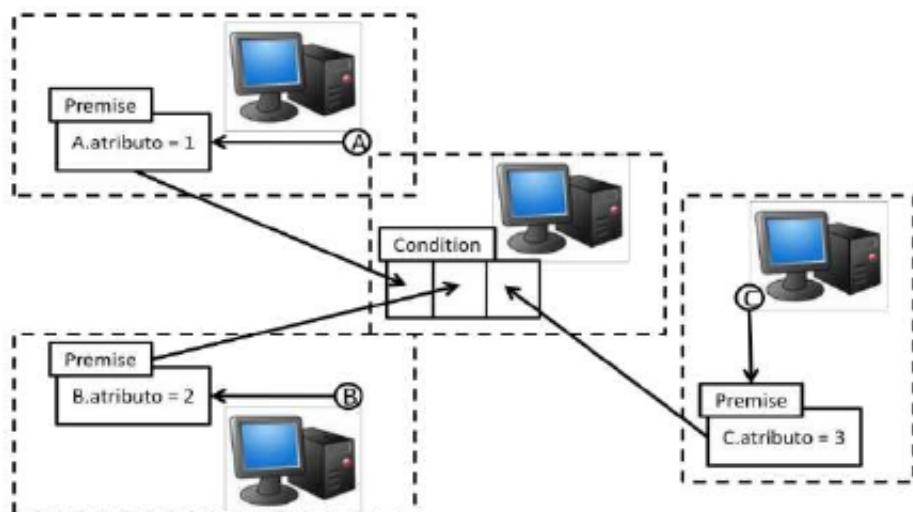


Figura 112 –
Fonte: Banaszewski (2009)

Como apresentado na Seção 2.4.6, mais especificamente na Tabela 12, o *Framework PON C++ 4.0* materializa as propriedades de programação em alto nível, paralelismo e desempenho do PON, porém esta implementação ainda não materializa a propriedade de distribuição. Desta forma, já é previsto o desenvolvimento de pesquisa considerando a implementação de

mecanismos que possibilitem atender a propriedade de distribuição com o *Framework PON C++ 4.0*. Já existe inclusive trabalho de mestrado em andamento desenvolvimento por Pelegrin Figueiredo nesse tópico.

A implementação do mecanismo de distribuição no *Framework PON C++ 4.0* pode ser feita aproveitando os conceitos apresentados, por exemplo, pelo PON *internet protocol* (PONIP) (TALAU, 2016). O PONIP apresenta um esquema independente de materialização para a comunicação via protocolo IP entre entidades do PON. Apesar de limitado, possibilitando a distribuição apenas de *Premises* e *Attributes* de tipos básicos, ele já oferece uma base para o desenvolvimento de aplicações distribuídas com o PON.

O PONIP poderia ser utilizado diretamente com o *Framework PON C++ 4.0*, porém, de forma a manter o nível de abstração e facilidade de programação propostos pelo *Framework PON C++ 4.0*, uma implementação de distribuição nativa ao *Framework PON C++ 4.0* seria mais interessante, ainda que seja implementada aproveitando os conceitos do PONIP.

5.2.4 NeuroPON

O NeuroPON, conforme apresentado na Seção 2.4.1.4, foi implementado utilizando o *Framework PON C++ 3.0*, buscando aproveitar os benefícios que seriam possibilitados pela aplicação do paralelismo possibilitado com tal *framework*. Entretanto, foi visto que a implementação de paralelismo no *Framework PON C++ 3.0* não apresenta resultados esperados no que diz respeito ao desempenho das aplicações.

Nesse sentido, seria interessante a implementação do NeuroPON com o *Framework PON C++ 4.0*, visto que, conforme os resultados apresentados no Capítulo 4, o *framework* possibilitou observar ganhos de desempenho na aplicação *Bitonic Sort*, sendo possível vislumbrar eventuais ganhos de desempenho com a parallelização na aplicação do NeuroPON.

5.2.5 Técnicas de depuração em PON

Uma das principais dificuldades apresentadas no desenvolvimento de *software* em PON é a dificuldade de depuração do código. Essa dificuldade advém do fluxo de execução não sequencial das aplicações em PON, que dificulta o uso de ferramentas de depuração usualmente disponíveis para as principais linguagens de programação.

Nesse sentido, o *Framework PON C++ 4.0*, seguindo o que já havia sido proposto com

o *Framework PON C++ 2.0*, apresenta um mecanismo de registro de *logs* que permitem ao desenvolvedor analisar a execução do *software*. Entretanto, o uso de tais ferramentas ainda tem efetividade limitada, de modo que o PON carece de ferramentas mais avançadas para a depuração de *software*.

Uma possibilidade vislumbrada para auxiliar depuração em PON é a criação de uma ferramenta que permita observar de forma gráfica esse fluxo de execução do *software* que é registrado por meio de *logs*. Tal ferramenta poderia utilizar como base os registros já existentes, interpretando e fornecendo ao desenvolvedor uma interface mais fácil de se analisar.

REFERÊNCIAS

ALEXANDRESCU, Andrei. **Modern C++ Design: Generic Programming and Design Patterns Applied.** USA: Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN 0201704315.

AMBLER, Scott W. 2006. Disponível em: <http://agiledata.org/essays/tdd.html>.

ASANOVIC, Krste; BODIK, Rastislav; DEMMEL, James; KEAVENY, Tony; KEUTZER, Kurt; KUBIATOWICZ, John; MORGAN, Nelson; PATTERSON, David; SEN, Koushik; WAWRZYNEK, John; WESSEL, David; YELICK, Katherine. A view of the parallel computing landscape. **Commun. ACM**, Association for Computing Machinery, v. 52, n. 10, p. 56–67, oct 2009. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/1562764.1562783>.

ATHAYDE, E. B.; NEGRINI, F. **Implementação de Compilação para C++ Namespaces para a LingPON e Otimizações no Tratamento de Premissas.** [S.l.]: PPGCA/UTFPR, 2016. Trabalho realizado na disciplina Tópicos Avançados em Engenharia de Software (CAES101).

AVACHEVA, T; PRUTZKOW, Alexander. The evolution of imperative programming paradigms as a search for new ways to reduce code duplication. **IOP Conference Series: Materials Science and Engineering**, v. 714, p. 012001, 01 2020.

BANASZEWSKI, R. F. **Paradigma Orientado a Notificações: Avanços e Comparações.** 2009. Dissertação (Mestrado) — UTFPR, 2009.

BARENDRREGT, Henk (Hendrik); BARENDSSEN, E. Introduction to lambda calculus. **Nieuw archief voor wiskunde**, v. 4, p. 337–372, 01 1984.

BARRETTO, Wagner R. M.; VENDRAMIN, A.; SIMÃO, J. M. Notification oriented paradigm for distributed systems. **Anais do Computer on the Beach**, p. 110–119, 2018.

BATCHER, Kenneth. Sorting networks and their applications. In: . [S.l.: s.n.], 1968. v. 32, p. 307–314.

BECK, Kent; KUNNINGHAM, Ward. **Test Driven Development: By Example.** USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN 0321146530.

BELMONTE, Danilo. **Método para Distribuição da Carga de Trabalho dos Softwares PON em Multicore.** [S.l.]: Trabalho de Qualificação de Doutorado, CPGEI, UTFPR, 2012.

BELMONTE, D. L.; LINHARES, R. R.; STADZISZ, P. C.; SIMAO, J. M. A new method for dynamic balancing of workload and scalability in multicore systems. **IEEE Latin America Transactions**, v. 14, n. 7, p. 3335–3344, 2016.

BHADWAL, Akhil. **Functional Programming: Concepts, Advantages, Disadvantages, and Applications**. hacker.io, 2020. Disponível em: <https://hackr.io/blog/functional-programming>.

BINDER, Robert V. **Testing Object-Oriented Systems: Models, Patterns, and Tools**. USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN 0201809389.

BOCCARA, Jonathan. **Beyond Locks, a Safer and More Expressive Way to Deal with Mutexes in C**. 2019. Disponível em: <https://www.fluentcpp.com/2019/04/26/how-to-write-safe-and-expressive-multi-threaded-code-in-cpp11/>.

BORKAR, Shekhar; CHIEN, Andrew A. The future of microprocessors. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 54, n. 5, p. 67–77, maio 2011. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/1941487.1941507>.

BOS, Patrick. **C++ Compile-Time Exceptions**. Netherlands eScience Center, 2019. Disponível em: <https://blog.esciencecenter.nl/c-compile-time-exceptions-5443f5bf06fe>.

BROOKSHEAR, J. G. **Computer Science: An Overview**. 9. ed. [S.l.]: Addison-Wesley, 2006.

CARDELLI, Luca; WEGNER, Peter. On understanding types, data abstraction, and polymorphism. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 17, n. 4, p. 471–523, dez. 1985. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/6041.6042>.

CARDOSO, Rafael; HÜBNER, Jomi; BORDINI, Rafael. Benchmarking communication in actor- and agent-based languages. In: . [S.l.: s.n.], 2013. p. 1267–1268.

CHIERICI, Gustavo Brunholi. **JuNOC++ e NOPL Lite: uma nova forma de compor aplicações do Paradigma Orientado a Notificações em alto nível por meio de um novo framework em C++ e um dialeto de NOPL**. [S.l.]: CPGEI/UTFPR, 2020. Trabalho realizado na disciplina Tópicos Especiais em EC: Paradigma Orientado A Notificações (TEC0301).

CRIMINISI, Antonio; SHOTTON, Jamie; KONUKOGLU, Ender. Decision forests: A unified framework for classification, regression, density estimation, manifold learning and semi-supervised learning. **Foundations and Trends in Computer Graphics and Vision**, v. 7, p. 81–227, 01 2011.

CRNKOVIC, Ivica; LARSSON, Magnus. Challenges of component-based development. **Journal of Systems and Software**, v. 61, p. 201–212, 04 2002.

DEANE, Ben; TURNER, Jason. **constexpr ALL the Things!** 2017. Disponível em: <https://www.youtube.com/watch?v=PJwd4JLYJJY&t=797s>.

DEHNERT, James; STEPANOV, Alexander. Fundamentals of generic programming. In: . [S.l.: s.n.], 1998. v. 1766, p. 1–11.

DOORENBOS, Robert B. Production matching for large learning systems. In: . [S.l.: s.n.], 1995.

D'SOUZA, Desmond F.; WILLS, Alan Cameron. **Objects, Components, and Frameworks with UML: The Catalysis Approach**. USA: Addison-Wesley Longman Publishing Co., Inc., 1998. ISBN 0201310120.

FAISON, T. **Event-Based Programming: Taking Events to the Limit**. [S.l.]: Apress, 2006.

FELDMAN, Richard. **Why Isn't Functional Programming the Norm?** Metosin, 2019. Disponível em: <https://www.youtube.com/watch?v=QyJZzq0v7Z4&t=776s>.

FERG, S. **Event-Driven Programming: Introduction, Tutorial, History**. 2006.

FILIPEK, Bartłomiej. **The Amazing Performance of C 17 Parallel Algorithms, is it Possible?** 2018. Disponível em: <https://www.cppstories.com/2018/11/parallel-alg-perf/>.

FORGY, Charles L. Rete: A fast algorithm for the many pattern/many object pattern match problem. **Artificial Intelligence**, v. 19, n. 1, p. 17–37, 1982. ISSN 0004-3702. Disponível em: <https://www.sciencedirect.com/science/article/pii/0004370282900200>.

GABBRIELLI, Maurizio; MARTINI, Simone. **Programming Languages: principles and Paradigms. Series: Undergraduate Topics in Computer Science**. 1. ed. [S.l.]: Springer London, 2010. 440 p. ISBN 978-1-84882-913-8.

GAMMA, E.; HELM, R.; JOHNSON, R; VLISSIDES, J. **Design Patterns: Elements of Reusable Object-Oriented Software**. [S.l.]: Addison Wesley, 1995.

GOOGLE. **google/benchmark**. 2020. Disponível em: <https://github.com/google/benchmark>.

GOOGLE. **google/googletest**. 2020. Disponível em: <https://github.com/google/googletest/blob/master/googletest/docs/primer.md>.

GRIMM, Rainer. **Concurrency with Modern C++**. 2017. Disponível em: <https://www.modernescpp.com/images/stories/pdfs/ConcurrencyWithModernC++.pdf>.

GRIMM, Rainer. **Modernes C++**. 2020. Disponível em: <https://www.modernescpp.com/index.php/more-powerful-lambdas-with-c-20>.

GWOSDZ, Medi Madelen. **If everyone hates it, why is OOP still so widely spread?** 2020. Disponível em: <https://stackoverflow.blog/2020/09/02/if-everyone-hates-it-why-is-oop-still-so-widely-spread/>.

HENNESSY, John; PATTERSON, David. **Computer architecture - a quantitative approach, 3rd Edition.** [S.l.: s.n.], 2003. ISBN 978-1-55860-596-1.

HENZEN, A. F. **Portabilidade do Framework PON de C++ standard para C# e Java.** PPGCA: UTFPR, 2015.

HURD, Tim. **Introduction to Data Types: Static, Dynamic, Strong and Weak.** SitePoint, 2021. Disponível em: <https://www.sitepoint.com/typing-versus-dynamic-typing/>.

ISO/IEC. **Programming Languages — C++.** [S.l.], 2017. Disponível em: <https://web.archive.org/web/20170325025026/http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4660.pdf>.

JENNINGS, Nicholas R. Agent-oriented software engineering. In: **Proceedings of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World: MultiAgent System Engineering.** Berlin, Heidelberg: Springer-Verlag, 1999. (MAAMAW '99), p. 1–7. ISBN 3540662812.

JEONG, Hwancheol; KIM, Sunghoon; LEE, Weonjong; MYUNG, Seok-Ho. **Performance of SSE and AVX Instruction Sets.** 2012.

KERSCHBAUMER, R. **Proposição do paradigma orientado a notificações no desenvolvimento de circuitos lógico-digitais reconfiguráveis.** 2018. Tese (Doutorado) — UTFPR, 2018.

KERSCHBAUMER, Ricardo; LINHARES, Robson R.; SIMÃO, Jean M.; STADZISZ, Paulo C.; LIMA, Carlos R. Erig. Notification-oriented paradigm to implement digital hardware. **Journal of Circuits, Systems and Computers**, v. 27, n. 08, p. 1850124, 2018. Disponível em: <https://doi.org/10.1142/S0218126618501244>.

KERSCHBAUMER, R.; SIMÃO, J. M.; FABRO, J. A.; LIMA, C. R. Erig; LINHARES, R. R. A tool for digital circuits synthesis based on notification oriented paradigm. **IEEE Latin America Transactions**, v. 16, n. 6, p. 1574–1586, 2018.

KOSSOSKI, C. **Proposta de um método de teste para processos de desenvolvimento de software usando o Paradigma Orientado a Notificações.** 2015. Dissertação (Mestrado) — UTFPR, 2015.

LADDAD, Ramnivas. **AspectJ in Action: Practical Aspect-Oriented Programming.** USA: Manning Publications Co., 2003. ISBN 1930110936.

LANGR, Jeff. **Modern C++ Programming with Test-Driven Development: Code Better, Sleep Better.** [S.l.]: Pragmatic Bookshelf, 2013. ISBN 1937785483.

LEE, Pou-Yung; CHENG, A.M.K. Hal: a faster match algorithm. In: . [S.l.: s.n.], 2002. v. 14, n. 5, p. 1047–1058.

LIMA, Anderson Eduardo de; NEVES, Felipe dos Santos; GARCIA, Lucas Tachini; SANTANA, Luis Henrique; BERTOL, Omero Francisco. **AVANÇOS NA APLICAÇÃO DE FUTEBOL DE ROBÔS EM FRAMEWORK PON C++ 2.0 DO PARADIGMA ORIENTADO A NOTIFICAÇÕES – PON.** CPGEI: UTFPR, 2020. Relatório em formato de artigo para a disciplina de Tópicos Especiais em EC: Paradigma Orientado a Notificações (segundo trimestre de 2020).

LINHARES, Robson Ribeiro. **Contribuição para o desenvolvimento de uma arquitetura de computação própria ao paradigma orientado a notificações.** 2015. Tese (Doutorado) — UTFPR, 2015.

LINHARES, Robson R.; PORDEUS, Leonardo F.; SIMÃO, Jean M.; STADZISZ, Paulo C. Noca — a notification-oriented computer architecture: Prototype and simulator. **IEEE Access**, v. 8, p. 37287–37304, 2020.

LIPPMAN, Stanley B.; LAJOIE, Josée; MOO, Barbara E. **C++ Primer (5th Edition).** [S.l.]: Addison-Wesley Professional, 2013. ISBN 0-321-71411-3.

MARTIN, Guilherme Henrique Kaehler; RONSZCKA, Adriano Francisco; FABRO, João Alberto; SIMÃO, Jean Marcelo. Multi-threading capability evaluation of the notification oriented programming language for the x86 architecture. In: **ICIST.** [S.l.: s.n.], 2021. p. 44–49.

MARTINI, Guilherme H. K.; SIMÃO, J.; LINHARES, R. **NOP on multi-core architecture computers.** PPGCA: UTFPR, 2019. Trabalho realizado na disciplina Tópicos Avançados Em Sistemas Embarcado (CAES102 – UTFPR).

MELO, C. A. **Adaptação do Paradigma Orientado a Notificações para Desenvolvimento de Sistemas Fuzzy.** 2016. Dissertação (Mestrado) — UTFPR, 2016.

MEYERS, Scott. **Effective Modern C++.** [S.l.]: O'Reilly, 2015. ISBN 978-1-491-90399-5.

MILES, Russell. **AspectJ Cookbook.** [S.l.]: O'Reilly Media, Inc., 2004. ISBN 0596006543.

MIRANKER, Daniel P. Treat: A better match algorithm for ai production systems. In: **Proceedings of the Sixth National Conference on Artificial Intelligence - Volume 1.** [S.l.]: AAAI Press, 1987. (AAAI'87), p. 42–47. ISBN 0934613427.

MIRANKER, Daniel P.; BRANT, David A. An algorithmic basis for integrating production systems and large databases. In: **Proceedings of the Sixth International Conference on Data Engineering, February 5-9, 1990, Los Angeles, California, USA.** [S.l.]: IEEE Computer Society, 1990. p. 353–360. ISBN 0-8186-2025-0.

MODY, Rahul. **Test Driven Development - Breaking Down Unit And Integration Tests.** Medium, 2017. Disponível em: <https://medium.com/@RahulTMody/test-driven-development-breaking-down-unit-integration-tests-d4a723817419>.

MUCHALSKI, Fernando José; MAZIERO, C. A.; STADZISZ, P. C.; SIMÃO, J. M. **Estudo Comparativo entre o Paradigma de Programação Orientado a Objetos e o Paradigma Orientado a Notificações em um Sistema para Cálculo de Produtividade.** [S.l.]: CPGEI/UTFPR, 2012. Trabalho realizado na disciplina Tópicos Avançados Em Sistemas Embarcados (CASE102).

MULLAPUDI, Amrutha. **Bitonic Sort.** [S.l.]: University at Buffalo, 2014. Lecture for CSE633: Parallel Algorithms (Spring 2014).

NAIK, Kshirasagar; TRIPATHY, Priyadarshi. **Software Testing and Quality Assurance: Theory and Practice.** 2nd. ed. [S.l.]: Wiley Publishing, 2018. ISBN 111919475X.

NAMBIAR; RAGHUNATH; POESS; MEIKEL. **Performance Evaluation and Benchmarking.** [S.l.]: Springer, 2009. ISBN 978-3-642-10423-7.

NEGRINI, F. **Tecnologia NOPL Erlang-Elixir – paradigma orientado a notificações via uma abordagem orientada a microatores assíncronos.** 2019. Dissertação (Mestrado) — UTFPR, CPGEI, 2019.

NEGRINI, Fabio; RONSZCKA, Adriano; LINHARES, Robson; FABRO, João; STADZISZ, Paulo; SIMÃO, Jean Marcelo. Nopl-erlang: Programação multicore transparente em linguagem de alto nível. In: **Anais da V Escola Regional de Alto Desempenho do Rio de Janeiro.** Porto Alegre, RS, Brasil: SBC, 2019. p. 16–20. ISSN 0000-0000. Disponível em: <https://sol.sbc.org.br/index.php/eradrj/article/view/9535>.

NEGRINI, Fabio; RONSZCKA, Adriano Francisco; LINHARES, Robson Ribeiro; FABRO, João Alberto; STADZISZ, Paulo Cézar; SIMÃO, Jean Marcelo. NOPL-Erlang: Programação multicore transparente em linguagem de alto nível. **Cadernos do IME**

- Série Informática, v. 43, n. 2, p. 70–74, 2019. ISSN 2317-2193. Disponível em: <https://www.e-publicacoes.uerj.br/index.php/cadinf/article/view/54404>.

NEVES, Felipe dos Santos; SIMÃO, Jean Marcelo; LINHARES, Robson Ribeiro. Application of generic programming for the development of a c++ framework for the notification oriented paradigm. In: ZDRAVKOVIĆ, M.; TRAJANOVIĆ, M.; KONJOVIĆ, Z. (Ed.). **ICIST 2021 Proceedings**. [S.l.: s.n.], 2021. p. 56–61.

NEVES, Felipe dos Santos. **Comparisons on Game Development with Unreal Engine 4 using Object Oriented Paradigm (OOP in C++) versus Notification Oriented Paradigm**. CPGEI: UTFPR, 2020. Relatório na forma de artigo para da disciplina de Paradigmas de Programação (1º trimestre de 2020).

OLIVEIRA, R. N. **Paradigma Orientado a Notificações Aplicado em Assistência à Autonomia Domiciliar**. 2019. Dissertação (Mestrado) — UTFPR, CPGEI, 2019.

OLIVEIRA, Rodrigo Nunes; ROTH, Valmir; HENZEN, Alexandre Felippeto; SIMAO, Jean Marcelo; NOHAMA, Percy; WILLE, Emilio Carlos Gomes. Notification oriented paradigm applied to ambient assisted living tool. **IEEE Latin America Transactions**, v. 16, n. 2, p. 647–653, 2018.

O’NEAL, Billy. **Using C++17 Parallel Algorithms for Better Performance**. Microsoft, 2018. Disponível em: <https://devblogs.microsoft.com/cppblog/using-c17-parallel-algorithms-for-better-performance/>.

OSHIRO, Larissa; RONSZCKA, Adriano; FABRO, João; SIMÃO, Jean. Linguagem e compilador para o paradigma orientado a notificações: Uma solução performante orientada a regras. In: **Anais da XII Escola Regional de Alto Desempenho de São Paulo**. Porto Alegre, RS, Brasil: SBC, 2021. p. 61–64. ISSN 0000-0000. Disponível em: <https://sol.sbc.org.br/index.php/eradsp/article/view/16706>.

PETERS, E. **Coprocessador para aceleração de aplicações desenvolvidas utilizando paradigma orientado a notificações**. 2012. Dissertação (Mestrado) — UTFPR, 2012.

PETERS, Eduardo; JASINSKI, Ricardo P.; PEDRONI, Volnei A.; SIMÃO, Jean M. A new hardware coprocessor for accelerating notification-oriented applications. In: **2012 International Conference on Field-Programmable Technology**. [S.l.: s.n.], 2012. p. 257–260.

PITSIANIS, Nikos. **CMake**. 2008. Disponível em: <https://courses.cs.duke.edu/fall08/cps196.1/Pthreads/bitonic.c>.

POO, Danny; KIONG, Derek; ASHOK, Swarnalatha. **Object-Oriented Programming and Java**. Berlin, Heidelberg: Springer-Verlag, 2007. ISBN 1846289629.

PORDEUS, L.F.; LINHARES, R.R.; STADZISZ, P.C.; SIMÃO, J.M. Nop-dh – evaluation over bitonic sort algorithm. **Microprocessors and Microsystems**, p. 104314, 2021. ISSN 0141-9331. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0141933121004750>.

PORDEUS, L. F. Simulação de uma arquitetura de computação própria ao paradigma orientado a notificações. 2017. Dissertação (Mestrado) — UTFPR, CPGEI, 2017.

PORDEUS, L. F. ArqTotalPON - Contribuição para Arquitetura de Computação própria e efetiva ao Paradigma Orientado a Notificações. 2020. Dissertação (Mestrado) — UTFPR, CPGEI, 2020.

PORDEUS, L. F.; LAZZARETTI, A. E.; LINHARES, R. R.; SIMÃO, J. M. NOP-DH - Evaluation Over Random Forest Algorithm. [S.l.]: UTFPR, 2020.

PRESSMAN, R.; MAXIM, B. **Engenharia de Software - 8^a Edição**. McGraw Hill Brasil, 2016. ISBN 9788580555349. Disponível em: <https://books.google.com.br/books?id=wexzCwAAQBAJ>.

RENAUX, Douglas P. B.; LINHARES, Robson Ribeiro; SIMAO, Jean Marcelo; STADZISZ, Paulo Cézar. **CTA Simulator - Concepts of Operations**. 2015. Disponível em: http://www.dainf.ct.utfpr.edu.br/~douglas/CTA_CONOPS.pdf.

RONSZCKA, A. F. Contribuição para a concepção de aplicações no paradigma orientado a notificações (PON) sob o viés de padrões. 2012. Dissertação (Mestrado) — UTFPR, CPGEI, 2012.

RONSZCKA, A. F. Método para a Criação de Linguagens de Programação e Compiladores para o Paradigma Orientado a Notificações em Plataformas Distintas. 2019. Tese (Doutorado) — UTFPR, CPGEI, 2019.

RONSZCKA, A. F.; VALENCA, G. Z.; LINHARES, R. R.; FABRO, J. A.; STADZISZ, P. C.; SIMÃO, J. M. Notification-oriented paradigm framework 2.0: An implementation based on design patterns. **IEEE Latin America Transactions**, v. 15, n. 11, p. 2220–2231, 2017.

ROY, Peter Van. Programming paradigms for dummies: What every programmer should know. 04 2012.

ROY, Peter Van; HARIDI, Seif. Concepts, Techniques, and Models of Computer Programming. 1st. ed. [S.l.]: The MIT Press, 2004. ISBN 0262220695.

SANTOS, L. A. Linguagem e Compilador para o Paradigma Orientado A Notificações: Avanços para Facilitar a Codificação e sua Validação em uma Aplicação de Controle de Futebol de Robôs. 2017. Dissertação (Mestrado) — UTFPR, CPGEI, 2017.

SCHÜTZ, F. NEURO-PON: Uma Abordagem para o Desenvolvimento de Redes Neurais Artificiais Utilizando o Paradigma Orientado a Notificações. 2019. Tese (Doutorado) — UTFPR, 2019.

SCHÜTZ, Fernando; FABRO, João; RONSZCKA, Adriano; STADZISZ, Paulo; SIMÃO, Jean. Proposal of a declarative and parallelizable artificial neural network using the notification-oriented paradigm. **Neural Computing and Applications**, v. 30, 09 2018.

SCHÜTZ, Fernando; RONSZCKA, Adriano. **Compilação C++ Estático**. [S.l.]: UTFPR, 2015. Relatório para Disciplina Linguagens e Compiladores PON.

SCOTT, M. L. **Programming Language Pragmatics**. [S.l.]: Morgan Kaufmann Publishers Inc, 2000.

SCOTT, Michael L. **Programming Language Pragmatics, Third Edition**. 3rd. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009. ISBN 0123745144.

SILISTEANU, Paul. **C++17 constexpr everything (or as much as the compiler can)**. 2017. Disponível em: <https://solarianprogrammer.com/2017/12/27/cpp-17-constexpr-everything-as-much-as-the-compiler-can/>.

SIMÃO, Jean. **A Contribution to the Development of a HMS Simulation Tool and Proposition of a Meta-Model for Holonic Control**. 06 2005. Tese (Doutorado) — UTFPR , Henri Poincaré University, 06 2005.

SIMÃO, J. M. **Proposta de uma Arquitetura de Controle para Sistemas Flexíveis de Manufatura Baseada em Regras e Agentes**. 2001. Dissertação (Mestrado) — UTFPR, CPGEI, 2001.

SIMÃO, Jean Marcelo; BANASZEWSKI, Roni Fábio; TACLA, Cesar Augusto; STADZISZ, Paulo Cézar. Notification oriented paradigm (nop) and imperative paradigm: A comparative study. **IEEE Transactions on Systems, Man and Cybernetics. Part A, Systems and Humans**, IEEE Press, v. 39, n. 1, p. 238–250, 2012.

SIMÃO, Jean Marcelo; STADZISZ, Paulo Cézar. An agent-oriented inference engine applied for supervisory control of automated manufacturing systems. In: ABE, J.; FILHO, J. Silva (Ed.). **Advances in Logic, Artificial Intelligence and Robotics**. [S.l.]: IOS Press Books, 2002. v. 85, p. 234–241.

SIMÃO, Jean Marcelo; STADZISZ, Paulo Cézar. **Paradigma Orientado a Notificações (PON) – Uma Técnica de Composição e Execução de Software Orientado a Notificações**. 2008. Pedido de Patente submetida ao INPI/Brazil (Instituto Nacional de Propriedade Industrial) em

2008 e a Agência de Inovação/UTFPR em 2007. No. INPI Provisório 015080004262. Patente submetida ao INPI.

SIMÃO, Jean Marcelo; TACLA, Cesar Augusto; STADZISZ, Paulo Cézar. Inference process based on notifications: The kernel of a holonic inference meta-model applied to control issues. **IEEE Transactions on Systems, Man and Cybernetics. Part A, Systems and Humans**, IEEE Press, v. 39, n. 1, p. 238–250, 2009.

SKORA, Lucas Eduardo Bonacio. **Criação de alvos de compilação para a NOPL e ferramentas de serialização-desserialização para o Grafo PON**. [S.l.]: CPGEI/UTFPR, 2020. Trabalho realizado na disciplina Paradigmas de Programação.

SKORA, Lucas Eduardo Bonacio. **ANALISADORES LÉXICO-SINTÁTICOS EM PON E POO**. [S.l.]: UTFPR, 2021. Relatório em formato de artigo para a disciplina de Estudo Especial em Paradigmas de Programação CSE20 no CPGEI - UTFPR.

STEPANOV, Alexander A.; ROSE, Daniel E. **From mathematics to generic programming**. [S.l.]: Addison-Wesley, 2015.

STROUSTRUP, Bjarne. Why c++ is not just an object-oriented programming language. *In: Addendum to the Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (Addendum)*. New York, NY, USA: Association for Computing Machinery, 1995. (OOPSLA '95), p. 1–13. ISBN 0897917219. Disponível em: <https://doi.org/10.1145/260094.260207>.

STROUSTRUP, Bjarne. Thriving in a crowded and changing world: C++ 2006–2020. **Proc. ACM Program. Lang.**, Association for Computing Machinery, New York, NY, USA, v. 4, n. HOPL, jun. 2020. Disponível em: <https://doi.org/10.1145/3386320>.

TALAU, Marcos. **PONIP: Uso do Paradigma Orientado a Notificações em Redes IP**. [S.l.]: CPGEI/UTFPR, 2016. Trabalho realizado na disciplina Paradigma Orientado a Notificações.

TEIXEIRA, Pedro. **Inside Windows 8: Pedro Teixeira - Thread pools**. Channel 9, 2012. Disponível em: <https://channel9.msdn.com/Shows/Going+Deep/Inside-Windows-8-Pedro-Teixeira-Thread-pool>.

THORSEN, Bo. **Using Templates to Avoid Code Duplication**. 2015. Disponível em: <https://www.vikingsoftware.com/using-templates-to-avoid-code-duplication/>.

TIOBE. **TIOBE Index for June 2021**. 2021. Disponível em: <https://www.tiobe.com/tiobe-index/>.

TURNER, Jason. **C++ Best Practices**. [S.l.]: Leanpub, 2021.

VALENÇA, G. Z. **Contribuição para materialização do paradigma orientado a notificações (PON) via framework e wizard.** 2012. Dissertação (Mestrado) — UTFPR, 2012.

WATT, D. **Programming Language Design Concepts.** [S.l.]: J. Willey & Sons,, 2004.

WILLIAMS, Laurie *et al.* On the effectiveness of unit test automation at microsoft. In: **2009 20th International Symposium on Software Reliability Engineering.** [S.l.: s.n.], 2009. p. 81–89.

XAVIER, R. D. **Paradigmas de desenvolvimento de software: comparação entre abordagens orientada a eventos e orientada a notificações.** 2014. Dissertação (Mestrado) — UTFPR, PPGCA, 2014.

APÊNDICES

APÊNDICE A

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DIRETORIA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO APLICADA (PPGCA)
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA E
INFORMÁTICA INDUSTRIAL (CPGEI)

AVANÇOS NA APLICAÇÃO DE FUTEBOL DE ROBÔS EM FRAMEWORK PON C++ 2.0 DO PARADIGMA ORIENTADO A NOTIFICAÇÕES – PON

Disciplinas:

Tópicos Especiais em EC: Paradigma Orientado a Notificações - TEC0301 (CPGEI)
Tópicos Avançados em Sistemas Embarcados 2 - CASE102 (PPGCA)

Alunos:

1. Anderson Eduardo de Lima
2. Felipe dos Santos Neves
3. Lucas Tachini Garcia
4. Luis Henrique Sant'Ana
5. Omero Francisco Bertol

CURITIBA
2020

LISTA DE FIGURAS

Figura 1 – Exemplo de uma <i>Rule</i>	8
Figura 2 – Oracle VM VirtualBox instalado	22
Figura 3 – Máquina virtual “Lubuntu+ROS_Kinetic” no Oracle VM Virtual Box	23
Figura 4 – Máquina virtual “Lubuntu+ROS_Kinetic” executando no Oracle VM Virtual Box	23
Figura 5 – Ambiente gráfico de simulação “grSim” em execução.....	24
Figura 6 – Futebol de Robôs rodando no ambiente gráfico de simulação “grSim”	24
Figura 7 – Robôs se movimentando após ações executadas no ambiente gráfico de simulação <i>Small Size League</i>	25
Figura 8 – Organização estrutural do projeto “RoboCup2012”	26
Figura 9 – Códigos-fontes do pacote “ControleF180” projeto “RoboCup2012”	27
Figura 10 – Diagrama de classes parcial do projeto “RoboCup2012”.....	29
Figura 11 – Softwares para partidas de Futebol de Robôs desenvolvido em Santos (2017)	48
Figura 12 – Diagrama de classes parcial do projeto “RoboCup2012” desenvolvido em Santos (2017).....	50
Figura 13 – Dimensão, em milímetros, do campo oficial da categoria <i>Small Size League</i> (SSL).....	51

LISTA DE QUADROS

Quadro 1 – Exemplo da declaração de <i>Attributes e MethodPointers</i>	9
Quadro 2 – Exemplo da implementação de <i>MethodsPointers</i>	9
Quadro 3 – Exemplo da implementação de <i>Premises</i>	10
Quadro 4 – Exemplo da implementação de <i>Rules</i>	11
Quadro 5 – Conjunto de <i>Rules</i> implementadas na aplicação para controle do Futebol de Robôs.....	17
Quadro 6 – Principais códigos-fontes do pacote “ControleF180” projeto “RoboCup2012”	27
Quadro 7 – Conjunto de <i>Premises</i> do projeto “RoboCup2012” desenvolvido em Santos (2017)	31
Quadro 8 – Conjunto de <i>Instigations</i> do projeto “RoboCup2012” desenvolvido em Santos (2017).....	37
Quadro 9 – Conjunto de <i>Rules</i> do projeto “RoboCup2012” desenvolvido em Santos (2017)	39
Quadro 10 – Principais códigos-fontes do projeto “RoboCup2012” desenvolvido em Santos (2017).....	49
Quadro 11 – Partes relevantes do arquivo “StrategyPON.cpp” para a <i>Rule</i> “rlPassWhenOff”	68
Quadro 12 – <i>Method</i> “attackMove” presente nos diferentes códigos de acordo com o posicionamento do jogador.....	69
Quadro 13 – <i>Method</i> “defenceMove” presente nos diferentes códigos de acordo com o posicionamento do jogador.....	71
Quadro 14 – <i>Method</i> “moveToStopPosition”.....	73
Quadro 15 – Alterações relevantes no arquivo “StrategyPonDF.cpp” para as novas movimentações de goleiro	74

SUMÁRIO

1 INTRODUÇÃO	4
1.1 CONTEXTUALIZAÇÃO	4
1.2 OBJETIVOS	4
1.3 ESTRUTURA DO TRABALHO	4
2 PARADIGMA ORIENTADO A NOTIFICAÇÕES – PON.....	7
2.1 ELEMENTOS FUNDAMENTAIS DO PON	7
2.2 ATTRIBUTES E METHODS	8
2.3 CONDITION E PREMISES	10
2.4 ACTION E INSTIGATIONS	10
2.5 RULES	11
2.6 FLUXO DE EXECUÇÃO NO PON	12
2.7 LINGUAGENS DE PROGRAMAÇÃO NO PON	12
3 INFORMAÇÕES SOBRE O FUNCIONAMENTO DO FUTEBOL DE ROBÔS	13
3.1 REQUISITOS FUNCIONAIS.....	13
3.2 REGRAS	14
3.3 NOTAS IMPORTANTES	15
4 DESENVOLVIMENTO DA APLICAÇÃO PARA CONTROLE DO FUTEBOL DE ROBÔS.....	16
4.1 ATIVIDADE PROPOSTA.....	16
4.2 DIFICULDADES ENCONTRADAS	16
4.3 TRABALHO DESENVOLVIDO	17
5 CONCLUSÃO.....	20
REFERÊNCIAS	21
APÊNDICE A: UTILIZAÇÃO DO AMBIENTE DE FUTEBOL DE ROBÔS EM MÁQUINA VIRTUAL.....	22
APÊNDICE B: PROJETO “ROBOCUP2012” EM AMBIENTE DE SIMULAÇÃO	26
APÊNDICE C: PROJETO “ROBOCUP2012” DESENVOLVIDO EM SANTOS (2017)	30
APÊNDICE D: RULES IMPLEMENTADAS NO PROJETO “ROBOCUP2012” PELOS ALUNOS ANDERSON EDUARDO DE LIMA, FELIPE DOS SANTOS NEVES, LUCAS TACHINI GARCIA E OMERO FRANCISCO BERTOL.....	51
APÊNDICE E: RULES IMPLEMENTADAS NO PROJETO “ROBOCUP2012” PELO ALUNO LUIS HENRIQUE SANT’ANA	67

1 INTRODUÇÃO

Nestas considerações iniciais serão apresentadas a contextualização do problema objeto deste trabalho acadêmico, os objetivos e a organização estrutural das seções que constituem este relatório técnico.

1.1 CONTEXTUALIZAÇÃO

Este trabalho tem por objetivo relatar o que foi colocado em prática no tocante aos tópicos abordados nas disciplinas referentes ao Paradigma Orientado a Notificações (PON) registradas como “Tópicos Especiais em EC: Paradigma Orientado a Notificações – TEC0301” no Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI) e como “Tópicos Avançados em Sistemas Embarcados 2 – CASE102” no Programa de Pós-Graduação em Computação Aplicada (PPGCA).

Para experimentar as técnicas explanadas nas disciplinas em questão em uma aplicação real, foi desenvolvido um projeto orientado a regras e baseado nos conceitos e propriedades do PON para um simulador de Futebol de Robôs (RoboCup) executado em máquina virtual.

O desenvolvimento deste estudo deve contemplar o regulamento de funcionamento de uma competição de Futebol de Robôs (RoboCup), um estudo teórico e prático dos elementos fundamentais da tecnologia PON, a configuração de um ambiente de simulação do Futebol de Robôs em máquina virtual (Apêndice A), a análise de *Regras (Rules)* implementadas em trabalhos existentes (Apêndices B e C) e o desenvolvimento no *Framework PON C++ 2.0* de um conjunto de *Rules* para um software de controle de uma partida de Futebol de Robôs (Apêndices D e E).

1.2 OBJETIVOS

Com o objetivo de colocar em prática os conceitos vistos em classe no contexto das disciplinas supracitadas, foi proposto o desenvolvimento de um aplicativo de controle de robôs em tecnologia PON, em contexto de Futebol de Robôs, para execução em ambiente de simulação.

1.3 ESTRUTURA DO TRABALHO

Este relatório técnico está dividido em 5 seções e 5 apêndices. Nesta presente e primeira seção, nomeadamente “Introdução”, foi explicado o assunto tema do trabalho e também foram abordados a motivação, os objetivos e a estrutura geral do trabalho.

Subsequentemente, na segunda seção nomeadamente “Paradigma Orientado a Notificações – PON” será mostrada a estrutura do PON, seus elementos essenciais na forma de entidades *Atributos (Attributes)*, *Métodos (Methods)*, *Condição (Condition)*, *Premissas (Premises)*, *Ação (Action)*, *Instigações (Instigations)* e *Regras (Rules)*, o fluxo de execução no PON e a apresentação das linguagens de programação que materializam a tecnologia PON.

Por sua vez, na terceira seção, nomeadamente “Informações sobre o Funcionamento do Futebol de Robôs”, são apresentadas informações sobre as *Rules* de como funciona uma partida de Futebol de Robôs (RoboCup).

Notadamente, na quarta seção, nomeadamente “Desenvolvimento da Aplicação para Controle do Futebol de Robôs”, será apresentado o desenvolvimento do software de controle dos robôs visando a simulação de uma partida de Futebol de Robôs em ambiente de simulação, o qual foi disponibilizado em uma máquina virtual.

Por fim, na quinta e última seção, nomeadamente de “Conclusão”, serão apresentadas as considerações finais apontando as conclusões alcançadas com o desenvolvimento deste trabalho acadêmico.

Outrossim, no “Apêndice A: Utilização do Ambiente de Futebol de Robôs em Máquina Virtual”, será apresentado um tutorial com os passos para configuração da máquina virtual e a execução do ambiente de simulação do Futebol de Robôs.

Ainda, no “Apêndice B: Projeto RoboCup2012 em Ambiente de Simulação” será detalhado o projeto do aplicativo de controle do Futebol de Robôs “RoboCup2012” em tecnologia PON, para execução no ambiente de simulação e que foi disponibilizado para este estudo.

Por sua vez, o “Apêndice C: Projeto RoboCup2012 Desenvolvido em Santos (2017)”, será discutido o trabalho de Santos (2017) que realizou a proposição de uma nova versão da linguagem de programação específica para o Paradigma Orientado a Notificações (PON), nomeada LingPON. A versão 1.2 implementada da LingPON foi empregada no desenvolvimento de um software que controla partidas de Futebol de Robôs (RoboCup) e que foi então comparado com outros dois softwares equivalentes, a saber, desenvolvidos utilizando o *Framework PON C++ 2.0* e a LingPON versão 1.0. Os softwares desenvolvidos em Santos (2017) nas diferentes linguagens de programação que materializam a tecnologia PON foram também disponibilizados para a realização deste estudo.

Em tempo, as *Rules* implementadas pelos discentes no Projeto “RoboCup2012” em ambiente de simulação serão apresentadas no Apêndice D as *Rules* implementadas pelos alunos Anderson Eduardo de Lima, Felipe dos Santos Neves, Lucas Tachini Garcia e Omero Francisco Bertol; e no Apêndice E as *Rules* implementadas pelo aluno Luis Henrique Sant’Ana.

2 PARADIGMA ORIENTADO A NOTIFICAÇÕES – PON

Nesta segunda seção serão feitas as considerações iniciais sobre o Paradigma Orientado a Notificações (PON), seus elementos essenciais na forma de *Atributos* (*Attributes*), *Métodos* (*Methods*), *Condição* (*Condition*), *Premissas* (*Premises*), *Ação* (*Action*), *Instigações* (*Instigations*) e *Regras* (*Rules*), o fluxo de execução no PON e a apresentação das linguagens de programação que materializam a tecnologia PON.

2.1 ELEMENTOS FUNDAMENTAIS DO PON

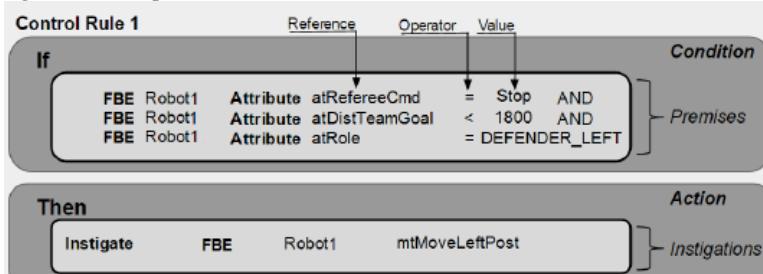
O Paradigma Orientado a Objeto (POO), classificado como subparadigma do Paradigma Imperativo, ou os Sistemas Baseados em Regras, englobados pelo Paradigma Declarativo, sofrem limitações intrínsecas de seus paradigmas (RONSZCKA, 2012, p. 27).

Esses paradigmas levam ao forte acoplamento em expressões lógico causais deste modo levando ao processamento desnecessário com redundâncias causais e/ou utilização custosa de estruturas de dados. Essas limitações com frequência comprometem o desempenho das aplicações. Nesse contexto, existem motivações para buscas de alternativas aos Paradigma Imperativo e Paradigma Declarativo, com o objetivo de diminuir ou eliminar as desvantagens desses paradigmas (RONSZCKA, 2012, p. 28).

Nesse contexto, o Paradigma Orientado a Notificações (PON) foi apresentado como uma alternativa. O PON pretende eliminar algumas deficiências dos paradigmas usuais em relação à avaliação de expressões causais desnecessárias e acopladas. Para tal, o PON faz uso de um mecanismo baseado no relacionamento de entidades computacionais que proporcionam um fluxo de execução de maneira reativa através de notificações precisas e pontuais (RONSZCKA, 2012, p. 28).

Na estrutura do Paradigma Orientado a Notificações (PON), as entidades computacionais que possuem *Atributos* (*Attributes*) e *Métodos* (*Methods*) são genericamente chamadas de *Fact Base Elements* (FBEs). Por meio de seus *Attributes* e *Methods*, as entidades FBEs são passíveis de correlação lógico-casual, normalmente proveniente de regras “se-então” (ou “If-Then”), por meio de entidades *Rules*, esquematizada na Figura 1, o que constituem elementos fundamentais do PON (SANTOS, 2017, p. 19; RONSZCKA, 2019, p. 28).

Figura 1 – Exemplo de uma Rule



Fonte: Santos (2017, p. 19).

Uma *Rule*, como mostra novamente a Figura 1, é decomposta em entidades *Condition* (*Condição*, trata da decisão da *Rule*) e *Action* (*Ação*, trata da execução das ações associadas a *Rule*). Por sua vez, a *Condition* é decomposta em uma ou mais entidades *Premises* (*Premissas*, realiza as verificações que definem a tomada de decisão) e uma *Action* é decomposta em uma ou mais entidades *Instigations* (*Instigações*, “instiga” *Methods* responsáveis por realizar serviços ou habilidades) (BANASZEWSKI, 2009, p. 10; SANTOS, 2017, p. 19; VALENÇA, 2012, p. 8-9; RONSZCKA, 2019, p. 28-29).

2.2 ATTRIBUTES E METHODS

Todo e qualquer recurso de uma aplicação expressa os estados ou valores de situações por meio de entidades chamadas de *Atributos* (*Attributes*), bem como disponibiliza seus serviços ou funcionalidades por meio de entidades chamadas de *Métodos* (*Methods*) (BANASZEWSKI, 2009, p. 9).

O conceito de *Atributos* (*Attributes*) e *Métodos* (*Methods*) representam uma evolução dos conceitos de *Atributos*, que definem a porção de dados; e *Métodos*, que implementam o comportamento, de classe do Paradigma Orientado a Objetos. A diferença, segundo Ronszcka (2012), está no desacoplamento implícito da classe proprietária e a colaboração pontual para com as entidades *Premises* e *Instigations*.

No contexto do desenvolvimento do aplicativo em tecnologia PON para controle de uma partida de Futebol de Robôs, tem-se como exemplo de *Attribute* PON o elemento “atClosestToGoal” (Quadro 1, Linha 4), implementado para definir se o jogador está próximo (*true*) ao gol ou não (*false*). Já no caso do *Attribute* “atPoxX” (Quadro 1, Linha 5), ele será utilizado para representar o valor da posição do robô em relação ao eixo “X” do gráfico cartesiano (corresponde a distância da linha de gol defendido, até a linha de gol atacado).

Quadro 1 – Exemplo da declaração de Attributes e MethodPointers

```
// Declaração dos Attributes “atClosestToGoal” e “atPosX” na classe “RobotPON”
// no arquivo de cabeçalhos “RobotPON.h”
1. class RobotPON : public Robot, public FBE {
2. public:
3. ...
4. Boolean* atClosestToGoal;
5. Double* atPosX;
6. };
```

Fonte: Autoria própria.

Em tempo, também no contexto do aplicativo de controle de uma partida de Futebol de Robôs, tem-se como exemplos de *Methods PON* os elementos “*mtClosestToGoal*” e “*mtNotClosestToGoal*” (Quadro 2, Linhas 4-5). Estes elementos do tipo *MethodPointers* permitem que métodos implementadas em C++ sejam utilizadas diretamente como *Methods*. Eles são criados atrelando o método que modifica o estado de um jogador para indicar se ele está próximo ao gol (*Attribute* “*atClosestToGoal*” é setado com o valor *true*), em “*closestToGoal*” (Quadro 2, Linhas 7 e 15-17); ou não (*Attribute* “*atClosestToGoal*” é setado com o valor *false*), com o método “*notClosestToGoal*” (Quadro 2, Linhas 8 e 18-20).

Quadro 2 – Exemplo da implementação de MethodsPointers

```
// Implementação das interfaces dos Methods na classe “StrategyPON”
// no arquivo de cabeçalhos “StrategyPON.h”
1. class StrategyPON : public Strategy, public FBE {
2. protected:
3. ...
4. MethodPointer<StrategyPON>* mtClosestToGoal;
5. MethodPointer<StrategyPON>* mtNotClosestToGoal;
6. ...
7. void closestToGoal();
8. void notClosestToGoal();
9. };

// Criação das Instigations “mtClosestToGoal” e “mtNotClosestToGoal”
// no arquivo de códigos-fontes StrategyPON.cpp Method que instância as Instigations
10. void StrategyPON::initMethodPointers() {
11. ...
12. mtClosestToGoal = new MethodPointer<StrategyPON>(this, &StrategyPON::closestToGoal);
13. mtNotClosestToGoal = new MethodPointer<StrategyPON>(this,
&StrategyPON::notClosestToGoal);
14. }

// Implementação do comportamento dos Methods no arquivo de códigos-fontes StrategyPON.cpp
15. void StrategyPON::closestToGoal(){
16. this->robot->atClosestToGoal->setValue(true);
17. }

18. void StrategyPON::notClosestToGoal(){
19. this->robot->atClosestToGoal->setValue(false);
20. }
```

Fonte: Autoria própria.

2.3 CONDITION E PREMISES

As *Premissas (Premises)* que compõem a entidade *Condição (Condition)*, como apresentadas no exemplo de *Rules* ainda na Figura 1, são responsáveis pela realização das verificações que definem a tomada de decisão de uma *Rule*.

A implementação da tecnologia PON, no aplicativo de controle de uma partida de Futebol de Robôs, no que diz respeito as *Premises* está exemplificada no Quadro 3. No primeiro momento do exemplo, declara-se a *Premise* “prNotClosestToGoalX” como um ponteiro (*) para objetos da classe “Premise” (Quadro 3, Linha 4). Na segunda etapa do processo no método “initPremises”, a *Premise* “prNotClosestToGoalX” é instanciada a partir da classe “Premise” no método “PREMISE” com o seguinte pseudocódigo da expressão lógica: *atPosX <= 7985* (Quadro 3, Linha 8). Neste contexto, a *Premise* “prNotClosestToGoal” é responsável por verificar se um robô “não” está próximo a linha do gol atacado verificando se o estado/valor do *Attribute* “*atPosX*” (Quadro 1, Linha 5) é menor ou igual (SMALLERTHAN) ao valor *double* 7985 (*new Double(this, 7980)*).

Quadro 3 – Exemplo da implementação de Premises

```
//Declaração do Atributo “prClosestToGoal”, como um ponteiro para objetos da classe “Premise”,  
//no arquivo de cabeçalhos “StrategyPON.h”  
1. class StrategyPON : public Strategy, public FBE {  
2. protected:  
3. ...  
4. Premise *prNotClosestToGoalX;  
5. };  
  
//Method que instancia as Premises: criação da Premise “prClosestToGoal”  
6. void StrategyPON::initPremises() {  
7. ...  
8. PREMISE(prNotClosestToGoalX, this->robot->atPosX, new Double(this, 7980),  
         Premise::SMALLERTHAN, Premise::STANDARD, false);  
9. }
```

Fonte: Autoria própria.

2.4 ACTION E INSTIGATIONS

As *Instigações (Instigations)*, que compõem a entidade *Ação (Action)*, como demonstradas no exemplo de *Rules* novamente na Figura 1, são responsáveis através de seus *Methods* pela realização dos serviços ou habilidades associadas a uma *Rule*.

A demonstração da forma de implementação das *Instigations*, novamente no aplicativo de controle da partida de Futebol de Robôs, pode ser observada no Quadro 2. A *Instigation* é criada e atrelada diretamente ao objeto *Rule* “rlNotClosestToGoalX” (Quadro 4, Linha 4) por meio do método *INSTIGATION* (Quadro 4, Linha 10). Esta

Instigation instigará o *Method* “*mtNotClosestToGoal*” (Quadro 2, Linha 5).

2.5 RULES

Uma *Rule* é decomposta em entidades *Condition* (condição, conjunto de *Premises*) e *Action* (ação, conjunto de *Instigations*), como pode ser observado ainda na Figura 1. A *Condition* é formada pelas *Premises* associadas à *Rule* de modo que a ativação delas define a execução ou não da *Action*. Cada *Rule* desempenha o papel de conjunção ou disjunção unicamente para todas as *Conditions* em que está atrelada.

Uma das *Rules* implementada no aplicativo de controle da partida de Futebol de Robôs (Apêndice D), pode ser observada no Quadro 4. Primeiramente, declaram-se a *Rule* “*rlNotClosestToGoalX*” como um ponteiro (*) para objetos da classe “*RuleObject*” (Quadro 4, Linha 4). A seguir, na próxima etapa do processo no *Method* “*initRules*”, cria-se a *Rule* com 3 (três) linhas de implementações: a) *Rule* “*rlNotClosestToGoalX*” é instânciada a partir da classe “*RuleObject*” no *Method* “*RULE*” (Quadro 4, Linha 8); b) adiciona-se a *Premise* “*prNotClosestToGoalX*” (Quadro 3, Linha 4) na *Rule* “*rlNotClosestToGoalX*” (Quadro 4, Linha 9); e c) adiciona-se a *Instigation* criada a partir do *MethodPointer* “*mtNotClosestToGoal*” com o método *INSTIGATION* na *Rule* “*rlNotClosestToGoalX*” (Quadro 4, Linha 10).

Quadro 4 – Exemplo da implementação de Rules

```
//Declaração da Rule "rlNotClosestToGoalX", como um ponteiro para objetos da classe "RuleObject"
//no arquivo de cabeçalhos StrategyPON.h
1. class StrategyPON : public Strategy, public FBE {
2. protected:
3. ...
4.     RuleObject* rlNotClosestToGoalX;
5. };

//Method que instancia as Rules: criação da Rule "rlNotClosestToGoalX"
6. void StrategyPON::initRules(){
7. ...
8.     RULE(rlNotClosestToGoalX, scheduler, Condition::SINGLE);
9.     rlNotClosestToGoalX->addPremise(prNotClosestToGoalX);
10.    rlNotClosestToGoalX->addInstigation(INSTIGATION(this->mtNotClosestToGoal));
11. }
```

Fonte: Autoria própria.

A *Rule* “*rlNotClosestToGoalX*”, implementada novamente no Quadro 4, se comporta de forma a determinar se o robô “não” está próximo do gol em relação a coordenada “x”, valor do *Attribute* “*atPosX*” (Quadro 1, Linha 5), expressão lógica implementada na *Premise* “*prNotClosestToGoalX*” (Quadro 3, Linha 4). Se não está próximo ao gol, então define o estado do *Attribute* “*atClosestToGoal*” (Quadro 1, Linha

- 4) como *false*, comportamento implementação no *Method* “*mtNotClosestToGoal*” (Quadro 2, Linha 11).

2.6 FLUXO DE EXECUÇÃO NO PON

No Paradigma Orientado a Notificações (PON) o fluxo de execução de uma aplicação inicia-se a partir da mudança de estado de um *Attribute* que “notifica” todas as *Premises* pertinentes, a fim de que estas reavaliem seus estados lógicos. Caso o valor lógico de uma *Premise* se altere, a *Premise* colabora com a avaliação lógica de uma ou de um conjunto de *Conditions* conectadas, o que ocorre por meio da “notificação” sobre a mudança relacionada ao seu estado lógico (VALENÇA, 2012, p. 29; RONSZCKA, 2019, p. 29; BANASZEWSKI, 2009, p. 11).

Consequentemente, cada *Condition* notificada avalia o seu valor lógico de acordo com as notificações recebidas das *Premises* e com um dado operador lógico, sendo este em geral um operador de conjunção (*and*) ou disjunção (*or*). Assim, no caso de uma conjunção, por exemplo, quando todas as *Premises* que integram uma *Condition* são satisfeitas, a *Condition* também é satisfeita. Isto resulta na aprovação de sua respectiva *Rule* que pode “então” ser executada (*Action*) (VALENÇA, 2012, p. 29).

Ainda segundo Valença (2012, p. 29), com a *Rule* aprovada a sua respectiva *Action* é ativada. Uma *Action*, por sua vez, é conectada a um ou vários *Instigations*. As *Instigations* colaboram com as atividades das *Actions*, acionando a execução de algum serviço por meio dos seus *Methods*. Usualmente, os *Methods* alteram os estados dos *Attributes*, recomeçando assim o ciclo de notificações.

2.7 LINGUAGENS DE PROGRAMAÇÃO NO PON

Santos (2017, p. 21) relata que os conceitos do PON foram primeiramente materializados no Paradigma Orientado a Objetos, através de um *framework* desenvolvido na Linguagem de Programação C++ em 2007 pelo Prof. Jean Marcelo Simão e que teve posteriormente a versão 1.0 desenvolvida no trabalho de Banaszewski (2009). Em 2012, a nova versão nomeada de *Framework PON C++ 2.0* foi implementada nos trabalhos de Valença (2012) e Ronszcka (2012).

No caso da categoria de linguagem de programação específica para o PON, nomeada de LingPON, a versão 1.0 com o respectivo compilador foi desenvolvida e apresentada no projeto de mestrado de Ferreira (2015). Posteriormente, o trabalho de Santos (2017) propôs a LingPON versão 1.2.

3 INFORMAÇÕES SOBRE O FUNCIONAMENTO DO FUTEBOL DE ROBÔS

Nesta segunda seção serão apresentadas informações de como uma partida de Futebol de Robôs é jogada. Tais informações são necessárias para que haja o entendimento de *Rules* implementadas.

3.1 REQUISITOS FUNCIONAIS

Aqui, estão apresentados requisitos funcionais baseados na regulamentação atual da competição de Futebol de Robôs (RoboCup), documento “Rules of the RoboCup Small Size League” (RULES, 2020), a fim de simplesmente garantir uma partida dentro desta regulamentação. São elas:

- a) Quando em “fora de jogo”, a bola não pode ser manipulada, ou seja, um jogador não pode realizar *dribbling* ou *shooting*. *Dribbling* é o ato manipular a bola de maneira geral, podendo mover-se com ela.
- b) Não é permitido executar *dribbling* por mais de 1m, ou seja, manter contato com a bola (segurá-la) por mais de um metro.
- c) Após cobranças, o jogador que cobrará não pode encostar na bola após ela ter se movido 50mm (o que torna a bola em jogo) sem que outro jogador a tenha tocado após esse movimento. Essa infração é chamada de toque duplo e possui uma tolerância equivalente a movimentação para a bola estar em jogo, há uma tolerância de 50mm para o toque duplo. Ou seja, o jogador deve executar passe ou chute quando em cobranças.
- d) Quando em *stop*, os robôs devem se movimentar abaixo de 1,5m/s, devendo manter-se a 500mm de distância da bola, não podendo manipulá-la, e devem se posicionar a 200mm da área de defesa do oponente. Há uma tolerância de 2 segundos.
- e) Quando em *halt*, nenhum robô pode mover-se ou manipular a bola. Há uma tolerância de 2 segundos.
- f) Em *direct kicks* pode ser realizado gol com o chute. Os jogadores defensores devem se posicionar a pelo menos 500mm da bola e os jogadores atacantes podem se posicionar próximos à bola.
- g) O *indirect kick* possui regras semelhantes ao *direct kick*, mas não pode ser realizado gol com o chute (pelo menos um jogador atacante que não o *kicker*, após a cobrança, deve tocar na bola antes da validade de um gol).

- h) Após o comando *force start*, ambos os times podem manipular a bola.
- i) Após o comando de pênalti, o goleiro defensor deve tocar a linha do gol e um jogador atacante pode se aproximar da bola e será o cobrador. Todos os outros robôs devem se posicionar a uma distância mínima demarcada por uma linha a 400mm da marca de pênalti, paralela à linha do gol.
- j) Após o comando *normal start*, deve ser avaliado também se o estado anterior era pênalti, para execução de chute do cobrador.
- k) Após o comando *normal start*, deve ser avaliado também se o estado anterior era *kick-off*, para execução de chute ou passe do cobrador.
- l) De mesmo modo, observa-se que apenas o goleiro pode entrar na sua área de defesa.
- m) Um robô não pode executar “chute sem mira”.
- n) Um robô não pode chutar a bola de forma que esta ultrapasse a velocidade de 6,5m/s.
- o) Um robô não deve ir de encontro direto a outro robô para evitar “colisão” e “empurrão”.
- p) Quando em chute a gol, a bola não pode ser elevada mais do que 150mm, ou o gol é declarado como inválido.
- q) Quando a bola ultrapassa as linhas de campo, há cobranças como tiro de meta ou laterais.

3.2 REGRAS

A seguir estão apresentadas *Regras (Rules)*, *Premissas (Premises)* e *Métodos (Methods)* necessários para que o funcionamento dos robôs siga de acordo com o estipulado pelos requisitos apresentados na seção “2.1 Requisitos Funcionais”. São elas:

- a) Premissa que avalia se o comando atual é *normal start*, se o anterior era pênalti, se o jogador é cobrador e método que realiza chute.
- b) Limitar a velocidade de chute ou passe nos respectivos métodos.
- c) Impedir avanço sobre robô adversário, especialmente se este está se deslocando na direção do robô em questão. Limitação em premissa ou método de movimentação.
- d) Limitação no método de chute.
- e) Não há razão para um robô posicionar-se além das linhas de campo.

3.3 NOTAS IMPORTANTES

Não é necessário haver um goleiro segundo o documento “Rules of the RoboCup Small Size League”, nas seções “3.1 Number Of Robots” e “4.3.5 Choosing Keeper Id” (RULES, 2020).

Pode-se segurar a bola, desde que não haja restrição a todos os seus graus de liberdade, pelo menos 80% da área da bola (RULES, 2020) deve ser visível quando em vista superior, não eleve a bola do chão e seja possível que outro robô adquira a posse de bola. Uma forma de implementação, portanto, pode-se ser exercer força limitada na bola em direção ao robô com posse.

4 DESENVOLVIMENTO DA APLICAÇÃO PARA CONTROLE DO FUTEBOL DE ROBÔS

Nesta quarta seção, será feita a apresentação da proposta de trabalho de conclusão de disciplina que envolve o desenvolvimento de *Rules* no software de controle dos robôs disponibilizado para estudo (Apêndice B) visando a simulação de uma partida de Futebol de Robôs em ambiente de simulação.

4.1 ATIVIDADE PROPOSTA

A atividade proposta durante a disciplina era de realizar a implementação de um novo conjunto de *Rules* com base no trabalho desenvolvido na dissertação do então mestrando Leonardo Araujo Santos (SANTOS, 2017), afim de melhorar o comportamento dos robôs.

4.2 DIFICULDADES ENCONTRADAS

Durante o desenvolvimento do trabalho foram encontradas diversas dificuldades, principalmente no que se refere ao ambiente proposto.

No uso da máquina virtual disponibilizada (Apêndice A), em computadores com processador AMD ocorriam algumas falhas de execução, e foi necessário o *download* do código fonte e a recompilação da aplicação “grSim”. Esse problema foi causado pelo fato das bibliotecas já virem pré-compiladas, e ao utilizar um processador diferente as mesmas eventualmente vem a apresentar falhas.

Além desse problema inicial também foi constatado que a versão de código disponibilizado neste ambiente não correspondia à versão final da dissertação, mas sim uma versão mais antiga.

Por conta disso inicialmente foi feita uma tentativa de atualizar o código (Apêndice B) com as novas *Rules*, utilizando um outro código disponibilizado da versão final de Santos (2017) (Apêndice C), porém isso se mostrou bastante complicado pois havia uma diferença muito grande entre a estrutura base da interface dos robôs sob a qual essas versões estavam implementadas, conforme a regulamentação de competição de Futebol de Robôs (RoboCup) descrito no capítulo 3, portanto ao tentar compilar e executar o código resultava em muitos erros de compilação, que quando resolvidos apenas geravam erros de execução (*core dump*). Esses erros podem ter sido causados devido a diferenças nas bibliotecas sob as quais os mesmos foram implementados.

Não obtendo sucesso foi observado que não seria muito produtivo prosseguir nessa tentativa de recompilar o código. Desta forma o que se seguiu foi uma tentativa de se adaptar as *Rules* propostas por Santos (2017) (Apêndice C), no código funcional, porém mais uma vez a tarefa se mostrou complicada, devido estrutura geral do código e falta de algumas partes, tornou-se impossível a mesclagem dos códigos, funcional e de Santos (2017) (Apêndice C).

4.3 TRABALHO DESENVOLVIDO

Considerando as dificuldades apresentadas foi decidido finalmente que cada um dos alunos ficaria responsável pela implementação de 5 *Rules* próprias (Apêndices D e E), de modo a exercitar melhor os conhecimentos individuais sobre o PON e também obter como resultado um código funcional no qual seria possível observar a aplicação das *Rules* implementadas durante uma partida do Futebol de Robôs.

O Quadro 5 apresenta as *Rules* implementadas pelos integrantes do trabalho. Neste quadro é descrito o nome da variável adicionada ao código para novas *Rules*, uma descrição de funcionamento dela e seu responsável. As *Rules* têm papel de modificar o comportamento dos jogadores durante a partida.

Quadro 5 – Conjunto de *Rules* implementadas na aplicação para controle do Futebol de Robôs
(continua)

Rule	Objetivo	Responsável
rlPositionToShoot	Posicionar jogador para chutar ao gol quando estiver próximo. Verifica <i>Attributes</i> de estado do jogo e executa o <i>Method</i> que seta flag “atReadyToShoot” para <i>true</i> .	Felipe dos Santos Neves (Apêndice D)
rlShootToGoal	Chutar ao gol após estar posicionado. Verifica <i>Attributes</i> de controle de chute a gol e executa o <i>Method</i> que produz a ação de chute “mtShootToGoal”.	Felipe dos Santos Neves (Apêndice D)
rlResetShootFlag	Resetar flags de controle após realizar chute. Verifica os mesmos <i>Attributes</i> de controle de chute e também posse de bola. O chute é considerado feito quando “atBallIsMine” é <i>false</i> . Então as flags de controle de chute são atribuídas como <i>false</i> .	Felipe dos Santos Neves (Apêndice D)
rlMaxDrible	Passar a bola após exceder determinado número de passes. O Atribute “atDribbleCount” é incrementado a cada toque e “atMaxDribleCount” é randomizado entre 1 e 6 a cada execução da Rule. O Method força um passe e o reset de “atDribleCount”.	Felipe dos Santos Neves (Apêndice D)

Quadro 5 – Conjunto de *Rules* implementadas na aplicação para controle do Futebol de Robôs
(continua)

Rules	Objetivo	Responsável
rlPassGkCloseGoal	Forçar goleiro a passar a bola ao ficar com a posse dela muito próximo ao gol. Verifica se a distância ao gol “atBallDistTeamGoal” é menor que 1800 e outros <i>Attributes</i> de controle de posse de bola. Força o passe a outro robô. Só é aplicada em robô do tipo defesa.	Felipe dos Santos Neves (Apêndice D)
rlPenalty	Posicionar o jogador atacante para a posição de pênalti do campo. Tratamento para movimentar o jogador atacante para a posição de pênalti. Posição calculada com base nas dimensões do campo. O estímulo desta <i>Rule</i> é realizado pelo painel <i>Referee Box</i> no botão <i>Penalty</i>	Anderson Eduardo de Lima (Apêndice D)
rlYellowCardBlue	Captura do evento de cartão amarelo do time azul. Realiza o tratamento do estímulo do painel <i>Referee Box</i> no botão <i>Yellow Card</i> para os jogadores do time azul.	Anderson Eduardo de Lima (Apêndice D)
rlRedCardBlue	Captura do evento de cartão vermelho do time azul. Realiza o tratamento do estímulo do painel <i>Referee Box</i> no botão <i>Red Card</i> para os jogadores do time azul.	Anderson Eduardo de Lima (Apêndice D)
rlLimiteHorizontal	Evitar a saída de robôs do limite horizontal do campo (regra da categoria <i>Small Size League - SSL</i>). Verifica se o robô encontra-se fora do limite horizontal, pré-estabelecido, do campo.	Anderson Eduardo de Lima (Apêndice D)
rlLimiteVertical	Evitar a saída de robôs do limite vertical do campo (regra do SSL). Verifica se o robô encontra-se fora do limite vertical, pré-estabelecido, do campo.	Anderson Eduardo de Lima (Apêndice D)
rlClosestToGoal	Determinar se o robô está próximo do gol. Se está próximo ao gol (coordenadas “x” e “y”) então define o estado “atClosestToGoal” como <i>true</i> .	Omero Francisco Bertol (Apêndice D)
rlNotClosestToGoalX	Determinar se o robô “não” está próximo do gol em relação a coordenada “x”. Se não está próximo ao gol na coordenada “x” então define o estado “atClosestToGoal” como <i>false</i> .	Omero Francisco Bertol (Apêndice D)
rlBallFar	Reposicionar o robô. Redefinir chute e drible quando a bola estiver longe. Baseada na “Rule 5” de Santos (2017, p. 179).	Omero Francisco Bertol (Apêndice D)
rlNotClosestToGoalY	Determinar se o robô “não” está próximo do gol em relação a coordenada “y”. Se não está próximo ao gol em uma das coordenadas “y” então define o estado “atClosestToGoal” como <i>false</i> .	Omero Francisco Bertol (Apêndice D)

**Quadro 5 – Conjunto de *Rules* implementadas na aplicação para controle do Futebol de Robôs
(conclusão)**

Rule	Objetivo	Responsável
rlClosestToGoalKick	Determinar se o robô deverá realizar o chute a gol. Se o jogador está pronto, está com a bola e está próximo ao gol então chuta para o gol.	Omero Francisco Bertol (Apêndice D)
rlRobotWantsToMoveX	Avalia necessidade de movimento em X.	Lucas Tachini Garcia (Apêndice D)
rlRobotWantsToMoveY	Avalia necessidade de movimento em Y.	Lucas Tachini Garcia (Apêndice D)
rlRobotWantsToMove	Confirma a necessidade de movimento em X ou Y.	Lucas Tachini Garcia (Apêndice D)
rlRobotNotRetrained	Esta <i>Rule</i> deve agrupar condições negadas que possam vir a restringir o movimento do jogador.	Lucas Tachini Garcia (Apêndice D)
rlRobotMoveGoalKeeper	Permite ou não que o movimento do goleiro seja executado.	Lucas Tachini Garcia (Apêndice D)
rlNextMoveToRestrictedArea	Avalia movimentação para áreas não permitidas.	Lucas Tachini Garcia (Apêndice D)
rlFixRobotNotGkmove	Modifica o movimento do jogador que não seja o goleiro.	Lucas Tachini Garcia (Apêndice D)
rlFixRobotGkmove	Modifica o movimento do goleiro.	Lucas Tachini Garcia (Apêndice D)
rlRedCardYellow	Aplica cartão vermelho para um jogador do time amarelo.	Lucas Tachini Garcia (Apêndice D)
rlBallOutsideEnemyTeam	Se a bola sai da área do jogo no campo adversário, força o jogador a voltar para sua posição inicial.	Lucas Tachini Garcia (Apêndice D)
rlPassWhenOff	Determinar se o robô deve passar a bola e retornar ao centro do campo. Jogador com a bola, avançado no campo adversário e fora de centro, passa a bola e retorna ao centro.	Luis Henrique Sant'Ana (Apêndice D)

Fonte: Autoria própria.

A finalidade das *Rules* criadas foi presenciar a modificação no comportamento dos jogadores durante simulação causada pela adição das instruções PON. Como muitas das funcionalidades descritas no projeto “RoboCup2012” desenvolvido em Santos (2017) estavam faltando algumas partes do código, desta forma impossibilitando a compilação, algumas destas novas funcionalidades não estão adequadas às regras vigentes para uma partida de Futebol de Robôs Oficial (RoboCup).

5 CONCLUSÃO

Com o objetivo de aplicar na prática os temas abordados nas disciplinas com enfoque no Paradigma Orientado a Notificações (PON) foi proposto a aplicação da tecnologia PON, configuração do ambiente de simulação (Apêndice A), análise de *Rules* em trabalhos existentes (Apêndices B e C) e o desenvolvimento de novas *Rules* do software para o simulador visando controle do Futebol de Robôs (Apêndices D e E).

Notadamente, foram levantadas informações sobre o regulamento da competição do Futebol de Robôs (RoboCup), como é realizada a condução de uma partida, *Rules*, *Premises* e *Methods* necessários para que o funcionamento de um jogador robô siga em acordo com os requisitos funcionais estabelecidos no regulamento.

Subsequentemente, realizou-se a apresentação do Paradigma Orientado a Notificações (PON) como uma alternativa para outros paradigmas de programação, entre eles, o Paradigma Imperativo e Paradigma Declarativo. O estudo da tecnologia PON mostrou seus elementos fundamentais materializados na forma de seus elementos essenciais na forma de *Atributos* (*Attributes*), *Métodos* (*Methods*), *Condição* (*Condition*), *Premissas* (*Premises*), *Ação* (*Action*), *Instigações* (*Instigations*) e *Regras* (*Rules*). O fluxo de execução de aplicações no PON foi destacado mostrando o desacoplamento em expressões lógico-causais, inexistente em outros paradigmas, o que evita o processamento desnecessário com redundâncias causais e/ou utilização custosa de estruturas de dados. Importância também foi dada as linguagens de programação que materializam a tecnologia PON, são elas: a) *Framework* PON C++ 2.0, desenvolvido na Linguagem de Programação C++; e b) LingPON, linguagem específica para o PON.

Por fim, foi apresentado o desenvolvimento de novas *Rules*, com base no trabalho de Santos (2017), afim de melhorar o comportamento dos jogadores no aplicativo para controle de Futebol de Robôs, executado em ambiente de simulação configurado em máquina virtual. Tarefa essa que apresentou diversas dificuldades, principalmente, no que se refere ao ambiente proposto como por exemplo, na questão das bibliotecas já estarem pré-compiladas e se mostraram incompatíveis com processadores diferentes provocando falhas na sua utilização. A nível de códigos-fontes a versão disponibilizada para estudo não correspondia à versão final de Santos (2017), mas sim uma versão anterior. Considerando as dificuldades, foram então apresentadas as *Rules* que cada um dos alunos implementou aplicando os conhecimentos individuais adquiridos sobre a tecnologia PON.

REFERÊNCIAS

- BANASZEWSKI, Roni Fabio. **Paradigma Orientado a Notificações: Avanços e comparações.** 2009. 285 f. Dissertação de Mestrado – Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial (CPGEI), Universidade Tecnológica Federal do Paraná (UTFPR). Curitiba, 2009. Disponível em: <<http://livros01.livrosgratis.com.br/cp087456.pdf>>. Acesso em: 16 fev. 2020.
- FERREIRA, Cleverson Avelino. **Linguagem e compilador para o paradigma orientado a notificações (PON): avanços e comparações.** 2015. 227 f. Dissertação (Mestrado em Computação Aplicada) – Universidade Tecnológica Federal do Paraná, Curitiba, 2015. Disponível em: <<http://repositorio.utfpr.edu.br/jspui/handle/1/1414>>. Acesso em: 22 mai. 2020.
- RONSZCKA, Adriano Francisco. **Contribuição para a concepção de aplicações no paradigma orientado a notificações (PON) sob o viés de padrões.** 2012. 224 f. Dissertação de Mestrado – Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI), Universidade Tecnológica Federal do Paraná (UTFPR). Disponível em: <http://repositorio.utfpr.edu.br/jspui/bitstream/1/327/1/CT_CPGEI_M_%20Ronszcka,%20Adriano%20Francisco_2012.pdf>. Acesso em: 23 mar. 2020.
- RONSZCKA, Adriano Francisco. **Método para a criação de linguagens de programação e compiladores para o Paradigma Orientado a Notificações em plataformas distintas.** 2019. 375 f. Tese de Doutorado – Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial (CPGEI), Universidade Tecnológica Federal do Paraná (UTFPR). Curitiba, 2019. Disponível em: <<http://repositorio.utfpr.edu.br/jspui/handle/1/4234>>. Acesso em: 25 fev. 2020.
- RULES of the RoboCup Small Size League 2019. **RoboCup Federation**, 19 mar. 2020. Disponível em: <<https://robocup-ssl.github.io/ssl-rules/sslrules.html>>. Acesso em: 08 fev. 2020.
- SANTOS, Leonardo Araujo. **Linguagem e compilador para o paradigma orientado a notificações: Avanços para facilitar a codificação e sua validação em uma aplicação de controle de Futebol de Robôs.** 2017. 293 f. Dissertação de Mestrado – Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial (CPGEI), Universidade Tecnológica Federal do Paraná (UTFPR). Curitiba, 2017. Disponível em: <<http://repositorio.utfpr.edu.br/jspui/handle/1/2778>>. Acesso em: 16 fev. 2020.
- VALENÇA, Glauber Zárate. **Contribuição para a materialização do Paradigma Orientado a Notificações (PON) via framework e wizard.** 2012. 205 f. Dissertação (Mestrado em Computação Aplicada) – Programa de Pós-graduação em Computação Aplicada (PPGCA), Universidade Tecnológica Federal do Paraná (UTFPR). Curitiba, 2012. Disponível em: <<http://repositorio.utfpr.edu.br:8080/jspui/handle/1/393>>. Acesso em: 16 fev. 2020.

APÊNDICE B

Comparisons on Game Development with Unreal Engine 4 using Object-Oriented Paradigm (OOP in C++) versus Notification Oriented Paradigm

Felipe dos Santos Neves

Graduate student at PPGCA¹ - DAINF² - UTFPR³

Report in article form to the course Programming Paradigms

(1st trimester of 2020) at CPGEI⁴ - UTFPR³ – Prof. Jean Marcelo Simão.

Curitiba, Paraná, Brazil

fneves@alunos.utfpr.edu.br

Abstract—This paper draws comparisons on the usage of the novel Notification Oriented Paradigm for game development with the traditional approach using Object-Oriented C++. For this purpose, a simple game is developed using the Unreal Engine 4, implementing the same game logic with both the Object-Oriented Paradigm and Notification Oriented Paradigm. A formal software development cycle, from requirements to design and implementation is used to ensure that both implementations have the same final result, enabling the comparisons of the time spent in development and number of lines of code necessary for each implementation. The software developed using the Notification Oriented Paradigm generated more lines of code, however less time was spent during development, showing that in this case using the Notification Oriented Paradigm was beneficial to reduce the software development time.

Index Terms—Game Development Methodologies, Unreal Engine 4, NOP Versus OOP Experimentation

I. INTRODUCTION

The development of electronic video games dates back to the early 1950s, where computers first began to be available to civilians, and the first graphical video game appeared in 1958, with the vastly known game called Pong, which was entirely hardware based [1]. Since then game Development has grown to a multibillion industry, reaching \$120.1 billion revenue in 2019 [2].

Game Development is a widely multidisciplinary field, involving fields such as arts, design, business and software development. And as digital games grow in size and complexity each year the industry pushes forward the fields of both hardware and software development.

In the early days of computer assisted game development most games had software built from the ground up specifically for that purpose, but in the core of any modern game is a Game Engine, which is the core software responsible for handling the doing the rendering, calculating physics, lighting, playing

sounds and much more. These engines provide the tools for the developers to bring their design to work without much more ease than having to rebuild everything for each game. Of course, some cutting edge development still requires the development of a new engine from time to time to better fit the requirements of the game that is being developed [3].

Given the nature of such applications, as computer games can be roughly described as a group of objects that interact with each other and that have to be rendered on a screen on a fixed interval, both the games and the engines themselves are almost always implemented in the Object-Oriented paradigm. The engine used in this study is the Unreal Engine version 4, which has its API available in C++. This is one of the most used tools in professional game development, and shows how important the Object-Oriented paradigm is in the game industry.

Given this context this paper's goal is to show how the novel Notification Oriented paradigm can be used as a great asset to extend toolset developers have, where the Object-Oriented paradigm shows its limitations. In its current implementation there is a version of the Notification Oriented Paradigm available as a C++ library [4], which makes it suitable and easy to interface with the Unreal Engine API, providing a convenient way to study the viability of the application of the Notification Oriented Paradigm for game development.

Given the objective of this paper to compare the two approaches to the development of a game we establish a set of requirements that should be present in both implementations. Also, after developing for the initial requirements there will be a second set of modified requirements, where some of the requirements are modified and new ones are added, for the purpose of evaluating how easy it would be to modify the software after it is already done for the initial requirements.

II. UNREAL ENGINE

The Unreal Engine was chosen for this project due to it being one of the most used engines in commercial game Development, as illustrated in Figure 1, from data gathered from Steam [5], the major games storefront for PC. Besides

¹Programa de Pós-Graduação em Computação Aplicada

²Departamento Acadêmico de Informática

³Universidade Tecnológica Federal do Paraná

⁴Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial

that, it also has its API in C++, making it easy to interface with the NOP C++ Framework 2.0 [4], which could be compiled as a library and loaded in the Unreal Engine project.

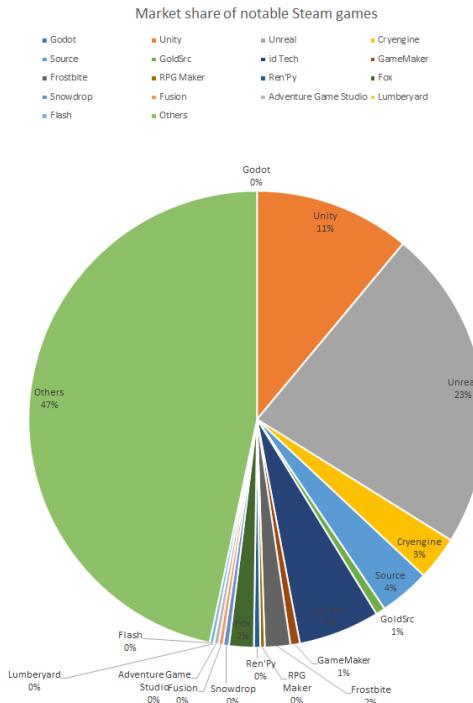


Fig. 1: Game engines market share from Steam [5]

While most of the game's implementations can be done in C++, the Unreal Engine also has its own visual programming language called *Blueprints*, that allows the users to use the API through visual blocks and connections. This is an excellent way to speed up the development, as it is very easy and intuitive, and also enables users without programming language to use the engine. There is extensive documentation provided for both the C++ and *Blueprints* and the Unreal Engine website [6], in this paper only the core concepts required in order to provide a basic understanding of the implementations of this project.

While being easier to use the *Blueprints* has a very robust interface to call methods from the native C++ API. The only downside is that it may be slower, and using the native API can result in a faster implementation, depending on how the code is written, and also allows for a more advanced use. For the most part for simpler games they can be made entirely using *Blueprints*, or used in complement with core implementations in C++, which the engine provides the tools to export to be used with *Blueprints*. This means that you can expand write your own classes in C++ and use the available tools from the

engine, through some clever use of macros, and convert it to a pure *Blueprints* object.

A. Core Concepts

One of the core concepts of the Unreal Engine is that every object is derived from the base class *UObject*, which contains the base structure for every object that you want to be handled by the engine. This is because the engine expects to handle the two core loops for every object *BeginPlay* and *Tick*. Also, every object that is created is inserted into a list of all objects in the game world, so that the engine can handle all of them. The engine handles both graphical and non-graphical objects as well.

The *BeginPlay* function relates to a usual C++ class constructor. The *BeginPlay* function is called every time a instance of the class is invoked into the game world. It is very similar to what a constructor function would do in a normal C++ program, but the difference in this case is that the engine uses the constructor to pre-load assets, such as graphics and sounds, into memory. This is done to provide better memory efficiency and speed, as you cannot have heavy objects being loaded into memory during gameplay, so the engine pre-loads all the classes before the level starts, that would be what happens during a loading phase. And then there is the *BeginPlay* function that is only called when the object is effectively spawned into the world.

Another very important function is the *Tick* function. This function is actually optional, as it can be very resource intensive, as it is called for every tick of the engine, or for every frame that is drawn on the render. Objects that do not need to run every tick can disable this feature, but it is usually enabled for most objects that have any sort of interaction.

As it becomes evident from these concepts, for graphical objects in general, the code must run loops that update on every frame, what could contradict the principles of the Notification Oriented Paradigm, that would avoid those loops altogether. But as the game engine abstracts most of these graphical elements and the Notification Oriented Paradigm can be used, not to substitute the traditional Object Oriented approach, but to complement and improve the design of the other non-graphical related aspects of the game code, such as the enemy behavior and game systems and rules.

III. NOTIFICATION ORIENTED PARADIGM

A. Objectives

The main objectives of the Notification Oriented Paradigm are to solve some of the many problems present in the Imperative and Declarative Paradigms [7] [8]. The first problem that can be mentioned is the redundancy present on the Imperative Paradigm, since it is strongly dependent on loops (such as if, while and for loops) for evaluating variable states during the program execution, but most of these variables will never change during each interaction, resulting in a code that will run multiple times without achieving any different results [7].

Besides that one of the other main problems is distribution, since the same variables can be evaluated in multiple different

sections of the code, and the passivity of its elements bring a close dependency to the data and commands within a program, making it harder to split into multiple execution units. In scenarios where you cannot control the execution order of the parts of your code it can cause many problems [8].

As such is the case of Unreal Engine objects, where the engine is responsible for choosing which elements to evaluate first, the developer has no input on that matter, and that can bring problems where there is a close dependency on variables that may change their states before a dependent state has its chance to be evaluated.

B. NOP Entities

This section presents in greater detail the structures that compose the Notification Oriented Paradigm implementation.

1) *Fact Base Element*: The Figure 2 describes the main entity of the Notification Oriented Paradigm, the Fact Base Element (FBE). It is a class used to describe the states and services of our objects. An FBE contains attributes and methods, which are collaborative objects. The difference between an FBE and a traditional Object Oriented class is that the attributes and methods are not simple passive entities [8].

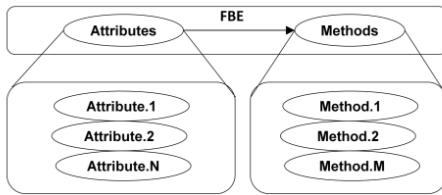


Fig. 2: FBE Structure
[8]

2) *Attribute*: The attributes store the values that represents the current states of the properties of given FBE. It is different from a traditional member variable from the point that it is able to notify its relevant premises when its state changes. The ability to notify the premises, that can be members of the same object or be part of a different entity is what gives the advantages in respect to parallel execution and decoupling.

3) *Premises*: The premises represent the evaluation of the state of a given attribute. When the attribute notifies the premises, it is then evaluated against a given value or against another attribute. The premises can be evaluated as true or false, and when its value changes it notifies the relevant conditions.

4) *Condition*: The condition is composed from premises, that are evaluated with either a conjunction or disjunction operation. It notifies the relevant rules when it is approved.

5) *Rule*: A rule, as illustrated in Figure 3, is composed by a condition and an action, in a causal manner, when the rule's condition becomes true it then the rule becomes approved for execution, executing its action. Indirectly, the rule is composed by premises, and instigations.

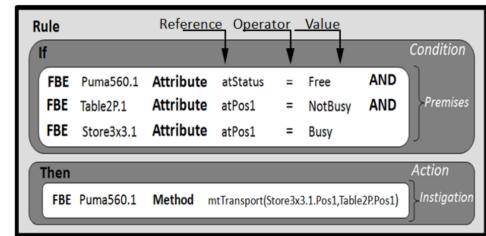


Fig. 3: Rule Structure
[8]

6) *Action*: The action may contain multiple instigations. Its activated when the rule is approved. It is responsible for calling the connected instigations.

7) *Instigation*: The responsibility of the instigation is to call the methods of the objects, passing their appropriate parameters, when required. These methods can either be NOP attribute changes or also regular C++ function calls. By allowing the call of the regular code functions it gives the flexibility to integrate with other frameworks such as the Unreal Engine API.

The chain of events from the attribute changes to the method execution as described in the Section III-B is also illustrated in the Figure 4.

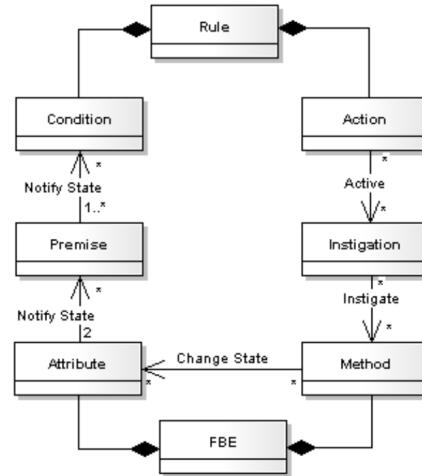


Fig. 4: NOP Notification Mechanism Diagram
[8]

IV. PROGRAMMING CONCEPTS

As defined in by Peter Van Roy [9], programming paradigms are composed of smaller conceptual units, called programming

concepts. He defines 4 concepts that he considers the most important.

1) *Record*: A data structure, as a group of references that can be accessed through indexes to each item. Is present in the PON C++ framework as inherited from C++. It can be seen in the FBE structure, which is composed of other structures such as attributes and methods.

2) *Lexically Scoped Closures*: In the PON C++ framework, as all the entities described in Section III-B are implemented as classes from C++ we inherit this concept.

3) *Independence (Concurrency)*: A program can be constructed as independent parts. When two parts do not interact, they are concurrent, as opposed to when the order of execution is defined and they are sequential. All the PON entities are by definition independent and support parallelism, as their behavior is determined by the notification mechanism, which supports concurrency. In this project it is very clear to observe this behavior as all the objects constructed in the Unreal Engine act as separate and independent threads therefore are executed concurrently.

4) *Named State*: Is the capacity of storing information in a computational element with a name, so that it can be accessed and modified through the code. It is very important in the PON framework so that attributes, premises, conditions and rules can be defined, and the instances of these elements can be shared through references to their names.

These concepts were later analyzed and classified in relation to the Notification Oriented Paradigm [10]. And these concepts are then validated and discussed below derived from the observations applying the Notification Oriented Paradigm through the C++ Framework in this project in Table I.

TABLE I: Programming Concepts in the NOP

Concept	Present
Record	Yes
Lexically Scoped Closures	Yes
Independence (Concurrency)	Yes
Named State	Yes

V. IMPLEMENTATION

Regarding the design of the game itself the was to reduce the complexity related to the graphics so that it would be possible to focus more on the software development itself. For that purpose the game was designed as a top-down shooter, which is composed of a player and enemies that are able to move in a two-dimensional environment. This is an appropriate approach for this study as it gives two degrees of freedom for the movement of the entities while still remaining fairly easy to implement when compared to full three dimensional environment. The graphics themselves are also implemented just as necessary to provide a minimum viable demo, and to give suitable player feedback so that the game can be enjoyed.

As for the NOP implementation specifics, it is still constrained to the Object-Oriented approach that comes from the

engine for the basic objects structure, but these objects are now also Fact-Based Entities. To comply with the NOP concepts, the basic iteration loops of the Unreal Engine are still used, but variable evaluation and function calls during these loops are avoided, being only used for the attribution of updated values to the FBE's attributes.

In Figure 5 is shown a screenshot of the implemented game, with the player in the center, surrounded by enemies with behavior following the game logic implemented using the NOP.



Fig. 5: In-Game screenshot

A. Requirements Specification

Given the fact that the two implementations have the same objective it was clear that writing a good set of requirements to guide both of them. Having a strict set of Functional and Non-Functional Requirements allows us to ensure that both implementations would have the same final functionality. These full set of requirements is described in Table II and III.

The initial set of requirements in Table III was designed to achieve a minimum viable product, a simple but functional game. We do not enter in much detail about the graphical aspects of the project, such as models, textures, sounds and effects as they are outside of the scope of the study.

As one of our objectives was to evaluate how both implementations could be modified a second set of modified requirements was defined with the objective to improve upon the original game design and is presented in Table IV. With these modifications it is possible to evaluate how easy is to modify an already finished code with the different approaches.

One of the benefits of developing with NOP based on functional requirements is how easy it is to relate the requirements with the code, due to how the program is structured, it more clearly reproduces the rules of a requirement into the code, making it easier to understand and to develop. This relationship between rules and requirements is extremely beneficial to the system, as it facilitates and encourages better traceability of the requirements during the whole product life cycle [11].

B. Class Diagram

The classes were designed to respect a similar hierarchy in both implementation, as well as reutilizing elements that

TABLE II: Non-Functional Requirements

Requirement	Description
NFR-1	The game shall be implemented using C++, the NOP C++ Framework 2.0 and blueprints with Unreal Engine 4
NFR-2	All elements shall be implemented using both only C++ and C++ with the NOP Framework while keeping the same functionality
NFR-3	The game shall be composed of a single level that can be played with both implementations

did not change in both implementations, such the classes for the abilities, derived from UAbility. The Object-Oriented only design is presented in Figure 6 and the NOP design in Figure 7. The only difference in both class diagrams is that in the NOP design the Player and Enemy classes derive from both AActor class and the FBE class and the names are different to separate both implementations, as they run simultaneously in the same application. The similar structure also allows to reutilize some of the code used to implement basic functionality, such as controlling the player, moving the objects and using weapons, only requiring minor modifications between each implementation and thus keeping both codes as similar as possible, and this is important to be able to evaluate the differences in the amount of lines of code as well.

C. Rules Diagram

For the NOP implementation there are a total of 8 classes that are derived from the FBE class, meaning that they have a set of rules in their implementation, and it is present below the textual description of all the rules and their conditions. These rules were designed in order to achieve the requirements presented in the section V-A. The rules are described in tables V to XXI.

One of the advantages of a software developed using NOP is that it can become very easy to trace the requirements into the code. As, ideally, all of the logic must be contained within the rules are linked directly to a requirement and the code where it is implemented, via the rules, premises, attributes and methods involved. In Table XXV it is shown the link between the requirements and the rules used for their implementation. Some requirements may not link to any rule as they were not implemented using rules, an therefore these requirements become harder to trace in the code. It can be observed that the requirements that describe the enemy and generic game behavior are the ones that have a rule, because these are the features that were more appropriate to be implemented with rules.

TABLE III: Initial Functional Requirements

Requirement	Description
FR-1	The player shall be able to choose from the C++ and NOP versions in the menu
FR-2	The level shall be composed of a player versus enemies inside a closed arena
FR-3	All the objects shall be able to move in two dimensions
FR-4	The player shall win when all the enemies are dead
FR-5	The player shall lose when it dies
FR-6	The player shall be able to move with WASD keys
FR-7	The player shall be able to shoot with directional keys
FR-8	The enemy type 1 shall follow the player while its HP is below half its maximum value
FR-9	The enemy type 1 shall flee from the player while its HP is at or below half its maximum value
FR-10	The enemy type 1 shall fire a projectile always when it has a clear line of shot towards the player and it is not on cooldown
FR-11	The enemy type 1 shall fire a missile always when it has a clear line of shot towards the player and it is not on cooldown
FR-12	The enemy type 2 shall flee from the player while its HP is at or below half its maximum value
FR-13	The enemy type 2 shall follow the player while its over 500 units away from the player
FR-14	The enemy type 2 shall follow a straight line its at or under 500 units away from the player
FR-15	The enemy type 2 shall place a bomb when the player has a clear path towards its current position and it is not on cooldown
FR-16	The enemy type 3 shall always follow the player
FR-17	The enemy type 3 shall explode when it collides
FR-18	The enemy type 3 shall cause 2 damage in a 200 units radius when exploding
FR-19	An entity shall not cause damage to itself
FR-20	The projectile ability shall have a cooldown of 2 seconds to the enemies
FR-21	The projectile ability shall have a cooldown of 1 second to the player
FR-22	The missile ability shall have a cooldown of 5 seconds to the enemies
FR-23	The bomb ability shall have a cooldown of 5 seconds to the enemies
FR-24	A projectile shall cause 1 damage to the object it collides
FR-25	A missile shall cause 2 damage to the object it collides
FR-26	A bomb shall detonate 3 seconds after being created
FR-27	A enemy type 3 shall cause 2 damage in a 200 units radius when exploding
FR-28	The player shall move at a speed of 500 units per second
FR-29	The enemy type 1 shall move at a speed of 300 units per second
FR-30	The enemy type 2 shall move at a speed of 250 units per second
FR-31	The enemy type 3 shall move at a speed of 400 units per second
FR-32	The objects shall not be able to move over each other
FR-33	The projectile shall follow a straight line with a speed of 1000 units per second
FR-34	The missile shall follow the player with a speed of 1000 units per second
FR-35	The level shall spawn a enemy of each type in 3 corners of the arena
FR-36	The player shall be able to pause the game
FR-37	The player shall be able to continue the game from the pause menu
FR-38	The enemy type 1 shall have 2 HP
FR-39	The enemy type 2 shall have 5 HP
FR-40	The enemy type 3 shall have 1 HP
FR-41	The player shall have 20 HP

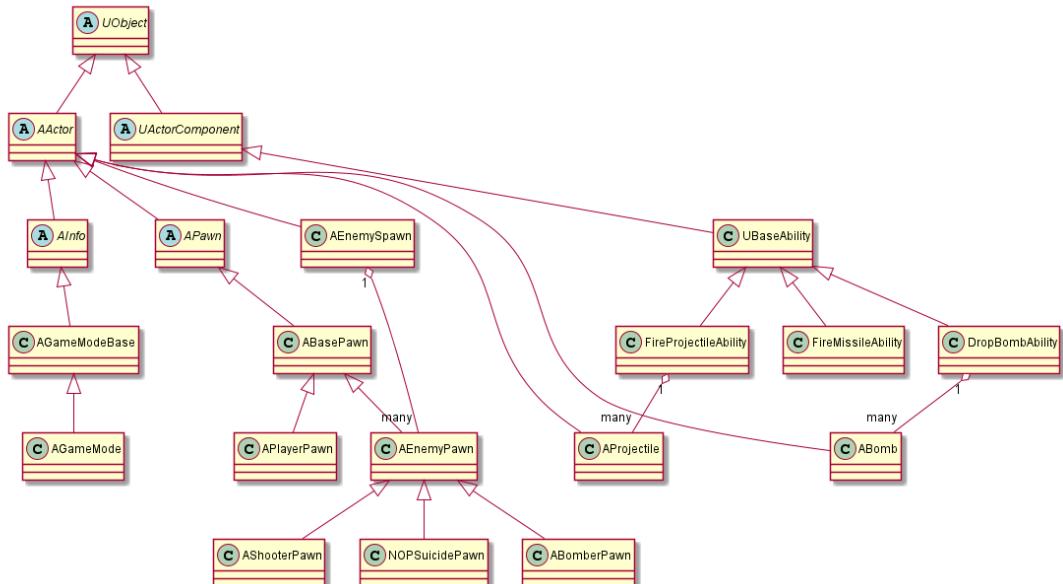


Fig. 6: OO Class Diagram

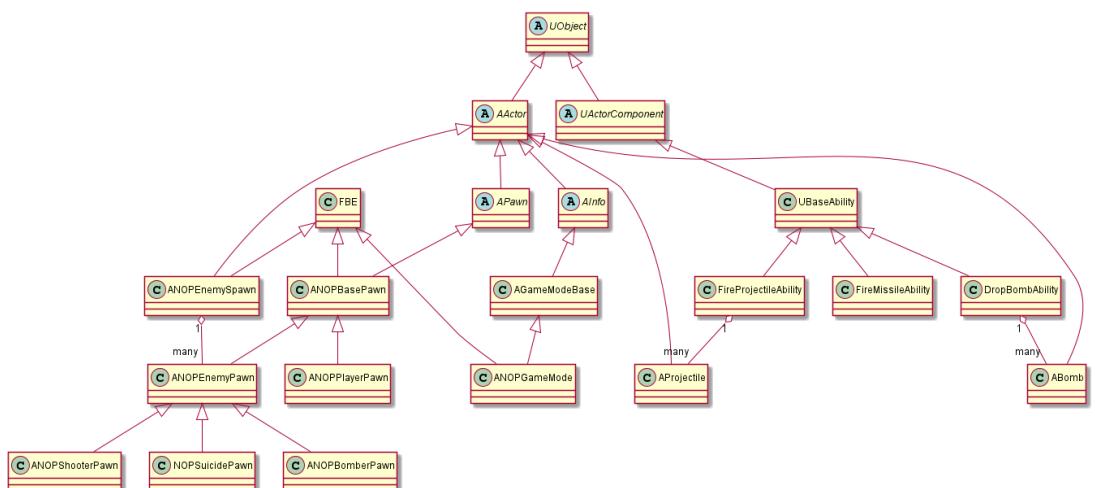


Fig. 7: NOP Class Diagram

TABLE IV: Modified Functional Requirements

Requirement	Description
FR-8	The enemy type 1 shall follow the player while it is over half of its maximum HP and it is the closest type 1 enemy to the player or it is over a distance of 800 units
FR-35	The level shall spawn enemies dynamically
FR-42	The player shall win when reaching 20 points
FR-43	The player shall be awarded 2 points when destroying a enemy type 1
FR-44	The player shall be awarded 3 points when destroying a enemy type 2
FR-45	The player shall be awarded 1 point when destroying a enemy type 3
FR-46	The player shall not be awarded any points when enemies destroy each other
FR-47	The level shall only spawn new enemies on the corners of the arena
FR-48	The level shall spawn a new enemy type 1 every 10 seconds until a maximum of 5 simultaneous enemies type 1 exist
FR-49	The level shall spawn a enemy type 3 for each two enemies of other types destroyed
FR-50	The level shall spawn a enemy type 2 every 5 seconds, while there are less than 5 enemies of any type in the arena
FR-51	The level shall initially spawn a enemy type 1 in each of the corners of the arena
FR-52	The enemy type 1 shall circle around the player with a 800 units radius while its HP is over half of its maximum HP and it is not the closest type 1 enemy to the player

TABLE V: Rule 1

RL-1	
Class	NOPBasePawn
Type	Conjunction
Description	Handle pawn' death
Premises	
Pawn's HP is at or below 0	

TABLE VI: Rule 2

RL-2	
Class	NOPBasePawn
Type	Conjunction
Description	Gives rewards on pawn's death
Premises	
Pawn's HP is equal or less than 0	

TABLE VII: Rule 3

RL-3	
Class	NOPBomberPawn
Type	Conjunction
Description	Use the drop bomb ability
Premises	
Ability cooldown is ready	
Player has free path to current position	

TABLE VIII: Rule 4

RL-4	
Class	NOPBomberPawn
Type	Conjunction
Description	Set strategy to flee
Premises	
Pawn's HP is equal or less than the maximum	

TABLE IX: Rule 5

RL-5	
Class	NOPBomberPawn
Type	Conjunction
Description	Set strategy to follow
Premises	
Pawn's HP is greater than half the maximum	
Distance to player is greater than 500 units	

TABLE X: Rule 6

RL-6	
Class	NOPBomberPawn
Type	Conjunction
Description	Set strategy to roam
Premises	
Pawn's HP is less or equal half the maximum	
Distance to player is less or equal 500 units	

TABLE XI: Rule 7

RL-6	
Class	NOPENEMYSpawner
Type	Conjunction
Description	Spawn an enemy type 1
Premises	
Spawn cooldown is ready	
Number of simultaneous enemies of type 1 is less than 5	

TABLE XII: Rule 8

RL-7	
Class	NOPENEMYSpawner
Type	Conjunction
Description	Spawn an enemy type 2
Premises	
Spawn cooldown is ready	
Number of simultaneous total enemies is less than 5	

TABLE XIII: Rule 9

RL-8	
Class	NOPENEMYSpawner
Type	Conjunction
Description	Spawn an enemy type 3
Premises	
Spawn cooldown is ready	
Number of simultaneous total enemies is less than 5	

TABLE XIV: Rule 10

RL-9	
Class	NOPShooterPawn
Type	Conjunction
Description	Fire a projectile
Premises	
Has clear line of shot to player	
Projectile ability cooldown is ready	

TABLE XV: Rule 11

RL-10	
Class	NOPShooterPawn
Type	Conjunction
Description	Fire a missile
Premises	
Has clear line of shot to player	
Missile ability cooldown is ready	

TABLE XVI: Rule 12

RL-11	
Class	NOPShooterPawn
Type	Conjunction
Description	Set strategy to flee
Premises	
Pawn's HP is less or equal half the maximum	

TABLE XVII: Rule 13

RL-12	
Class	NOPShooterPawn
Type	Disjunction
Description	Set strategy to follow
Premises	
Sub condition 1	Type: Conjunction
Pawn's HP is greater than half the maximum	
Pawn is the closest of its type to the player	
Sub condition 2	Type: Conjunction
Pawn's HP is greater than half the maximum	
Pawn over 800 units away from the player	

TABLE XVIII: Rule 14

RL-13	
Class	NOPShooterPawn
Type	Conjunction
Description	Set strategy to roam
Premises	
Pawn is not the closest of its type to the player	
Pawn's HP is greater than half the maximum	
Pawn under 800 units away from the player	

TABLE XIX: Rule 15

RL-14	
Class	NOPUnrealGameMode
Type	Conjunction
Description	Player wins the game
Premises	
Game has started	
Player has 20 or more points	
Player is alive	

TABLE XX: Rule 16

RL-15	
Class	NOPUnrealGameMode
Type	Conjunction
Description	Player loses the game
Premises	
Game has started	
Player is dead	

TABLE XXI: Rule 17

RL-16	
Class	NOPUnrealGameMode
Type	Conjunction
Description	Set game as started
Premises	
Game has not started	
Game startup is finished	

VI. EVALUATIONS AND COMPARISONS

For an objective perspective in comparing the amount of effort required to develop the software it was chosen to compare both implementations by evaluating the difference in lines of code and time required to develop the software using both approaches, as they can be useful as indicators of how much effort was required during the development. Given that the developer was not experienced with neither the Unreal Engine API nor the NOP C++ Framework it was chosen to only evaluate their time used in the development of the second set of requirements, since most of the time spent during the development of the first version was actually spent learning how to use the tools as well as writing basic functions used for both versions and therefore would not reflect the real difference in the amount of effort required.

For the time evaluation each requirement, either new or modified, is linked to the amount of time spent in the Object-Oriented and Notification Oriented approaches, detailed in Table XXII. For this purpose it is only considered the time effectively spent writing code.

TABLE XXII: Time spent for development

Requirement	State	NOP	OOP
FR-35	Modified	0 min	0 min
FR-42	New	5 min	20 min
FR-43	New	15 min	5 min
FR-44	New	5 min	5 min
FR-45	New	5 min	5 min
FR-46	New	0 min	0 min
FR-47	New	5 min	5 min
FR-48	New	10 min	30 min
FR-49	New	10 min	30 min
FR-50	New	10 min	30 min
FR-51	New	10 min	10 min
FR-8	Modified	30 min	90 min
FR-9	Modified	30 min	30 min
Total		135 min	220 min

In the same manner the total lines of code are compared using the *cloc* tool [12]. This tool can count the total lines of code while eliminating comments and blank lines. Only the source code used for each implementation is considered for this purpose, and code that is common for both implementations is accounted for in each one of them. The results are shown in Table XXIII. The number of files is the same for both implementations.

TABLE XXIII: Lines of code written

Requirement Version	OOP	NOP
Initial	1272 lines	1407 lines
Final	1583 lines	1776 lines

Another analysis was to count the number of words in the same manner, this time using the command *wc* from *Linux*.

The results are shown in Table XXIV.

TABLE XXIV: Words written

Requirement Version	OOP	NOP
Initial	54800 words	59315 words
Final	67682 words	75329 words

VII. CONCLUSION AND FUTURE WORKS

The results presented clearly show that working with NOP can bring great improvements to the process of game development, and other types of software as well, whether by reducing the development time due to a more comprehensive code structure, specially when writing code for behavior that can be translated well into rules, which is the case in many aspects of game development, since a lot of the effort is dedicated to writing the game logic and non-playable characters behavior.

As shown in the results from Table XXII, the time spent modifying the NOP code was 38.62% when compared with the OOP only code. And another major improvement is that all the NOP rules can be directly traced into specific requirements, meaning that any changes in the project can be more easily translated into code, as it reduces the risk of having the logic spread among to many places in the code, which makes it harder to do any changes without breaking other behaviors or leaving deprecated code in some places, which may easily lead to bugs and inconsistency in the software behavior.

There are however some clear drawbacks, that come from working with a novel technology. The framework itself can be very verbose, as it is shown in the increased amount of code written necessary for the same logic, as show in Table XXIII and Table XXIV, as the NOP code has 12.16% more lines and 11.13% more words, however this is small problem, since this comes from having to copy the initialization and declaration from the NOP structure, and doing so still took less time than the OOP approach.

Further development of the NOP technology and the C++ framework itself can help yield even better results, reducing the total time and amount of code required for software development. The development of more projects with the NOP framework can also be useful to ground the statement that the use of the technology can help reduce the effort required in software development.

REFERENCES

- [1] J. Gold, *Object-oriented game development*. Pearson Education Limited, 2004.
- [2] D. Takahashi, “Superdata: Games hit 120.1 billion in 2019, with Fortnite topping 1.8 billion,” Dec 2019. [Online]. Available: <https://venturebeat.com/2020/01/02/superdata-games-hit-120-1-billion-in-2019-with-fortnite-topping-1-8-billion/>
- [3] J. Gregory, *Game engine architecture*. CRC Press, 2019.
- [4] A. F. Ronszka, “Contribuição para a concepção de aplicações no paradigma orientado a notificações (pon) sob o viés de padrões,” Master’s thesis, UTFPR, 2012.
- [5] L. D. Wen, “Steam-engines,” <https://github.com/linddingwen/Steam-Engines>, 2018.
- [6] E. Games. (2020) Unreal engine 4 documentation. [Online]. Available: <https://docs.unrealengine.com/en-US/index.html>

TABLE XXV: Correlation Matrix

Requirement	RL-1	RL-2	RL-3	RL-4	RL-5	RL-6	RL-7	RL-8	RL-9	RL-10	RL-11	RL-12	RL-13	RL-14	RL-15	RL-16	RL-17
FR-1																	
FR-2																	
FR-3																	
FR-4																	
FR-5	x															x	
FR-6																	
FR-7																	
FR-8													x				
FR-9											x						
FR-10									x								
FR-11										x							
FR-12		x															
FR-13			x														
FR-14				x													
FR-15		x															
FR-16																	
FR-17																	
FR-18																	
FR-19																	
FR-20								x									
FR-21																	
FR-22									x								
FR-23		x															
FR-24																	
FR-25																	
FR-26																	
FR-27																	
FR-28																	
FR-29																	
FR-30																	
FR-31																	
FR-32																	
FR-33																	
FR-34																	
FR-35																	
FR-36																	
FR-37																	
FR-38																	
FR-39																	
FR-40																	
FR-41																	
FR-42												x		x			
FR-43	x																
FR-44	x																
FR-45	x																
FR-46	x																
FR-47																	
FR-48						x											
FR-49								x									
FR-50								x									
FR-51														x			
FR-52												x					

APÊNDICE D

Conjunto de testes do *Framework PON C++ 4.0*

```

#include <atomic>
#include <queue>
#include <random>
#include <thread>

#include "gtest/gtest.h"
#include "libnop/framework.h"

TEST(Attribute, Int)
{
    const int val{123456};
    NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(val);
    EXPECT_EQ(at1->GetValue(), val);
}

TEST(Attribute, Bool)
{
    const bool val{true};
    NOP::SharedAttribute<bool> at1 = NOP::BuildAttribute(val);
    EXPECT_EQ(at1->GetValue(), val);
}

TEST(Attribute, Class)
{
    class Rectangle
    {
        int width, height;

    public:
        Rectangle(int w, int h) : width{w}, height{h} {};
        inline bool operator==(Rectangle i) const
        {
            return (width == i.width) && (height == i.height);
        }
        inline bool operator!=(Rectangle i) const
        {
            return (width != i.width) || (height != i.height);
        }
    };

    const Rectangle val1{1, 2};
    NOP::SharedAttribute<Rectangle> at1 = NOP::BuildAttribute(val1);
    EXPECT_EQ(at1->GetValue(), val1);

    const Rectangle val2{3, 4};
    NOP::SharedAttribute<Rectangle> at2 = NOP::BuildAttribute(val2);
    EXPECT_NE(at1->GetValue(), at2->GetValue());

    at2->SetValue(val1);
    EXPECT_EQ(at1->GetValue(), at2->GetValue());
}

TEST(Attribute, String)
{
    const std::string val{"test"};
    NOP::SharedAttribute<std::string> at1 = NOP::BuildAttribute(val);
    EXPECT_EQ(at1->GetValue(), val);
}

TEST(Premise, Simple)
{
    NOP::SharedAttribute<int> at1 = NOP::BuildAttribute<int>(-1);
    NOP::SharedAttribute<int> at2 = NOP::BuildAttribute<int>(-2);

    NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());

    // On initialization should not be approved
    EXPECT_FALSE(pr1->Approved());
    at1->SetValue(1);
    EXPECT_FALSE(pr1->Approved());
}

```

```

        at2->SetValue(1);
        EXPECT_TRUE(pr1->Approved());
    }

TEST(Premise, Custom)
{
    NOP::SharedAttribute<int> at1 = NOP::BuildAttribute<int>(1);
    NOP::SharedAttribute<int> at2 = NOP::BuildAttribute<int>(1);
    NOP::SharedPremise prDouble = NOP::BuildPremise(
        at1, at2, [] (auto a1, auto a2) { return a1 == 2 * a2; });

    // On initialization should not be approved
    EXPECT_FALSE(prDouble->Approved());
    at1->SetValue(2 * at2->GetValue());
    EXPECT_TRUE(prDouble->Approved());
    at1->SetValue(0);
    EXPECT_FALSE(prDouble->Approved());
}

TEST(Condition, Conjunction)
{
    NOP::SharedAttribute<int> at1 = NOP::BuildAttribute<int>(-1);
    NOP::SharedAttribute<int> at2 = NOP::BuildAttribute<int>(-2);
    NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());

    NOP::SharedAttribute<int> at3 = NOP::BuildAttribute<int>(-3);
    NOP::SharedAttribute<int> at4 = NOP::BuildAttribute<int>(-4);
    NOP::SharedPremise pr2 = NOP::BuildPremise(at3, at4, NOP::Equal());

    NOP::SharedCondition cn1 =
        NOP::BuildCondition(CONDITION(*pr1 && *pr2), pr1, pr2);

    // On initialization should not be approved
    EXPECT_FALSE(cn1->Approved());
    at1->SetValue(1);
    at2->SetValue(1);
    EXPECT_FALSE(cn1->Approved());
    at3->SetValue(1);
    at4->SetValue(1);
    EXPECT_TRUE(cn1->Approved());
}

TEST(Condition, Disjunction)
{
    NOP::SharedAttribute<int> at1 = NOP::BuildAttribute<int>(-1);
    NOP::SharedAttribute<int> at2 = NOP::BuildAttribute<int>(-2);
    NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());

    NOP::SharedAttribute<int> at3 = NOP::BuildAttribute<int>(-3);
    NOP::SharedAttribute<int> at4 = NOP::BuildAttribute<int>(-4);
    NOP::SharedPremise pr2 = NOP::BuildPremise(at3, at4, NOP::Equal());

    NOP::SharedCondition cn1 =
        NOP::BuildCondition(CONDITION(*pr1 || *pr2), pr1, pr2);

    // On initialization should not be approved
    EXPECT_FALSE(cn1->Approved());
    at1->SetValue(1);
    at2->SetValue(1);
    EXPECT_TRUE(cn1->Approved());
    at3->SetValue(1);
    at4->SetValue(1);
    EXPECT_TRUE(cn1->Approved());
    at1->SetValue(-1);
    EXPECT_TRUE(cn1->Approved());
    at3->SetValue(-1);
    EXPECT_FALSE(cn1->Approved());
}

TEST(Condition, SubCondition)
{
    NOP::SharedAttribute<int> at1 = NOP::BuildAttribute<int>(-1);
    NOP::SharedAttribute<int> at2 = NOP::BuildAttribute<int>(-2);
    NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());

    NOP::SharedAttribute<int> at3 = NOP::BuildAttribute<int>(-3);
    NOP::SharedAttribute<int> at4 = NOP::BuildAttribute<int>(-4);
    NOP::SharedPremise pr2 = NOP::BuildPremise(at3, at4, NOP::Equal());

    NOP::SharedCondition cn1 = NOP::BuildCondition(CONDITION(*pr1), pr1);
    NOP::SharedCondition cn2 =

```

```

    NOP::BuildCondition(CONDITION(*pr2 && *cn1), pr2, cn1);

    // On initialization should not be approved
    EXPECT_FALSE(cn2->Approved());
    at1->SetValue(1);
    at2->SetValue(1);
    EXPECT_TRUE(cn1->Approved());
    EXPECT_FALSE(cn2->Approved());
    at3->SetValue(1);
    at4->SetValue(1);
    EXPECT_TRUE(cn2->Approved());
}

TEST(Condition, MasterRule)
{
    NOP::SharedAttribute<int> at1 = NOP::BuildAttribute<int>(-1);
    NOP::SharedAttribute<int> at2 = NOP::BuildAttribute<int>(-2);
    NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());

    NOP::SharedAttribute<int> at3 = NOP::BuildAttribute<int>(-3);
    NOP::SharedAttribute<int> at4 = NOP::BuildAttribute<int>(-4);
    NOP::SharedPremise pr2 = NOP::BuildPremise(at3, at4, NOP::Equal());

    NOP::SharedCondition cn1 = NOP::BuildCondition(CONDITION(*pr1), pr1);
    NOP::SharedRule r11 =
        NOP::BuildRule(cn1, NOP::BuildAction(NOP::BuildInstigation()));

    NOP::SharedCondition cn2 =
        NOP::BuildCondition(CONDITION(*pr2 && *r11), pr2, r11);

    EXPECT_FALSE(cn2->Approved());
    at1->SetValue(1);
    at2->SetValue(1);
    NOP::Scheduler::Instance().FinishAll();
    EXPECT_TRUE(r11->Approved());
    EXPECT_FALSE(cn2->Approved());
    at3->SetValue(1);
    at4->SetValue(1);
    NOP::Scheduler::Instance().FinishAll();
    EXPECT_TRUE(cn2->Approved());
}

TEST(Complete, Basic)
{
    NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(-1);
    NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(-2);
    NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());
    NOP::SharedCondition cn1 = NOP::BuildCondition(CONDITION(*pr1), pr1);

    std::atomic<int> executionCounter{0};
    NOP::Method mt = [&] { executionCounter++; };
    NOP::SharedInstigation in1 = NOP::BuildInstigation(mt);
    NOP::SharedAction ac1 = NOP::BuildAction(in1);

    NOP::SharedRule r11 = NOP::BuildRule(cn1, ac1);
    NOP::Scheduler::Instance().FinishAll();

    // Nothing should be approved upon initialization
    EXPECT_FALSE(pr1->Approved());
    EXPECT_FALSE(cn1->Approved());
    EXPECT_FALSE(r11->Approved());

    at1->SetValue(1);
    NOP::Scheduler::Instance().FinishAll();

    // Nothing should be approved yet
    EXPECT_FALSE(pr1->Approved());
    EXPECT_FALSE(cn1->Approved());
    EXPECT_FALSE(r11->Approved());

    at2->SetValue(1);
    NOP::Scheduler::Instance().FinishAll();

    // Everything should be approved
    EXPECT_TRUE(pr1->Approved());
    EXPECT_TRUE(cn1->Approved());
    EXPECT_TRUE(r11->Approved());
    EXPECT_EQ(executionCounter, 1);

    at2->SetValue(10);
    NOP::Scheduler::Instance().FinishAll();
}

```

```

EXPECT_FALSE(pr1->Approved());
EXPECT_FALSE(cn1->Approved());
EXPECT_FALSE(r11->Approved());

at1->SetValue(10);
NOP::Scheduler::Instance().FinishAll();

// Everything should be approved
EXPECT_TRUE(pr1->Approved());
EXPECT_TRUE(cn1->Approved());
EXPECT_TRUE(r11->Approved());
EXPECT_EQ(executionCounter, 2);
}

#ifdef LIBNOP_LOG_ENABLE
TEST(Complete, Logger)
{
    NOP::Logger::Instance().SetLogFileName("test.log");
    NOP::Logger::Instance().SetLogFileName("test2.log");
    NOP::Logger::Instance().SetLogFileName("test.log");

    struct Test : NOP::FBE
    {
        explicit Test(const std::string_view name) : FBE(name) {}
        NOP::SharedAttribute<int> at1 =
            NOP::BuildAttributeNamed("at1", this, -1);
        // Build without parent
        NOP::SharedAttribute<int> at2 =
            NOP::BuildAttributeNamed("at2", nullptr, -2);
        NOP::SharedPremise pr1 =
            NOP::BuildPremiseNamed("pr1", this, at1, at2, NOP::Equal());
        NOP::SharedCondition cn1 =
            NOP::BuildConditionNamed("cn1", this, CONDITION(*pr1), pr1);

        std::atomic<int> executionCounter{0};
        NOP::Method mt = [&] { executionCounter++; };
        NOP::SharedInstigation in1 =
            NOP::BuildInstigationNamed("in1", this, mt);
        NOP::SharedAction acl = NOP::BuildActionNamed("acl", this, in1);

        NOP::SharedRule r11 = NOP::BuildRuleNamed("r11", this, cn1, acl);
    };
}

Test test{"TestFBE"};
NOP::Scheduler::Instance().FinishAll();

// Nothing should be approved upon initialization
EXPECT_FALSE(test.pr1->Approved());
EXPECT_FALSE(test.cn1->Approved());
EXPECT_FALSE(test.r11->Approved());

test.at1->SetValue(1);
NOP::Scheduler::Instance().FinishAll();

// Nothing should be approved yet
EXPECT_FALSE(test.pr1->Approved());
EXPECT_FALSE(test.cn1->Approved());
EXPECT_FALSE(test.r11->Approved());

test.at2->SetValue(1);
NOP::Scheduler::Instance().FinishAll();

// Everything should be approved
EXPECT_TRUE(test.pr1->Approved());
EXPECT_TRUE(test.cn1->Approved());
EXPECT_TRUE(test.r11->Approved());
EXPECT_EQ(test.executionCounter, 1);

test.at2->SetValue(10);
NOP::Scheduler::Instance().FinishAll();

EXPECT_FALSE(test.pr1->Approved());
EXPECT_FALSE(test.cn1->Approved());
EXPECT_FALSE(test.r11->Approved());

test.at1->SetValue(10);
NOP::Scheduler::Instance().FinishAll();

// Everything should be approved

```

```

    EXPECT_TRUE(test.prl->Approved());
    EXPECT_TRUE(test.cn1->Approved());
    EXPECT_TRUE(test.rll->Approved());
    EXPECT_EQ(test.executionCounter, 2);
}
#endif

TEST(Complete, MultipleCycles)
{
    NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(-1);
    NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(1);
    NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());
    NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Single>(pr1);

    std::atomic<int> executionCounter{0};
    NOP::Method mt = [&] { executionCounter++; };
    NOP::SharedInstigation in1 = NOP::BuildInstigation(mt);
    NOP::SharedAction acl = NOP::BuildAction(in1);

    NOP::SharedRule rll = NOP::BuildRule(cn1, acl);

    for (int i = 1; i <= INT16_MAX; i++)
    {
        at1->SetValue(1);
        NOP::Scheduler::Instance().FinishAll();

        at1->SetValue(-1);
        EXPECT_EQ(executionCounter, i);
    }

    EXPECT_EQ(executionCounter, INT16_MAX);
}

TEST(Counter, Default)
{
    NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(1);
    NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(-1);
    NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());
    NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Single>(pr1);

    std::atomic<int> executionCounter{0};
    NOP::Method mt = [&] { executionCounter++; };
    NOP::SharedInstigation in1 = NOP::BuildInstigation(mt);
    NOP::SharedAction acl = NOP::BuildAction(in1);

    NOP::SharedRule rll = NOP::BuildRule(cn1, acl);
    NOP::Scheduler::Instance().FinishAll();

    constexpr int executions{INT8_MAX};
    for (int i = 0; i < executions; i++)
    {
        at2->SetValue(-1);
        at2->SetValue(1);
        NOP::Scheduler::Instance().FinishAll();
    }
    EXPECT_EQ(executionCounter, executions);
}

TEST(Complete, InitialStatesDisapproved)
{
    NOP::Scheduler& scheduler = NOP::Scheduler::Instance();

    NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(1);
    NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(2);
    NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());
    NOP::SharedCondition cn1 = NOP::BuildCondition(CONDITION(*pr1), pr1);

    std::atomic<int> executionCounter{0};
    NOP::Method mt = [&] { executionCounter++; };
    NOP::SharedInstigation in1 = NOP::BuildInstigation(mt);
    NOP::SharedAction acl = NOP::BuildAction(in1);

    NOP::SharedRule rll = NOP::BuildRule(cn1, acl);
    scheduler.FinishAll();

    EXPECT_FALSE(pr1->Approved());
    EXPECT_FALSE(cn1->Approved());
    EXPECT_FALSE(rll->Approved());
    NOP::Scheduler::Instance().FinishAll();
    EXPECT_EQ(executionCounter, 0);
}
}

```

```

TEST(Complete, InitialStatesApproved)
{
    NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(1);
    NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(1);
    NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());
    NOP::SharedCondition cn1 = NOP::BuildCondition(CONDITION(*pr1), pr1);

    std::atomic<int> executionCounter{0};
    NOP::Method mt = [&] { executionCounter++; };
    NOP::SharedInstigation in1 = NOP::BuildInstigation(mt);
    NOP::SharedAction ac1 = NOP::BuildAction(in1);

    NOP::SharedRule r11 = NOP::BuildRule(cn1, ac1);

    NOP::Scheduler::Instance().FinishAll();

    EXPECT_TRUE(pr1->Approved());
    EXPECT_TRUE(cn1->Approved());
    EXPECT_TRUE(r11->Approved());

    EXPECT_EQ(executionCounter, 1);
}

TEST(Complete, SharedEntities1)
{
    NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(1);
    NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(1);
    NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());
    NOP::SharedCondition cn1 = NOP::BuildCondition(CONDITION(*pr1), pr1);

    std::atomic<int> executionCounter{0};
    NOP::Method mt = [&] { executionCounter++; };
    // clone methods
    NOP::SharedInstigation in1 = NOP::BuildInstigation(mt);
    NOP::SharedAction ac1 = NOP::BuildAction(in1);
    NOP::SharedRule r11 = NOP::BuildRule(cn1, ac1);

    NOP::SharedInstigation in2 = NOP::BuildInstigation(mt);
    NOP::SharedAction ac2 = NOP::BuildAction(in2);
    NOP::SharedRule r12 = NOP::BuildRule(cn1, ac2);

    NOP::Scheduler::Instance().FinishAll();

    EXPECT_EQ(executionCounter, 2);
}

TEST(Complete, SharedEntities2)
{
    NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(1);
    NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(1);
    NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());
    NOP::SharedCondition cn1 = NOP::BuildCondition(CONDITION(*pr1), pr1);
    NOP::SharedCondition cn2 = NOP::BuildCondition(CONDITION(*pr1), pr1);
    std::atomic<int> executionCounter{0};
    NOP::Method mt = [&] { executionCounter++; };
    // clone methods
    NOP::SharedInstigation in1 = NOP::BuildInstigation(mt);
    NOP::SharedAction ac1 = NOP::BuildAction(in1);
    NOP::SharedRule r11 = NOP::BuildRule(cn1, ac1);
    NOP::SharedRule r12 = NOP::BuildRule(cn2, ac1);

    NOP::Scheduler::Instance().FinishAll();

    EXPECT_EQ(executionCounter, 2);
}

TEST(Complete, Renotification)
{
    NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(-1);
    NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(-2);
    NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());
    NOP::SharedCondition cn1 = NOP::BuildCondition(CONDITION(*pr1), pr1);
    std::atomic<int> executionCounter{0};
    NOP::SharedInstigation in1 =
        NOP::BuildInstigation([&] { executionCounter++; });
    NOP::SharedAction ac1 = NOP::BuildAction(in1);
    NOP::SharedRule r11 = NOP::BuildRule(cn1, ac1);

    at1->SetValue(1, false);
    at2->SetValue(1, false);
}

```

```

NOP::Scheduler::Instance().FinishAll();
EXPECT_EQ(executionCounter, 1);

at2->SetValue(1, true);
NOP::Scheduler::Instance().FinishAll();
EXPECT_EQ(executionCounter, 2);

at2->SetValue(1, true);
NOP::Scheduler::Instance().FinishAll();
EXPECT_EQ(executionCounter, 3);
}

TEST(Complete, RecursiveRule)
{
    NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(0);
    NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(0);
    NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, std::less<>());
    NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Single>(pr1);
    int executionCounter{0};
    auto test = [&]()
    {
        executionCounter++;
        at1->SetValue<NOP::ReNotify>(at1->GetValue() + 1);
    };
    NOP::SharedInstigation in1 = NOP::BuildInstigation(test);
    NOP::SharedAction acl = NOP::BuildAction(in1);
    NOP::SharedRule r11 = NOP::BuildRule(cn1, acl);

    at2->SetValue(10);
    NOP::Scheduler::Instance().FinishAll();

    EXPECT_EQ(executionCounter, 10);
}
TEST(Condition, Composition)
{
    NOP::SharedAttribute<int> at1 = NOP::BuildAttribute<int>(-1);
    NOP::SharedAttribute<int> at2 = NOP::BuildAttribute<int>(-2);
    NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());

    NOP::SharedAttribute<int> at3 = NOP::BuildAttribute<int>(-3);
    NOP::SharedAttribute<int> at4 = NOP::BuildAttribute<int>(-4);
    NOP::SharedPremise pr2 = NOP::BuildPremise(at3, at4, NOP::Equal());

    NOP::SharedAttribute<int> at5 = NOP::BuildAttribute<int>(0);
    NOP::SharedAttribute<int> at6 = NOP::BuildAttribute<int>(0);
    NOP::SharedPremise pr3 = NOP::BuildPremise(at5, at6, NOP::Equal());

    NOP::SharedCondition cn1 =
        NOP::BuildCondition(CONDITION((*pr1 || *pr2) && !*pr3), pr1, pr2, pr3);

    at1->SetValue(1);
    at2->SetValue(1);
    NOP::Scheduler::Instance().FinishAll();
    EXPECT_TRUE(pr1->Approved());
    EXPECT_FALSE(pr2->Approved());
    EXPECT_TRUE(pr3->Approved());
    EXPECT_FALSE(cn1->Approved());

    at5->SetValue(1);
    NOP::Scheduler::Instance().FinishAll();
    EXPECT_FALSE(pr3->Approved());
    EXPECT_TRUE(cn1->Approved());

    at2->SetValue(-1);
    NOP::Scheduler::Instance().FinishAll();
    EXPECT_FALSE(pr1->Approved());
    EXPECT_FALSE(cn1->Approved());

    at3->SetValue(1);
    at4->SetValue(1);
    NOP::Scheduler::Instance().FinishAll();
    EXPECT_TRUE(pr2->Approved());
    EXPECT_TRUE(cn1->Approved());
}
TEST(Flag, Exclusive)
{
    NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(0);
    NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(0);

    NOP::SharedAttribute<bool> atExclusive = NOP::BuildAttribute(false);
}

```

```

NOP::SharedPremise pr1 = NOP::BuildPremise(at1, 1, NOP::Equal());
NOP::SharedPremise pr2 = NOP::BuildPremise(at2, 2, NOP::Equal());

NOP::SharedPremise prExclusive =
    NOP::BuildPremise(atExclusive, true, NOP::Equal());

NOP::SharedCondition cn1 =
    NOP::BuildCondition<NOP::Conjunction>(pr1, prExclusive);
NOP::SharedCondition cn2 =
    NOP::BuildCondition<NOP::Conjunction>(pr2, prExclusive);

std::atomic<int> executionCounter1{0};
auto test1 = [&]()
{
    std::this_thread::sleep_for(std::chrono::milliseconds(1));
    EXPECT_TRUE(atExclusive->GetValue());
    atExclusive->SetValue<NOP::Exclusive>(false);
    std::this_thread::sleep_for(std::chrono::milliseconds(20));
    ++executionCounter1;
    at1->SetValue(0);
    atExclusive->SetValue<NOP::Exclusive>(true);
};

NOP::SharedInstigation in1 = NOP::BuildInstigation(test1);
NOP::SharedAction ac1 = NOP::BuildAction(in1);
NOP::SharedRule rl1 = NOP::BuildRule(cn1, ac1);

std::atomic<int> executionCounter2{0};
auto test2 = [&]()
{
    std::this_thread::sleep_for(std::chrono::milliseconds(5));
    EXPECT_TRUE(atExclusive->GetValue());
    atExclusive->SetValue<NOP::Exclusive>(false);
    std::this_thread::sleep_for(std::chrono::milliseconds(10));
    ++executionCounter2;
    at2->SetValue(0);
    atExclusive->SetValue<NOP::Exclusive>(true);
};

NOP::SharedInstigation in2 = NOP::BuildInstigation(test2);
NOP::SharedAction ac2 = NOP::BuildAction(in2);
NOP::SharedRule rl2 = NOP::BuildRule(cn2, ac2);

atExclusive->SetValue<NOP::Exclusive>(true);
{
    std::thread t1{[&]() { at1->SetValue<NOP::Exclusive>(1); }};
    std::thread t2{[&]() { at2->SetValue<NOP::Exclusive>(2); }};
    std::this_thread::sleep_for(std::chrono::milliseconds(10));
    t1.join();
    t2.join();
}

NOP::Scheduler::Instance().FinishAll();

EXPECT_EQ(executionCounter1, 1);
EXPECT_EQ(executionCounter2, 1);
}

TEST(Example, Alarm)
{
    struct Alarm
    {
        int alarmCount{0};

        NOP::SharedAttribute<bool> atState{NOP::BuildAttribute<bool>(false)};
        NOP::SharedPremise prAlarm{
            NOP::BuildPremise(atState, true, NOP::Equal())};
        NOP::SharedCondition cnAlarm{
            NOP::BuildCondition(CONDITION(*prAlarm), prAlarm)};
        NOP::Method mtNotify { METHOD(alarmCount++); };
        NOP::SharedInstigation inAlarm{NOP::BuildInstigation(mtNotify)};
        NOP::SharedAction acAlarm{NOP::BuildAction(inAlarm)};
        NOP::SharedRule rlAlarm{NOP::BuildRule(cnAlarm, acAlarm)};
    };

    Alarm all1;
    Alarm al2;

    for (int i = 0; i < 10; i++)
    {
        all1.atState->SetValue(!all1.atState->GetValue());

```

```

        al2.atState->SetValue<NOP::ReNotify>(true);
        NOP::Scheduler::Instance().FinishAll();
    }
    EXPECT_EQ(al1.alarmCount, 5);
    EXPECT_EQ(al2.alarmCount, 10);
}

TEST(Example, AlarmSimplified)
{
    struct Alarm
    {
        int alarmCount{0};

        NOP::SharedAttribute<bool> atState{NOP::BuildAttribute<bool>(false)};
        NOP::SharedPremise prAlarm{
            NOP::BuildPremise(atState, true, NOP::Equal())};
        NOP::SharedRule rlAlarm
        {
            NOP::BuildRule(
                NOP::BuildCondition(CONDITION(*prAlarm), prAlarm),
                NOP::BuildAction(NOP::BuildInstigation(METHOD(alarmCount++))));
        };
    };

    Alarm al1;
    Alarm al2;

    for (int i = 0; i < 10; i++)
    {
        al1.atState->SetValue(!al1.atState->GetValue());
        al2.atState->SetValue<NOP::ReNotify>(true);
        NOP::Scheduler::Instance().FinishAll();
    }
    EXPECT_EQ(al1.alarmCount, 5);
    EXPECT_EQ(al2.alarmCount, 10);
}

#ifdef LIBNOP_SCHEDULER_ENABLE
TEST(Complete, Competing)
{
    NOP::SharedAttribute<bool> atShouldExecute = NOP::BuildAttribute(true);
    NOP::SharedAttribute<bool> atResourceAvailable = NOP::BuildAttribute(false);
    NOP::SharedPremise prShouldExecute =
        NOP::BuildPremise(atShouldExecute, true, NOP::Equal());
    NOP::SharedPremise prResourceAvailable =
        NOP::BuildPremise(atResourceAvailable, true, NOP::Equal());

    NOP::SharedCondition cn1 =
        NOP::BuildCondition(CONDITION(*prShouldExecute && *prResourceAvailable),
                           prShouldExecute, prResourceAvailable);

    std::atomic<int> executionCounter1{0};
    NOP::Method mt1 = [&]()
    {
        atResourceAvailable->SetValue(false);
        ++executionCounter1;
    };
    NOP::Method mt2 = [&]()
    {
        atResourceAvailable->SetValue(false);
        ++executionCounter1;
    };
    NOP::SharedInstigation in1 = NOP::BuildInstigation(mt1);
    NOP::SharedAction ac1 = NOP::BuildAction(in1);
    NOP::SharedInstigation in2 = NOP::BuildInstigation(mt2);
    NOP::SharedAction ac2 = NOP::BuildAction(in2);

    NOP::SharedRule rl1 = NOP::BuildRule(cn1, ac1);
    NOP::SharedRule rl2 = NOP::BuildRule(cn1, ac2);

    atResourceAvailable->SetValue(true);
    NOP::Scheduler::Instance().FinishAll();

    EXPECT_EQ(executionCounter1, 1);

    // Use function just to apply code coverage
    mt1();
    mt2();
}
#endif
}

```

```

TEST(Condition, EnumConjunction)
{
    NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(-1);
    NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(-1);
    NOP::SharedPremise pr1 = NOP::BuildPremise(at1, 1, NOP::Equal());
    NOP::SharedPremise pr2 = NOP::BuildPremise(at2, 2, NOP::Equal());

    NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Conjunction>(pr1, pr2);

    std::atomic<int> executionCounter{0};
    NOP::Method mt = [&] { ++executionCounter; };
    NOP::SharedInstigation in1 = NOP::BuildInstigation(mt);
    NOP::SharedAction acl = NOP::BuildAction(in1);

    NOP::SharedRule r11 = NOP::BuildRule(cn1, acl);
    NOP::Scheduler::Instance().FinishAll();
    EXPECT_FALSE(r11->Approved());
    EXPECT_EQ(executionCounter, 0);

    at1->SetValue(1);
    NOP::Scheduler::Instance().FinishAll();
    EXPECT_FALSE(r11->Approved());
    EXPECT_EQ(executionCounter, 0);

    at2->SetValue(2);
    NOP::Scheduler::Instance().FinishAll();
    EXPECT_TRUE(r11->Approved());
    EXPECT_EQ(executionCounter, 1);
}

TEST(Condition, EnumDisjunction)
{
    NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(-1);
    NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(-1);
    NOP::SharedPremise pr1 = NOP::BuildPremise(at1, 1, NOP::Equal());
    NOP::SharedPremise pr2 = NOP::BuildPremise(at2, 2, NOP::Equal());

    NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Disjunction>(pr1, pr2);

    std::atomic<int> executionCounter{0};
    NOP::Method mt = [&] { ++executionCounter; };
    NOP::SharedInstigation in1 = NOP::BuildInstigation(mt);
    NOP::SharedAction acl = NOP::BuildAction(in1);

    NOP::SharedRule r11 = NOP::BuildRule(cn1, acl);
    NOP::Scheduler::Instance().FinishAll();
    EXPECT_FALSE(r11->Approved());
    EXPECT_EQ(executionCounter, 0);

    at1->SetValue(1);
    NOP::Scheduler::Instance().FinishAll();
    EXPECT_TRUE(r11->Approved());
    EXPECT_EQ(executionCounter, 1);

    at2->SetValue(2);
    NOP::Scheduler::Instance().FinishAll();
    EXPECT_TRUE(r11->Approved());
    EXPECT_EQ(executionCounter, 1);
}

TEST(Flag, NoNotify)
{
    NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(-1);
    NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(-2);
    NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());
    NOP::SharedCondition cn1 = NOP::BuildCondition(CONDITION(*pr1), pr1);

    std::atomic<int> executionCounter{0};
    NOP::Method mt = [&] { executionCounter++; };
    NOP::SharedInstigation in1 = NOP::BuildInstigation(mt);
    NOP::SharedAction acl = NOP::BuildAction(in1);

    NOP::SharedRule r11 = NOP::BuildRule(cn1, acl);
    at1->SetValue<NOP::Default>(1);
    at2->SetValue<NOP::NoNotify>(1);
    NOP::Scheduler::Instance().FinishAll();

    EXPECT_FALSE(r11->Approved());
    EXPECT_EQ(executionCounter, 0);
}

```

```

TEST(Flag, ParallelNoNotify)
{
    NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(-1);
    NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(-2);
    NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());
    NOP::SharedCondition cn1 = NOP::BuildCondition(CONDITION(*pr1), pr1);

    std::atomic<int> executionCounter{0};
    NOP::Method mt = [&] { executionCounter++; };
    NOP::SharedInstigation in1 = NOP::BuildInstigation(mt);
    NOP::SharedAction acl = NOP::BuildAction(in1);

    NOP::SharedRule r11 = NOP::BuildRule(cn1, acl);

    at1->SetValue<NOP::Parallel>(1);
    at2->SetValue<NOP::Parallel | NOP::NoNotify>(1);
    NOP::Scheduler::Instance().FinishAll();

    EXPECT_FALSE(r11->Approved());
    EXPECT_EQ(executionCounter, 0);
}

TEST(Flag, ReNotify)
{
    NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(-1);
    NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(-2);
    NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());
    NOP::SharedCondition cn1 = NOP::BuildCondition(CONDITION(*pr1), pr1);

    std::atomic<int> executionCounter{0};
    NOP::Method mt = [&] { executionCounter++; };
    NOP::SharedInstigation in1 = NOP::BuildInstigation(mt);
    NOP::SharedAction acl = NOP::BuildAction(in1);

    NOP::SharedRule r11 = NOP::BuildRule(cn1, acl);

    at1->SetValue<NOP::Default>(1);
    at2->SetValue<NOP::Default>(1);

    NOP::Scheduler::Instance().FinishAll();

    EXPECT_TRUE(r11->Approved());
    EXPECT_EQ(executionCounter, 1);

    at2->SetValue<NOP::ReNotify>(1);
    NOP::Scheduler::Instance().FinishAll();

    EXPECT_TRUE(r11->Approved());
    EXPECT_EQ(executionCounter, 2);
}

TEST(Flag, Parallel)
{
    NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(-1);
    NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(-2);
    NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());
    NOP::SharedCondition cn1 = NOP::BuildCondition(CONDITION(*pr1), pr1);

    std::atomic<int> executionCounter{0};
    NOP::Method mt = [&] { executionCounter++; };
    NOP::SharedInstigation in1 = NOP::BuildInstigation(mt);
    NOP::SharedAction acl = NOP::BuildAction(in1);

    NOP::SharedRule r11 = NOP::BuildRule(cn1, acl);
    NOP::SharedRule r12 = NOP::BuildRule(cn1, acl);

    at1->SetValue<NOP::Parallel>(1);
    at2->SetValue<NOP::Parallel>(1);
    NOP::Scheduler::Instance().FinishAll();

    EXPECT_TRUE(r12->Approved());
    EXPECT_TRUE(r11->Approved());
    EXPECT_EQ(executionCounter, 2);
}

TEST(Flag, ReNotifyParallel)
{
    NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(-1);
    NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(-2);
    NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());
    NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Single>(pr1);
}

```

```

auto ptr = NOP::BuildPremise(at1, 1, NOP::Equal());

std::atomic<int> executionCounter{0};
NOP::Method mt = [&] { executionCounter++; };
NOP::SharedInstigation in1 = NOP::BuildInstigation(mt);
NOP::SharedAction ac1 = NOP::BuildAction(in1);

NOP::SharedRule r11 = NOP::BuildRule(cn1, ac1);
NOP::SharedRule r12 = NOP::BuildRule(cn1, ac1);

at1->SetValue<NOP::Parallel>(1);
at2->SetValue<NOP::Parallel>(1);
NOP::Scheduler::Instance().FinishAll();

EXPECT_TRUE(r12->Approved());
EXPECT_TRUE(r11->Approved());
EXPECT_EQ(executionCounter, 2);

at1->SetValue<NOP::Parallel | NOP::ReNotify>(1);
NOP::Scheduler::Instance().FinishAll();

// Everything should be approved
EXPECT_TRUE(r12->Approved());
EXPECT_TRUE(r11->Approved());
EXPECT_EQ(executionCounter, 4);
}

TEST(Flag, ReNotifyNoNotify)
{
    NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(-1);
    NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(-2);
    NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());
    NOP::SharedCondition cn1 = NOP::BuildCondition(CONDITION(*pr1), pr1);

    std::atomic<int> executionCounter{0};
    NOP::Method mt = [&] { executionCounter++; };
    NOP::SharedInstigation in1 = NOP::BuildInstigation(mt);
    NOP::SharedAction ac1 = NOP::BuildAction(in1);

    NOP::SharedRule r11 = NOP::BuildRule(cn1, ac1);

    at1->SetValue<NOP::NoNotify | NOP::ReNotify>(1);
    at2->SetValue<NOP::NoNotify | NOP::ReNotify>(1);
    NOP::Scheduler::Instance().FinishAll();

    EXPECT_FALSE(r11->Approved());
    EXPECT_EQ(executionCounter, 0);
}

#define LIBNOP_SCHEDULER_ENABLE
TEST(Scheduler, FIFO)
{
    NOP::Scheduler::Instance().SetStrategy(NOP::FIFO);
    NOP::SharedAttribute<bool> atDelay = NOP::BuildAttribute(false);
    NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(0);
    NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(0);
    NOP::SharedPremise pr1 = NOP::BuildPremise(at1, 1, NOP::Equal());
    NOP::SharedPremise pr2 = NOP::BuildPremise(at2, 2, NOP::Equal());
    NOP::SharedPremise prDelay = NOP::BuildPremise(atDelay, true, NOP::Equal());

    NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Single>(pr1);
    NOP::SharedCondition cn2 = NOP::BuildCondition<NOP::Single>(pr2);
    NOP::SharedCondition cnDelay = NOP::BuildCondition<NOP::Single>(prDelay);

    std::atomic<int> executionCounter{0};
    NOP::Method mt1 = [&] { EXPECT_EQ(1, ++executionCounter); };
    NOP::SharedInstigation in1 = NOP::BuildInstigation(mt1);
    NOP::SharedAction ac1 = NOP::BuildAction(in1);

    NOP::Method mt2 = [&] { EXPECT_EQ(2, ++executionCounter); };
    NOP::SharedInstigation in2 = NOP::BuildInstigation(mt2);
    NOP::SharedAction ac2 = NOP::BuildAction(in2);

    NOP::SharedRule r11 = NOP::BuildRule(cn1, ac1);
    NOP::SharedRule r12 = NOP::BuildRule(cn2, ac2);

    NOP::Method mtDelay = [&]
    { std::this_thread::sleep_for(std::chrono::milliseconds(10)); };
    NOP::SharedInstigation inDelay = NOP::BuildInstigation(mtDelay);
    NOP::SharedAction acDelay = NOP::BuildAction(inDelay);
    NOP::SharedRule r1Delay = NOP::BuildRule(cnDelay, acDelay);
}

```

```

atDelay->SetValue(true);
at1->SetValue(1);
at2->SetValue(2);
NOP::Scheduler::Instance().FinishAll();

EXPECT_EQ(executionCounter, 2);
}

TEST(Scheduler, LIFO)
{
    NOP::Scheduler::Instance().SetStrategy(NOP::LIFO);
    NOP::SharedAttribute<bool> atDelay = NOP::BuildAttribute(false);
    NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(0);
    NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(0);
    NOP::SharedPremise pr1 = NOP::BuildPremise(at1, 1, NOP::Equal());
    NOP::SharedPremise pr2 = NOP::BuildPremise(at2, 2, NOP::Equal());
    NOP::SharedPremise prDelay = NOP::BuildPremise(atDelay, true, NOP::Equal());

    NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Single>(pr1);
    NOP::SharedCondition cn2 = NOP::BuildCondition<NOP::Single>(pr2);
    NOP::SharedCondition cnDelay = NOP::BuildCondition<NOP::Single>(prDelay);

    std::atomic<int> executionCounter{0};
    NOP::Method mt1 = [&] { EXPECT_EQ(2, ++executionCounter); };
    NOP::SharedInstigation in1 = NOP::BuildInstigation(mt1);
    NOP::SharedAction ac1 = NOP::BuildAction(in1);

    NOP::Method mt2 = [&] { EXPECT_EQ(1, ++executionCounter); };
    NOP::SharedInstigation in2 = NOP::BuildInstigation(mt2);
    NOP::SharedAction ac2 = NOP::BuildAction(in2);

    NOP::SharedRule r11 = NOP::BuildRule(cn1, ac1);
    NOP::SharedRule r12 = NOP::BuildRule(cn2, ac2);

    NOP::Method mtDelay = [&]
    { std::this_thread::sleep_for(std::chrono::milliseconds(100)); };
    NOP::SharedInstigation inDelay = NOP::BuildInstigation(mtDelay);
    NOP::SharedAction acDelay = NOP::BuildAction(inDelay);
    NOP::SharedRule r1Delay = NOP::BuildRule(cnDelay, acDelay);

    atDelay->SetValue(true);
    std::this_thread::sleep_for(std::chrono::milliseconds(10));
    at1->SetValue(1);
    at2->SetValue(2);
    NOP::Scheduler::Instance().FinishAll();

    EXPECT_EQ(executionCounter, 2);
}

TEST(Scheduler, Priority)
{
    NOP::Scheduler::Instance().SetStrategy(NOP::Priority);
    NOP::SharedAttribute<bool> atDelay = NOP::BuildAttribute(false);
    NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(0);
    NOP::SharedPremise pr1 = NOP::BuildPremise(at1, 1, NOP::Equal());
    NOP::SharedPremise prDelay = NOP::BuildPremise(atDelay, true, NOP::Equal());

    NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Single>(pr1);
    NOP::SharedCondition cnDelay = NOP::BuildCondition<NOP::Single>(prDelay);

    std::atomic<int> executionCounter{0};
    NOP::Method mt1 = [&] { EXPECT_EQ(2, ++executionCounter); };
    NOP::SharedInstigation in1 = NOP::BuildInstigation(mt1);
    NOP::SharedAction ac1 = NOP::BuildAction(in1);

    NOP::Method mt2 = [&] { EXPECT_EQ(1, ++executionCounter); };
    NOP::SharedInstigation in2 = NOP::BuildInstigation(mt2);
    NOP::SharedAction ac2 = NOP::BuildAction(in2);

    NOP::Method mt3 = [&] { EXPECT_EQ(3, ++executionCounter); };
    NOP::SharedInstigation in3 = NOP::BuildInstigation(mt3);
    NOP::SharedAction ac3 = NOP::BuildAction(in3);

    NOP::SharedRule r11 = NOP::BuildRule(cn1, ac1, 0);
    NOP::SharedRule r12 = NOP::BuildRule(cn1, ac2, 123);
    NOP::SharedRule r13 = NOP::BuildRule(cn1, ac3, -1);

    NOP::Method mtDelay = [&]
    { std::this_thread::sleep_for(std::chrono::milliseconds(10)); };
    NOP::SharedInstigation inDelay = NOP::BuildInstigation(mtDelay);
}

```

```

NOP::SharedAction acDelay = NOP::BuildAction(inDelay);
NOP::SharedRule rlDelay = NOP::BuildRule(cnDelay, acDelay);

atDelay->SetValue(true);
at1->SetValue(1);
NOP::Scheduler::Instance().FinishAll();

EXPECT_EQ(executionCounter, 3);
}

#endif

TEST(Methods, Async)
{
    NOP::SharedAttribute<int> atl = NOP::BuildAttribute(-1);
    NOP::SharedPremise prl = NOP::BuildPremise(atl, 1, NOP::Equal());

    NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Single>(prl);

    std::atomic<int> executionCounter{0};
    NOP::SharedInstigation in1 =
        NOP::BuildInstigation(ASYNC_METHOD(++executionCounter));
    NOP::SharedAction acl = NOP::BuildAction(in1);

    NOP::SharedRule rl1 = NOP::BuildRule(cn1, acl);
    atl->SetValue(1);
    NOP::Scheduler::Instance().FinishAll();
    EXPECT_TRUE(rl1->Approved());
    // Should not be executed yet
    EXPECT_EQ(executionCounter, 0);
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    EXPECT_EQ(executionCounter, 1);
}

TEST(Methods, AsyncMultiple)
{
    NOP::Scheduler::Instance().SetStrategy(NOP::EStrategy::None);
    NOP::SharedAttribute<int> atl = NOP::BuildAttribute(-1);
    NOP::SharedPremise prl = NOP::BuildPremise(atl, 1, NOP::Equal());

    NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Single>(prl);

    std::atomic<int> executionCounter{0};
    NOP::SharedInstigation in1 =
        NOP::BuildInstigation(ASYNC_METHOD(++executionCounter));
    NOP::SharedAction acl = NOP::BuildAction(in1);

    NOP::SharedRule rl1 = NOP::BuildRule(cn1, acl);
    atl->SetValue(1);
    atl->SetValue(0);
    atl->SetValue(1);
    atl->SetValue(0);
    atl->SetValue(1);
    NOP::Scheduler::Instance().FinishAll();
    std::this_thread::sleep_for(std::chrono::milliseconds(200));
    EXPECT_EQ(executionCounter, 3);
}

TEST(Methods, AsyncMultipleSlow)
{
    NOP::Scheduler::Instance().SetStrategy(NOP::EStrategy::None);
    NOP::SharedAttribute<int> atl = NOP::BuildAttribute(-1);
    NOP::SharedPremise prl = NOP::BuildPremise(atl, 1, NOP::Equal());

    NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Single>(prl);

    std::atomic<int> executionCounter{0};
    NOP::SharedInstigation in1 = NOP::BuildInstigation(
        ASYNC_METHOD(std::this_thread::sleep_for(std::chrono::milliseconds(10));
                    ++executionCounter));
    NOP::SharedAction acl = NOP::BuildAction(in1);

    NOP::SharedRule rl1 = NOP::BuildRule(cn1, acl);
    atl->SetValue(1);
    atl->SetValue(0);
    atl->SetValue(1);
    atl->SetValue(0);
    atl->SetValue(1);
    NOP::Scheduler::Instance().FinishAll();
    std::this_thread::sleep_for(std::chrono::milliseconds(20));
    EXPECT_EQ(executionCounter, 3);
}

```

```

TEST(Parallel, Action)
{
    NOP::Scheduler::Instance().SetStrategy(NOP::EStrategy::None);
    NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(-1);
    NOP::SharedPremise pr1 = NOP::BuildPremise(at1, 1, NOP::Equal());
    NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Single>(pr1);

    std::atomic<int> executionCounter{0};
    NOP::SharedInstigation in1 =
        NOP::BuildInstigation(METHOD(++executionCounter));
    NOP::SharedInstigation in2 =
        NOP::BuildInstigation(METHOD(++executionCounter));
    NOP::SharedAction ac1 = NOP::BuildAction<NOP::Parallel>(in1, in2);

    NOP::SharedRule r11 = NOP::BuildRule(cn1, ac1);
    at1->SetValue(1);
    at1->SetValue(0);
    at1->SetValue(1);
    at1->SetValue(0);
    at1->SetValue(1);
    NOP::Scheduler::Instance().FinishAll();
    EXPECT_EQ(executionCounter, 6);
}

TEST(Parallel, Instigation)
{
    NOP::Scheduler::Instance().SetStrategy(NOP::EStrategy::None);
    NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(-1);
    NOP::SharedPremise pr1 = NOP::BuildPremise(at1, 1, NOP::Equal());
    NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Single>(pr1);

    std::atomic<int> executionCounter{0};
    NOP::SharedInstigation in1 = NOP::BuildInstigation<NOP::Parallel>(
        METHOD(++executionCounter), METHOD(++executionCounter));

    NOP::SharedAction ac1 = NOP::BuildAction<NOP::Parallel>(in1);

    NOP::SharedRule r11 = NOP::BuildRule(cn1, ac1);
    at1->SetValue(1);
    at1->SetValue(0);
    at1->SetValue(1);
    at1->SetValue(0);
    at1->SetValue(1);
    NOP::Scheduler::Instance().FinishAll();
    EXPECT_EQ(executionCounter, 6);
}

```

APÊNDICE E

README.md

7/25/2021

Framework NOP C++ 4.0 (libnop)

This framework was developed to make it easier to build applications using the NOP in C++. It uses CMake to build the library and applications.

Requirements

- Unix or Windows (OSX is untested)
- CMake 3.14
- C++20 compliant compiler (MSVC 14.2 / GCC 10 or GCC 11 is recommended)

Usage instructions

Check the template project for a ready to use template: <https://nop.dainf.ct.utfpr.edu.br/nop-applications/framework-application/framework-cpp-4-application/template-application>

Check libnop_gtest/test.cpp file for more examples on how to use the framework.

Build options

The following options must be set during cmake generation.

Usage options:

- LIBNOP_SCHEDULER_ENABLE: Enable use of scheduler for executing Rules (reduces performance)
- LIBNOP_LOG_ENABLE: Enable using logs (heavy performance hit)

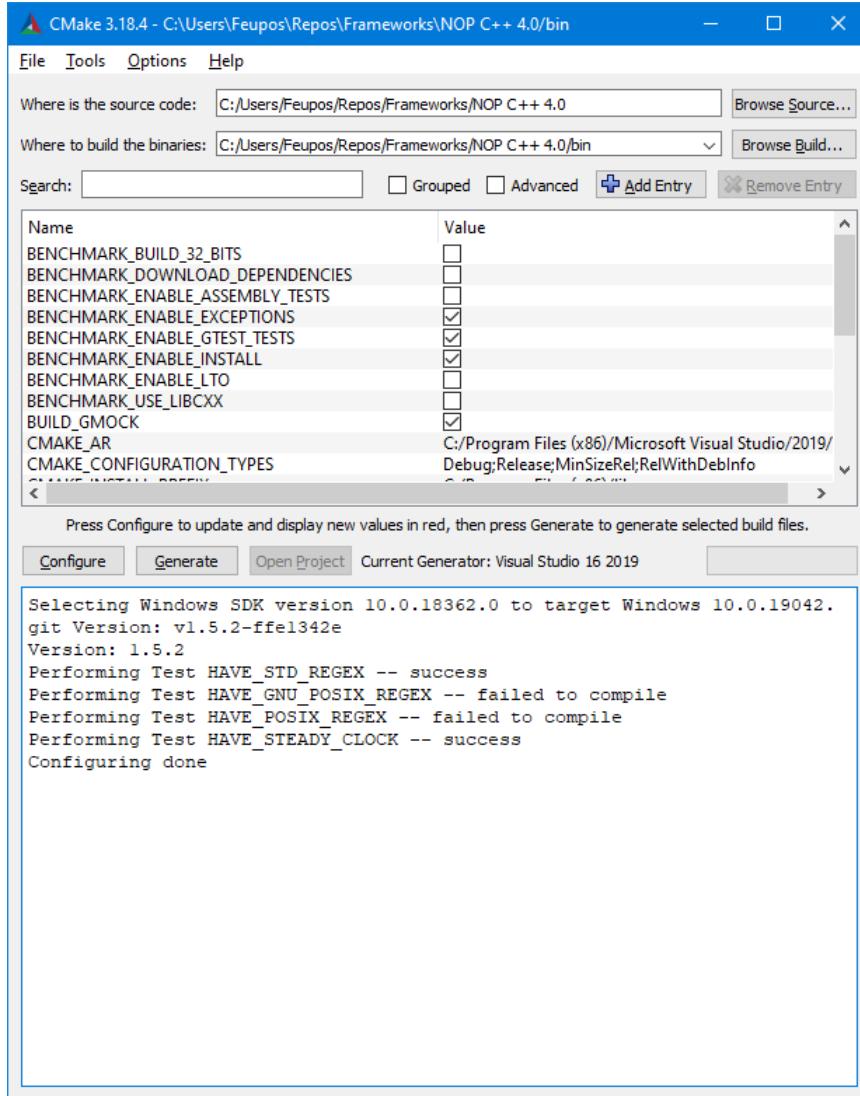
Development options:

- LIBNOP_TEST_ENABLE: Enable building unity tests
- LIBNOP_COVERAGE_ENABLE: Enable code coverage
- LIBNOP_BENCHMARK_ENABLE: Enable build benchmarks
- LIBNOP_BENCHMARK_SENSOR_ENABLE: Enable build sensor benchmark
- LIBNOP_BENCHMARK_BITONIC_ENABLE: Enable build bitonic benchmark
- LIBNOP_BENCHMARK_RANDOMFOREST_ENABLE: Enable build random forest benchmark
- LIBNOP_FW2_TEST_ENABLE: Enable build NOP C++ Framework 2.0 tests

For maximum performance, all options are disabled by default. As an alternative you can add "#define LIBNOP_SCHEDULER_ENABLE" before including "#include "NOPFramework.h"" or "#include "libnop/framework.h"" to enable the scheduler.

Hot to build (Windows)

Download and install the CMake GUI for windows and generate the project for your IDE.
<https://cmake.org/download/>



1. Set source folder - Ex: nop-framework-cpp-4/
2. Set build folder - Ex: nop-framework-cpp-4/build
3. Configure the desired options
4. Generate
5. Open Project

Hot to build (Unix)

On Ubuntu 20.04 you must install gcc 10/11 and libtbb and cmake as follows:

1. `sudo apt-get install -y g++-11`

2. `sudo apt-get install -y libtbb-dev`
3. `sudo apt-get install python3-pip`
4. `sudo pip install cmake`

With the dependencies installed you can compile:

1. `mkdir build && cd build`
2. `cmake .. -DCMAKE_CXX_COMPILER=g++-11`
3. `sudo make install`

Tests and benchmark (Unix)

After building you can run the tests and benchmarks:

1. `libnop_gtest/libnop_gtest`
2. `libnop_gbench/libnop_gbench`

How to use

Include the header "libnop/framework.h" in your files.

To utilize this library in other CMake projects you can add the following lines to your CMakeLists.txt:

```
find_package(libnop)
if(NOT libnop_FOUND)
    include(FetchContent)
    FetchContent_Declare(libnop
        GIT_REPOSITORY https://nop.dainf.ct.utfpr.edu.br/nop-
implementations/frameworks/nop-framework-cpp-4.git
        GIT_TAG master
    )

    FetchContent_GetProperties(libnop)
    if(NOT libnop_POPULATED)
        FetchContent_MakeAvailable(libnop)
    endif()
endif()
```

C++20 is required for use:

```
set_target_properties(<target> PROPERTIES
    CXX_STANDARD 20
)
```

Alternatively you can clone the repository and install the library in your system:

```
1. git clone https://nop.dainf.ct.utfpr.edu.br/nop-implementations/frameworks/nop-framework-cpp-4.git
2. cd nop-framework-cpp-4.git
3. mkdir build
4. cd build
5. cmake .. -DCMAKE_CXX_COMPILER=g++-11
6. sudo make install
```

Then import with CMake with `find_package`:

```
find_package(libnop)
```

CI/CD

To make use of code coverage make sure you have a valid runner:

<https://nop.dainf.ct.utfpr.edu.br/help/ci/runners/README> <https://docs.gitlab.com/runner/install/>

How to use

To use this framework you can just define and initialize the NOP entities as desired.

For more details about the NOP entities you may check the NOP bibliography, such as the paper regarding this framework available [here](#).

All the NOP entities are defined as shared pointers, so they can be easily shared in the code.

You must declare the entity and the initialize the pointer using one of the available builders.

The following header must be included: `#include "libnop/framework.h"`.

Attribute

The Attribute has a template parameter that can be any data type. It has a single builder that receives the initial value of the attribute.

```
template <typename T>
using SharedAttribute = std::shared_ptr<Attribute<T>>;
```

Example:

```
NOP::SharedAttribute<int> atl = NOP::BuildAttribute(1);
```

Premise

The Premise can be built with two Attributes or an Attribute and a value. The third parameter is the Premise operation between it Attributes.

```
template <typename T, typename Func>
auto BuildPremise(std::shared_ptr<Attribute<T>> lhs, T rhs, Func op);

template <typename T, typename Func>
auto BuildPremise(std::shared_ptr<Attribute<T>> lhs,
                  std::shared_ptr<Attribute<T>> rhs, Func op);

/* The operation op can be:
   NOP::Equal(),
   NOP::Different(),
   NOP::Greater(),
   NOP::GreaterEqual(),
   NOP::Less(),
   NOP::LessEqual()
*/
```

Example:

```
NOP::SharedAttribute<int> at1 = NOP::BuildAttribute<int>(-1);
NOP::SharedAttribute<int> at2 = NOP::BuildAttribute<int>(-2);

NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());
NOP::SharedPremise pr2 = NOP::BuildPremise(at1, 0, NOP::Different());
```

Condition

The Condition is built using as arguments one or more Premises, Conditions (as Sub-Bonditions) or Rules (as Master Rules).

The simple condition builder with the condition type as a template parameter:

```
template <EConditionType type, typename... Args>
auto BuildCondition(Args... args);

/* The condition type can be:
   NOP::Single,
   NOP::Conjunction,
   NOP::Disjunction
*/
```

More complex conditions can be composed as a boolean expression, and passing the pointers as arguments. The operation can be passed as a lambda expression, where the CONDITION macro can be used to facilitate the definition.

```
template <typename... Args>
auto BuildCondition(std::function<bool(void)> operation, Args... args);

#define CONDITION(expression) [=, this]() { return bool(expression); }
```

Example:

```
NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Conjunction>(pr1,
pr2);
NOP::SharedCondition cn2 =
    NOP::BuildCondition(CONDITION((*pr1 || *pr2) && !*pr3), pr1, pr2,
pr3);
```

Method

The Method is stored as a std::function with no parameters. It can be initialized with a lambda expression. The macro METHOD can be used to facilitate the definition.

```
using Method = std::function<void(void)>;
#define METHOD(expression) [&]() { expression }
```

Example:

```
int counter{0};
NOP::Method mt1{ METHOD(counter++) };
```

Instigation

The Instigation is built with one or more Methods as parameters. A template parameter can be used to define a parallel Instigation (all Methods are executed in parallel).

```
template <NOPFlag Flag = Default, typename... Args>
auto BuildInstigation(Args... methods)
```

Example:

```
NOP::SharedInstigation in1 = NOP::BuildInstigation(mt1, mt2);
NOP::SharedInstigation in2 = NOP::BuildInstigation<NOP::Parallel>(mt3,
mt4);
```

Action

The Action is built with one or more Instigations as parameters. A template parameter can be used to define a parallel Action (all Instigations are instigated in parallel).

```
template <NOPFlag Flag = Default, typename... Args>
auto BuildAction(Args... instigation);
```

Example:

```
NOP::SharedAction ac1 = NOP::BuildAction(in1, in2);
NOP::SharedAction ac2 = NOP::BuildAction<NOP::Parallel>(in3, in4);
```

Rule

The Rule is build using an Action and a Condition. When using the Scheduler you can also define a priority value (bigger values has higher priority).

```
std::shared_ptr<Rule> BuildRule(std::shared_ptr<Condition> condition,
                                    std::shared_ptr<Action> action,
                                    const int priority = 0)
```

Example:

```
NOP::SharedRule rl1 = NOP::BuildRule(cn1, ac1);
NOP::SharedRule rl2 = NOP::BuildRule(cn1, ac1, 10);
```

FBE

The FBE class can be inherited by any classes containing Attributes or Methods. It is not required, but facilitates organizing the NOP entities.

```
class Example: public NOP::FBE
{
    NOP::SharedAttribute<int> atl;
};
```

Scheduler

The Scheduler can be used to ensure determinism when executing the Rules. It executes Rules one by one, while rechecking the condition for each one. Beware, performance is heavily

impacted when using the Scheduler.

To use the Scheduler you must use the compile definition -DLIBNOP_SCHEDULER_ENABLE-=ON or define `#define LIBNOP_SCHEDULER_ENABLE` before including `#include "libnop/framework.h"`.

The Scheduler execution strategy can be set at any point. The ignore_conflict flag may be used if conflict resolution is not desired.

```
void SetStrategy(EStrategy strategy, bool ignore_conflict = false);

/* Valid strategy types are:
NOP::None,
NOP::FIFO,
NOP::LIFO,
NOP::Priority
*/
```

Example:

```
NOP::Scheduler::Instance().SetStrategy(NOP::FIFO);
```

Logger

The framework provides debuggin facilities by means of a Logger.

The Logger can be used to log the NOP execution flow. Beware, performance is heavily impacted when using the Logger.

To use the Scheduler you must use the compile definition -DLIBNOP_LOGGER_ENABLE-=ON or define `#define LIBNOP_LOGGER_ENABLE` before including `#include "libnop/framework.h"`.

The log file location can be set with:

```
void Logger::SetLogFileName(const std::string& file_name);
```

All the NOP entities proved auxiliary builder that can name the entities for logging purposes. Those builders are prefixed Named, and have an additional name parameter. The FBE constructor also accepts a name parameter. When those entities are members of an FBE they may also pass the FBE pointer as a parameter.

Example:

```
NOP::Logger::Instance().SetLogFileName("test.log");
```

```

struct Test : NOP::FBE
{
    explicit Test(const std::string_view name) : FBE(name) {}
    NOP::SharedAttribute<int> at1 =
        NOP::BuildAttributeNamed("at1", this, -1);
    NOP::SharedAttribute<int> at2 =
        NOP::BuildAttributeNamed("at2", this, -2);
    NOP::SharedPremise pr1 =
        NOP::BuildPremiseNamed("pr1", this, at1, at2, NOP::Equal());
    NOP::SharedCondition cn1 =
        NOP::BuildConditionNamed<NOP::Single:(cn1", this, pr1);

    std::atomic<int> executionCounter{0};
    NOP::Method mt = [&] { executionCounter++; };
    NOP::SharedInstigation in1 =
        NOP::BuildInstigationNamed("in1", this, mt);
    NOP::SharedAction ac1 = NOP::BuildActionNamed("ac1", this, in1);

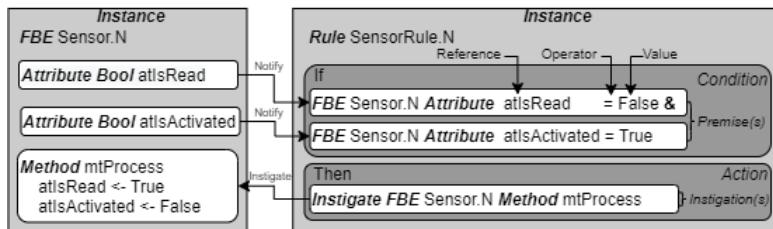
    NOP::SharedRule rl1 = NOP::BuildRuleNamed("rl1", this, cn1, ac1);
};

Test test{"TestFBE"};

```

FBE example

The following FBE and Rule can be implemented as an example.



As described using NOPL:

```

fbe Sensor
public boolean atIsRead = false
public boolean atIsActivated = false
private method mtProcess
    attribution
        this.atIsRead = true
        this.atIsActivated = false
    end_attribution
end_method
rule rlSensor
    condition
        premise prIsActivated
            this.atIsActivated == true

```

```
end_premise
and
premise prIsNotRead
    this.atIsRead == false
end_premise
end_condition
action sequential
    instigation sequential
        call this.mtProcess()
    end_instigation
end_action
end_rule
end_fbe
```

And defined using the framework:

```
struct NOPSensor: NOP::FBE
{
    NOP::SharedAttribute<bool> atIsRead{NOP::BuildAttribute(false)};
    NOP::SharedAttribute<bool> atIsActivated{NOP::BuildAttribute(false)};
    NOP::Method mtProcess{METHOD(atIsRead->SetValue(true); atIsActivated-
>SetValue(false));};

    NOP::SharedPremise prIsActivated{
        NOP::BuildPremise(atIsActivated, true, NOP::Equal());
    }
    NOP::SharedPremise prIsNotRead{
        NOP::BuildPremise(atIsRead, false, NOP::Equal());
    }

    NOP::SharedRule rlSensor{NOP::BuildRule(
        NOP::BuildCondition<NOP::Conjunction>(
            NOP::BuildPremise(atIsActivated, true, NOP::Equal()),
            NOP::BuildPremise(atIsRead, false, NOP::Equal())
        ),
        NOP::BuildAction(
            NOP::BuildInstigation(mtProcess)
        )
    );
};
```

APÊNDICE F

A aplicação mira ao alvo tem como objetivo ser uma aplicação que realce os problemas de redundâncias estruturais e temporais, permitindo assim comparações de desempenho do PON com os demais paradigmas. Neste cenário, ilustrado na Figura 112, um conjunto de arqueiros deve atirar em um conjunto de maçãs, de acordo com o estado das maçãs e da arma de fogo que tem a função de sinalizar o disparo de início desta interação.

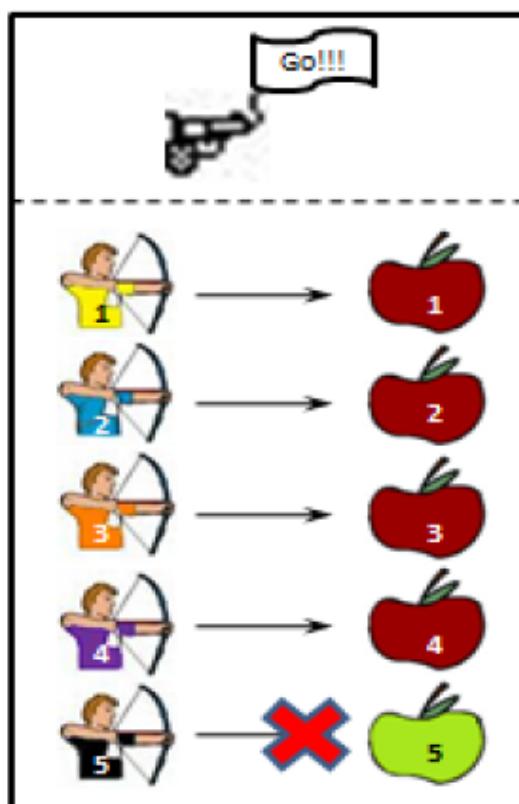


Figura 113 – Aplicação mira ao alvo
Fonte: Banaszewski (2009)

O Código 1 apresenta o desenvolvimento deste cenário com o *Framework PON C++ 4.0*. A maçã, arqueiro e arma de fogo são representados, respectivamente, por *NOP4Apple*, *NOP4Archer* e *NOP4Gun*, enquanto a *Rule* é implementada em *NOP4GUn*.

Código 1 – Código da estrutura do mira ao alvo com o *Framework PON C++ 4.0*

```
struct NOP4Apple : NOP::FBE {
    NOP::SharedAttribute<std::string> atColor =
        NOP::BuildAttribute<std::string>("GREEN");
    NOP::SharedAttribute<bool> atStatus = NOP::BuildAttribute(true);
    NOP::SharedAttribute<bool> atIsCrossed = NOP::BuildAttribute(false);
};

struct NOP4Archer : NOP::FBE {
    NOP::SharedAttribute<bool> atStatus = NOP::BuildAttribute(true);
```

```

};

struct NOP4Gun : NOP::FBE {
    NOP::SharedAttribute<bool> atStatus = NOP::BuildAttribute(false);
};

struct NOP4ArcherRule {
    NOP::SharedRule rule;
    NOP4ArcherRule(std::shared_ptr<NOP4Apple>& apple,
                    std::shared_ptr<NOP4Archer>& archer,
                    NOP::SharedPremise& prGunIsTrue) {
        rule = NOP::BuildRule(
            NOP::BuildCondition<NOP::Conjunction>(
                NOP::BuildPremise<std::string>(apple->atColor, "RED",
                                                NOP::Equal()),
                NOP::BuildPremise(apple->atStatus, true, NOP::Equal()),
                NOP::BuildPremise(archer->atStatus, true, NOP::Equal()),
                prGunIsTrue),
            NOP::BuildAction(NOP::BuildInstigation(
                [=]() { apple->atIsCrossed->SetValue(true); })));
    }
};
}

```

Fonte: Autoria própria

Durante os experimentos realizados por Banaszewski (2009), a aplicação desenvolvida com o *Framework PON C++ 1.0* apresentou desempenho um pouco inferior à mesma aplicação desenvolvida com o POO, conforme apresentado na Figura 113. Porém, nos novos testes realizados com o *Framework PON C++ 4.0*, a aplicação em PON não conseguiu chegar perto dos mesmos resultados, conforme apresentado na Figura 114.

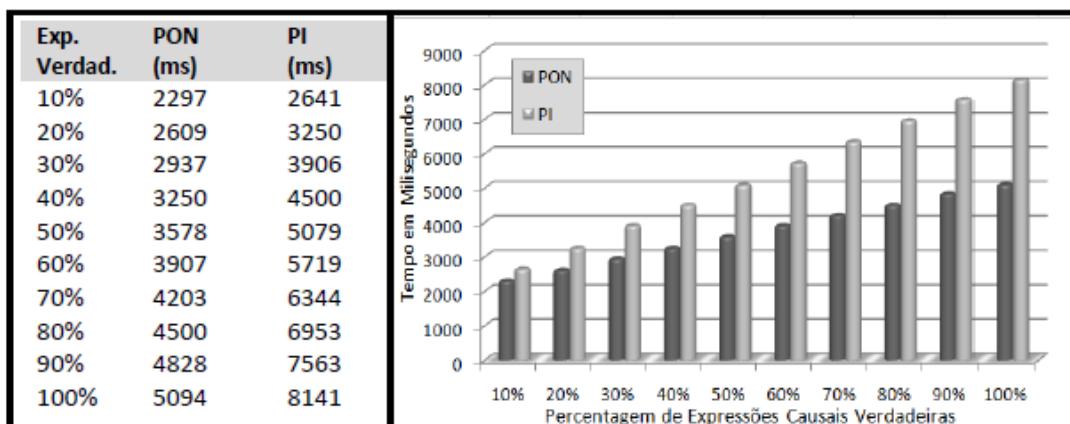


Figura 114 – Resultado do experimento mira ao alvo com o *Framework PON C++ 1.0*
Fonte: Banaszewski (2009)

A dificuldade em se reproduzir os mesmos resultados pode ser justificada pela dificuldade em se determinar as condições originais do teste, visto que a versão do compilador e opções de otimização podem influenciar de forma significativa o desempenho da aplicação. Além disso, processadores com arquiteturas mais modernas, com os utilizados nos novos testes¹, apresentam recursos que otimizam a execução de operações condicionais e laços de repetição, de modo que são capazes de compensar os efeitos das redundâncias temporais e estruturais.

¹ Teste executado em um computador com processador Ryzen 5 3600 a 3.6GHz e 16 GB de RAM DDR4 a 3000MHz (dual channel) e sistema operacional Windows 10 64 bits.

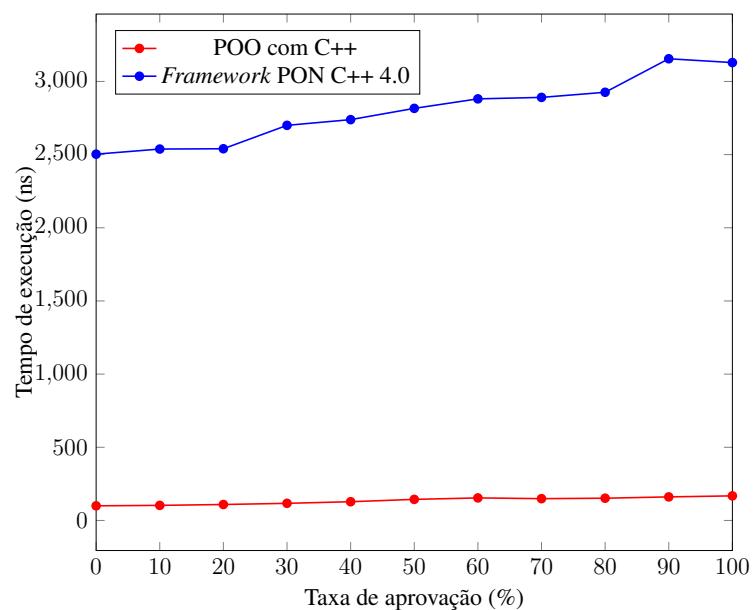


Figura 115 – Resultado do experimento mira ao alvo com o Framework PON C++ 4.0
Fonte: Autoria própria

APÊNDICE G

CÓDIGO-FONTE DA APLICAÇÃO BITONIC SORT EM PON NO *FRAMEWORK PON C++* 4.0

```

template <typename T = int>
struct NOPBitonicSorter
{
    std::map<int, std::vector<NOP::SharedAttribute<T>>> elements;
    std::vector<NOP::SharedPremise> premises;
    std::vector<NOP::SharedRule> rules;
    int stages{0};
    std::vector<T> out;
    size_t size;
    std::vector<int> indexes;

    NOPBitonicSorter(size_t len) : size{len}
    {
        out.resize(size);
        indexes.resize(len);
        std::iota(indexes.begin(), indexes.end(), 0);

        Init();
    }

    auto Sort(const std::vector<T>& input)
    {
        for (auto i = 0; i < input.size(); i++)
        {
            elements[0][i]->SetValue(input[i], (i % 2));
        }
        return out;
    }

    auto ParallelSort(const std::vector<T>& input)
    {
        std::for_each(
            std::execution::par_unseq, indexes.begin(), indexes.end(),
            [&](int i) {
                elements[0][i]->template SetValue<NOP::ReNotify>(input[i]);
            });
        return out;
    }

    void MoveAttributes(NOP::SharedAttribute<T>& at1,
                         NOP::SharedAttribute<T>& at2, int stage, int x, int y)
    {
        if (stage < stages - 1)
        {
            elements[stage + 1][x]->template SetValue<NOP::ReNotify>(
                at1->GetValue());
            elements[stage + 1][y]->template SetValue<NOP::ReNotify>(
                at2->GetValue());
        }
        else
        {
            out[x] = at1->GetValue();
            out[y] = at2->GetValue();
        }
    }

    void CreateComparatorFBE(int x, int y, int stage, bool reverse = false)
    {
        auto& at1 = elements[stage][x];
        auto& at2 = elements[stage][y];
        NOP::SharedPremise pr1;
        NOP::SharedPremise pr2;
        if (!reverse)
        {
            pr1 = NOP::BuildPremise(at2, at1, NOP::Greater());
            pr2 = NOP::BuildPremise(at2, at1, NOP::LessEqual());
        }
    }
}

```

```

    }
    else
    {
        pr2 = NOP::BuildPremise(at2, at1, NOP::Greater());
        pr1 = NOP::BuildPremise(at2, at1, NOP::LessEqual());
    }
    premises.push_back(pr1);
    premises.push_back(pr2);

    // Rule to move
    rules.push_back(NOP::BuildRule(
        NOP::BuildCondition(CONDITION(*pr1), pr1),
        NOP::BuildAction(NOP::BuildInstigation([&, stage, x, y] () {
            MoveAttributes(at1, at2, stage, x, y);
        })));
    // Rule to swap
    rules.push_back(NOP::BuildRule(
        NOP::BuildCondition(CONDITION(*pr2), pr2),
        NOP::BuildAction(NOP::BuildInstigation([&, stage, x, y] () {
            MoveAttributes(at1, at2, stage, y, x);
        })));
}
}

void Init()
{
    elements.clear();
    premises.clear();
    rules.clear();

    const size_t half = size / 2;
    const size_t quarter = size / 4;

    int n = 1;
    for (int i = 1; i < log2(size); i++)
    {
        stages += n++;
    }
    for (int s = 0; s < stages; s++)
    {
        elements[s].resize(size);
        for (int i = 0; i < size; i++)
        {
            elements[s][i] = NOP::BuildAttribute(T());
        }
    }

    int stage = 0;
    for (auto div = half; div > 1; div = div / 2)
    {
        bool reverse{false};
        int internal_stage = stage;
        for (auto start = 0; start < size; start += size / div)
        {
            internal_stage = stage;
            for (auto step = half / div; step > 0; step = step / 2)
            {
                for (auto i = 0; i < step; i++)
                {
                    for (auto j = 0; j < size / div; j = j + 2 * step)
                    {
                        int x = start + i + j;
                        int y = start + i + j + step;
                        CreateComparatorFBE(x, y, internal_stage, reverse);
                    }
                }
                internal_stage++;
            }
            reverse = !reverse;
        }
        stage += (internal_stage - stage);
    }

    int sort_stages = log2(size);
    int curr_stage = stages;
    for (int s = 0; s < sort_stages; s++)
    {
        stages++;
        elements[curr_stage + s].resize(size);
        for (int i = 0; i < size; i++)
        {
            elements[curr_stage + s][i] = NOP::BuildAttribute(T());
        }
    }
}

```

```

        }

    stage = curr_stage;
    for (auto start = 0; start < size; start += size)
    {
        for (auto step = half; step > 0; step = step / 2)
        {
            for (auto i = 0; i < step; i++)
            {
                for (auto j = 0; j < size; j = j + 2 * step)
                {
                    auto x = start + i + j;
                    auto y = start + i + j + step;
                    CreateComparatorFBE(x, y, stage);
                }
            }
            stage++;
        }
    }
};

template <typename T = int>
struct NOPBitonicSorterStages
{
    std::vector<NOP::SharedAttribute<bool>> stages;
    std::map<int, NOP::SharedPremise> stage_premises;

    std::vector<NOP::SharedAttribute<T>> elements;
    std::vector<NOP::SharedPremise> premises;
    std::vector<NOP::SharedRule> rules;

    std::vector<int> out{};
    size_t size;

    NOPBitonicSorterStages(size_t len) : size{len}
    {
        Init();
        out.resize(len);
    }

    void Input(std::vector<T>& input)
    {
        for (size_t i = 0; i < input.size(); i++)
        {
            elements[i]->SetValue(input[i], NOP::NoNotify);
        }
    }

    auto ParallelSort()
    {
        for (const auto& stage : stages)
        {
            stage->SetValue<NOP::Parallel>(&true);
            stage->SetValue<NOP::Parallel>(&false);
        }
        for (auto i = 0; i < elements.size(); i++)
        {
            out[i] = elements[i]->GetValue();
        }
        return out;
    }

    auto Sort()
    {
        for (const auto& stage : stages)
        {
            stage->SetValue(&true);
            stage->SetValue(&false);
        }
        for (auto i = 0; i < elements.size(); i++)
        {
            out[i] = elements[i]->GetValue();
        }
        return out;
    }

    void Init()
    {

```

```

elements.clear();
premises.clear();
rules.clear();

for (auto i = 0; i < size; i++)
{
    elements.push_back(NOP::BuildAttribute(T()));
}
const size_t half = size / 2;
const size_t quarter = size / 4;

int n = 1;
int total_stages = 0;
for (int i = 1; i < log2(size); i++)
{
    total_stages += n++;
}

for (int s = 0; s < total_stages; s++)
{
    NOP::SharedAttribute<bool> atStage(NOP::BuildAttribute(false));
    stages.push_back(atStage);
    NOP::SharedPremise prStage =
        NOP::BuildPremise(atStage, true, NOP::Equal());
    stage_premises[s] = prStage;
}

int stage = 0;
for (auto div = half; div > 1; div = div / 2)
{
    bool reverse{false};
    int internal_stage = stage;
    for (auto start = 0; start < size; start += size / div)
    {
        internal_stage = stage;
        for (auto step = half / div; step > 0; step = step / 2)
        {
            for (auto i = 0; i < step; i++)
            {
                for (auto j = 0; j < size / div; j = j + 2 * step)
                {
                    auto x = start + i + j;
                    auto y = start + i + j + step;
                    auto& at1 = elements[x];
                    auto& at2 = elements[y];
                    NOP::SharedPremise pr;
                    if (reverse)
                    {
                        pr = NOP::BuildPremise(at1, at2, NOP::Less());
                    }
                    else
                    {
                        pr =
                            NOP::BuildPremise(at1, at2, NOP::Greater());
                    }
                    premises.push_back(pr);

                    rules.push_back(NOP::BuildRule(
                        NOP::BuildCondition<NOP::Conjunction>(
                            pr, stage_premises[internal_stage]),
                        NOP::BuildAction(NOP::BuildInstigation(
                            METHOD(SwapAttributes(at1, at2))))));
                }
            }
            internal_stage++;
        }
        reverse = !reverse;
    }
    stage += (internal_stage - stage);
}

int sort_stages = log2(size);
int curr_stage = total_stages;
for (int s = 0; s < sort_stages; s++)
{
    NOP::SharedAttribute<bool> atStage(NOP::BuildAttribute(false));
    stages.push_back(atStage);
    NOP::SharedPremise prStage =
        NOP::BuildPremise(atStage, true, NOP::Equal());
    stage_premises[curr_stage + s] = prStage;
}

```

```
stage = curr_stage;
for (auto start = 0; start < size; start += size)
{
    for (auto step = half; step > 0; step = step / 2)
    {
        for (auto i = 0; i < step; i++)
        {
            for (auto j = 0; j < size; j = j + 2 * step)
            {
                auto x = start + i + j;
                auto y = start + i + j + step;
                auto& at1 = elements[x];
                auto& at2 = elements[y];
                NOP::SharedPremise pr =
                    NOP::BuildPremise(at1, at2, NOP::Greater());
                premises.push_back(pr);
                rules.push_back(NOP::BuildRule(
                    NOP::BuildCondition<NOP::Conjunction>(
                        pr, stage_premises[stage]),
                    NOP::BuildAction(NOP::BuildInstigation(
                        METHOD(SwapAttributes(at1, at2))))));
            }
        }
        stage++;
    }
};
```

APÊNDICE H

CÓDIGO-FONTE DA APLICAÇÃO BITONIC SORT NO PP EM LINGUAGEM DE PROGRAMAÇÃO C

```

/*
bitonic.c

This file contains two different implementations of the bitonic sort
recursive version
imperative version : impBitonicSort()

The bitonic sort is also known as Batcher Sort.
For a reference of the algorithm, see the article titled
Sorting networks and their applications by K. E. Batcher in 1968

The following codes take references to the codes available at
http://www.cag.lcs.mit.edu/streamit/results/bitonic/code/c/bitonic.c
http://www.tools-of-computing.com/tc/CS/Sorts/bitonic_sort.htm
http://www.itи.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonicen.htm
 */

/*
----- -----
Nikos Pitsianis, Duke CS
----- -----
*/

#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

struct timeval startwtime, endwtime;
double seq_time;

int N; // data array size
int *a; // data array to be sorted

const int ASCENDING = 1;
const int DESCENDING = 0;

void init(void);
void print(void);
void sort(void);
void test(void);
inline void exchange(int i, int j);
void compare(int i, int j, int dir);
void bitonicMerge(int lo, int cnt, int dir);
void recBitonicSort(int lo, int cnt, int dir);
void impBitonicSort(void);

/** the main program */
int main(int argc, char **argv) {

    if (argc != 2) {
        printf("Usage: %s n\n where n is problem size (power of two)\n", argv[0]);
        exit(1);
    }

    N = atoi(argv[1]);
    a = (int *) malloc(N * sizeof(int));

    init();
}

```

```

gettimeofday (&startwtime, NULL);
impBitonicSort();
gettimeofday (&endwtime, NULL);

seq_time = (double) ((endwtime.tv_usec - startwtime.tv_usec)/1.0e6
                     + endwtime.tv_sec - startwtime.tv_sec);

printf("Imperative wall clock time = %f\n", seq_time);

test();

init();
gettimeofday (&startwtime, NULL);
sort();
gettimeofday (&endwtime, NULL);

seq_time = (double) ((endwtime.tv_usec - startwtime.tv_usec)/1.0e6
                     + endwtime.tv_sec - startwtime.tv_sec);

printf("Recursive wall clock time = %f\n", seq_time);

test();

// print();
}

/** ----- SUB-PROCEDURES ----- **/

/** procedure test() : verify sort results **/
void test() {
    int pass = 1;
    int i;
    for (i = 1; i < N; i++) {
        pass &= (a[i-1] <= a[i]);
    }

    printf(" TEST %s\n", (pass) ? "PASSED" : "FAILED");
}

/** procedure init() : initialize array "a" with data **/
void init() {
    int i;
    for (i = 0; i < N; i++) {
        // a[i] = rand() % N; // (N - i);
        a[i] = (N - i);
    }
}

/** procedure print() : print array elements **/
void print() {
    int i;
    for (i = 0; i < N; i++) {
        printf("%d\n", a[i]);
    }
    printf("\n");
}

/** INLINE procedure exchange() : pair swap **/
inline void exchange(int i, int j) {
    int t;
    t = a[i];
    a[i] = a[j];
    a[j] = t;
}

/** procedure compare()
   The parameter dir indicates the sorting direction, ASCENDING
   or DESCENDING; if (a[i] > a[j]) agrees with the direction,
   then a[i] and a[j] are interchanged.
*/
void compare(int i, int j, int dir) {
    if (dir==(a[i]>a[j]))
        exchange(i,j);
}

```

```

/** Procedure bitonicMerge()
   It recursively sorts a bitonic sequence in ascending order,
   if dir = ASCENDING, and in descending order otherwise.
   The sequence to be sorted starts at index position lo,
   the parameter cnt is the number of elements to be sorted.
*/
void bitonicMerge(int lo, int cnt, int dir) {
    if (cnt>1) {
        int k=cnt/2;
        int i;
        for (i=lo; i<lo+k; i++)
            compare(i, i+k, dir);
        bitonicMerge(lo, k, dir);
        bitonicMerge(lo+k, k, dir);
    }
}

/** function recBitonicSort()
   first produces a bitonic sequence by recursively sorting
   its two halves in opposite sorting orders, and then
   calls bitonicMerge to make them in the same order
*/
void recBitonicSort(int lo, int cnt, int dir) {
    if (cnt>1) {
        int k=cnt/2;
        recBitonicSort(lo, k, ASCENDING);
        recBitonicSort(lo+k, k, DESCENDING);
        bitonicMerge(lo, cnt, dir);
    }
}

/** function sort()
   Caller of recBitonicSort for sorting the entire array of length N
   in ASCENDING order
*/
void sort() {
    recBitonicSort(0, N, ASCENDING);
}

/*
   imperative version of bitonic sort
*/
void impBitonicSort() {

    int i,j,k;

    for (k=2; k<=N; k=2*k) {
        for (j=k>>1; j>0; j=j>>1) {
            for (i=0; i<N; i++) {
                int ij=i^j;
                if ((ij)>i) {
                    if ((i&k)==0 && a[i] > a[ij])
                        exchange(i,ij);
                    if ((i&k)!=0 && a[i] < a[ij])
                        exchange(i,ij);
                }
            }
        }
    }
}

```

APÊNDICE I

CÓDIGO-FONTE DA APLICAÇÃO RANDOM FOREST EM PON NO *FRAMEWORK PON C++ 4.0*

```

#pragma once
#include "libnop/framework.h"

std::vector<std::vector<int>> test_data{
    {580, 280, 509, 240}, {600, 220, 400, 100}, {550, 420, 140, 20},
    {730, 290, 630, 180}, {1500, 340, 150, 20}, {630, 330, 600, 250},
    {500, 350, 130, 30}, {670, 310, 470, 150}, {680, 280, 480, 140},
    {610, 280, 400, 130}, {610, 260, 560, 140}, {640, 320, 450, 150},
    {610, 280, 470, 120}, {650, 280, 459, 150}, {610, 290, 470, 140},
    {490, 360, 140, 10}, {600, 290, 450, 150}, {550, 260, 440, 120},
    {480, 300, 140, 30}, {540, 390, 130, 40}, {560, 280, 490, 200},
    {560, 300, 450, 150}, {480, 340, 190, 20}, {440, 290, 140, 20},
    {620, 280, 480, 180}, {459, 360, 100, 20}, {509, 380, 190, 40},
    {620, 290, 430, 130}, {500, 229, 330, 100}, {500, 340, 160, 40}};
std::vector<int> result_data{2, 1, 0, 2, 0, 1, 1, 1, 2, 1, 1, 1, 1,
                            0, 1, 1, 0, 0, 2, 1, 0, 0, 2, 0, 0, 1, 1, 0};

struct Forest
{
    int result_class{-1};

    NOP::SharedAttribute<int> attr0 = NOP::BuildAttribute<int>(0);
    NOP::SharedAttribute<int> attr1 = NOP::BuildAttribute<int>(0);
    NOP::SharedAttribute<int> attr2 = NOP::BuildAttribute<int>(0);
    NOP::SharedAttribute<int> attr3 = NOP::BuildAttribute<int>(0);
    NOP::SharedAttribute<int> count_setosa = NOP::BuildAttribute<int>(0);
    NOP::SharedAttribute<int> count_versicolor = NOP::BuildAttribute<int>(0);
    NOP::SharedAttribute<int> count_virginica = NOP::BuildAttribute<int>(0);
    NOP::SharedAttribute<bool> trigger_tree_0 =
        NOP::BuildAttribute<bool>(false);
    NOP::SharedAttribute<bool> trigger_tree_1 =
        NOP::BuildAttribute<bool>(false);
    NOP::Method mt_count_setosa = [&]() {
        count_setosa->SetValue(count_setosa->GetValue() + 1);
    };
    NOP::Method mt_RST_count_setosa = [&]() { count_setosa->SetValue(0); };
    NOP::Method mt_count_versicolor = [&]() {
        count_versicolor->SetValue(count_versicolor->GetValue() + 1);
    };
    NOP::Method mt_RST_count_versicolor = [&]() {
        count_versicolor->SetValue(0);
    };
    NOP::Method mt_count_virginica = [&]() {
        count_virginica->SetValue(count_virginica->GetValue() + 1);
    };
    NOP::Method mt_RST_count_virginica = [&]() {
        count_virginica->SetValue(0);
    };
    NOP::Method mtTrigger0 = [&]() { trigger_tree_0->SetValue(true); };
    NOP::Method mtRstTrigger0 = [&]() { trigger_tree_0->SetValue(false); };
    NOP::Method mtTrigger1 = [&]() { trigger_tree_1->SetValue(true); };
    NOP::Method mtRstTrigger1 = [&]() { trigger_tree_1->SetValue(false); };

    NOP::SharedRule rlTree_0_1 = NOP::BuildRule(
        NOP::BuildCondition<NOP::Conjunction>(
            NOP::BuildPremise(attr3, 75, NOP::LessEqual()),
            NOP::BuildPremise(trigger_tree_0, true, NOP::Equal())),
        NOP::BuildAction(
            NOP::BuildInstigation(mt_count_setosa, mtTrigger1, mtRstTrigger0)));
    NOP::SharedRule rlTree_0_5 = NOP::BuildRule(
        NOP::BuildCondition<NOP::Conjunction>(
            NOP::BuildPremise(attr3, 75, NOP::Greater()),
            NOP::BuildPremise(attr2, 495, NOP::LessEqual()),
            NOP::BuildPremise(attr3, 165, NOP::LessEqual()),
            NOP::BuildPremise(trigger_tree_0, true, NOP::Equal())),
        NOP::BuildAction(NOP::BuildInstigation(mt_count_versicolor, mtTrigger1,

```

```

        mtRstTrigger0)));
NOP::SharedRule rlTree_0_7 = NOP::BuildRule(
    NOP::BuildCondition<NOP::Conjunction>(
        NOP::BuildPremise(attr3, 75, NOP::Greater()),
        NOP::BuildPremise(attr2, 495, NOP::LessEqual()),
        NOP::BuildPremise(attr3, 165, NOP::Greater()),
        NOP::BuildPremise(attr1, 310, NOP::LessEqual()),
        NOP::BuildPremise(trigger_tree_0, true, NOP::Equal())),
    NOP::BuildAction(NOP::BuildInstigation(mt_count_virginica, mtTrigger1,
        mtRstTrigger0)));
NOP::SharedRule rlTree_0_8 = NOP::BuildRule(
    NOP::BuildCondition<NOP::Conjunction>(
        NOP::BuildPremise(attr3, 75, NOP::Greater()),
        NOP::BuildPremise(attr2, 495, NOP::LessEqual()),
        NOP::BuildPremise(attr3, 165, NOP::Greater()),
        NOP::BuildPremise(attr1, 310, NOP::Greater()),
        NOP::BuildPremise(trigger_tree_0, true, NOP::Equal())),
    NOP::BuildAction(NOP::BuildInstigation(mt_count_versicolor, mtTrigger1,
        mtRstTrigger0)));
NOP::SharedRule rlTree_0_11 = NOP::BuildRule(
    NOP::BuildCondition<NOP::Conjunction>(
        NOP::BuildPremise(attr3, 75, NOP::Greater()),
        NOP::BuildPremise(attr2, 495, NOP::Greater()),
        NOP::BuildPremise(attr2, 505, NOP::LessEqual()),
        NOP::BuildPremise(attr3, 185, NOP::LessEqual()),
        NOP::BuildPremise(trigger_tree_0, true, NOP::Equal())),
    NOP::BuildAction(NOP::BuildInstigation(mt_count_versicolor, mtTrigger1,
        mtRstTrigger0)));
NOP::SharedRule rlTree_0_12 = NOP::BuildRule(
    NOP::BuildCondition<NOP::Conjunction>(
        NOP::BuildPremise(attr3, 75, NOP::Greater()),
        NOP::BuildPremise(attr2, 495, NOP::Greater()),
        NOP::BuildPremise(attr2, 505, NOP::LessEqual()),
        NOP::BuildPremise(attr3, 185, NOP::Greater()),
        NOP::BuildPremise(trigger_tree_0, true, NOP::Equal())),
    NOP::BuildAction(NOP::BuildInstigation(mt_count_virginica, mtTrigger1,
        mtRstTrigger0)));
NOP::SharedRule rlTree_0_15 = NOP::BuildRule(
    NOP::BuildCondition<NOP::Conjunction>(
        NOP::BuildPremise(attr3, 75, NOP::Greater()),
        NOP::BuildPremise(attr2, 495, NOP::Greater()),
        NOP::BuildPremise(attr2, 505, NOP::Greater()),
        NOP::BuildPremise(attr1, 275, NOP::LessEqual()),
        NOP::BuildPremise(attr3, 175, NOP::LessEqual()),
        NOP::BuildPremise(trigger_tree_0, true, NOP::Equal())),
    NOP::BuildAction(NOP::BuildInstigation(mt_count_versicolor, mtTrigger1,
        mtRstTrigger0)));
NOP::SharedRule rlTree_0_16 = NOP::BuildRule(
    NOP::BuildCondition<NOP::Conjunction>(
        NOP::BuildPremise(attr3, 75, NOP::Greater()),
        NOP::BuildPremise(attr2, 495, NOP::Greater()),
        NOP::BuildPremise(attr2, 505, NOP::Greater()),
        NOP::BuildPremise(attr1, 275, NOP::LessEqual()),
        NOP::BuildPremise(attr3, 175, NOP::Greater()),
        NOP::BuildPremise(trigger_tree_0, true, NOP::Equal())),
    NOP::BuildAction(NOP::BuildInstigation(mt_count_virginica, mtTrigger1,
        mtRstTrigger0)));
NOP::SharedRule rlTree_0_14 = NOP::BuildRule(
    NOP::BuildCondition<NOP::Conjunction>(
        NOP::BuildPremise(attr3, 75, NOP::Greater()),
        NOP::BuildPremise(attr2, 495, NOP::Greater()),
        NOP::BuildPremise(attr2, 505, NOP::Greater()),
        NOP::BuildPremise(attr1, 275, NOP::Greater()),
        NOP::BuildPremise(trigger_tree_0, true, NOP::Equal())),
    NOP::BuildAction(NOP::BuildInstigation(mt_count_virginica, mtTrigger1,
        mtRstTrigger0)));
NOP::SharedRule rlEnd = NOP::BuildRule(
    NOP::BuildCondition<NOP::Single>(
        NOP::BuildPremise(trigger_tree_1, true, NOP::Equal())),
    NOP::BuildAction(NOP::BuildInstigation(
        [&] () {
            result_class =
                count_setosa->GetValue() < count_versicolor->GetValue() ? 1
    }));

```

```

        : 0;
    result_class =
        count_setosa->GetValue() < count_virginica->GetValue() &&
        count_versicolor->GetValue() <
        count_virginica->GetValue()
    ? 2
    : result_class;
},
mt_rst_count_setosa, mt_rst_count_versicolor,
mt_rst_count_virginica, mtRstTrigger1));
}

int Classify(int a0, int a1, int a2, int a3)
{
    attr0->SetValue(a0);
    attr1->SetValue(a1);
    attr2->SetValue(a2);
    attr3->SetValue(a3);
    trigger_tree_0->SetValue(true);
    return result_class;
}
};

```

CÓDIGO-FONTE DA APLICAÇÃO RANDOM FOREST EM PP NA LINGUAGEM DE PROGRAMAÇÃO C

```

int randomForestClassifier(int attr[4]);
static int sumOfResult[3];
static int result[1][3];
static void tree0(int attr[4]);
static int voting();
void tree0(int attr[4])
{
    if (attr[3] <= 75)
    {
        result[0][0] += 100;
    }
    else
    {
        if (attr[2] <= 495)
        {
            if (attr[3] <= 165)
            {
                result[0][1] += 100;
            }
            else
            {
                if (attr[1] <= 310)
                {
                    result[0][2] += 100;
                }
                else
                {
                    result[0][1] += 100;
                }
            }
        }
        else
        {
            if (attr[2] <= 505)
            {
                if (attr[3] <= 185)
                {
                    result[0][1] += 100;
                }
                else
                {
                    result[0][2] += 100;
                }
            }
            else
            {
                if (attr[1] <= 275)

```

```

    {
        if (attr[3] <= 175)
        {
            result[0][1] += 100;
        }
        else
        {
            result[0][2] += 100;
        }
    }
}
int voting()
{
    for (int t = 0; t < 1; t++)
    {
        for (int c = 0; c < 3; c++)
        {
            sumOfResult[c] += result[t][c];
        }
    }
    int maxIndex, maxValue = 0;
    for (int i = 0; i < 3; i++)
    {
        if (sumOfResult[i] >= maxValue)
        {
            maxValue = sumOfResult[i];
            maxIndex = i;
        }
    }
    return maxIndex;
}
void rst()
{
    for (int i = 0; i < 3; i++)
    {
        sumOfResult[i] = 0;
    }
    for (int t = 0; t < 1; t++)
    {
        for (int c = 0; c < 3; c++)
        {
            result[t][c] = 0;
        }
    }
}
int randomForestClassifier(int attr[4])
{
    rst();
    tree0(attr);
    return voting();
}

```

APÊNDICE J

CÓDIGO-FONTE DA APLICAÇÃO CTA EM LINGPON

```

fbe Semaphore_CTA

private Integer atSemaphoreState = 5
public Integer atSeconds = 0

private method mtResetTimer
    assignment
        this.atSeconds = 0
    end_assignment
end_method

private method mtHorizontalTrafficLightGREEN
    assignment
        this.atSemaphoreState = 0
    end_assignment
end_method

private method mtHorizontalTrafficLightYELLOW
    assignment
        this.atSemaphoreState = 1
    end_assignment
end_method

private method mtHorizontalTrafficLightRED
    assignment
        this.atSemaphoreState = 2
    end_assignment
end_method

private method mtVerticalTrafficLightGREEN
    assignment
        this.atSemaphoreState = 3
    end_assignment
end_method

private method mtVerticalTrafficLightYELLOW
    assignment
        this.atSemaphoreState = 4
    end_assignment
end_method

private method mtVerticalTrafficLightRED
    assignment
        this.atSemaphoreState = 5
    end_assignment
end_method

rule rlHorizontalTrafficLightGreen
    condition
        premise prSeconds
            this.atSeconds == 2
        end_premise
        and
        premise prSemaphoreState
            this.atSemaphoreState == 5
        end_premise
    end_condition
    action sequential
        instigation parallel
            call this.mtHorizontalTrafficLightGREEN()
        end_instigation
    end_action
end_rule

rule rlHorizontalTrafficLightYellow
    condition
        premise prSeconds2
            this.atSeconds == 40
        end_premise
        and

```

```

premise prSemaphoreState2
    this.atSemaphoreState == 0
end_premise
end_condition
action sequential
    instigation parallel
        call this.mtHorizontalTrafficLightYELLOW()
    end_instigation
end_action
end_rule

rule rlHorizontalTrafficLightRed
    condition
        premise prAtSeconds3
            this.atSeconds == 45
        end_premise
        and
        premise prSemaphoreState3
            this.atSemaphoreState == 1
        end_premise
    end_condition
    action sequential
        instigation parallel
            call this.mtHorizontalTrafficLightRED()
        end_instigation
    end_action
end_rule

rule rlVerticalTrafficLightGreen
    condition
        premise prAtSeconds4
            this.atSeconds == 47
        end_premise
        and
        premise prSemaphoreState4
            this.atSemaphoreState == 2
        end_premise
    end_condition
    action sequential
        instigation parallel
            call this.mtVerticalTrafficLightGREEN()
        end_instigation
    end_action
end_rule

rule rlVerticalTrafficLightYellow
    condition
        premise prAtSeconds5
            this.atSeconds == 85
        end_premise
        and
        premise prSemaphoreState5
            this.atSemaphoreState == 3
        end_premise
    end_condition
    action sequential
        instigation parallel
            call this.mtVerticalTrafficLightYELLOW()
        end_instigation
    end_action
end_rule

rule rlVerticalTrafficLightRed
    condition
        premise prAtSeconds6
            this.atSeconds == 90
        end_premise
        and
        premise prSemaphoreState6
            this.atSemaphoreState == 4
        end_premise
    end_condition
    action sequential
        instigation parallel
            call this.mtVerticalTrafficLightRED()
            call this.mtResetTimer()
        end_instigation
    end_action
end_rule

end_fbe

```

```

fbe Semaphore_CBCL

private Integer atSemaphoreState = 5
public Integer atSeconds = 0
    private integer atHVSS = 0
    private integer atVVSS = 0

private method mtRT
    assignment
        this.atSeconds = 0
    end_assignment
end_method

private method mtHTLG
    assignment
        this.atSemaphoreState = 0
    end_assignment
end_method

private method mtHTLY
    assignment
        this.atSemaphoreState = 1
    end_assignment
end_method

private method mtHTLR
    assignment
        this.atSemaphoreState = 2
    end_assignment
end_method

private method mtVTLG
    assignment
        this.atSemaphoreState = 3
    end_assignment
end_method

private method mtVTLY
    assignment
        this.atSemaphoreState = 4
    end_assignment
end_method

private method mtVTLR
    assignment
        this.atSemaphoreState = 5
    end_assignment
end_method

private method mtHTLGCBL
    assignment
        this.atSemaphoreState = 6
    end_assignment
end_method

private method mtHTLYCBCL
    assignment
        this.atSemaphoreState = 7
    end_assignment
end_method

private method mtVTLGCBL
    assignment
        this.atSemaphoreState = 8
    end_assignment
end_method

private method mtVTLYCBCL
    assignment
        this.atSemaphoreState = 9
    end_assignment
end_method

rule rLCBCL1
    condition
        premise prSeconds
            this.atSeconds == 2
        end_premise
        and
        premise prSemaphoreState

```

```

        this.atSemaphoreState == 5
    end_premise
end_condition
action sequential
instigation parallel
    call this.mtHTLG()
        call this.mtRT()
    end_instigation
end_action
end_rule

rule r1CBCL2
    condition
        premise prSeconds2
            this.atSeconds == 38
        end_premise
        and
        premise prSemaphoreState2
            this.atSemaphoreState == 0
        end_premise
    end_condition
    action sequential
        instigation parallel
            call this.mtHTLY()
                call this.mtRT()
        end_instigation
    end_action
end_rule

rule r1CBCL3
    condition
        premise prSecondsCBCL2
            this.atSeconds == 30
        end_premise
        and
        premise prSemaphoreStateCBCL2
            this.atSemaphoreState == 6
        end_premise
    end_condition
    action sequential
        instigation parallel
            call this.mtHTLY()
                call this.mtRT()
        end_instigation
    end_action
end_rule

rule r1CBCL4
    condition
        premise prSeconds3
            this.atSeconds == 5
        end_premise
        and
        premise prSemaphoreState3
            this.atSemaphoreState == 1
        end_premise
    end_condition
    action sequential
        instigation parallel
            call this.mtHTLR()
                call this.mtRT()
        end_instigation
    end_action
end_rule

rule r1CBCL5
    condition
        premise prSecondsCBCL3
            this.atSeconds == 6
        end_premise
        and
        premise prSemaphoreStateCBCL3
            this.atSemaphoreState == 7
        end_premise
    end_condition
    action sequential
        instigation parallel
            call this.mtHTLR()
                call this.mtRT()
        end_instigation
    end_action

```

```

end_rule

rule r1CBCL6
  condition
    premise prSeconds4
      this.atSeconds == 2
    end_premise
    and
    premise prSemaphoreState4
      this.atSemaphoreState == 2
    end_premise
  end_condition
  action sequential
    instigation parallel
      call this.mtVTLG()
      call this.mtRT()
    end_instigation
  end_action
end_rule

rule r1CBCL7
  condition
    premise prSeconds5
      this.atSeconds == 38
    end_premise
    and
    premise prSemaphoreState5
      this.atSemaphoreState == 3
    end_premise
  end_condition
  action sequential
    instigation parallel
      call this.mtVTLY()
      call this.mtRT()
    end_instigation
  end_action
end_rule

rule r1CBCL8
  condition
    premise prSecondsCBCL5
      this.atSeconds == 30
    end_premise
    and
    premise prSemaphoreStateCBCL5
      this.atSemaphoreState == 8
    end_premise
  end_condition
  action sequential
    instigation parallel
      call this.mtVTLY()
      call this.mtRT()
    end_instigation
  end_action
end_rule

rule r1CBCL9
  condition
    premise prSeconds6
      this.atSeconds == 5
    end_premise
    and
    premise prSemaphoreState6
      this.atSemaphoreState == 4
    end_premise
  end_condition
  action sequential
    instigation parallel
      call this.mtVTLR()
      call this.mtRT()
    end_instigation
  end_action
end_rule

rule r1CBCL10
  condition
    premise prSecondsCBCL6
      this.atSeconds == 6
    end_premise
    and
    premise prSemaphoreStateCBCL6

```

```

        this.atSemaphoreState == 9
    end_premise
end_condition
action sequential
    instigation parallel
        call this.mtVTLR()
            call this.mtRT()
    end_instigation
end_action
end_rule

rule rLCBCL11
    condition
        premise prSeconds7
            this.atSeconds <= 17
        end_premise
        and
        premise prSemaphoreState7
            this.atSemaphoreState == 0
        end_premise
        and
            premise prVehicleSensorState7
                this.atVVSS == 1
            end_premise
    end_condition
    action sequential
        instigation parallel
            call this.mtHTLGCBL()
        end_instigation
    end_action
end_rule

rule rLCBCL12
    condition
        premise prSeconds7Full
            this.atSeconds <= 17
        end_premise
        and
        premise prSemaphoreState7Full
            this.atSemaphoreState == 0
        end_premise
        and
            premise prVehicleSensorState7Full
                this.atVVSS == 2
            end_premise
    end_condition
    action sequential
        instigation parallel
            call this.mtHTLGCBL()
        end_instigation
    end_action
end_rule

rule rLCBCL13
    condition
        premise prSeconds8
            this.atSeconds >= 18
        end_premise
        and
            premise prSecondsSup8
                this.atSeconds < 32
            end_premise
            and
                premise prSemaphoreState8
                    this.atSemaphoreState == 0
                end_premise
                and
                    premise prVehicleSensorState8
                        this.atVVSS == 1
                    end_premise
    end_condition
    action sequential
        instigation parallel
            call this.mtHTLYCBL()
                call this.mtRT()
        end_instigation
    end_action
end_rule

rule rLCBCL14
    condition

```

```

premise prSeconds8Full
    this.atSeconds >= 18
end_premise
and
    premise prSecondsSup8Full
        this.atSeconds < 32
end_premise
and
premise prSemaphoreState8Full
    this.atSemaphoreState == 0
end_premise
and
    premise prVehicleSensorState8Full
        this.atVVSS == 2
    end_premise
end_condition
action sequential
instigation parallel
    call this.mtHTLYCBCL()
        call this.mtRT()
end_instigation
end_action
end_rule

rule rLCBCL15
condition
    premise prSeconds9
        this.atSeconds <= 17
    end_premise
and
premise prSemaphoreState9
    this.atSemaphoreState == 3
end_premise
and
    premise prVehicleSensorState9
        this.atSemaphoreState == 1
    end_premise
end_condition
action sequential
instigation parallel
    call this.mtVTLGCBCL()
end_instigation
end_action
end_rule

rule rLCBCL16
condition
    premise prSeconds9Full
        this.atSeconds <= 77
    end_premise
and
premise prSemaphoreState9Full
    this.atSemaphoreState == 3
end_premise
and
    premise prVehicleSensorState9Full
        this.atSemaphoreState == 2
    end_premise
end_condition
action sequential
instigation parallel
    call this.mtVTLGCBCL()
end_instigation
end_action
end_rule

rule rLCBCL17
condition
    premise prSeconds10
        this.atSeconds >= 18
    end_premise
and
premise prSecondsSup10
    this.atSeconds < 32
end_premise
and
    premise prSemaphoreState10
        this.atSemaphoreState == 3
    end_premise
and
premise prVehicleSensorState10

```

```

        this.atHVSS == 1
    end_premise
end_condition
action sequential
instigation parallel
    call this.mtVTLYCBCL()
    call this.mtRT()
end_instigation
end_action
end_rule

rule rLCBCL18
    condition
        premise prSeconds10Full
            this.atSeconds >= 18
        end_premise
        and
        premise prSecondsSup10Full
            this.atSeconds < 32
        end_premise
        and
            premise prSemaphoreState10Full
                this.atSemaphoreState == 3
            end_premise
            and
            premise prVehicleSensorState10Full
                this.atHVSS == 2
            end_premise
        end_condition
        action sequential
            instigation parallel
                call this.mtVTLYCBCL()
                call this.mtRT()
            end_instigation
        end_action
    end_rule

end_fbe

```

CÓDIGO-FONTE DA APLICAÇÃO CTA EM PON NO FRAMEWORK PON C++ 4.0

```

#include "libnop/framework.h"

class SemaphoreCTA
{
public:
    NOP::SharedAttribute<int> atSeconds;

private:
    NOP::SharedAttribute<int> atSemaphoreState;
    const NOP::SharedPremise prSeconds;
    const NOP::SharedPremise prSemaphoreState;
    NOP::SharedRule rlHorizontalTrafficLightGreen;
    const NOP::SharedPremise prAtSeconds3;
    const NOP::SharedPremise prSemaphoreState3;
    NOP::SharedRule rlHorizontalTrafficLightRed;
    const NOP::SharedPremise prSeconds2;
    const NOP::SharedPremise prSemaphoreState2;
    NOP::SharedRule rlHorizontalTrafficLightYellow;
    const NOP::SharedPremise prAtSeconds4;
    const NOP::SharedPremise prSemaphoreState4;
    NOP::SharedRule rlVerticalTrafficLightGreen;
    const NOP::SharedPremise prAtSeconds6;
    const NOP::SharedPremise prSemaphoreState6;
    NOP::SharedRule rlVerticalTrafficLightRed;
    const NOP::SharedPremise prAtSeconds5;
    const NOP::SharedPremise prSemaphoreState5;
    NOP::SharedRule rlVerticalTrafficLightYellow;

public:
    SemaphoreCTA();

private:

```

```

    void mtHorizontalTrafficLightGREEN();
    void mtHorizontalTrafficLightRED();
    void mtHorizontalTrafficLightYELLOW();
    void mtResetTimer();
    void mtVerticalTrafficLightGREEN();
    void mtVerticalTrafficLightRED();
    void mtVerticalTrafficLightYELLOW();
};

SemaphoreCTA::SemaphoreCTA()
: atSeconds{NOP::BuildAttribute<int>(0)},
  atSemaphoreState{NOP::BuildAttribute<int>(5)},
  prSeconds{NOP::BuildPremise<>(atSeconds, 2, NOP::Equal())},
  prSemaphoreState{
    NOP::BuildPremise<int>(atSemaphoreState, 5, NOP::Equal()),
    prAtSeconds3{NOP::BuildPremise<int>(atSeconds, 45, NOP::Equal())},
    prSemaphoreState3{
      NOP::BuildPremise<int>(atSemaphoreState, 1, NOP::Equal()),
      prSeconds2{NOP::BuildPremise<int>(atSeconds, 40, NOP::Equal())},
      prSemaphoreState2{
        NOP::BuildPremise<int>(atSemaphoreState, 0, NOP::Equal()),
        prAtSeconds4{NOP::BuildPremise<int>(atSeconds, 47, NOP::Equal())},
        prSemaphoreState4{
          NOP::BuildPremise<int>(atSemaphoreState, 2, NOP::Equal()),
          prAtSeconds6{NOP::BuildPremise<int>(atSeconds, 90, NOP::Equal())},
          prSemaphoreState6{
            NOP::BuildPremise<int>(atSemaphoreState, 4, NOP::Equal()),
            prAtSeconds5{NOP::BuildPremise<int>(atSeconds, 85, NOP::Equal())},
            prSemaphoreState5{
              NOP::BuildPremise<int>(atSemaphoreState, 3, NOP::Equal())
            }
          }
        }
      }
    }
  rlHorizontalTrafficLightGreen = NOP::BuildRule(
    NOP::BuildCondition(CONDITION(*prSeconds && *prSemaphoreState),
                         prSeconds, prSemaphoreState),
    NOP::BuildAction(NOP::BuildInstigation<NOP::Parallel>(
      METHOD(mtHorizontalTrafficLightGREEN())))
  );
  rlHorizontalTrafficLightRed = NOP::BuildRule(
    NOP::BuildCondition(CONDITION(*prAtSeconds3 && *prSemaphoreState3),
                         prAtSeconds3, prSemaphoreState3),
    NOP::BuildAction(NOP::BuildInstigation<NOP::Parallel>(
      METHOD(mtHorizontalTrafficLightRED())))
  );
  rlHorizontalTrafficLightYellow = NOP::BuildRule(
    NOP::BuildCondition(CONDITION(*prSeconds2 && *prSemaphoreState2),
                         prSeconds2, prSemaphoreState2),
    NOP::BuildAction(NOP::BuildInstigation<NOP::Parallel>(
      METHOD(mtHorizontalTrafficLightYELLOW())))
  );
  rlVerticalTrafficLightGreen = NOP::BuildRule(
    NOP::BuildCondition(CONDITION(*prAtSeconds4 && *prSemaphoreState4),
                         prAtSeconds4, prSemaphoreState4),
    NOP::BuildAction(NOP::BuildInstigation<NOP::Parallel>(
      METHOD(mtVerticalTrafficLightGREEN())))
  );
  rlVerticalTrafficLightRed = NOP::BuildRule(
    NOP::BuildCondition(CONDITION(*prAtSeconds6 && *prSemaphoreState6),
                         prAtSeconds6, prSemaphoreState6),
    NOP::BuildAction(NOP::BuildInstigation<NOP::Parallel>(
      METHOD(mtVerticalTrafficLightRED()), METHOD(mtResetTimer())))
  );
  rlVerticalTrafficLightYellow = NOP::BuildRule(
    NOP::BuildCondition(CONDITION(*prAtSeconds5 && *prSemaphoreState5),
                         prAtSeconds5, prSemaphoreState5),
    NOP::BuildAction(NOP::BuildInstigation<NOP::Parallel>(
      METHOD(mtVerticalTrafficLightYELLOW())))
  );
}

void SemaphoreCTA::mtHorizontalTrafficLightGREEN()
{
  atSemaphoreState->SetValue<NOP::Parallel>(0);
}

void SemaphoreCTA::mtHorizontalTrafficLightRED()

```

```

{
    atSemaphoreState->SetValue<NOP::Parallel>(2);
}

void SemaphoreCTA::mtHorizontalTrafficLightYELLOW()
{
    atSemaphoreState->SetValue<NOP::Parallel>(1);
}

void SemaphoreCTA::mtResetTimer() { atSeconds->SetValue<NOP::Parallel>(0); }

void SemaphoreCTA::mtVerticalTrafficLightGREEN()
{
    atSemaphoreState->SetValue<NOP::Parallel>(3);
}

void SemaphoreCTA::mtVerticalTrafficLightRED()
{
    atSemaphoreState->SetValue<NOP::Parallel>(5);
}

void SemaphoreCTA::mtVerticalTrafficLightYELLOW()
{
    atSemaphoreState->SetValue<NOP::Parallel>(4);
}

class Semaphore_CBCL
{
    public:
        NOP::SharedAttribute<int> atSeconds;

    private:
        NOP::SharedAttribute<int> atHVSS;
        NOP::SharedAttribute<int> atSemaphoreState;
        NOP::SharedAttribute<int> atVVSS;
        const NOP::SharedPremise prSeconds;
        const NOP::SharedPremise prSemaphoreState;
        NOP::SharedRule rLCBCL1;
        const NOP::SharedPremise prSecondsCBCL6;
        const NOP::SharedPremise prSemaphoreStateCBCL6;
        NOP::SharedRule rLCBCL10;
        const NOP::SharedPremise prSeconds7;
        const NOP::SharedPremise prSemaphoreState7;
        const NOP::SharedPremise prVehicleSensorState7;
        NOP::SharedRule rLCBCL11;
        const NOP::SharedPremise prSeconds7Full;
        const NOP::SharedPremise prSemaphoreState7Full;
        const NOP::SharedPremise prVehicleSensorState7Full;
        NOP::SharedRule rLCBCL12;
        const NOP::SharedPremise prSeconds8;
        const NOP::SharedPremise prSecondsSup8;
        const NOP::SharedPremise prSemaphoreState8;
        const NOP::SharedPremise prVehicleSensorState8;
        NOP::SharedRule rLCBCL13;
        const NOP::SharedPremise prSeconds8Full;
        const NOP::SharedPremise prSecondsSup8Full;
        const NOP::SharedPremise prSemaphoreState8Full;
        const NOP::SharedPremise prVehicleSensorState8Full;
        NOP::SharedRule rLCBCL14;
        const NOP::SharedPremise prSeconds9;
        const NOP::SharedPremise prSemaphoreState9;
        const NOP::SharedPremise prVehicleSensorState9;
        NOP::SharedRule rLCBCL15;
        const NOP::SharedPremise prSeconds9Full;
        const NOP::SharedPremise prSemaphoreState9Full;
        const NOP::SharedPremise prVehicleSensorState9Full;
        NOP::SharedRule rLCBCL16;
        const NOP::SharedPremise prSeconds10;
        const NOP::SharedPremise prSecondsSup10;
        const NOP::SharedPremise prSemaphoreState10;
        const NOP::SharedPremise prVehicleSensorState10;
        NOP::SharedRule rLCBCL17;
        const NOP::SharedPremise prSeconds10Full;
        const NOP::SharedPremise prSecondsSup10Full;
        const NOP::SharedPremise prSemaphoreState10Full;
        const NOP::SharedPremise prVehicleSensorState10Full;
        NOP::SharedRule rLCBCL18;
        const NOP::SharedPremise prSeconds2;
        const NOP::SharedPremise prSemaphoreState2;
        NOP::SharedRule rLCBCL2;
        const NOP::SharedPremise prSecondsCBCL2;
}

```

```

const NOP::SharedPremise prSemaphoreStateCBCL2;
NOP::SharedRule r1CBCL3;
const NOP::SharedPremise prSeconds3;
const NOP::SharedPremise prSemaphoreState3;
NOP::SharedRule r1CBCL4;
const NOP::SharedPremise prSecondsCBCL3;
const NOP::SharedPremise prSemaphoreStateCBCL3;
NOP::SharedRule r1CBCL5;
const NOP::SharedPremise prSeconds4;
const NOP::SharedPremise prSemaphoreState4;
NOP::SharedRule r1CBCL6;
const NOP::SharedPremise prSeconds5;
const NOP::SharedPremise prSemaphoreState5;
NOP::SharedRule r1CBCL7;
const NOP::SharedPremise prSecondsCBCL5;
const NOP::SharedPremise prSemaphoreStateCBCL5;
NOP::SharedRule r1CBCL8;
const NOP::SharedPremise prSeconds6;
const NOP::SharedPremise prSemaphoreState6;
NOP::SharedRule r1CBCL9;

public:
Semaphore_CBCL();

private:
void mtHTLG();
void mtHTLGBCL();
void mtHTLR();
void mtHTLY();
void mtHTLYCBCL();
void mtRT();
void mtVTLG();
void mtVTLGBCL();
void mtVTLR();
void mtVTLY();
void mtVTLYCBCL();
};

Semaphore_CBCL::Semaphore_CBCL()
: atSeconds{NOP::BuildAttribute<int>(0)},
atHVSS{NOP::BuildAttribute<int>(0)},
atSemaphoreState{NOP::BuildAttribute<int>(5)},
atVVSS{NOP::BuildAttribute<int>(0)},
prSeconds{NOP::BuildPremise<int>(atSeconds, 2, NOP::Equal())},
prSemaphoreState{
    NOP::BuildPremise<int>(atSemaphoreState, 5, NOP::Equal()),
    prSecondsCBCL6{NOP::BuildPremise<int>(atSeconds, 6, NOP::Equal())},
    prSemaphoreStateCBCL6{
        NOP::BuildPremise<int>(atSemaphoreState, 9, NOP::Equal()),
        prSeconds7{NOP::BuildPremise<int>(atSeconds, 17, NOP::LessEqual())},
        prSemaphoreState7{
            NOP::BuildPremise<int>(atSemaphoreState, 0, NOP::Equal()),
            prVehicleSensorState7{NOP::BuildPremise<int>(atVVSS, 1, NOP::Equal())},
            prSeconds7Full{NOP::BuildPremise<int>(atSeconds, 17, NOP::LessEqual())},
            prSemaphoreState7Full{
                NOP::BuildPremise<int>(atSemaphoreState, 0, NOP::Equal()),
                prVehicleSensorState7Full{
                    NOP::BuildPremise<int>(atVVSS, 2, NOP::Equal()),
                    prSeconds8{NOP::BuildPremise<int>(atSeconds, 18, NOP::GreaterEqual())},
                    prSecondsSup8{NOP::BuildPremise<int>(atSeconds, 32, NOP::Less())},
                    prSemaphoreState8{
                        NOP::BuildPremise<int>(atSemaphoreState, 0, NOP::Equal()),
                        prVehicleSensorState8{NOP::BuildPremise<int>(atVVSS, 1, NOP::Equal())},
                        prSeconds8Full{
                            NOP::BuildPremise<int>(atSeconds, 18, NOP::GreaterEqual()),
                            prSecondsSup8Full{NOP::BuildPremise<int>(atSeconds, 32, NOP::Less())},
                            prSemaphoreState8Full{
                                NOP::BuildPremise<int>(atSemaphoreState, 0, NOP::Equal()),
                                prVehicleSensorState8Full{
                                    NOP::BuildPremise<int>(atVVSS, 2, NOP::Equal()),
                                    prSeconds9{NOP::BuildPremise<int>(atSeconds, 17, NOP::LessEqual())},
                                    prSemaphoreState9{
                                        NOP::BuildPremise<int>(atSemaphoreState, 3, NOP::Equal()),
                                        prVehicleSensorState9{
                                            NOP::BuildPremise<int>(atSemaphoreState, 1, NOP::Equal()),
                                            prSeconds9Full{NOP::BuildPremise<int>(atSeconds, 77, NOP::LessEqual())},
                                            prSemaphoreState9Full{
                                                NOP::BuildPremise<int>(atSemaphoreState, 3, NOP::Equal()),
                                                prVehicleSensorState9Full{
                                                    NOP::BuildPremise<int>(atSemaphoreState, 2, NOP::Equal()),
                                                    prSeconds10{NOP::BuildPremise<int>(atSeconds, 18, NOP::GreaterEqual())},
                                                }
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}

```

```

prSecondsSup1{NOP::BuildPremise<int>(atSeconds, 32, NOP::Less())},
prSemaphoreState10{
    NOP::BuildPremise<int>(atSemaphoreState, 3, NOP::Equal()),
    prVehicleSensorState10{NOP::BuildPremise<int>(atHVSS, 1, NOP::Equal())},
    prSeconds10Full{
        NOP::BuildPremise<int>(atSeconds, 18, NOP::GreaterEqual()),
        prSecondsSup10Full{NOP::BuildPremise<int>(atSeconds, 32, NOP::Less())},
        prSemaphoreState10Full{
            NOP::BuildPremise<int>(atSemaphoreState, 3, NOP::Equal()),
            prVehicleSensorState10Full{
                NOP::BuildPremise<int>(atHVSS, 2, NOP::Equal()),
                prSeconds2{NOP::BuildPremise<int>(atSeconds, 38, NOP::Equal())},
                prSemaphoreState2{
                    NOP::BuildPremise<int>(atSemaphoreState, 0, NOP::Equal()),
                    prSecondsCBCL2{NOP::BuildPremise<int>(atSeconds, 30, NOP::Equal())},
                    prSemaphoreStateCBCL2{
                        NOP::BuildPremise<int>(atSemaphoreState, 6, NOP::Equal()),
                        prSeconds3{NOP::BuildPremise<int>(atSeconds, 5, NOP::Equal())},
                        prSemaphoreState3{
                            NOP::BuildPremise<int>(atSemaphoreState, 1, NOP::Equal()),
                            prSecondsCBCL3{NOP::BuildPremise<int>(atSeconds, 6, NOP::Equal())},
                            prSemaphoreStateCBCL3{
                                NOP::BuildPremise<int>(atSemaphoreState, 7, NOP::Equal()),
                                prSeconds4{NOP::BuildPremise<int>(atSeconds, 2, NOP::Equal())},
                                prSemaphoreState4{
                                    NOP::BuildPremise<int>(atSemaphoreState, 2, NOP::Equal()),
                                    prSeconds5{NOP::BuildPremise<int>(atSeconds, 38, NOP::Equal())},
                                    prSemaphoreState5{
                                        NOP::BuildPremise<int>(atSemaphoreState, 3, NOP::Equal()),
                                        prSecondsCBCL5{NOP::BuildPremise<int>(atSeconds, 30, NOP::Equal())},
                                        prSemaphoreStateCBCL5{
                                            NOP::BuildPremise<int>(atSemaphoreState, 8, NOP::Equal()),
                                            prSeconds6{NOP::BuildPremise<int>(atSeconds, 5, NOP::Equal())},
                                            prSemaphoreState6{
                                                NOP::BuildPremise<int>(atSemaphoreState, 4, NOP::Equal())
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
};

rlCBCL1 = NOP::BuildRule(
    NOP::BuildCondition(CONDITION(*prSeconds && *prSemaphoreState),
        prSeconds, prSemaphoreState),
    NOP::BuildAction(NOP::BuildInstigation<NOP::Parallel>(METHOD(mtHTLG());,
        METHOD(mtRT();)))
);

rlCBCL10 =
    NOP::BuildRule(NOP::BuildCondition(
        CONDITION(*prSecondsCBCL6 && *prSemaphoreStateCBCL6),
        prSecondsCBCL6, prSemaphoreStateCBCL6),
        NOP::BuildAction(NOP::BuildInstigation<NOP::Parallel>(
            METHOD(mtVTLR()), METHOD(mtRT();)))
    );
;

rlCBCL11 = NOP::BuildRule(
    NOP::BuildCondition(CONDITION(*prSeconds7 && *prSemaphoreState7 &&
        *prVehicleSensorState7),
        prSeconds7, prSemaphoreState7,
        prVehicleSensorState7),
    NOP::BuildAction(
        NOP::BuildInstigation<NOP::Parallel>(METHOD(mtHTLGCBCL();)))
);

rlCBCL12 = NOP::BuildRule(
    NOP::BuildCondition(
        CONDITION(*prSeconds7Full && *prSemaphoreState7Full &&
            *prVehicleSensorState7Full),
        prSeconds7Full, prSemaphoreState7Full, prVehicleSensorState7Full),
    NOP::BuildAction(
        NOP::BuildInstigation<NOP::Parallel>(METHOD(mtHTLGCBCL();)))
);

rlCBCL13 = NOP::BuildRule(
    NOP::BuildCondition(
        CONDITION(*prSeconds8 && *prSecondsSup8 && *prSemaphoreState8 &&
            *prVehicleSensorState8),
        prSeconds8, prSecondsSup8, prSemaphoreState8,
        prVehicleSensorState8),
    NOP::BuildAction(NOP::BuildInstigation<NOP::Parallel>(
        METHOD(mtHTLYCBCL()), METHOD(mtRT();)))
);

rlCBCL14 = NOP::BuildRule(
    NOP::BuildCondition(

```



```

);
rlCBCL7 = NOP::BuildRule(
    NOP::BuildCondition(CONDITION(*prSeconds5 && *prSemaphoreState5),
                         prSeconds5, prSemaphoreState5),
    NOP::BuildAction(NOP::BuildInstigation<NOP::Parallel>(METHOD(mtVTLY()),,
                    METHOD(mtRT()))))

);
rlCBCL8 =
    NOP::BuildRule(NOP::BuildCondition(
        CONDITION(*prSecondsCBCL5 && *prSemaphoreStateCBCL5),
        prSecondsCBCL5, prSemaphoreStateCBCL5),
    NOP::BuildAction(NOP::BuildInstigation<NOP::Parallel>(
        METHOD(mtVTLY()), METHOD(mtRT()))))

);
rlCBCL9 = NOP::BuildRule(
    NOP::BuildCondition(CONDITION(*prSeconds6 && *prSemaphoreState6),
                         prSeconds6, prSemaphoreState6),
    NOP::BuildAction(NOP::BuildInstigation<NOP::Parallel>(METHOD(mtVTLR()),,
                    METHOD(mtRT()))))

);
}

void Semaphore_CBCL::mtHTLG() { atSemaphoreState->SetValue<NOP::Parallel>(0); }

void Semaphore_CBCL::mtHTLGBCL()
{
    atSemaphoreState->SetValue<NOP::Parallel>(6);
}

void Semaphore_CBCL::mtHTLR() { atSemaphoreState->SetValue<NOP::Parallel>(2); }

void Semaphore_CBCL::mtHTLY() { atSemaphoreState->SetValue<NOP::Parallel>(1); }

void Semaphore_CBCL::mtHTLYCBCL()
{
    atSemaphoreState->SetValue<NOP::Parallel>(7);
}

void Semaphore_CBCL::mtRT() { atSeconds->SetValue<NOP::Parallel>(0); }

void Semaphore_CBCL::mtVTLG() { atSemaphoreState->SetValue<NOP::Parallel>(3); }

void Semaphore_CBCL::mtVTLGBCL()
{
    atSemaphoreState->SetValue<NOP::Parallel>(8);
}

void Semaphore_CBCL::mtVTLR() { atSemaphoreState->SetValue<NOP::Parallel>(5); }

void Semaphore_CBCL::mtVTLY() { atSemaphoreState->SetValue<NOP::Parallel>(4); }

void Semaphore_CBCL::mtVTLYCBCL()
{
    atSemaphoreState->SetValue<NOP::Parallel>(9);
}

```

APÊNDICE K

Avaliação do Framework PON C++ 4.0

Gostaria de receber feedback referente a utilização dos frameworks do PON, de modo a permitir comparações Frameworks PON C++ 2.0, que será utilizado na composição da minha dissertação. Suas respostas podem ser baseadas tanto na sua experiência pessoal utilizando os frameworks ou de conhecimento adquirido através de apresentações e leitura de materiais como artigos, dissertações e teses.

*Obrigatório

1. Assinale as versões de Framework (ou NOPL) do PON que você já utilizou para o desenvolvimento de programas *

Marque todas que se aplicam.

- C++ Prototipal
- C++ 1.0
- C++ 2.0
- C++ 3.0
- C++ 4.0
- Java / C#
- C# IoT
- Elixir/Erlang
- Akka
- JuNOC++
- LingPON
- NOPLite

Outro:

2. Como você avalia o Framework PON C++ 2.0 nas seguintes características *

Marcar apenas uma oval por linha.

	Fraco	Moderado	Satisfatório	Muito bom	Excelente
Facilidade de aprendizado	<input type="radio"/>				
Facilidade de uso	<input type="radio"/>				
Verbosidade	<input type="radio"/>				
Versatilidade	<input type="radio"/>				
Desempenho	<input type="radio"/>				
Aderência aos fundamentos do PON	<input type="radio"/>				

3. Como você avalia o Framework PON C++ 4.0 nas seguintes características *

Marcar apenas uma oval por linha.

	Fraco	Moderado	Satisfatório	Muito bom	Excelente
Facilidade de aprendizado	<input type="radio"/>				
Facilidade de uso	<input type="radio"/>				
Verbosidade	<input type="radio"/>				
Versatilidade	<input type="radio"/>				
Desempenho	<input type="radio"/>				
Aderência aos fundamentos do PON	<input type="radio"/>				

4. Numa escala de 0 a 5, o quanto você considera que o Framework PON C++ 4.0 apresenta melhorias sobre o Framework PON C++ 2.0, sendo que 0 significa que não houve melhoria. *

Marcar apenas uma oval.

0 1 2 3 4 5

5. Numa escala de 0 a 5, o quanto você considera viável dar manutenção ao código apresentado pelo Framework PON C++ 4.0, sendo que 0 significa que não seria viável. Nessa avaliação considere a complexidade e quantidade de código apresentada. *

Marcar apenas uma oval.

6. Qual a probabilidade de você escolher utilizar os seguintes Frameworks no desenvolvimento de uma aplicação em PON *

Marcar apenas uma oval por linha.

	Improvável	Pouco provável	Provável	Muito provável
C++ Prototipal	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
C++ 1.0	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
C++ 2.0	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
C++ 3.0	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
C++ 4.0	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Java / C#	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
C# IoT	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Elixir/Erlang	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Akka	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
JuNOC++	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
LingPON	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
NOPLite	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

7. Considerando que o Framework PON C++ 4.0 ainda se encontra em desenvolvimento ativo, possuí alguma sugestão para o desenvolvimento do mesmo? Considere aqui sugestões que de forma geral podem melhorar a experiência de desenvolvimento de programas com os frameworks em PON, sejam estas questões de arquitetura, interface e até mesmo de compilação.

Este conteúdo não foi criado nem aprovado pelo Google.

Google Formulários