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

FABIO NEGRINI

TECNOLOGIA NOPL ERLANG-ELIXIR – PARADIGMA ORIENTADO A NOTIFICAÇÕES VIA UMA ABORDAGEM ORIENTADA A MICROATORES ASSÍNCRONOS

DISSERTAÇÃO

CURITIBA
2019

FABIO NEGRINI

TECNOLOGIA NOPL ERLANG-ELIXIR – PARADIGMA ORIENTADO A NOTIFICAÇÕES VIA UMA ABORDAGEM ORIENTADA A MICROATORES ASSÍNCRONOS

Dissertação de Mestrado apresentado ao Programa de Pós-Graduação em Engenharia Elétrica e informática Industrial da Universidade Tecnológica Federal do Paraná – Área de concentração: Engenharia de Computação

Orientador: Prof. Dr. Jean Marcelo Simão
Coorientador: Prof. Dr. Robson Ribeiro Linhares

CURITIBA

2019

Dados Internacionais de Catalogação na Publicação

Negrini, Fabio

Tecnologia NOPL Erlang-Elixir [recurso eletrônico] : paradigma orientado a notificações via uma abordagem orientada a microatores assíncronos / Fabio Negrini.-- 2019.

1 arquivo texto (298 f.) : PDF ; 13,3 MB.

Modo de acesso: World Wide Web

Título extraído da tela de título (visualizado em 18 dez. 2019)

Texto em português com resumo em inglês

Dissertação (Mestrado em Engenharia Elétrica e Informática Industrial) - Universidade Tecnológica Federal do Paraná, Programa de Pós-Graduação em Engenharia Elétrica e informática Industrial, Curitiba, 2019

Bibliografia: p. 206-212.

1. Engenharia elétrica - Dissertações. 2. Microprocessadores. 3. Paradigma orientado a notificações. 4. Linguagem de programação (Computadores). 5. Teoria das filas. 6. Processamento paralelo (Computadores). 7. Programação paralela (Computação). I. Simão, Jean Marcelo. II. Linhares, Robson Ribeiro. III. Universidade Tecnológica Federal do Paraná. Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial. IV. Título.

CDD: Ed. 23 – 621.3

Biblioteca Central da UTFPR, Câmpus Curitiba

Bibliotecário: Adriano Lopes CRB-9/1429

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

TECNOLOGIA NOPL ERLANG-ELIXIR – PARADIGMA ORIENTADO A NOTIFICAÇÕES VIA UMA ABORDAGEM ORIENTADA A MICROATORES ASSÍNCRONOS

por

Fabio Negrini

Orientador: Prof. Dr. Jean Marcelo Simão

Coorientador: Prof. Dr. Robson Ribeiro Linhares

Esta dissertação foi apresentada como requisito parcial à obtenção do grau de MESTRE EM CIÊNCIAS – Área de concentração: **Engenharia de Computação**, pelo Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial – CPGEI - da Universidade Tecnológica Federal do Paraná – UTFPR, às **09:00h** do dia **25 de outubro de 2019**. O trabalho foi aprovado pela Banca Examinadora, composta pelos doutores:

Paulo Cézar Stadzisz (PRESIDENTE – UTFPR)	Carlos Alberto Maziero (UFPR)
Adriano Francisco Ronszcka GranMoney	Adolfo Gustavo Serra Seca Neto (UTFPR)

AGRADECIMENTO

Agradecer a todas as pessoas que tornaram este trabalho possível não caberia em uma mera menção dedicatória. De qualquer forma, alguns nomes se destacam em meio à multidão de pessoas que contribuíram direta ou indiretamente para este trabalho. Este agradecimento cita as principais, mas de forma alguma se limita a apenas elas.

Inicio meus agradecimentos aos docentes do CPGEI, em particular meus orientadores, Jean Marcelo Simão e Robson Ribeiro Linhares, que não pouparam esforços para formar meu pensamento *stricto sensu* durante esta jornada. Contudo, há outros professores de extrema importância no alcance deste objetivo, como os professores Paulo Cezar Stadzisz, João Alberto Fabro, Hugo Vieira Neto e Adolfo Gustavo Serra Seca Neto. Agradeço também aos membros da banca Paulo Cézar Stadzisz, Carlos Alberto Maziero e Adriano Francisco Ronszcka pela disponibilidade em avaliar este trabalho.

Nesta caminhada alguns (então) discentes *stricto sensu* (mestrandos e doutorandos) também deixaram suas marcas, seja pelo apoio científico-intelectual, seja pela amizade e conselhos, e que merecem meu agradecimento. Cito, principalmente, Adriano Francisco Ronszcka, Leonardo Faix Pordeus e Fernando Schütz que tiveram contribuição crucial para os resultados deste trabalho.

Agradeço também a toda minha família, em especial minha esposa que, além de me dar todo o apoio de forma indireta, com carinho e compreensão para que pudesse concluir este trabalho, também me apoiou de forma direta com conselhos, críticas e revisões de forma a transformar este trabalho em algo realmente legível.

Por fim e mais importante, agradeço a Deus, meu criador e redentor, que me deu toda a capacidade física e intelectual e arquitetou todas as pessoas e circunstâncias com grande maestria, colocando cada elemento nesta caminhada no momento certo para que este trabalho pudesse ser feito.

RESUMO

NEGRINI, Fabio. **Tecnologia NOPL Erlang-Elixir – Paradigma Orientado a Notificações via uma abordagem orientada a microatores assíncronos**. 2019. 298 f. Dissertação de mestrado. Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI). Universidade Tecnológica Federal do Paraná (UTFPR). Curitiba, 2019.

A arquitetura *multicore* consiste na implementação de múltiplos núcleos em uma mesma pastilha, sendo uma alternativa à estagnação das taxas de velocidade dos microprocessadores. Em tese, com o aumento do número de unidades de processamento paralelas, permite-se melhorar o desempenho de execução. Entretanto, na prática, isto depende de softwares desenvolvidos especificamente para explorarem esta característica. Este tipo de desenvolvimento de software traz maior dificuldade em relação à usual programação sequencial. Neste contexto, apresenta-se uma técnica alternativa de desenvolvimento de software chamada de Paradigma Orientado a Notificações (PON), a qual consiste em entidades notificantes sucintas e colaborativas. O PON proporciona desacoplamento natural entre suas entidades, o que beneficia o uso de paralelismo/distribuição. Neste ambiente foi previamente proposta a NOPL, uma linguagem de programação de alto nível para o PON e uma respectiva tecnologia de compilação que auxilia a composição de compiladores. Esta tecnologia NOPL, entretanto, não é específica para uma plataforma e precisa ser devidamente aplicada em cada plataforma alvo. Isto posto, esta dissertação tem por objetivo apresentar uma solução PON baseada na tecnologia NOPL para ambiente *multicore*. Para isto, é proposto primeiramente um *framework* na linguagem Elixir, que reproduz cada elemento do paradigma PON em microatores no ambiente Erlang. Subsequentemente, há o desenvolvimento integrativo no tocante à tecnologia NOPL. Com esta sinergia de tecnologias, é almejado o aproveitamento da concorrência e balanceamento da arquitetura Erlang aliado ao desacoplamento implícito das entidades PON e à programação de alto nível disponibilizada pela NOPL. Efetivamente, experimentos realizados nesta tecnologia proposta apresentam considerável melhoria de desempenho à medida que se aumenta o número de núcleos, os quais se mantêm com taxas de ocupação apropriadas e balanceadas. Como resultado, tem-se uma programação *multicore* em alto nível que aproveita o paralelismo de núcleos de processamento de maneira transparente para o desenvolvedor.

Palavras-chave: Paradigma Orientado a Notificações. *Multicore.* Software paralelo. Balanceamento de cargas de trabalho. NOPL. Erlang. Elixir.

ABSTRACT

NEGRINI, Fabio. **NOPL Erlang-Elixir Technology – Notification-Oriented Paradigm via an asynchronous micro-actor-oriented approach**. 2019. 298 p. Text of individual dissertation (Master of Science in Electrical Engineering and Industrial Informatics). Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI). Universidade Tecnológica Federal do Paraná (UTFPR). Curitiba, 2019.

The multicore architecture consists of the implementation of multiple cores in the same chip, which is becoming an alternative to the stagnation of microprocessor speed rates. In theory, by increasing the number of parallel processing units, it is possible to improve execution performance. However, in practice, this depends on software designed specifically to explore such feature. This type of *software* development brings more difficulty when compared to the usual sequential programming. In this context, an alternative *software* development technique called Notification Oriented Paradigm (NOP) was proposed, which is based on minimal and collaborative notifying entities. NOP provides natural decoupling among its entities, which benefits the use of parallelism and distribution. In this environment, the NOPL was previously proposed, which is a high-level programming language for NOP and a corresponding compilation technology that assists compiler composition. This NOPL technology, however, is not platform-specific and needs to be properly applied to each target platform. That said, this M.Sc. dissertation aims to present a NOP solution based on NOPL technology for multicore environments. In order to achieve this objective, an Elixir framework is first proposed, which reproduces each element of the NOP as an Erlang micro-actor. Subsequently, this framework is integrated to the NOPL technology. With this synergy of technologies, it is aimed to join the concurrency and balance of Erlang architecture with the implicit decoupling of NOP entities and the high-level programming provided by NOPL. Indeed, experiments performed on this proposed technology show considerable performance improvement as the number of cores increases, by keeping appropriate and balanced load percentages. As a result, the object of this research presents itself as a high-level multicore development platform that takes advantage of parallel processing cores in a transparent way for the developer.

Keywords: Notification-oriented paradigm. Multicore. Parallel software. Workload balancing. NOPL. Erlang. Elixir.

LISTA DE FIGURAS

Figura 1. Processador *Hyperthreading*.....32
Figura 2. Arquitetura *multicore*.....33
Figura 3. Ilustração dos modelos de consistência.....39
Figura 4. Estados de uma tarefa em um sistema de tempo compartilhado.....43
Figura 5. Relacionamento entre Processo, Tarefa e Thread.....45
Figura 6. Exemplo de *speedup* máximo teórico de um algoritmo.....47
Figura 7. Dois processos acessando a memória compartilhada simultaneamente...49
Figura 8. Solução de Peterson para exclusão mútua em C.....51
Figura 9. Diagrama para representação de um ator.....56
Figura 10. Envio de mensagens entre atores sem garantia de precedência.....57
Figura 11. Processos Erlang sendo executados pela VM.....63
Figura 12. Informações de um processo Erlang, com destaque para o controle de estado.....64
Figura 13. Comunicação entre processos.....65
Figura 14. Taxonomia de Paradigmas de programação.....70
Figura 15. Classificação dos paradigmas de programação.....72
Figura 16. Relação entre o PON, o PI e o PD.....77
Figura 17. Integração da entidade de manufatura Kuka386 a um sistema computacional.....78
Figura 18. Exemplo de uma regra PON.....79
Figura 19. Colaboração por notificações.....81
Figura 20. Relação entre objetos PON.....83
Figura 21. Etapas de Compilação de NOPL.....90
Figura 22. Tempos de execução (em milissegundos) do treinamento de RNA MLP com método BP, utilizando o modelo computacional NeuroPON em materializações de software do PON, para a base de dados da flor de Iris.....92
Figura 23. Visão geral estrutural do MCPON.....94
Figura 24. Visão geral das etapas do MCPON.....95
Figura 25. Diagrama de classes da representação do Grafo PON.....97
Figura 26. Representação ilustrativa de um de Grafo PON.....99
Figura 27. Modelo genérico de ator com separação entre estado e processo.....103

Figura 28. Diagrama de elementos PON modelados em atores 104
Figura 29. Modelo alto nível de ator *FBE* 106
Figura 30. Modelo alto nível de ator *Attribute* 108
Figura 31. Modelo alto nível de ator *Method* 109
Figura 32. Modelo alto nível de ator *Premise* 111
Figura 33. Modelo alto nível de ator *Subcondition* 113
Figura 34. Modelo alto nível de ator *Condition* 116
Figura 35. Modelo alto nível de ator *Rule* 118
Figura 36. Modelo alto nível de ator Action 120
Figura 37. Modelo alto nível de ator *Instigation* 121
Figura 38. Modelagem UML dos elementos do paradigma PON 124
Figura 39. Modelagem UML para *ElementBase* 125
Figura 40. Modelagem UML para o elemento PON *FBE* 126
Figura 41. Modelagem UML para o elemento PON *Attribute* 127
Figura 42. Modelagem UML do elemento *LogicElement* 128
Figura 43. Modelagem UML do elemento *Premise* 129
Figura 44. Modelagem UML do elemento PON *Subcondition* 129
Figura 45. Modelagem UML do elemento *Condition* 130
Figura 46. Modelagem UML do elemento *Rule* 131
Figura 47. Modelagem UML do elemento PON *Action* 131
Figura 48. Modelagem do elemento PON Instigation 132
Figura 49. Modelagem UML dos elementos auxiliares 132
Figura 50. Modelagem UML da classe Application 133
Figura 51. Modelagem UML da classe *ServiceBase* 134
Figura 52. Modelagem UML da classe *ServiceFBE* responsável por supervisionar objetos *FBE* 135
Figura 53. Modelagem UML da classe *ServiceAttribute* responsável por supervisionar objetos *Attribute* 135
Figura 54. Modelagem UML da classe *ServicePremise* responsável por supervisionar objetos *Premise* 135
Figura 55. Modelagem UML da classe *ServiceSubcondition* responsável por supervisionar objetos *Subcondition* 135
Figura 56. Modelagem UML da classe *ServiceCondition* responsável por supervisionar objetos *Condition* 136

Figura 57. Modelagem UML da classe *ServiceRule* responsável por supervisionar objetos *Rule* 136
Figura 58. Modelagem UML da classe *ServiceAction* responsável por supervisionar objetos *Action* 136
Figura 59. Modelagem UML da classe *ServiceInstigation* responsável por supervisionar objetos *Instigation* 136
Figura 60. Modelagem UML simplificada para desenvolvimento do *framework* NOP Elixir 137
Figura 61. Diagrama UML dos elementos complementares adaptado para o *framework* NOP Elixir 139
Figura 62. Estrutura de arquivos do *framework* NOP Elixir 140
Figura 63. Estrutura de aplicações supervisoras em uma aplicação NOP Elixir 146
Figura 64. Log de execução de um programa PON em *Framework* NOP Elixir 148
Figura 65. Demonstração alto nível da melhoria: atribuição por expressão 149
Figura 66. Demonstração alto nível da melhoria: leitura múltipla de atributos 150
Figura 67. Demonstração alto nível da melhoria: atribuição com valores encadeados 151
Figura 68. Fluxograma das etapas de compilação 156
Figura 69. Fluxograma para tradução do Grafo PON 159
Figura 70. Código para criação de definição de FBE 160
Figura 71. Materialização das contribuições para o NOPL 167
Figura 72. Exemplo de saída do comando top para acompanhamento de taxa de ocupação de um ambiente monoprocessado 172
Figura 73. Exemplo de saída do comando top para acompanhamento de ocupação de memória em um ambiente monoprocessado 173
Figura 74. Ambiente de simulação CTA: 10 ruas verticais - 10 ruas horizontais 174
Figura 75. Diagrama de estados da aplicação "Controle de Semáforos" 175
Figura 76. Taxa de ocupação do núcleo em ambiente VM02 179
Figura 77. Taxa de média ocupação por núcleo em ambiente VM04 180
Figura 78. Taxa de média ocupação por núcleo em ambiente VM08 181
Figura 79. Taxa de média ocupação por núcleo em ambiente VM16 182
Figura 80.Consumo de memória do sistema operacional médio durante simulações 183
Figura 81. Tempo médio de execução por ambiente virtualizado 184

Figura 82. Diagrama de estados da estratégia CBCF 185
Figura 83. Taxa de ocupação do núcleo em ambiente VM02 188
Figura 84. Taxa de média ocupação por núcleo em ambiente VM04 189
Figura 85. Taxa de média ocupação por núcleo em ambiente VM08 190
Figura 86. Taxa de média ocupação por núcleo em ambiente VM16 191
Figura 87. Consumo de memória do sistema operacional médio durante simulações 192
Figura 88. Tempo médio de execução por ambiente virtualizado 193
Figura 89. Comparativo de fator tempo sobre total de notificações 194
Figura 90. Diferença de fator tempo sobre total de notificações 194
Figura 91. Comparativo de variação dos estudos entre ambientes virtuais 195
Figura 92. Percentual de variação de tempo da estratégia CBCF em função da estratégia CTA 196
Figura 93. Representação gráfica dos tempos médios de execução NeuroPON de uma RNA MLP para a função XOR em Erlang 214
Figura 94. Histograma de tempos de execução por ambiente virtual para caso de estudo I 216
Figura 95. Histograma de tempos de execução por ambiente virtual para caso de estudo II 218
Figura 96. Comparativo entre NOPL Erlang-Elixir e NOPL-Namespaces para estratégia de controle independente 269
Figura 97. Comparativo de desempenho entre *Framework* NOP Elixir e programação Elixir especialista para estratégia de tráfego independente 271
Figura 98. Comparativo de desempenho entre *Framework* NOP Elixir e programação Elixir especialista para estratégia de congestionamento facilitado 273
Figura 99. Comparativo do *Framework* NOP Elixir com (NEGRINI, PORDEUS e SIMÃO, 2019) na estratégia de controle de tráfego independente 275
Figura 100. Comparativo do *Framework* NOP Elixir com (NEGRINI, PORDEUS e SIMÃO, 2019) na estratégia de controle de congestionamento facilitado 275

LISTA DE TABELAS

Tabela 1. Arquiteturas *multicore* de propósito geral 40
Tabela 2. Arquiteturas *multicore* de alto desempenho 41
Tabela 3. Aderência do elemento *FBE* em modelo de atores 106
Tabela 4. Aderência do elemento *Attribute* em modelo de atores 108
Tabela 5. Aderência do elemento *Method* em modelo de atores 110
Tabela 6. Aderência do elemento *Premise* em modelo de Atores 112
Tabela 7. Aderência do elemento *Subcondition* em modelo de Atores 114
Tabela 8. Aderência do elemento *Condition* em modelo de Atores 116
Tabela 9. Aderência do elemento *Rule* em modelo de Atores 118
Tabela 10. Aderência do elemento *Action* em modelo de Atores 120
Tabela 11. Aderência do elemento *Instigation* em modelo de Atores 122
Tabela 12. Organização dos módulos desenvolvidos por pasta e arquivo 140
Tabela 13. Detalhamentos dos tipos de máquinas usadas no experimento 170
Tabela 14. Resultados da execução do experimento NeuroPON de uma RNA MLP para a função XOR em Erlang 213
Tabela 15. Resultados das execuções do caso de estudo I 215
Tabela 16. Resultados das execuções do caso de estudo II 217
Tabela 17. Percentual de variação de tempo entre estratégias IND e CBCF 219
Tabela 18. Comparativo de melhora de desempenho entre os ambientes 219
Tabela 19. Comparativo de fator tempo dividido por número de notificações 219
Tabela 20. Comparativo de tempo de desenvolvimento entre Elixir e NOPL para estratégia de tráfego independente 270
Tabela 21. Comparativo de tempo de desenvolvimento entre Elixir e NOPL para estratégia de congestionamento facilitado 272

LISTA DE CÓDIGOS

Código 1. Exemplo de polimorfismo em Elixir 68
Código 2. Implementação de trata mento da mensagem *set_attribute* em microator *FBE* na linguagem Elixir 142
Código 3. Exemplo de implementação de *FBE* como especialização do módulo *NOP.Element.FBE* 143
Código 4. Exemplo de implementação de Rule como especialização de *NOP.Element.Rule* 144
Código 5. Trecho de código de inicialização do módulo Application do *framework* 145
Código 6. Exemplo de utilização de estatísticas do *framework* NOP Elixir 147
Código 7. Extensão do analisador léxico para reconhecer a palavra reservada *ELIXIR* 153
Código 8. Extensão do analisador sintático para reconhecimento de código específico em Elixir 154
Código 9. Extensão no analisador sintático para integração com a classe de compilação para framework Elixir 155
Código 10. Código para a construção da estrutura de arquivos 157
Código 11. Exemplo de arquivo de configuração mix.exs 158
Código 12. Exemplo de tradução de um Attribute para *Framework* NOP Elixir 161
Código 13. Exemplo de tradução de um *Method* para *Framework* NOP Elixir 162
Código 14. Exemplo da tradução de lógica de uma Rule para *framework* NOP Elixir 163
Código 15. Exemplo de função *initialize* declarada em todos os módulos FBE 164
Código 16. Exemplo de compilação de uma *Main Rule* 165
Código 17. Exemplo de função *initialize* e *create_main_rules* para o *FBE Main* 166
Código 18. Exemplo de um *FBE* da aplicação "Controle de semáforos" 176
Código 19. Exemplo de um Método de um *FBE* da aplicação "Controle de semáforos" 176
Código 20. Exemplo de uma Rule de aplicação de controle de semáforos 177
Código 21. *FBE* para estratégia CBCF em NOPL 186
Código 22. Rule para estratégia CBCF em NOPL 186
Código 23. Exemplo de tradução de *Method* PON para target NOPL-*Namespaces* 268

LISTA DE SIGLAS, ACRÔNIMOS E ABREVIATURAS

SIGLA	Original	Tradução
API	Application Process Interface	Interface de processo de aplicação
BEAM	Bogumil's/Björn's Abstract Machine	Máquina virtual do Bogumil/Björn
CMP	Chip-level MultiProcessor	Multiprocessador a nível de chip
CISC	Complex Instruction Set Computer	Computador de conjunto de instruções complexo
CPU	Central Processing Unit	Unidade Central de Processamento
DECT	Digital Enhanced Cordless Telecommunications	Telecomunicações sem fio digital aprimoradas
DSM	Distributed Shared Memory	Memória compartilhada distribuída
ERTS	Erlang Run-time System	Sistema Erlang em Tempo de execução
ETSI	European Telecommunications Standards Institute	Instituto Europeu de Padrões de Telecomunicação
FCFS	First Come First Served	Primeiro a chegar, primeiro a ser servido
GPU	Graphics Processing Unit	Unidade de processamento gráfico
MIMD	Multiple Instruction Stream, Multiple Data Stream	Fluxo de instruções múltiplas, fluxo de dados múltiplos
MISD	Multiple Instruction Stream, Single Data Stream	Múltiplos Fluxos de Instrução, Único Fluxo de Dados
MIT	Massachusetts Institute of Technology	Instituto de Tecnologia de Massachusetts
MP	Message Passing	Passagem de mensagens
PON	Notification-Oriented Paradigm	Paradigma orientado a notificações

NOPL	Notification-Oriented Programming Language	Linguagem de Programação orientada a Notificações
NUMA	Non-Uniform Memory Access	Acesso não uniforme à memória
OTP	Online Telecom Platform	Plataforma de Telecomunicações Online
POSIX	Portable Operating System Interface	Interface do sistema operacional portátil
PThreads	POSIX Threads	Threads do POSIX
RAM	Random Access Memory	Memória de acesso direto
RISC	Reduced Instruction Set Computer	Computador com conjunto de instruções reduzido
RR	Round Robin	Arredondamento Robin
SSD	Solid-state Drive	Unidade de Estado Sólido
SIMD	Single Instruction Stream, Multiple Data Stream	Fluxo de Instrução Única, Fluxo de Dados Múltiplos
SISD	Single Instruction Stream, Single Data Stream	Fluxo de instrução única, fluxo de dados único
SJF	Shortest Job First	Trabalho mais curto primeiro
SMP	Symmetric Multiprocessor	Multiprocessadores simétricos
TBB	Threading Building Blocks	Encadeando blocos de construção
TLP	Thread Level Parallelism	Paralelismo de nível de thread
UMA	Uniform Memory Access	Acesso Uniforme à Memória
UML	Unified Modeling Language	Linguagem de Modelagem Unificada

SUMÁRIO

1 INTRODUÇÃO 22

1.1 CONTEXTO DA PESQUISA 22

1.1.1 PON e NOPL 25

1.1.2 Erlang e Elixir 27

1.2 OBJETIVOS 28

1.3 MOTIVAÇÃO 29

1.4 ORGANIZAÇÃO DO DOCUMENTO 30

2 FUNDAMENTAÇÃO TEÓRICA 31

2.1 MULTICORE 31

2.1.1 Histórico 31

2.1.2 *Multicore* e *Manycore* 34

2.1.3 Classificação 35

2.1.4 Modelos Comerciais *Multicore* 40

2.2 ORGANIZAÇÃO E PARALELISMO DE SOFTWARE 42

2.2.1 Tarefas 42

2.2.2 Processos 43

2.2.3 Threads 44

2.2.4 Concorrência e paralelismo 45

2.3 COMUNICAÇÃO ENTRE PROCESSOS 48

2.3.1 Problemas de comunicação entre processos 48

2.3.2 Solução de Peterson 51

2.3.3 Semáforo 52

2.3.4 Mutexes 52

2.3.5 Mensagens assíncronas 53

2.4 MODELO DE ATORES 54

2.4.1 Histórico 54

2.4.2 Conceitos fundamentais 55

2.4.3 Semântica de passagem de mensagens 57

2.4.4 Escopo e localidade 58
2.4.5 Solução para problema de comunicação entre processos 59
2.5 ERLANG 60
2.5.1 Histórico 60
2.5.2 Processos Erlang 62
2.5.3 Concorrência 64
2.5.4 Suporte a *Multicore* 65
2.5.5 Considerações finais 66
2.6 ELIXIR 67
2.6.1 Histórico 67
2.6.2 Principais características 68
2.7 PARADIGMAS DE PROGRAMAÇÃO 69
2.7.1 Taxonomia dos paradigmas de programação 69
2.7.2 Barreiras dos paradigmas tradicionais 73
2.8 CONCLUSÕES DO CAPÍTULO 74
3 O PARADIGMA ORIENTADO A NOTIFICAÇÕES (PON) 76
3.1 INTRODUÇÃO 76
3.2 CONTEXTO HISTÓRICO 77
3.3 FUNDAMENTOS 80
3.4 METAMODELO E MECANISMO DE NOTIFICAÇÕES 82
3.5 MATERIALIZAÇÕES EM HARDWARE 84
3.6 MATERIALIZAÇÕES EM SOFTWARE 87
3.6.1 Arquétipo ou *Framework* PON 87
3.6.2 Linguagem e Compilador PON 88
3.6.3 Materializações do PON em *multicore* 90
3.7 ESTADO DA ARTE DO PON EM SOFTWARE 92
3.7.1 MCPON 92
3.7.2 Grafo PON 96
3.8 CONSIDERAÇÕES SOBRE O CAPÍTULO 100

4 DESENVOLVIMENTO DO TRABALHO 101

4.1 PON E MODELO DE ATORES 101
4.1.1 Elementos PON definidos como atores 103
4.1.2 Considerações sobre PON como Microatores 122

4.2 MODELAGEM UML 123
4.2.1 Modelagem 123
4.2.2 Elementos do paradigma PON 124
4.2.3 Elementos complementares 132

4.3 *FRAMEWORK* NOP ELIXIR 136
4.3.1 Adaptação da modelagem UML dos elementos PON 137
4.3.2 Adaptação da modelagem UML dos elementos complementares 139
4.3.3 Estrutura de arquivos 139
4.3.4 Elementos do paradigma PON 141
4.3.5 Elementos complementares 144
4.3.6 Funcionalidades complementares 147
4.3.7 Melhorias 148
4.3.8 Empacotamento e distribuição 152

4.4 COMPILADOR NOPL ERLANG-ELIXIR 152
4.4.1 Compilador e o Grafo PON 152
4.4.2 Extensão dos analisadores léxico e sintático 153
4.4.3 Classe de compilação 155

4.5 CONSIDERAÇÕES FINAIS 166

5 CASOS DE ESTUDO 168

5.1 CONSIDERAÇÕES INICIAIS 168

5.2 MATERIAIS E MÉTODOS 169
5.2.1 Recursos Materiais e Métodos 169
5.2.2 Cálculo de tempo de execução 170
5.2.3 Cálculo da taxa de ocupação dos núcleos 171
5.2.4 Cálculo de ocupação da memória 172
5.2.5 Ambiente de Simulação 173

5.3 CASO DE ESTUDO I: ESTRATÉGIA INDEPENDENTE 175

5.3.1 Configurações dos experimentos 178
5.3.2 Resultados e discussões 178
5.3.3 Consumo de memória 182
5.3.4 Considerações finais 183
5.4 CASO DE ESTUDO II: CONTROLE BASEADO EM CONGESTIONAMENTO FACILITADO 184
5.4.1 Configurações dos experimentos 187
5.4.2 Resultados e discussões 187
5.4.3 Consumo de memória 191
5.4.4 Considerações finais 192
5.5 COMPARATIVOS ENTRE OS ESTUDOS DE CASO 193
5.6 CONSIDERAÇÕES FINAIS 197
6 CONCLUSÕES E TRABALHOS FUTUROS 198
6.1 RESUMO DA PESQUISA 198
6.2 PRINCIPAIS CONTRIBUIÇÕES E RESULTADOS 201
6.3 TRABALHOS FUTUROS 203
REFERÊNCIAS 206
APÊNDICE A - Estudo de caso NeuroPON 213
APÊNDICE B - Tabulação de resultados do caso de estudo I 215
APÊNDICE C - Tabulação de resultados do caso de estudo II 217
APÊNDICE D - Tabulação dos comparativos entre os estudos de caso 219
APÊNDICE E - Códigos fonte em linguagem NOPL do caso de estudo I 220
APÊNDICE F - Códigos fonte em linguagem NOPL do caso de estudo II 229
APÊNDICE G - Manual de utilização do *Framework* NOP Elixir 242
APÊNDICE H - Comparativo NOPL Erlang-Elixir com NOPL-Namespaces 268

APÊNDICE I - Comparativo *Framework* NOP Elixir com solução *multicore* especialista Elixir 270

APÊNDICE J - Comparativo *Framework* NOP Elixir com (NEGRINI, PORDEUS e SIMÃO, 2019) 274

APÊNDICE K – Código fonte da classe *CodeGenerationElixir*.....277

1 INTRODUÇÃO

Este capítulo se inicia apresentando o avanço das tecnologias dos microprocessadores no tocante a *multicore* e os efeitos colaterais na área de software. Também faz uma breve explanação acerca do PON e da Notification Oriented Programing Language (NOPL) juntamente com as linguagens funcionais Erlang e Elixir, salientando as potenciais sinergias entre estas tecnologias para *multicore*. Em seguida é apresentada uma proposta de fusão destas tecnologias como elemento do objetivo geral deste trabalho, bem como os demais elementos que o compõe e suas motivações. Por fim, tem-se uma orientação geral aos demais capítulos.

1.1 CONTEXTO DA PESQUISA

Historicamente, o rápido crescimento da demanda por softwares aliado à complexidade crescente dos problemas a serem resolvidos com eles, criou uma procura por softwares cada vez mais rápidos, eficientes e eficazes. Contudo, o atendimento desta crescente demanda não era alcançado. Edsger Dijkstra (1972) apresentou pela primeira vez, no início dos anos 1970, a causa desta incapacidade de suprir esta demanda. Ele afirmou que a complexidade dos processos de software aliado à relativa imaturidade da engenharia seriam as causas para isto. Esta incapacidade ficou conhecida como “crise de software” (DIJKSTRA, 1972). Apesar dos avanços aliados à engenharia de software, esta crise ainda tem se mantido e agravado ao longo das décadas (BAUTSCH, 2007).

Aliado à intitulada crise de software, tem-se também a limitação da frequência de operação dos processadores (*clock*), cujo incremento não é mais possível em função da limitação dos materiais usados atualmente para a construção destes componentes (DEBENEDICTIS, 2017). Na busca em suplantar esta limitação e expandir a capacidade de processamento, a multiplicação do número de unidades de processamento (CPU) tem sido uma das estratégias amplamente empregada. Uma destas tecnologias de multiplicação de CPUS é conhecida como *multicore* (PILLA, SANTOS e CAVALHEIRO, 2009).

A tecnologia *multicore* consiste na integração de dois ou mais núcleos de processamento (*cores*) em um único processador (*chip*). O processador *dualcore*, por

exemplo, contém dois núcleos de processamento, enquanto o *quadcore* contém quatro núcleos de processamento. Como exemplo contemporâneo desta tecnologia, pode-se citar a segunda geração do processador AMD Ryzen™ Threadripper™ 2990WX com 32 núcleos (AMD, 2019).

Contudo, estes avanços em tecnologia *multicore* tornaram o desenvolvimento de software ainda mais complexo, pois para o correto aproveitamento desta tecnologia exigem-se softwares criados com sub-rotinas desacopladas para serem executadas nesta arquitetura (DEBENEDICTIS, 2017). De fato, para o devido aproveitamento dos núcleos em paralelo, particularmente alcançando concorrência e paralelismo fino, estes ambientes geralmente impõem dificuldades de abstração aos desenvolvedores, tradicionalmente habituados com processamento e desenvolvimento de softwares sequenciais (HENNESSY e PATTERSON, 2011).

Dito de outra forma, mais precisamente inclusive, pode-se atribuir esta dificuldade de uso apropriado de multicore a basicamente dois fatores principais e correlatos:

- O fato que, para escrever programas que aproveitem efetivamente os múltiplos núcleos de processamento, é necessário que o desenvolvedor despenda maior esforço na programação concorrente-paralela em relação à usual programação sequencial, de maneira a obter módulos desacoplado-paralelizáveis e harmonizados entre si (PILLA, SANTOS e CAVALHEIRO, 2009) (BANASZEWSKI, 2009);
- A tendência a alto nível de acoplamento de código entre as partes dos programas que compõem o software, causada pela natureza sequencial, prolixa e redundante dos atuais paradigmas de programação, o que é uma das principais deficiências destes por dificultar inclusive a composição de módulos desacoplados, particularmente com granularidade fina que seria útil para paralelização equilibrada entre núcleos de processamento (SIMÃO e STADZISZ, 2009) (SIMÃO, BANASZEWSKI, *et al.*, 2012) (BELMONTE, 2012)

Ainda assim, mesmo que o desenvolvedor despenda esforço no desmembramento do software em partes desacopladas, estas partes serializáveis ainda apresentarão fluxos de processamento de diferentes tamanhos, limitando a

melhoria de desempenho do mesmo quando executado em ambientes *multicore* (AMDAHL, 1967).

Entretanto, estas dificuldades poderiam ser minimizadas ou até eliminadas com o uso de tecnologias alternativas adequadas. A proposição de novos paradigmas ou mesmo técnicas de programação é uma delas. Neste âmbito, uma solução alternativa é o chamado Paradigma Orientado a Notificações (PON) que, dentre suas propriedades, tem a tendência de naturalmente gerar código de desacoplamento fino (SIMÃO, 2005). Esta propriedade é particularmente pertinente para aplicações *multicore*, o que se confirmou em primeiros experimentos feitos com tecnologia acentuadamente prototipal do PON para este domínio (WEBER, 2010) (BELMONTE, 2012) (BARRETO, 2016) (OLIVEIRA, 2016) (TALAU, 2016).

Ainda, outra alternativa no âmbito dado seria o uso de programação orientada ao modelo de atores, a qual foi definida como uma abordagem baseada em elementos primitivos universais pensados para computação concorrente (HEWITT, 1973). Aliado a isto, soma-se o fato de já haver materializações apropriadas fundamentadas neste modelo que permitem o desenvolvimento de aplicações que executam suas sub-rotinas de forma desacoplada e concorrente, alcançando assim uma execução concorrente e paralelizável quando executadas em ambientes *multicore*. Como exemplos relevantes desta tecnologia citam-se a linguagem Erlang e a linguagem Elixir, que constituem linguagens funcionais concorrentes com orientação a atores (CESARINI e THOMSON, 2009).

Neste contexto dado é apresentada a proposta deste trabalho: uma solução, com tecnologia decorrente, que surge com a fusão do PON, um paradigma emergente, com as linguagens Erlang e Elixir. Esta fusão se dá através da tecnologia NOPL, uma nova linguagem de programação de alto nível do PON e respectivo metamodelo de compilação para plataformas distintas. Assim, a suposição inicial é que a nova solução/tecnologia resultante dessa fusão posta seria uma plataforma de desenvolvimento para ambiente *multicore*, transparente para o desenvolvedor, com paralelização fina de processamento entre seus núcleos. Isto posto, as naturezas de cada tecnologia pertinente para tal, bem como suas limitações e sinergias são introduzidas a seguir.

1.1.1 PON e NOPL

O Paradigma Orientado a Notificações (PON) foi proposto embrionariamente e depois efetivamente por J. M. Simão, sendo desenvolvido por ele e demais pesquisadores do Laboratório de Sistemas Inteligentes [de Produção] (LSI[P]) da UTFPR e afins (SIMÃO, 2001) (SIMÃO, 2005) (SIMÃO, 2018). O PON vem sendo desenvolvido e materializado por já quase duas décadas desde sua criação embrionária na forma do então Controle Orientado a Notificações (CON) (SIMÃO, 2005), passando por Inferência Orientada a Notificações (ION), alcançando a forma de paradigma chamado PON (SIMÃO e STADZISZ, 2008).

Em suma, atualmente, considera-se que o PON apresenta três propriedades elementares que o destacam como paradigma:

- Facilidade de desenvolvimento de software em alto nível por meio de concepção orientada a regras, sempre à luz da estruturação organizada por entidades notificantes do modelo computacional do paradigma em questão. Basicamente, este modelo é composto por entidades facto-execucionais na forma de elementos notificantes e por entidades lógico-causais na forma de regras notificáveis (RONSZCKA, FERREIRA, *et al.*, 2017).
- Isenção de redundâncias estruturais (i.e., repetição de código) e temporais (i.e., reavaliação desnecessária) de expressões lógico-causais, em virtude do modelo baseado em notificações pontuais. Isto permite inclusive a construção de programas com alta reatividade e, por consequência, com baixo tempo de processamento (BANASZEWSKI, 2009) (SIMÃO, BANASZEWSKI, *et al.*, 2012) (RONSZCKA, BANASZEWSKI, *et al.*, 2015).
- Desacoplamento explícito (ou acoplamento mínimo) das entidades que compõem o modelo e mesmo de suas partes devido à organização baseada em notificações. Isto viabiliza, dentre outros, a construção de programas com execução paralela e/ou distribuída tão fina quanto a arquitetura computacional permitir (PETERS, 2012) (LINHARES, 2015) (BELMONTE, LINHARES, *et al.*, 2016) (BARRETO, VENDRAMIN e SIMÃO, 2018) (OLIVEIRA, ROTH, *et al.*, 2018) (SCHÜTZ, FABRO, *et al.*, 2018) (KERSCHBAUMER, LINHARES, *et al.*, 2018).

Neste contexto, o PON, com desacoplamento implícito entre as partes das entidades permite que estas possam ser separadas em módulos de granularidade fina para executarem concorrentemente e paralelamente em diferentes núcleos de processamento. Esta característica torna o PON atrativo para sua aplicação no desenvolvimento de software paralelo.

Primeiramente, visando demonstrar o PON, houve previamente implementações dele na forma de arquétipos ou *frameworks* sobre outras linguagens de programação usuais orientando-as, portanto, a trabalhar por notificações (BANASZEWSKI, 2009) (VALENÇA, 2012). Subsequentemente, visando estabelecer o PON, houve naturalmente a necessidade de desenvolvimento de uma linguagem própria ao PON. Ocorreu, portanto, a definição da chamada tecnologia NOPL (Notification Oriented Programing Language). Esta pesquisa foi liderada pelo pesquisador A. F. Ronszcka, envolvendo desenvolvimento de linguagens e tecnologia de compiladores (FERREIRA, 2015) (RONSZCKA, 2019).

Nesta sua última versão, a Tecnologia NOPL apresenta um Método de Compilação do PON (MCPON) que comporta uma solução de compilação de linguagens próprias do PON em uma única estrutura de dados uniforme intermediária em formato de grafo de entidades notificantes – o Grafo PON. A partir do Grafo PON, por meio de navegações neste grafo, pode-se então compor compiladores para geração de código final para plataformas distintas (RONSZCKA, 2019).

Portanto, na verdade, o Grafo PON se constitui em um elemento de desacoplamento entre os programas feito em PON com as entidades que os realizarão e a plataforma na qual os programas serão executados. Inclusive, isto traz uma maior facilidade na materialização de programas PON em outras plataformas que ainda não tenham sido objeto de integração ou afins com o novo paradigma em voga. Neste âmbito, ainda não há resultados satisfatórios em software PON *multicore*, o que demanda busca de sinergia do PON com tecnologias pertinentes (SCHÜTZ, 2019). Este presente trabalho, portanto, se interessa pela integração da tecnologia NOPL com plataforma de execução baseada em Linguagem Erlang com sua arquitetura concorrente-paralelizável, juntamente com a linguagem Elixir e suas funcionalidades complementares.

1.1.2 Erlang e Elixir

O Erlang surgiu em meados da década de 1980 nos laboratórios de Ciência da Computação da Ericsson, com o intuito de resolver problemas de *time-to-market* os quais não era possível de alcançar com as tecnologias disponíveis à época. Tendo seus autores Joe Armstrong, Robert Virding e Mike Williams, sua concepção teve influências de linguagens funcionais como ML e Miranda, linguagens concorrentes como ADA, Modula e Chill e, ainda linguagens de programação lógica como Prolog (CESARINI e THOMSON, 2009).

Criado para concorrência, paralelismo e distribuição, Erlang introduz o contexto de processos independentes que possuem *heap* e pilha próprios. Toda a comunicação entre estes processos é feita através de mensagens assíncronas tornando a arquitetura concorrente, paralelizável, distribuível e potencialmente tolerante a falhas (CESARINI e THOMSON, 2009). Entretanto, esta arquitetura materializada no Erlang não é algo novo, tendo sido fundamentada no modelo matemático de atores. Este modelo surgiu na década de 1970 (HEWITT, 1973).

Apesar da vantagem de permitir concorrência, paralelismo e distribuição via um modelo efetivo de atores funcionais, Erlang é conhecida por ser uma linguagem de difícil programação, principalmente para programadores oriundos da programação orientada a objetos. Erlang não comporta elementos como: metaprogramação[1], polimorfismo[2] e ferramentas de primeira classe[3].

Para suprir estas deficiências inerentes à linguagem Erlang, surge, então, a linguagem Elixir. Sendo executada sobre a plataforma Erlang, Elixir valoriza seus fundamentos funcionais ao mesmo tempo que foca na produtividade com estes recursos adicionais (THOMAS, 2018).

---

[1]Metaprogramação é a construção de programas que escrevem ou manipulam outros programas (ou a si próprios) assim como seus dados, ou que fazem parte do trabalho em tempo de compilação. Em alguns casos, isso permite que os programadores sejam mais produtivos ao evitar que parte do código seja escrita manualmente.

[2]Polimorfismo permite que referências de tipos de classes mais abstratas representem o comportamento das classes concretas que referenciam. Assim, é possível tratar vários tipos de maneira homogênea.

[3]Ferramentas de Primeira Classe (também tipo, objeto, entidade ou valor de primeira classe) em uma linguagem de programação é uma entidade que suporta todas as operações geralmente disponíveis para outras entidades. Estas operações incluem, normalmente, serem passados como um argumento, retornar uma função e atribuir a uma variável.

Todavia, nem Elixir/Erlang, nem o modelo de atores tem conseguido isentar os desenvolvedores de projetarem seus códigos pensando em concorrência. Tampouco produzir fluxos de processamento pequenos e facilmente escalonáveis. Nesta lacuna posta, via NOPL, o PON se apresenta como uma alternativa viável e pertinente. A hipótese inicial é que o nível de desacoplamento implícito das entidades do PON permita alcançar, via microatores (atores com processamento leve), os quais seriam mais facilmente balanceáveis e paralelizáveis em ambiente *multicore* pelas tecnologias de escalonamento da arquitetura Erlang.

1.2 OBJETIVOS

Esta dissertação explora o contexto de pesquisa apresentado e tem o seguinte objetivo geral:

Propor uma solução via Tecnologia NOPL que traduza os elementos PON, naturalmente desacoplantes, em microatores NOPL Erlang-Elixir permitindo assim, sua execução em plataforma Erlang com suporte *multicore*. Com esta tecnologia, possibilitar a execução concorrente e paralela com balanceamento apropriado. Com isto, alcançar-se-ia a programação *multicore* apropriada e transparente, ou seja, sem conhecimento anterior de concorrência e paralelismo por parte do desenvolvedor.

Para atingir o objetivo geral apresentado, são propostos os seguintes objetivos específicos:

- Propor uma modelagem dos elementos do PON com passagem de mensagens assíncronas compatível com atores, porém com processamento leve. Por este motivo, atores resultantes desta modelagem serão nomeados microatores.
- Implementar um *Framework* dos elementos do PON em forma de microatores em Elixir permitindo a execução de sistemas em PON nesta plataforma. Este *framework* chamar-se-á *Framework* NOP Elixir.
- Implementar um compilador a partir da Tecnologia NOPL 2.0 que, a partir da Linguagem NOPL 2.0 permitir gerar, via o Grafo PON, os respectivos

programas em Elixir suportados pelo *Framework* PON Elixir. Esta solução como um todo se chamará Tecnologia NOPL Erlang-Elixir.

- Avaliar a solução proposta por meio de experimentos envolvendo testes da tecnologia gerada em várias máquinas virtuais com diferentes quantidades de núcleos, a fim de verificar a adequação da Tecnologia NOPL Erlang-Elixir, inclusive para alcançar concorrência e paralelismo de forma transparente para o desenvolvedor.
- A partir dos experimentos realizados, observar a taxa de ocupação e em cada núcleo e o balanceamento durante os experimentos, a fim de verificar a adequação da Tecnologia NOPL Erlang-Elixir para alcançar concorrência e paralelismo de forma apropriada nos núcleos.

1.3 MOTIVAÇÃO

O PON surgiu como uma abordagem inovadora para a concepção e execução de aplicações, sendo que suas propriedades têm se mostrado apropriadas a execuções paralelas e distribuídas. Esta inovação também seria apropriada, portanto, quando é levada em consideração a evolução dos processadores atuais para um ambiente com múltiplos núcleos de processamento, ou seja, um ambiente de execução *multicore*, no qual o software executa com concorrência e paralelismos.

Outrossim, a linguagem Erlang e sua arquitetura têm se mostrado robusta e tem melhorado através dos anos seu suporte *multicore* (CESARINI e THOMSON, 2009). A linguagem Elixir, por sua vez, vem cobrir lacunas de programação sem subtrair os ganhos alcançados pela linguagem Erlang. Contudo, estas melhorias não isentam desenvolvedores de projetarem seus programas pensando em concorrência e paralelismo.

Neste sentido, a motivação deste trabalho é propor uma tecnologia nova, à luz da conjunção dessas elencadas, que possibilite um melhor aproveitamento dos recursos de um ambiente *multicore* sem que haja esforço do desenvolvedor para isto. Ademais, esta solução, do ponto de vista do PON, permitiria que experimentos outrora executados em ambientes sequenciais pudessem ser elaborados de forma concorrente e com os paralelismos do *multicore* sem esforços suplementares.

1.4 ORGANIZAÇÃO DO DOCUMENTO

Este documento está organizado em seis capítulos. O presente capítulo contextualizou o problema que levou à motivação para a elaboração deste trabalho, bem como seu objetivo e motivações.

O Capítulo 2 apresenta a revisão da literatura relacionada à elaboração deste trabalho de pesquisa, abrangendo primeiramente a tecnologia *multicore* com princípios de escalonamento de tarefas e arquiteturas paralelas. Em seguida, são apresentados os principais conceitos de organização e paralelismo de software. Por fim, o modelo de atores, a arquitetura Erlang e a linguagem Elixir são, então, visitados.

O Capítulo 3 apresenta o Paradigma Orientado a Notificações (PON), não somente como uma revisão bibliográfica dele, mas como uma contextualização técnica mais aprofundada. Este capítulo é baseado em um conjunto de conhecimento consolidado dentro do corpo de pesquisadores deste tema junto à UTFPR.

O Capítulo 4 apresenta o desenvolvimento do trabalho. Neste capítulo são apresentadas as etapas de modelagem do PON em uma modelagem assíncrona baseada em atores. Após isto, é apresentada a composição do *Framework* NOP Elixir. Por fim, o capítulo demonstra a construção da Tecnologia NOPL Erlang-Elixir, ou seja, do compilador de Linguagem do PON, a saber NOPL, para integração ao *Framework* NOP Elixir.

O Capítulo 5 apresenta as experimentações no qual são apresentados dois casos de estudo e comparativos.

Por fim, o Capítulo 6 apresenta conclusões deste trabalho e faz reflexões sobre trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os principais conceitos para estruturar esta dissertação. Primeiramente, a seção 2.1 apresenta os conteúdos relativos às arquiteturas paralelas e introduz a tecnologia *multicore*. A seção 2.2, por sua vez, fornece uma contextualização sobre os principais conceitos de concorrência e paralelismo em sistemas operacionais. A seção 2.3.5 traz as formas de comunicação segura entre processos concorrentes e paralelos. A seção 2.7 traz uma taxonomia dos paradigmas de programação e as limitações dos modelos mais utilizados, particularmente no tocante à programação concorrente e paralela. A seção 2.4, por sua vez, introduz o modelo matemático de atores, uma modelagem pensada para programação concorrente e paralela. Ainda, As seções 2.5 e 2.6 apresentam respectivamente as linguagens baseada em modelo de atores Erlang e Elixir. Por fim, a seção 2.8 apresenta as considerações finais deste capítulo.

2.1 MULTICORE

Esta seção fornece uma contextualização sobre a tecnologia *multicore*. Inicia-se com um breve histórico desta arquitetura, passando pelos marcos mais importantes. Após, algumas definições desta arquitetura são apresentadas no tocante a diferenciá-la da arquitetura *manycore.* Em seguida é apresentada uma classificação das diversas arquiteturas *multicore* disponíveis. Por fim, são apresentados alguns modelos comerciais *multicore* disponíveis atualmente.

2.1.1 Histórico

A história da computação *multicore* pode ser contada a partir de meados da década de 1960, com o surgimento do computador Solomon. Esta arquitetura, porém, exigia uma programação vetorial. Neste tipo de programação vetorial, diferentemente da programação tradicional na qual a instrução é executada em apenas um dado, cada instrução é executada em um conjunto de dados independente de maneira simultânea (SATISH, KIM, *et al.*, 2012). Porém, a dificuldade deste tipo de programação restringiu inicialmente seu uso a cientistas e engenheiros. Nas décadas

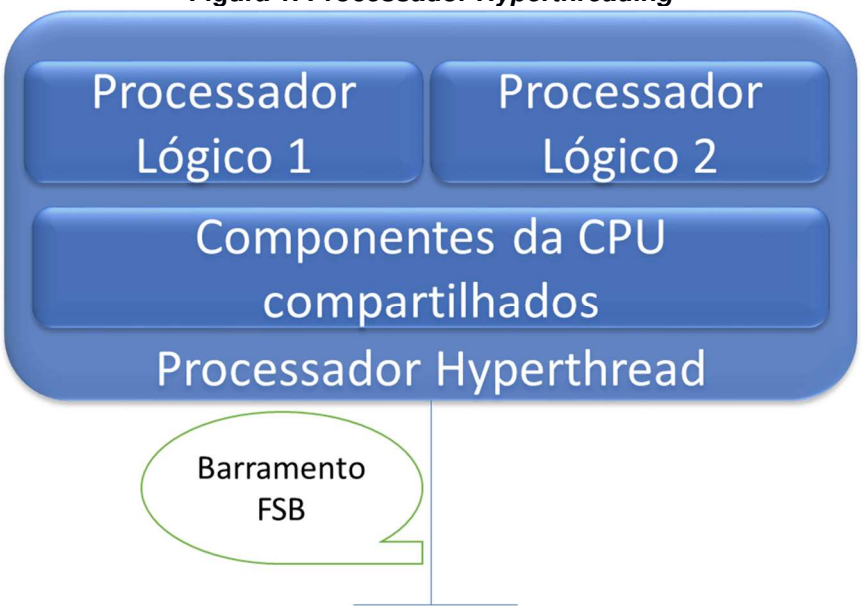
subsequentes, pelo mesmo motivo, iniciativas de muitas companhias na criação de máquinas paralelas se mostraram insatisfatórias (BLAKE, DRESLINSKI e MUDGE, 2009).

Uma das exceções à esta frustração foi a máquina vetorial Cray, pois esta máquina possuía, além de uma unidade central de processamento (CPU), também um processador escalar muito rápido que podia ser programado de maneira apropriada a ela (BLAKE, DRESLINSKI e MUDGE, 2009).

Durante os anos subsequentes, os avanços no esforço de melhorar o desempenho dos processadores levaram os fabricantes a implementarem estruturas mais complexas de forma a explorarem o paralelismo no nível de instrução (do inglês, *pipelining*). Este tipo de paralelismo permitia que a CPU realizasse a busca de uma ou mais instruções além da próxima a ser executada e as colocasse em uma fila de memória dentro do processador. Desta forma, as diferentes instruções ocorriam de maneira concorrente dentro da CPU (BLAKE, DRESLINSKI e MUDGE, 2009).

Outra linha de pesquisa foi a criação de uma lógica capaz de gerenciar instruções oriundas de várias *threads* em execução dentro de uma única CPU de forma simultânea. Este trabalho ficou conhecido a partir do trabalho de Tullsen, Eggers e Levy (1995), que foram quem introduziram o termo SMT (acrônimo de *Simultaneous Multithreading*).

Figura 1. Processador *Hyperthreading*



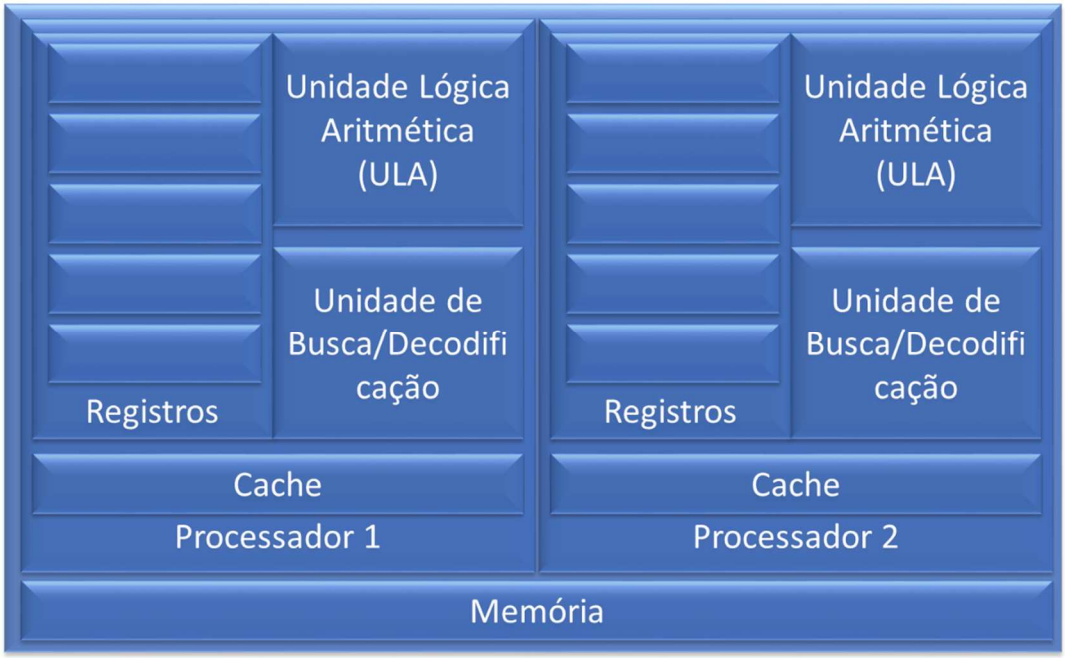
Fonte: adaptado de (HUGHES, 2003)

O termo SMT é também conhecido como *Hyperthreading.* Um diagrama desta arquitetura é apresentado na Figura 1. Máquinas *Hyperthreading* são comumente referenciadas como HT pelas fabricantes (e.g. processor Pentium 4 HT 2 threads da Intel) e tiveram poucas puramente SMT. Ainda que as implementações SMT/HT suplantassem em parte o problema inicial de limitação de frequência de processamento, a demanda por mais capacidade de processamento ainda era crescente.

O paradigma de evolução, até então focado em melhoras de arquitetura monoprocessada e aumento da frequência de operação (do inglês, *clock)* levou a um aumento no consumo energético de forma desproporcional ao ganho de desempenho adquirido. Em função disto, o superaquecimento destes processadores tornou-se uma de suas principais limitações (SAVAGE e ZUBAIR, 2008). Por conta disso, pesquisas relacionadas às arquiteturas *multicore* iniciaram-se.

No início dos anos 1990, então, o ritmo de evolução das máquinas paralelas mudou drasticamente com a adoção por parte das grandes companhias ao modelo de computação de múltiplos núcleos. Esta foi a forma encontrada por seus arquitetos para manter a taxa de aumento de desempenho esperada pelos consumidores.

Figura 2. Arquitetura *multicore*



Fonte: adaptado de (HUGHES, 2003)

A arquitetura básica dos processadores *multicore* baseia-se na arquitetura SMP (*Symmetric MultiProcessors*). Nessa arquitetura os diferentes núcleos são capazes de executar fluxos de instruções de forma independente e compartilham uma área de memória centralizada (WANG e WANG, 2006). Este modelo pode ser visto no diagrama da Figura 2.

Atualmente, há uma grande quantidade de arquiteturas *multicore* sendo oferecidas no mercado e direcionadas a todos os nichos, desde o segmento de sistemas embarcados, passando pelo de desktops de propósito geral até o segmento de servidores (BLAKE, DRESLINSKI e MUDGE, 2009).

2.1.2 *Multicore* e *Manycore*

Um processador *multicore* é um circuito integrado, ao qual dois ou mais processadores foram conectados em uma mesma pastilha, para melhorar o desempenho, reduzir o consumo de energia e processar simultaneamente mais eficientemente várias *threads*. Este tipo de arquitetura é projetado para executar com eficiência código paralelo e serial e, portanto, coloca mais ênfase no alto desempenho de *thread* única. Esta arquitetura também é identificada por ser fruto da evolução de processadores de um único núcleo e, por este motivo, muitos exemplares desta arquitetura mantém compatibilidade com versões anteriores de apenas um núcleo. O número de núcleos contidos neste tipo de arquitetura normalmente é baixo, sendo comuns processadores com dois, quatro ou oito núcleos (BRYAN, 2008).

Já o termo *manycore* caracteriza-se por núcleos independentes e mais simples. Exemplares deste tipo de arquitetura acabam possuindo muito mais núcleos em comparação com a arquitetura *multicore*, podendo chegar a milhares de núcleos (MATTSON, 2010). Este tipo de arquitetura é projetado para um alto grau de processamento paralelo. Por este motivo, os processadores *manycore* são mais comumente usados na computação de alto desempenho (VAJDA, 2011).

2.1.3 Classificação

Em virtude desta grande quantidade de arquiteturas *multicore* oferecidas atualmente, fazia-se necessário uma classificação. Blake et al. (2009), portanto, definiu uma classificação baseada nos seguintes elementos: classe de aplicação, relação de consumo de energia com desempenho, elementos de processamento e o sistema de memória. Esta classificação se popularizou e vem sendo uma das mais utilizadas. Entretanto, é importante reforçar que há outras formas de classificar a arquitetura *multicore* as quais não serão tratadas neste trabalho. A seguir, são detalhados cada um destes elementos de classificação.

Classe de Aplicação

A classificação por classe de aplicação de uma determinada arquitetura *multicore* refere-se ao fato de esta ter sido concebida ou adaptada para atender a um domínio específico de aplicação. Desta forma, espera-se que esta tenha um desempenho superior para este domínio em detrimento de arquiteturas de propósito geral ou não projetadas para o mesmo. A adequação de uma arquitetura para um domínio específico pode trazer consequências positivas, como a melhoria de desempenho e/ou economia de energia (BLAKE, DRESLINSKI e MUDGE, 2009).

A seguir, algumas classes de aplicação mais comuns são apresentadas (BLAKE, DRESLINSKI e MUDGE, 2009):

- *Processamento de dados*: Nessas arquiteturas, a computação baseia-se em uma sequência de operações sobre um fluxo de dados no qual pouca ou nenhuma informação é reusada e uma alta taxa de transferência é necessária. Estas operações podem frequentemente ser feitas em paralelo. Por isso, elas favorecem uma arquitetura que tenha tantos elementos de processamento quanto possível, desde que mantenha uma boa relação consumo de energia e desempenho. Entre as mais conhecidas, cita-se a rasterização[4] de gráficos, os processamentos de imagens e de áudio e o processamento de comunicações sem fio.

[4] Rasterização, é a tarefa de converter uma imagem vetorial (curvas funcionais) em uma imagem *raster* (pixels ou pontos) para a possível leitura do documento. O termo rasterização também é utilizado para converter uma imagem formada por vetores para um arquivo de formato bitmap.

- *Processamento de controles:* Nesses tipos de aplicação, os programas tendem a possuir muitos desvios condicionais, dificultando o paralelismo. Além disso, eles muitas vezes precisam manter uma série de informações sobre seu próprio estado e normalmente fazem reuso de uma grande quantidade de dados. Por isso, essas aplicações favorecem um número modesto de elementos de processamento de propósito geral para lidar com a natureza desestruturada do seu código. Nesta classe enquadram-se as arquiteturas voltadas para processamentos como compressão/descompressão de arquivos, processamentos em rede e a execução de operações transacionais.

Relação Consumo de Energia/Desempenho

O desempenho dos processadores sempre foi o principal objetivo dos projetistas. Porém, na última década, o baixo consumo de energia também passou a ser tratado como um requisito de grande importância. Em grande parte, isto se deve ao crescimento do mercado de telefones celulares e de computadores portáteis, no qual o tamanho e a vida útil das baterias são restrições críticas (BLAKE, DRESLINSKI e MUDGE, 2009).

Mais recentemente, o crescimento dos *data centers* como suporte à computação em nuvem também fez com que o consumo de energia se tornasse um atributo importante no projeto de computadores não portáteis (BLAKE, DRESLINSKI e MUDGE, 2009).

Elementos de Processamento

Os elementos de processamento podem ser abordados de duas formas, quanto à sua arquitetura e quanto à sua microarquitetura. A arquitetura está relacionada ao conjunto de instruções (em inglês, *ISA*) e define a interface entre o hardware e o software. A microarquitetura, por sua vez, refere-se a como são implementados estes conjuntos de instruções (BLAKE, DRESLINSKI e MUDGE, 2009).

Em processadores *multicore* convencionais, a *ISA* de cada núcleo é tipicamente um *ISA* legado dos processadores mais antigos com pequenas modificações para suportar paralelismos, tal como a adição de instruções atômicas para sincronização.

A vantagem dessas *ISA*s é o fato de manter o legado das instruções e a sua compatibilidade tanto com as implementações quanto com as ferramentas de programação já existentes (BLAKE, DRESLINSKI e MUDGE, 2009).

Os ISAs podem ser classificados como reduzidos (em inglês, *Reduced Instruction Set Computer* - RISC) ou complexos (em inglês, *Complex Instruction Set Computer* - CISC). Os códigos em uma máquina CISC são menores em função da riqueza semântica das instruções. Por outro lado, as máquinas RISC facilitam o trabalho dos compiladores e sua microarquitetura é mais fácil de ser implementada. Além das definições básicas do ISA, os fabricantes têm continuadamente melhorado o desempenho de operações mais comuns através de extensões adicionais ao conjunto de instruções (BLAKE, DRESLINSKI e MUDGE, 2009).

A microarquitetura dos elementos de processamento é responsável, em muitos aspectos, pelo desempenho e pelo consumo de energia que podem ser esperados de uma arquitetura *multicore*. Essa microarquitetura é normalmente associada à classe de aplicação para o qual o processador *multicore* é desenhado. Apesar de grandes fabricantes oferecem ao mercado processadores *multicore* com núcleos homogêneos, a combinação de diferentes tipos de elementos de processamento em uma mesma arquitetura já se mostra como uma solução promissora.

Uma forma de organização heterogênea consiste no emprego de um processador de controle para comandar as atividades de um grupo de núcleos mais simples dedicados ao processamento de dados. Um dos benefícios dessa organização é reduzir o consumo de energia sem perda de desempenho. Porém, o modelo de programação desse tipo de arquitetura torna-se normalmente mais complicado (BLAKE, DRESLINSKI e MUDGE, 2009)..

Sistema de Memória

Em arquiteturas monoprocessadas, o sistema de memória é relativamente simples, consistindo em alguns poucos níveis de memória cache[5] para alimentar o processador com dados e instruções. Nas arquiteturas *multicore*, as memórias cache são apenas parte do sistema de memória. Os outros componentes incluem o modelo de consistência dos dados e a forma de interconexão entre elas. Esses componentes

[55] Memória cache é uma memória de alto desempenho localizada dentro do processador e que serve para aumentar a velocidade no acesso aos dados e instruções armazenados na memória RAM.

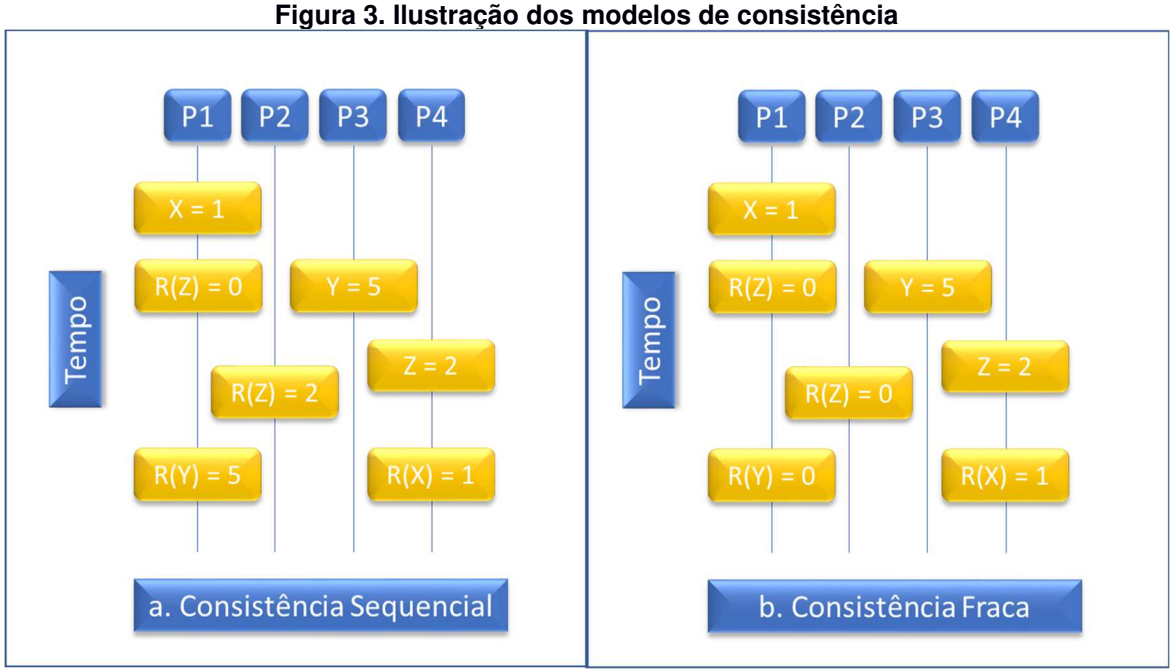
determinam como os núcleos se comunicam e impactam na programação, no desempenho das aplicações paralelas e no número de núcleos que o sistema comporta (BLAKE, DRESLINSKI e MUDGE, 2009).

O modelo de consistência define como as operações em memória podem ser reordenadas enquanto o código está sendo executado. Ele determina o quanto de esforço é requerido do programador ao escrever códigos paralelos. Modelos mais fracos exigem que o programador defina explicitamente como o código deve ser agendado no processador e possui protocolos de sincronização mais complexos. Por outro lado, os modelos mais robustos requerem menos esforço do programador com relação às sincronizações. O uso de um ou de outro também causa um impacto direto no desempenho (BLAKE, DRESLINSKI e MUDGE, 2009).

Os modelos de consistência mais comuns são os de consistência sequencial e consistência fraca. O modelo de consistência sequencial, um exemplar de modelos mais robustos, exige que todos os processadores do sistema enxerguem todas as leituras e escritas tal como elas ocorrem globalmente e no programa. Já nos modelos mais fracos, os processadores não se preocupam em enxergar as leituras e escritas feitas por outros processadores na ordem em que elas acontecem. Em função dessa baixa consistência, os modelos mais fracos requerem que os programadores façam uso de primitivas conhecidas como barreiras de memória[6] para forçar a consistência quando ela é necessária (BLAKE, DRESLINSKI e MUDGE, 2009).

[6] Uma barreira de memória (do inglês, *barrier* ou *fence)*, é um tipo de instrução de barreira que faz com que uma unidade central de processamento (CPU) ou compilador imponha uma restrição de ordem às operações de memória emitidas antes e depois da instrução. Isso garante que as operações emitidas antes da barreira são executadas antes e as operações emitidas após a barreira, executadas após.

Figura 3. Ilustração dos modelos de consistência



Fonte: (BLAKE, DRESLINSKI e MUDGE, 2009)

Para melhor ilustrar a diferença entre os modelos de consistência, um exemplo é apresentado na Figura 3. As linhas verticais representam a linha do tempo em cada processador *P1, P2, P3* e *P4*. A função *R(X)* representa a leitura de um registrador da variável atribuída como parâmetro. Neste exemplo, cada um dos processadores P1 a P4 estão despachando escritas (e.g., *X* = 1) e leituras (e.g., R(Z) = 0). O modelo de consistência sequencial (Figura 3.a) determina que todas as leituras e escritas para todos os endereços sejam vistas na mesma ordem por todos os processadores. Isso ocorre, quando o processador P2 lê Z e o valor retornado foi escrito previamente pelo processador P4. Essa comunicação é tipicamente forçada por uma mediação estabelecida através da rede de interconexão (BLAKE, DRESLINSKI e MUDGE, 2009).

No caso do modelo de consistência fraca (Figura 3.b), o processador P2 lê Z e o resultado retornado é zero. Neste caso, o modelo de consistência permite que núcleos diferentes enxerguem diferentes ordens globais de execução. Por possuírem uma sincronização relaxada, esses modelos são mais fáceis de serem implementados em hardware, deixando o esforço por conta do desenvolvedor do software. Já os modelos mais robustos, além de complicados, também são lentos pelo fato de não obterem as vantagens do acesso fora de ordem à memória (BLAKE, DRESLINSKI e MUDGE, 2009).

Outro elemento do sistema de memória passível de classificação é a memória cache. As memórias caches são muito importantes nas arquiteturas *multicore*, no qual muitos elementos de processamento tentam acessar a memória principal, que é custosa em termos de processamento. Há um esforço por parte dos desenvolvedores de arquiteturas *multicore* para definir o tamanho eficiente, tamanho este influenciado diretamente pela classe de aplicação a qual se deseja atender (BLAKE, DRESLINSKI e MUDGE, 2009).

Ainda na classificação de memórias, outro elemento que determina a classificação das arquiteturas *multicore* é o tipo de interconexão, isto é, a forma como os diferentes níveis da hierarquia de memória troca informações. Barramentos, anéis e *crossbars* são alguns exemplos de interconexão entre as memórias. (BLAKE, DRESLINSKI e MUDGE, 2009).

2.1.4 Modelos Comerciais *Multicore*

Como já explanado, há uma grande quantidade de arquiteturas multicore sendo oferecidas no mercado e direcionadas a todos os nichos, desde o segmento de sistemas embarcados, passando pelo de desktops de propósito geral até o segmento de servidores. Para oferecer um panorama geral dos modelos oferecidos, Blake et al nos apresenta uma tabela na qual alguns dos principais modelos são destacados (BLAKE, DRESLINSKI e MUDGE, 2009).

Tabela 1. Arquiteturas *multicore* de propósito geral

MODEL	ISA	NUMBER OF CORES	CACHE	MAX POWER	FREQUENCY	OPS/CLOCK
AMD PHENOM	X86	FOUR	64KB IL1 AND DL1/ CORE, 256 KB L2/CORE, 2-6 MB L3	140 W	2.5 GHZ - 3.0 GHZ	12-48
INTEL CORE I7	X86	TWO TO EIGHT	32KB IL1 AND DL1/ CORE, 256 KB L2/CORE, 8 MB L3	130 W	2.66 GHZ - 3.33 GHZ	18-128
SUN NIAGARA	SPARC	EIGHT	16KB IL1 AND 8KB DL1/ CORE, 4MB L2	60-123 W	900 MHZ - 1.4 GHZ	16
INTEL ATOM	X86	ONE TO TWO	32KB IL1 AND DL1/ CORE, 256 KB L2/CORE	2-8 W	800 MHZ - 1.6 GHZ	2-16
ARM CORTEX A9	ARM	ONE TO FOUR	(16,32,64)KB IL1 AND DL1/ CORE, 256 KB L2/CORE, UP TO 2 MB L2	1 W (NO CACHE)	N/A	3-12
XMOS XSI-G4	XCORE	FOUR	64 KB LCL STORE/CORE	0.2 W	400 MHZ	4

Fonte: Adaptado de (BLAKE, DRESLINSKI e MUDGE, 2009)

As quatro primeiras linhas da Tabela 1 representam uma seleção feita pelos autores de *multicores* de propósito geral. Todos esses modelos empregam um número limitado de processadores idênticos com caches grandes. Essas arquiteturas são direcionadas aos mercados de desktops e de servidores os quais o baixo consumo de energia não é um fator prioritário. As demais linhas da Tabela 1 também apresentam arquiteturas *multicore*, porém destinadas ao mercado de computação móvel e embarcada. Nesse contexto, o consumo de energia é muito importante pelo fato de que eles são alimentados por baterias.

Tabela 2. Arquiteturas *multicore* de alto desempenho

MODEL	ISA	NUMBER OF CORES	CACHE	MAX POWER	FREQUENCY	OPS/CLOCK
AMD RADEON	N/A	160 CORES 16 CORES PER SMD BLOCK 10 BLOCKS	16 KB LCL STORE/ SIMD BLOCK	15O W	750 MHZ	800-1600
NVIDIA G200	N/A	240 CORES 8 CORES PER SMD UNIT 30 SMD UNITS	16 KB LCL STORE/ EIGHT CORES	183 W	1.2 GHZ	240-720
INTEL LARRABEE	X86	UP TO 48 CORES	32 KB IL1 AND 32 KB DL1/ CORE, 4 MB L2	N/A	N/A	96-1536
IBM CELL	POWER	1 PPU, 8 SPUS	PPU: 32 KB IL1 AND 32 KB DL1/ 512 KB L2; SPU: 256 KB LCL STORE 32 KB IL1 AND 32 KB DL1/ CORE, 1 MB L2	100 W	3.2 GHZ	72
MICROSOFT XENON	POWER	3 CORES	32 KB IL1 AND 32 KB DL1/ CORE, 1 MB L2	60 W	3.2 GHZ	6-24

Fonte: Adaptado de (BLAKE, DRESLINSKI e MUDGE, 2009)

Na Tabela 2, as arquiteturas são mais especializadas, e dirigidas a computação de alto desempenho. Neste sentido, elas empregam um número expressivo de núcleos. Como é possível observar, nas arquiteturas AMR R700 e NVIDIA G200, esse número é na casa das centenas. A arquitetura IBM Cell é heterogênea, com um número pequeno de núcleos altamente especializados. Normalmente, essas soluções consomem bastante energia, com potência variando entre 100W e 180W.

Nesta seção foi apresentada a tecnologia *multicore*. A próxima seção, por sua vez, introduz conceitos de sistemas operacionais modernos para a organização e paralelismo de software. Nela é apresentada, de maneira geral, como os sistemas operacionais evoluíram para disponibilizar os benefícios da tecnologia *multicore* por meio de processos e subprocessos concorrentes.

2.2 ORGANIZAÇÃO E PARALELISMO DE SOFTWARE

Esta seção é focada em definições necessárias para o entendimento do restante deste trabalho. Inicialmente, é apresentada uma definição de Tarefas, Processos e Threads e como se relacionam em sistemas operacionais modernos. Por fim, são apresentadas as definições de concorrência e paralelismo de forma a traçar um diferencial claro entre elas.

2.2.1 Tarefas

Segundo Tanenbaum (TANENBAUM e WOODHULL, 2006), uma tarefa é definida como sendo a execução de um fluxo sequencial de instruções construído para atender uma finalidade específica. Desta forma, é um conceito dinâmico, que possui um estado bem definido a cada instante. Ainda, as tarefas definem as atividades a serem realizadas dentro do software. Geralmente, existem mais tarefas a realizar que processadores disponíveis e as tarefas têm diferentes prioridades, desse modo há necessidade de gerenciamento dessas tarefas. Portanto, a gerência de tarefas tem grande importância dentro de um Sistema Operacional (SO). Cabe ao SO organizar as tarefas definindo uma ordem para executá-las.

Os SOs atuais gerenciam suas tarefas por meio dos intitulados sistemas de tempo compartilhado (*time-sharing*). Na solução do gerenciamento de tarefas por meio dos sistemas de tempo compartilhado, cada atividade que detém o processador recebe um limite de tempo de processamento (*quantum*). Esgotado seu *quantum*[7], a

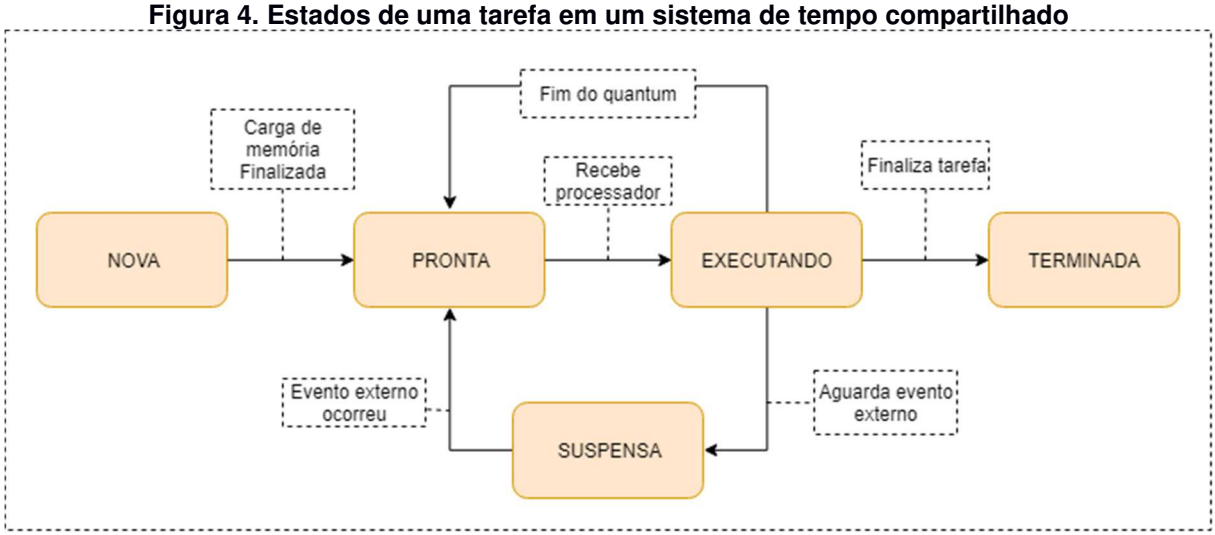
[7] *Quantum* pode ser entendido como o tempo que o sistema operacional dá para que os processos usem a CPU. Quando o quantum de um processo termina, mesmo que ele ainda não tenha terminado a execução de suas instruções, o contexto dele é trocado, é salvo na sua pilha de execução onde ele parou, e então ele volta para o estado de “pronto”, até que o sistema operacional por meio do seu

tarefa em execução perde o processador e volta para uma fila de tarefas “prontas”, que estão na memória aguardando sua oportunidade de executar. O mecanismo que permite a retirada de um recurso (o processador, por exemplo) de uma tarefa, é chamado de preempção, ou algoritmos preemptivos. (TANENBAUM e WOODHULL, 2006).

Em algoritmos preemptivos, após a interrupção de uma tarefa, ocorre a intitulada troca de contexto. Neste evento, o conteúdo dos registradores é salvo e a memória utilizada pelo processo concede a outra tarefa o privilégio de executar no processador; restaurando assim o contexto desta última (TANENBAUM e WOODHULL, 2006).

O diagrama apresentado na Figura 4 é conhecido na literatura como “Ciclo de vida das tarefas”. Nele, é possível observar o controle de estados em função da preempção e da troca de contexto a elas imposta (MAZIERO, 2017).

Figura 4. Estados de uma tarefa em um sistema de tempo compartilhado



Fonte: adaptado de (MAZIERO, 2017)

2.2.2 Processos

Um processo pode ser definido como um conjunto dos recursos, ou container, alocados a uma tarefa para sua execução. Assim, para cada tarefa ativa em um software, um conjunto de recursos para executar e cumprir seu objetivo é necessário, além de seu próprio código (TANENBAUM e WOODHULL, 2006).

escalonador de processos volte a disponibilizar a CPU para ele, que então volta pro estado de “executando”.

Conforme Tanenbaum (2010), os conceitos de processo e tarefa se confundem, principalmente porque os SOs mais antigos suportavam somente uma tarefa por processo. Mas, quase todos os SOs atuais suportam mais de uma tarefa por processo (RUSSINOVICH e SOLOMON, 2008) (LOVE, 2004).

Os SOs convencionais atuais associam uma tarefa por processo, o que corresponde à execução de um programa sequencial (i.e., um único fluxo de instruções dentro do processo). Caso se deseje associar mais tarefas ao mesmo processo, o desenvolvedor deve escrever o código necessário para isto (MAZIERO, 2017).

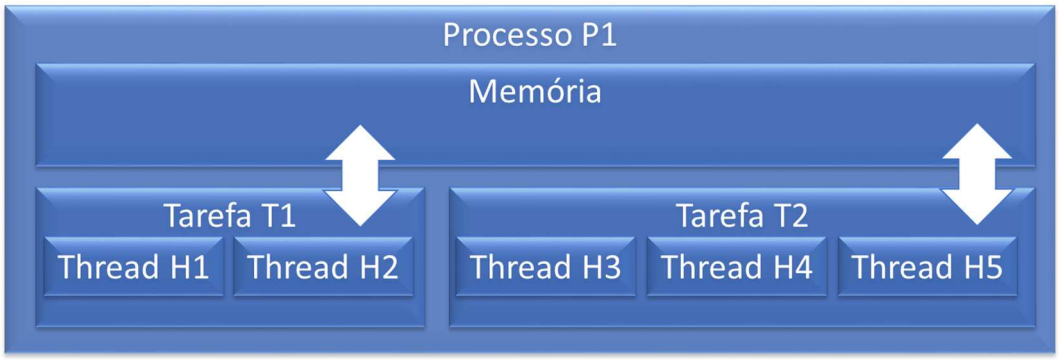
Processos são isolados entre si, inclusive, por meio de mecanismos de proteção em nível de hardware. Neste âmbito, processos não compartilham memória, possuem níveis de operação e a lista de chamadas do SO a serem executadas. Como os recursos são atribuídos aos processos, as tarefas fazem o uso deles a partir do processo. Dessa forma, uma determinada tarefa de um processo não consegue acessar um recurso (a memória, por exemplo) de uma tarefa de outro processo (TANENBAUM, 2010).

2.2.3 Threads

De forma geral, cada fluxo de execução do sistema associado a um processo é denominada *thread* (TANENBAUM e WOODHULL, 2006). Os processos podem ter uma série de *threads* associadas. Estas *threads* são conhecidas como *threads* de usuário (em inglês, *User Level Thread* - ULT). Uma *thread* é uma linha de execução dentro de um processo. Cada *thread* tem o seu próprio estado de processador e a sua própria pilha, mas compartilha a memória atribuída ao processo com as outras *threads* associadas ao mesmo processo (TANENBAUM, 2010).

A Figura 5 apresenta um diagrama exemplificando um processo **P1** que foi atribuído a duas tarefas **T1 e T2.** Estas**,** por sua vez, estão utilizando as threads **H1** e **H2**, e **H3**, **H4** e **H5** respectivamente às tarefas **T1** e **T2.**

Figura 5. Relacionamento entre Processo, Tarefa e Thread



Fonte: adaptado de (TANENBAUM, 2010)

Existem *threads* controladas pelo sistema operacional, executadas em modo-kernel (em inglês, *Kernel Level Thread* – KLT[8]). Por não serem foco deste trabalho, não será detalhado este tipo de *thread*.

2.2.4 Concorrência e paralelismo

Ainda que intimamente ligados, os conceitos de concorrência e paralelismo são distintos. De maneira geral, concorrência refere-se a gerenciar várias atividades ao mesmo tempo, enquanto paralelismo trata-se de executar várias atividades ao mesmo tempo (TANENBAUM, 2010). Neste âmbito, um dos criadores da linguagem GO traz uma definição simples, porém bastante esclarecedora.

"Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once." Rob Pike.

"Concorrência diz respeito a lidar com um conjunto de coisas conjuntamente. Paralelismo diz respeito a fazer um conjunto de coisa conjuntamente". Tradução livre da frase de Rob Pike, citada acima.

Concorrência diz respeito à disputa por recursos do sistema de forma sequencial por um conjunto de processos. Recursos como, memória e tempo de processamento

[8] KLT ou *Kernel Level Threads* são *threads* implementadas no núcleo (*kernel*) dos sistemas operacionais para controlar atividades internas que o sistema operacional precisa executar/cuidar.

são demandados por tarefas atribuídas a processos. Estes processos, por sua vez, são atendidos por um escalonador do sistema operacional, que se utiliza de diversas técnicas de abstração e priorização para fazer os atendimentos de forma a melhor aproveitar os recursos disponíveis. (TANENBAUM, 2010).

Paralelismo, por outro lado, diz respeito à execução paralela de tarefas, ou seja, se refere ao número de tarefas executadas de maneira simultânea, que depende da quantidade de núcleos do processador. Quanto mais núcleos, mais tarefas paralelas podem ser executadas. Paralelismo lida com linhas de execuções (threads) que são executadas simultaneamente com tarefas em um processo (TANENBAUM, 2010).

É possível fazer uma analogia destes conceitos com uma praça de pedágio. Neste exemplo, os vários carros atravessam a praça de pedágio de maneira simultânea, isto é, os carros atravessam o pedágio em paralelo. Em cada guichê, porém, os carros aguardam na fila para serem atendidos, isto é, os carros estão concorrendo para passar pelo guichê. Importante observar que, mesmo havendo paralelismo (entre guichês), existe a concorrência (fila de carros em cada guichê).

Nem sempre um recurso está sempre disponível apenas para uma linha de execução. Em um Sistema Operacional (SO), quem gerencia o que cada núcleo vai executar em dado momento do tempo é o seu escalonador de processos. Na verdade, tarefas e processos concorrentes são base para o projeto e implementação de sistemas multitarefas (i.e., concorrentes) e para uso de *multicore* (i.e., paralelos) (TANENBAUM, 2010).

Contudo, melhorar o desempenho de software na execução de programas não depende somente de SOs e seus escalonadores sendo executados em arquiteturas paralelas como *multicore*. Naturalmente, há também uma dependência de estes softwares serem elaborados com desacoplamento tal que permitam explorar estes recursos e, desta forma, terem seu desempenho melhorado com o uso do paralelismo e concorrência (AMDAHL, 1967).

Amdahl (1967) foi quem primeiro apresentou um argumento que afirmava que o máximo de melhora de um algoritmo correspondia ao maior fluxo não paralelizável, não importando o número de núcleos que lhe fosse disponibilizado para processamento paralelo. Esta melhora de desempenho, (do inglês, *speedup*) poderia ser, então, calculada uma vez conhecendo os tamanhos destes algoritmos não paralelizáveis (AMDAHL, 1967).

Segundo Amdahl, o *speedup* que pode ser alcançado usando *N* processadores pode ser calculado pela seguinte fórmula:

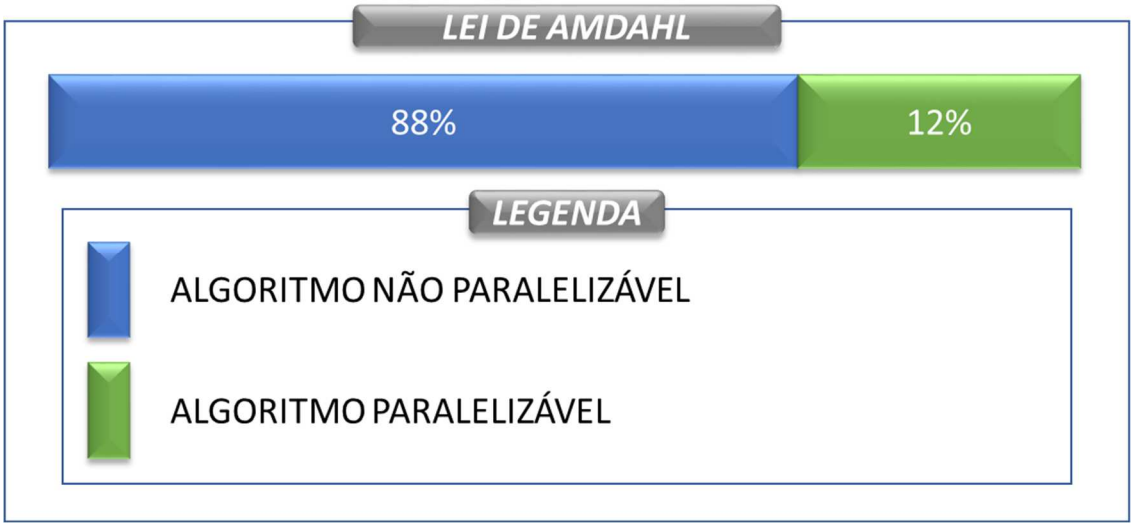
$$S(N) = \frac{1}{(1 - \mathrm{P}) + \frac{P}{N}}$$

Na fórmula dada, *P* representa a proporção de um programa que pode ser executado paralelamente. Portanto, ao tender-se *N* ao infinito, no limite esta fórmula pode ser reduzida para a seguinte fórmula para a determinação do *speedup* máximo teórico:

$$S(N) = \frac{1}{(1 - \mathrm{P})}$$

A fórmula fica mais entendível aplicando-se a um exemplo prático. A Figura 6 exemplifica este tipo de cálculo no qual um determinado algoritmo possui 88% não paralelizável. O argumento de Amdahl infere, portanto, que a melhora estará limitada à redução da parte paralelizável do algoritmo, ou seja, os 12% restantes. Portanto o s*peedup* máximo teórico este para o exemplo da Figura 6 será de = 1/(1 - 0,12) = 1,136 vezes. Este argumento ficou conhecido como “lei de Amdahl” (AMDAHL, 1967).

Figura 6. Exemplo de *speedup* máximo teórico de um algoritmo



Fonte: adaptado de (AMDAHL, 1967)

Portanto, segundo Amdahl, quanto menor for a parte não paralelizável de um algoritmo, maior seu potencial de *speedup* em arquiteturas paralelas. Falando de uma outra forma, softwares planejados com os seus algoritmos leves e desacoplados, ou seja, paralelizáveis, seriam naturalmente beneficiados em arquiteturas *multicore*. Será

discutido em mais detalhes na seção 2.7 algumas barreiras para o desenvolvimento de software paralelizável.

Além das questões postas aqui, uma consequência da aplicação do paralelismo de algoritmos para melhoria de desempenho de aplicações, foi a necessidade de comunicação entre estas partes paralelizáveis. Estas partes paralelizáveis são conhecidas como tarefas de processos. Esta demanda de comunicação traz, então, problemas de acesso à recursos compartilhados que precisam ser devidamente gerenciados (TANENBAUM, 2010).

Em tempo, as tarefas de um processo podem trocar informações com facilidade, pois compartilham a mesma área de memória. No entanto, tarefas de processos distintos não conseguem essa comunicação facilmente, pois estão em áreas diferentes de memória. Esse problema é resolvido com um grupo de mecanismos conhecido como intercomunicação entre processos (do inglês, *Inter-Process Communication* - IPC) (TANENBAUM, 2010), tema focal a ser detalhado na seção a seguir.

2.3 COMUNICAÇÃO ENTRE PROCESSOS

A seção 2.2 apresentou como os sistemas operacionais modernos distribuem as tarefas de forma concorrente permitindo assim, um aproveitamento dos recursos *multicore.* Entretanto, como já comentado, a necessidade de comunicação entre processos é uma consequência natural. Nesta seção, portanto, são apresentados os principais problemas de comunicação entre processos em um ambiente *multicore*. Por fim, são apresentadas técnicas computacionais para seu correto gerenciamento.

2.3.1 Problemas de comunicação entre processos

Segundo Tanenbaum (2010), muito resumidamente, há três tópicos em relação à comunicação entre processos que devem ser abordados: o primeiro refere-se a formas de passagem de informação entre processos; o segundo discute como garantir que dois ou mais processos não entrem em conflito, por exemplo, dois processos em um sistema de reserva de linha aérea, cada qual tentando conseguir o último assento do avião para clientes diferentes; por fim, o terceiro está relacionado com uma

sequência adequada quando existem dependências. Por exemplo, se o processo *A* produz dados e o processo *B* os imprime, *B* deve esperar até que *A* produza alguns dados antes de iniciar a impressão (TANENBAUM, 2010).

Os problemas que serão discutidos a seguir, o serão sob o contexto de processo, contudo, é importante reforçar que os mesmos problemas e soluções também se aplicam às *threads* (TANENBAUM, 2010).

Condições de corrida

Segundo Tanenbaum (2010), o problema conhecido como "Condições de corrida" (em inglês, *race conditions)* se dá quando dois ou mais processos estão lendo ou escrevendo algum dado compartilhado e cujo resultado final depende de quem executa precisamente e quando. A depuração de programas que contenham condições de corrida é bastante trabalhosa, pois os resultados dos testes, em sua grande maioria, não apresentam problemas. Contudo, em um raro momento algo incorreto ocorre e sua causa é exatamente a condição de corrida que não seguiu o esperado em função das trocas de contexto entre processos.

Figura 7. Dois processos acessando a memória compartilhada simultaneamente

Fonte: (TANENBAUM, 2010)

Um exemplo do problema deste tipo pode ser visto na Figura 7. Esta, representa um *Spool* de impressão o qual as vagas 1 a 3, 7 e 8 estão vazias. As variáveis *out* e *in* representam, respectivamente, o próximo arquivo a ser impresso e a próxima vaga

livre a ser preenchida no *Spool.* Se o processo *A* ler a variável *in* recebendo o valor *7* e seu processamento for interrompido pelo fim do seu *quantum* (vide subseção 2.2.1) e o Processo *B* iniciar sua execução em seguida fazendo a mesma leitura da variável *in*, este irá colocar seu arquivo de impressão posição 7 e ajustará o valor *in* para a próxima vaga livre, ou seja, *8*. Quando o processo *A* retornar para o contexto, irá sobrescrever o arquivo escrito pelo processo *B* na vaga *7* fazendo com que a impressão solicitada pelo processo *B* nunca ocorra.

Regiões críticas

O problema apresentado na subseção anterior foi um caso de 'região crítica' (em inglês, *critical region*) ou 'seção crítica' (em inglês, *critical section*). Segundo Tanenbaum (2010), uma região crítica é uma parte do programa na qual há acesso à memória compartilhada. A resposta para evitar esse problema é encontrar algum modo de impedir que mais de um processo leia e escreva ao mesmo tempo neste tipo de recurso. Em outras palavras, é necessária uma exclusão mútua (em inglês, *mutual exclusion)* (TANENBAUM, 2010).

Para que processos paralelos cooperem de forma correta e eficiente usando recursos compartilhados, são necessárias quatro condições para se alcançar uma solução que garanta coesão dos dados compartilhados, a saber (TANENBAUM, 2010):

- Dois processos nunca podem estar simultaneamente na mesma região crítica.
- Nada pode ser afirmado sobre a velocidade ou sobre o número de CPUs.
- Nenhum processo executando fora de sua região crítica pode bloquear outros processos
- Nenhum processo deve esperar eternamente para entrar em sua região crítica.

Resumidamente, então, o problema de comunicação entre processos está no acesso às regiões críticas de forma a garantir que as condições de corrida não afetem a coesão dos dados compartilhados. A seguir, serão apresentadas alternativas para a realização da exclusão mútua de forma a garantir um acesso saudável aos recursos compartilhados evitando, assim, problemas como os aqui vistos.

2.3.2 Solução de Peterson

Em 1981, G. L. Peterson propôs um algoritmo para exclusão mútua que permitiu que dois ou mais processos compartilhassem recursos em suas regiões críticas sem conflitos. O respectivo algoritmo pode ser visto de maneira simplificada na Figura 8.

Figura 8. Solução de Peterson para exclusão mútua em C

```
#define FALSE 0
#define TRUE  1
#define N     2

int turn;
int interested[N];

void enter_region(int process);
{
    int other;
    other = 1-process;
    interested[process] = TRUE;
    turn = process;
    while(turn==process && interested[other]==TRUE)
}

void leave_region(int process);
{
    interested[process] = FALSE;
}
```

Fonte: (TANENBAUM, 2010)

O algoritmo consiste em duas rotinas: *enter_region* e *leave_region* que devem ser chamadas, respectivamente, antes de entrar na região crítica e antes de sair da região crítica. Cada processo chama cada uma das rotinas de controle com seu próprio identificador de processo como parâmetro. (PETERSON, 1981). Desta forma, dois processos chamam *enter_region* quase simultaneamente. Ambos armazenarão seus números de processo na variável *turn*. O que armazenou por último é o que conta (o primeiro é sobreposto e perdido). Quando ambos os processos chegam ao comando *while*, o processo que armazenou por último não executa o laço e entra em sua região crítica, enquanto o outro processo executa o laço, aguardando sua vez

para entrar na região crítica que será liberada quando o processo executando na região crítica executar a instrução *leave_region* (TANENBAUM, 2010).

2.3.3 Semáforo

A solução de Peterson está correta no sentido de garantir o acesso exclusivo à uma região crítica. Entretanto, quando um processo aguarda para entrar em uma região crítica, fica em um laço esperando até que lhe seja permitida a entrada. Este tipo de comportamento chama-se 'espera ociosa' e acaba consumindo tempo de processamento (TANENBAUM, 2010).

Dijkstra (1965) propôs um algoritmo para acesso a recursos compartilhados que introduziu duas operações: *down* e *up*, generalizações para adormecer e acordar processos, respectivamente. Este algoritmo valia-se de uma variável identificada como *semáforo*, cujo valor seria um número inteiro. A operação *down* sobre um semáforo verifica se seu valor é maior que 0. Se for, decrementará o valor (isto é, gasta um sinal de acordar armazenado) e prosseguirá. Se o valor for 0, o processo será posto para dormir, sem terminar o *down*. Verificar o valor, alterá-lo e possivelmente colocá-lo para dormir são tarefas executadas de forma atômica[9]. Esta atomicidade é absolutamente essencial para resolver problemas de sincronização e evitar condições de corrida (TANENBAUM, 2010).

2.3.4 Mutexes

Quando não é preciso usar a capacidade do semáforo de contar, lança-se mão de uma versão simplificada do algoritmo do semáforo, a este algoritmo dá-se o nome de *mutex*, abreviação para *mutual exclusion.* Mutexes são adequados apenas para gerenciar a exclusão mútua de algum recurso ou parte de código compartilhada. São fáceis de implementar e eficientes, o que os torna especialmente úteis em pacotes de *threads* implementados totalmente no espaço de usuário (TANENBAUM, 2010).

[9] Uma execução atômica é uma execução indivisível, ou seja, garante-se que, uma vez iniciada, nenhum outro processo pode ter acesso ao semáforo até que a operação tenha terminado ou sido bloqueada.

Mutex é uma variável que pode estar em um dos dois estados: impedido e desimpedido. Duas rotinas são usadas com mutexes: *mutex_lock*, quando um processo (ou *thread*) precisa ter acesso a uma região crítica; e *mutex_unlock*, quando o processo (ou *thread*) estiver por sair da região crítica. De certa forma, mutex é muito parecido com a solução de Peterson, porém a diferença se dá pelo seguinte aspecto: quando o processo falha em verificar a variável de trava faz uma chamada ao escalonador de forma a baixar sua prioridade, abrindo mão, assim, de processamento (TANENBAUM, 2010).

2.3.5 Mensagens assíncronas

De acordo com Tanenbaum (2010), diferentemente das soluções apresentadas até agora, que visam gerenciar e coordenar as condições de corrida a uma região crítica, a troca de mensagens assíncronas apresenta uma abordagem completamente diferente. O objetivo desta técnica é evitar regiões críticas, fazendo com que as comunicações ocorram sem qualquer tipo de compartilhamento.

A troca de mensagens consiste na implementação de duas rotinas: a rotina *send* para enviar uma determinada informação a um processo; e a rotina *receive* que trata de processar os dados recebidos pelo processo. Esta última pode ou não conhecer o processo remetente da mensagem (TANENBAUM, 2010).

Van Roy também define que troca de mensagens assíncronas é o estilo natural de um sistema distribuído, ou seja, um conjunto de computadores que podem se comunicar entre si por meio de uma rede. É natural porque reflete a estrutura do sistema e seus custos. Sistemas distribuídos estão se tornando onipresentes devido à expansão contínua da Internet. Tecnologias mais antigas para programação de sistemas distribuídos, como RPC, CORBA e RMI, são baseadas em comunicação síncrona. Novas tecnologias, como serviços da Web, são assíncronas (VAN ROY, 2004, p. 345-346).

Também importante dizer que este tipo de solução se apresenta como parte fundamental na construção de sistemas altamente confiáveis. Como as entidades de transmissão de mensagens são independentes, se uma falhar, as outras podem

continuar a executar. Em um sistema adequadamente projetado, os outros se reorganizam para continuar fornecendo um serviço.

A troca de mensagens é, portanto, um estilo de programação no qual um programa consiste em entidades independentes que interagem enviando mensagens umas às outras assincronamente, ou seja, sem esperar por uma resposta. (VAN ROY, 2004, p. 345-346). Isto posto, a troca de mensagens assíncronas é, também, a estrutura básica para sistemas multiagentes[10]. Um exemplar deste tipo de sistema é o modelo de atores, o qual é apresentado em maiores detalhes na seção a seguir.

2.4 MODELO DE ATORES

Como pode ser visto na seção 2.3, foram apresentadas soluções para os problemas de comunicação entre processos. Em particular, a passagem de mensagens assíncronas que é base para sistemas multiagentes. Nesta seção é apresentado um tipo particular de sistemas multiagentes: o modelo de atores. A seção inicia com um breve histórico, em seguida são apresentados os conceitos fundamentais. Após, então, são detalhadas as formas de interação entre os elementos deste modelo.

2.4.1 Histórico

"O modelo de atores foi motivado pela perspectiva de máquinas de computação altamente paralelas, consistindo em dezenas, centenas ou até mesmo milhares de microprocessadores independentes, cada um com seu próprio processador local de memória e comunicações, comunicando-se através de uma rede de comunicações de alto desempenho". Carl Eddie Hewitt (1973), criador do modelo de atores.

A arquitetura *multicore* ainda dava seus primeiros passos quando Hewitt (1973) apresentou pela primeira vez modelo de atores. Este modelo surgiu como um modelo

[10] Sistemas multiagentes consistem em uma disciplina que vê sistemas complexos como um conjunto de "agentes" em interação. Os agentes são entidades independentes que trabalham em direção a seus próprios objetivos locais. Se a interação for projetada adequadamente, os agentes também poderão atingir metas globais (VAN ROY, 2004, p. 345-346).

matemático[11] em ciência da computação, que trata atores como elementos primitivos universais da computação concorrente. Entretanto, ao contrário dos modelos anteriores de computação, o modelo do ator foi inspirado pela física, incluindo a relatividade geral e a mecânica quântica. Ele também foi influenciado pelas linguagens de programação *Lisp*, *Simula* e versões anteriores do *Smalltalk*, bem como por sistemas baseados em recursos e comutação de pacotes.

Seguindo a publicação de 1973 de Hewitt, a pesquisadora Irene Greif desenvolveu uma semântica operacional[12] para o modelo de ator como parte de sua pesquisa de doutorado (GREIF, 1975). Dois anos depois, Henry Baker e Hewitt publicaram um conjunto de leis axiomáticas para os sistemas de atores (HEWITT e BAKER, 1977). Outros marcos importantes incluem a dissertação de William Clinger (1981), que introduz uma semântica denotacional[13] baseada em domínios de poder e a dissertação de Gul Agha (1985), que desenvolveu ainda um modelo semântico baseado em transição, complementar ao modelo de Clinger. Isso resultou no desenvolvimento completo da teoria do modelo de ator.

2.4.2 Conceitos fundamentais

No modelo de atores, tal como ocorre com conceitos abstratos como objetos do paradigma orientado a objetos, todos os elementos são atores. De maneira bem resumida, um ator é uma entidade computacional que, em resposta a uma mensagem que recebe, pode de maneira concorrente aos demais atores (HEWITT, 1973):

- Enviar um número finito de mensagens para outros atores;

[11] Um modelo matemático é uma representação ou interpretação simplificada da realidade, ou uma interpretação de um fragmento de um sistema, segundo uma estrutura de conceitos mentais ou experimentais.

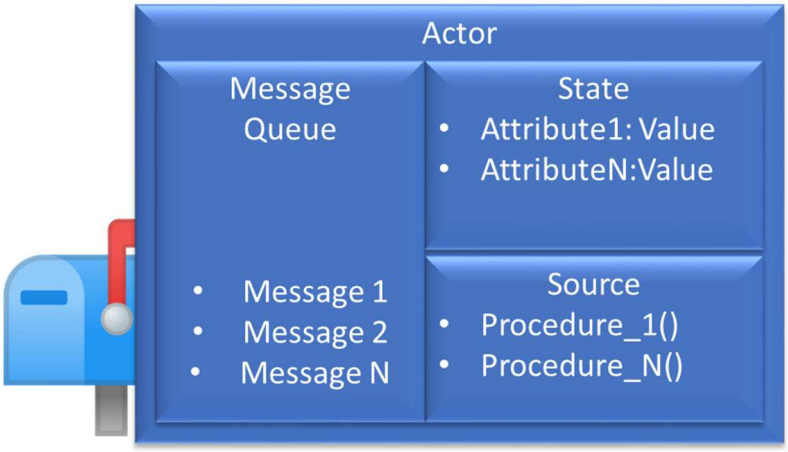
[12] Semântica operacional é uma das abordagens de semântica formal, em que o significado de uma construção da linguagem é especificado pela computação que ela induz quando executada em uma máquina hipotética. A semântica operacional preocupa-se mais em como os programas são executados do que meramente com os resultados destas computações. A semântica formal é uma das áreas de estudo de ciência da computação, preocupada em atribuir significado às construções das linguagens de programação.

[13] Semântica denotacional designa uma abordagem de semântica formal. A semântica formal é uma das áreas de estudo de ciência da computação, preocupada em atribuir significado às construções das linguagens de programação. Nesta abordagem, os significados são modelados por objetos matemáticos, geralmente funções semânticas definidas composicionalmente, que representam o efeito de executar uma estrutura.

- Criar um número finito de novos atores;
- Designar o comportamento a ser usado para a próxima mensagem que receber.

Em uma mensagem, o remetente não é diretamente acoplado a ela. Este, por sua vez, é representado por um endereço. Este desacoplamento entre remetente e destinatário determina um avanço fundamental ao modelo dado, que permite, desta forma, uma comunicação assíncrona e estruturas de controle como padrões de transmissão de mensagens (HEWITT, 1977).

Figura 9. Diagrama para representação de um ator



Fonte: adaptado de (HEWITT, 1977)

A Figura 9 representa um diagrama com as partes de um ator, a saber: a fila de mensagens na qual estão as mensagens a serem processadas pelo ator por ordem de chegada; a parte relativa ao estado, na qual são armazenadas as informações necessárias para controle do ator; a parte do código fonte, na qual há a lógica para processar as mensagens.

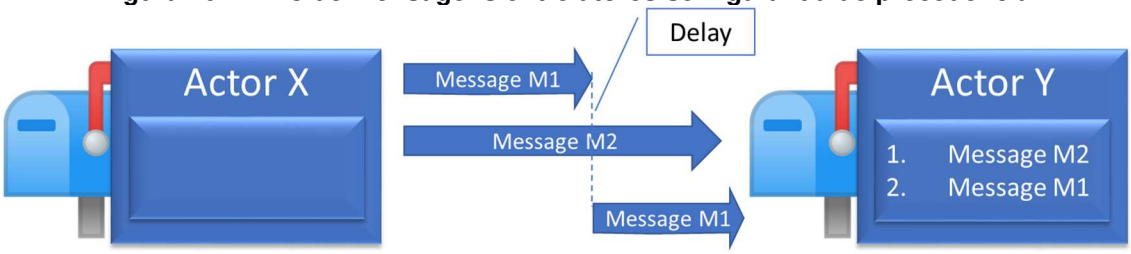
Em suma, o modelo de ator é caracterizado pela concorrência inerente de computação dentro e entre os atores, criação dinâmica de atores, inclusão de endereços de atores em mensagens e interação apenas por meio da passagem direta de mensagens assíncronas sem restrição à ordem de chegada da mensagem (HEWITT, 1977).

2.4.3 Semântica de passagem de mensagens

A passagem de mensagens é a essência do modelo de atores. Pode-se dizer, portanto, que modelo de atores é definido sobre a semântica da passagem de mensagens. A seguir, são apresentados os principais postulados da semântica de passagem de mensagens do modelo de atores.

Ordem de Precedência

Figura 10. Envio de mensagens entre atores sem garantia de precedência



Fonte: (HEWITT e BAKER, 1977)

No modelo de atores não há garantias de ordem de chegada, ou seja, se um ator X enviou uma mensagem M1 para um ator Y, e mais tarde X enviou outra mensagem M2 para Y, não há exigência de que M1 chegue em Y antes de M2 (HEWITT e BAKER, 1977). A Figura 10 apresenta um exemplo de envio de mensagens, no qual, a mensagem M1 sofreu um atraso. Desta forma não houve uma garantia da precedência.

Portanto, se a ordem das mensagens de saída for desejada, ela poderá ser modelada por um ator de fila que forneça essa funcionalidade. Este ator de fila deverá, então, enfileirar as mensagens que chegavam para que pudessem ser recuperadas na ordem FIFO ou através de indexação das mensagens recebidas.

Processamento das mensagens

Uma ação no modelo de atores pode ser interpretada como um objeto ator enviando mensagens para outros atores, chamados de destinatários dessas mensagens. O recebimento destas mensagens, portanto, dispara um evento de processamento. Portanto, um evento *E* é o recebimento da mensagem *Message(E)*

pelo ator *Target(E)*. Após o recebimento desta mensagem no evento E, o destinatário consulta seu código fonte e usa seu estado local atual e a mensagem como parâmetros. Como resultado o ator pode, então: enviar novas mensagens para outros atores, calcular um novo estado local para si próprio e/ou criar atores (HEWITT e BAKER, 1977).

A caixa de entrada

Todas as mensagens enviadas são eventualmente recebidas por seus destinos. Se muitas mensagens estiverem na rota para um único destinatário, a chegada simultânea delas poderá sobrecarregar o ator ou fazer com que as mensagens interfiram umas nas outras. Portanto, para resistir a essas possibilidades e garantir um tipo de princípio de superposição de mensagens, Hewitt postulou um árbitro na frente de cada ator, o que permite que apenas uma mensagem seja recebida por vez (HEWITT e BAKER, 1977).

Inicialmente postulado sob a forma genérica de árbitro, literaturas mais recentes apresentam esta definição como uma caixa de entrada (do inglês, *mailbox*). Esta definição contempla todas as premissas postuladas por Hewiit em 1977, porém, com a vantagem de ser um elemento mais claro e intuitivo para o entendimento deste postulado (IMAM e SARKAR, 2014), (BAUDE e VIDAL-NAQUET, 1991) (AGHA, 1985).

2.4.4 Escopo e localidade

Conforme já relatado, no modelo de atores as informações são transmitidas somente através de mensagens. Contudo, um ator A pode enviar uma mensagem para outro ator B apenas se souber sobre B, ou seja, se souber o nome de B. No entanto, um ator não pode conhecer o nome de um ator, a menos que tenha sido criado com esse conhecimento ou adquirido como resultado de uma mensagem. Além disso, um ator não pode enviar uma mensagem para outro ator transmitindo nomes que ele não conhece (HEWITT e BAKER, 1977).

Os endereços que identificam um ator são chamados de "endereço para correspondência". Um ator pode receber o endereço de outro ator a partir de uma

mensagem, ou pode conhecer o endereço por ter sido o ator que o criou. Desta forma, como é possível concluir, não há mensagens do tipo “para todos” (do inglês, *broadcast)* no modelo de atores (HEWITT, 1977).

Atores conhecidos

Um ator recebe um vetor inicial finito de conhecidos quando é criado. Cada elemento desse vetor inicial deve ser um participante do evento de criação do ator. Intuitivamente, um ator pode inicialmente saber nada mais que seus pais, conhecidos de seus pais e seus irmãos. Esse vetor de conhecimento pode mudar como resultado das mensagens que o ator recebe; quando um ator recebe uma mensagem, pode adicionar ao seu vetor conhecido qualquer nome mencionado na mensagem, sendo que cada mensagem pode mencionar apenas alguns nomes finitos. Também é permitido esquecer conhecidos a qualquer momento. No pior dos casos (para armazenamento), um ator poderia lembrar os nomes de todos os atores que conhecia quando criados, bem como todos os nomes que já foram contados em uma mensagem (HEWITT e BAKER, 1977).

2.4.5 Solução para problema de comunicação entre processos

Van Roy define o modelo de atores como uma extensão do modelo concorrente declarativo, ou seja, o modelo de mensagens estende o modelo declarativo concorrente, adicionando apenas um novo conceito: um canal de comunicação assíncrono. Isso significa que qualquer cliente pode enviar mensagens pelo canal de comunicação a qualquer momento e o servidor pode ler todas as mensagens deste canal (VAN ROY, 2004).

Isto posto, o modelo de atores se apresenta como uma forma de implementação para a solução de problemas de comunicação concorrente conhecida como troca de mensagens assíncronas, apresentada na subseção 2.3.5. (VAN ROY, 2004). Entretanto, esta modelagem traz mais uma inovação: isolar o estado, isto é, os atores não compartilham o estado entre si, mas trocam informações usando mensagens

assíncronas. Este estado isolado significa que os atores geram o mínimo de interrupção nos processos concorrentes (ARMSTRONG, 2013).

Todos estes atributos apresentados pelo modelo de atores inspiraram a produção de várias linguagens de programação compatíveis a ele. Um exemplar é a linguagem Erlang, que surgiu de dentro dos laboratórios da Ericsson para atender demandas da telefonia – um ambiente concorrente por natureza (ARMSTRONG, 2007). A linguagem Erlang é, portanto, o tema focal da seção a seguir.

2.5 ERLANG

A seção 2.4 apresentou o modelo de atores como uma proposta de implementação para mensagens assíncronas, que traz, ainda, uma inovação referente ao isolamento do estado de cada ator. A presente seção, por sua vez, apresenta a linguagem Erlang, a qual é baseada neste modelo. Nesta seção é apresentado um breve histórico da linguagem, bem como suas principais características no tocante à programação concorrente e os avanços em plataformas *multicore*.

2.5.1 Histórico

Em meados da década de 1980, o Laboratório de Ciência da Computação da Ericsson recebeu a tarefa de investigar linguagens de programação adequadas para programar a próxima geração de produtos de telecomunicações. Joe Armstrong, Robert Virding e Mike Williams, sob a supervisão de Bjarne Däcker, passaram dois anos prototipando aplicativos de telecomunicações com todas as linguagens de programação disponíveis da época. A conclusão deles foi que, embora muitas das linguagens tivessem características interessantes e relevantes, nenhuma linguagem individual englobava todas elas. Como resultado, eles decidiram inventar uma nova linguagem (CESARINI e THOMSON, 2009).

Projetada para fornecer uma maneira melhor de programar aplicativos de telefonia, Erlang teve em ambientes concorrentes o seu ponto forte, pois os aplicativos de telefonia são altamente concorrentes por natureza. Por exemplo, um único *switch* deve manipular dezenas ou centenas de milhares de transações simultâneas (ARMSTRONG, 2007). Isto posto, Erlang sofreu influência de linguagens cuja alguma

característica pudesse contribuir com concorrências, como as linguagens funcionais ML e Miranda, linguagens concorrentes ADA, Modula e Chill e, ainda, a linguagem de programação lógica Prolog (CESARINI e THOMSON, 2009).

Sua primeira versão, ainda prototipal, foi disponibilizada em 1986 e era executada em uma máquina virtual baseada em Prolog. Esta versão foi usada durante os quatro anos enquanto Mike Wiliams escrevia uma nova máquina virtual baseada em C dentro dos laboratórios da Ericsson. Um ano depois de liberada a VM em C, em 1991, o primeiro projeto comercial com uma pequena equipe de desenvolvedores foi lançado. O projeto era um servidor de mobilidade, permitindo que os usuários de telefones sem fio da DECT[14] passassem pelas redes de escritórios privados. A linguagem foi ganhando espaço dentro da Ericsson no decorrer dos anos até que, em 1994, já estava praticamente em todos os DECTs da empresa. Durante o ano que se seguiu, recebeu melhorias e novos recursos a partir de *feedbacks* de seus usuários. Foi então que a linguagem foi considerada madura o suficiente para ser usada em grandes projetos com centenas de desenvolvedores, incluindo as soluções de comutação de banda larga, GPRS e ATM da Ericsson (ARMSTRONG, 2007).

Em conjunto com esses projetos, o *framework* OTP[15] foi desenvolvido e lançado em 1996. O OTP fornece um conjunto de ferramentas para estruturar sistemas Erlang, oferecendo robustez e tolerância a falhas, juntamente com um conjunto de bibliotecas. A história do Erlang é importante para entender sua filosofia. Embora muitas linguagens tenham sido desenvolvidas antes de encontrar seu nicho, Erlang foi desenvolvido para resolver os requisitos de *time-to-market*[16] de sistemas em tempo real distribuídos, tolerantes a falhas, massivamente concorrentes e em tempo real (CESARINI e THOMSON, 2009).

Sistemas como plataformas web, bancos, serviços de voz sobre IP, sistemas de mensagens e sistemas de integração empresarial são alguns exemplos de sistemas que compartilham os mesmos requisitos de concorrência dos sistemas de

---

[14] *DECT* (*Digital Enhanced Cordless Telecommunications*), é uma norma *ETSI* (*European Telecommunications Standards Institute*) muito utilizada em telefones portáteis. Os postos de conversação DECT utilizam comunicação digital sem fios. A norma *DECT* também pode ser utilizada para comunicações sem fios de dados digitais.

[15] OTP vem do acrônimo inglês *Open Telecom Platform.* Apesar de ter sido originalmente concebida como uma plataforma para *Telecom* como o nome sugere, atualmente ela já não é mais específica para este mercado e se tornou uma plataforma de propósito geral, no entanto o nome se manteve através dos anos.

[16] *Time-to-market* (TTM) é o tempo que leva de um produto que está sendo concebido até que ele esteja disponível para venda. O TTM é importante em indústrias onde os produtos são ultrapassados rapidamente.

telecomunicações, este fato explica por que a Erlang está avançando nesses setores. A Ericsson tomou a decisão de lançar o Erlang como código aberto em dezembro de 1998 usando a licença EPL, um derivativo da Licença Pública Mozilla. Isso foi feito sem orçamento ou *press releases*, nem com a ajuda do departamento de marketing corporativo. Em janeiro de 1999, o site erlang.org tinha cerca de 36.000 impressões de páginas. Dez anos depois, esse número subiu para 2,8 milhões. (CESARINI e THOMSON, 2009).

2.5.2 Processos Erlang

Conforme mencionado, as aplicações Erlang são executadas em VM própria. Esta arquitetura, então, permite um controle fino de seus fluxos de execução. Estes fluxos de execução são conhecidos na literatura como “Processos Erlang” (CESARINI e THOMSON, 2009). Porém, faz-se a ressalva de não os confundir com processos de SO, que foram apresentados na subseção 2.2.2.

Isto posto, um Processo Erlang, é leve em se comparado aos processos e *threads* de sistemas operacionais vistos na seção 2.2. Este fato se dá, porque Erlang não cria uma nova *thread* do sistema operacional para cada um de seus processos, mas sim as controla diretamente na sua máquina virtual (VM). Processos Erlang são criados, programados e manipulados na VM, independentemente do sistema operacional em que a VM esteja operando. Como resultado, o tempo de criação do processo é da ordem de microssegundos e independente do número de processos existentes simultaneamente. (CESARINI e THOMSON, 2009).

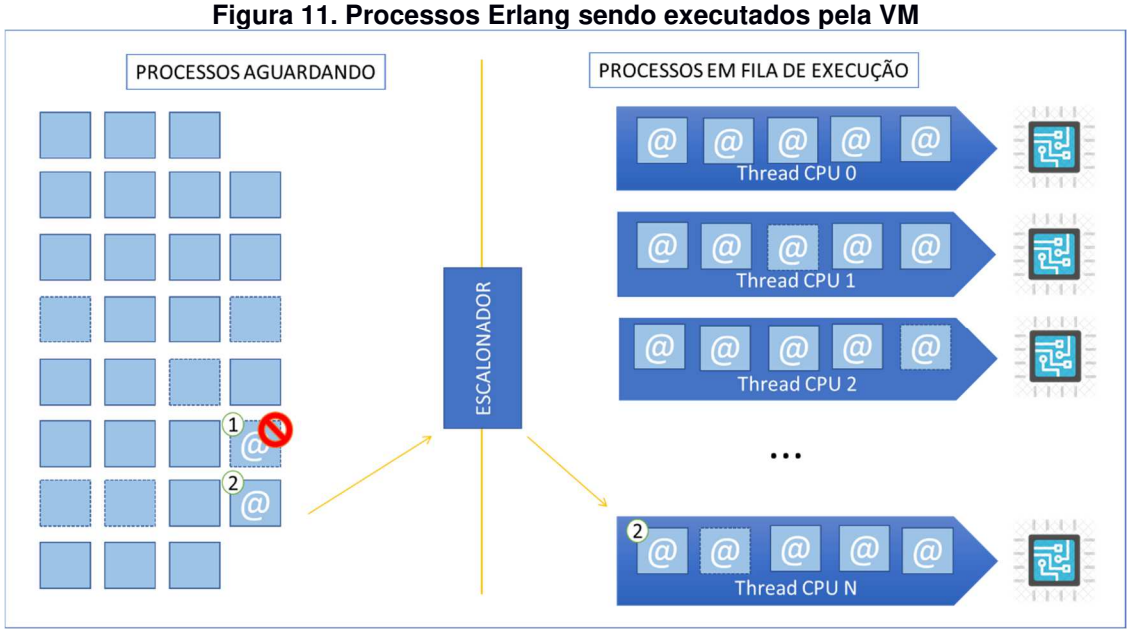
Um processo Erlang recém-criado usa 309 bytes de memória no emulador sem suporte SMP. Em tempo, o suporte SMP depende do ambiente de execução e é adicionado a este tamanho. Cada processo Erlang é executado em seu próprio espaço de memória e possui seu próprio *heap*[17] e *stack*[18], ambos controlados e

[17] O *Heap*, ou área de alocação dinâmica, é um espaço reservado para variáveis e dados criados durante a execução do programa.

[18] A *stack*, é pilha de execução funções também é uma área disponibilizada dentro do espaço de endereçamento do processo. Essa área funciona como uma estrutura de dados LIFO (*last in first out*). Quando uma função é chamada durante a execução de um programa, um bloco de memória é empilhado no topo da pilha de funções. Nesse bloco existem referências para todas as variáveis criadas ou apontadas dentro da função chamada. Ao término da execução da função, esse bloco é desempilhado/desalocado.

gerenciados pela VM. Isto posto, a criação de processos Erlang torna-se rápida e escalável (CESARINI e THOMSON, 2009).

Figura 11. Processos Erlang sendo executados pela VM



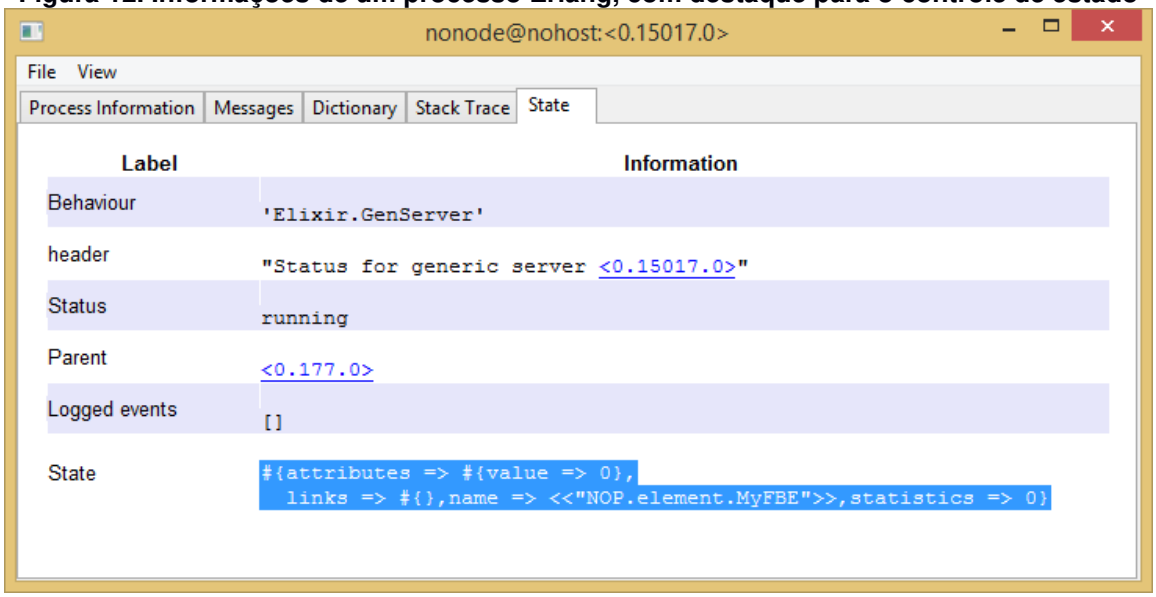
Fonte: adaptado de (CESARINI e THOMSON, 2009)

Em Erlang, a máquina virtual cria uma *thread* do SO por processador (ou núcleo). Esta thread é responsável por executar os processos Erlang que são direcionados a ela de forma balanceada por um escalonador próprio (do inglês, *scheduler*). A Figura 11 apresenta o relacionamento entre processos Erlang e *threads* do SO. O processo *1*, apesar de possuir uma mensagem a ser processada, está impedido de ser executado, i.e., aguardando o retorno de outro processo. O processo *2* está apto a ser processado e foi direcionado para a fila de execuções conforme decidido pelo escalonador.

A linguagem Erlang, por ser um exemplar do paradigma funcional[19], não possui suporte a estados nomeados. Esta deficiência da linguagem também é resolvida pelo processo Erlang. Processos Erlang possuem um suporte para armazenamento de estado (CESARINI e THOMSON, 2009). A Figura 12 apresenta as informações do estado de um processo Erlang extraído da funcionalidade *Observer*, esta nativa em versões mais recentes da VM Erlang.

[19] Detalhes sobre paradigma funcional e a taxonomia dos paradigmas será abordada em detalhes na seção 2.7.

Figura 12. Informações de um processo Erlang, com destaque para o controle de estado



Fonte: Autoria própria

B*enchmarks* que comparam os modelos de concorrência Erlang com seus correspondentes em C# ou Java são de melhor magnitude, especialmente ao aumentar o número de processos concorrentes (CESARINI e THOMSON, 2009) (GARCIA, BLANCO, *et al.*, 2007) (VINOSKI, 2007).

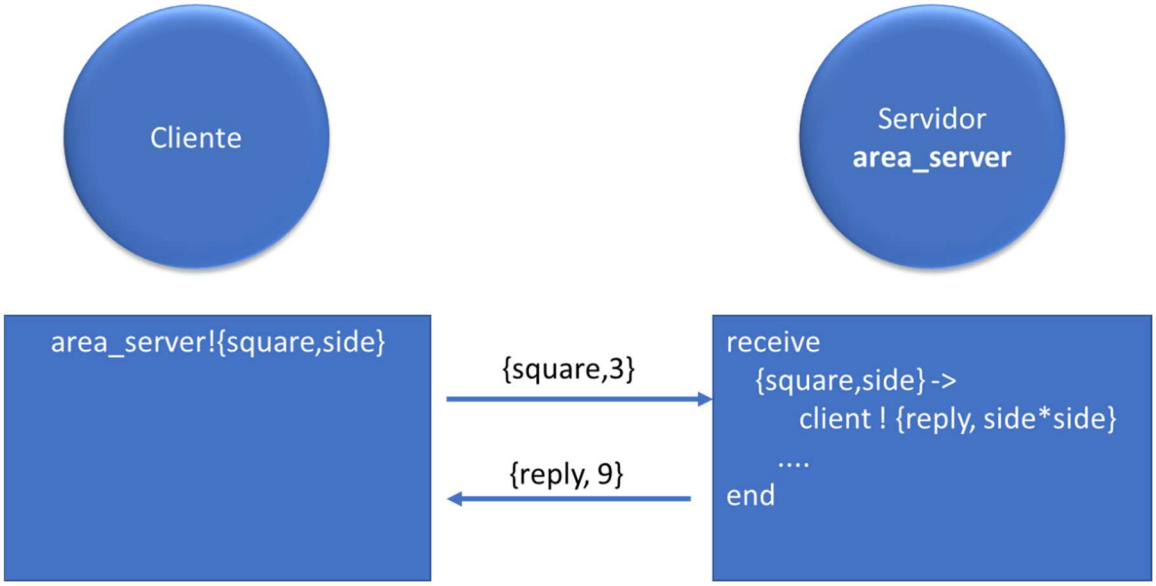
2.5.3 Concorrência

Concorrência em Erlang é fundamental para o seu sucesso. Como anteriormente visto, os processos Erlang possuem sua própria área de memória (do inglês, *heap) e*m vez de fornecer processos que compartilham memória. Desta forma, os processos não podem interferir uns com os outros inadvertidamente, como é muito fácil em modelos de *threading*, levando a *deadlocks* e outros problemas comuns à programação concorrente.

Processos se comunicam entre si através de passagem de mensagens, na qual a mensagem pode ser qualquer valor de dados Erlang. A passagem de mensagens é assíncrona, portanto, uma vez que uma mensagem é enviada, o processo pode continuar o processamento. As mensagens são recuperadas da caixa de correio (*mailbox)* do processo seletivamente, portanto, não é necessário processar mensagens na ordem em que são recebidas. Isso torna a simultaneidade mais

robusta, principalmente quando os processos são distribuídos entre diferentes computadores e a ordem em que as mensagens são recebidas dependerá das condições da rede ambiente. A Figura 13 mostra um exemplo no qual um processo de “servidor de área” calcula áreas de formas para um cliente em um modelo alto nível de construção (CESARINI e THOMSON, 2009).

Figura 13. Comunicação entre processos



Fonte: adaptado de (CESARINI e THOMSON, 2009)

As propriedades alcançadas com a arquitetura modelada a atores, que capacitaram inicialmente o Erlang para uma programação distribuída e tolerante a falhas, também o capacitaram para o paralelismo em *multicore*. Em virtude disto, iniciaram-se estudos e desenvolvimentos da máquina virtual Erlang visando o aproveitamento do paralelismo em ambientes *multicore* (CESARINI e THOMSON, 2009).

2.5.4 Suporte a *Multicore*

O modelo Erlang para concorrência naturalmente se transfere para processadores *multicore* de forma transparente. Desta forma, é possível transferir programas Erlang para hardwares mais potentes sem precisar reescrevê-los. Para alcançar esta escalabilidade de forma transparente, entretanto, não foi um processo simples. O suporte a multiprocessamento simétrico (SMP) em Erlang vem sendo

aprimorado na sua VM desde o final da década de 1990, sendo atualmente parte integrante da versão padrão.

O desafio da equipe de desenvolvimento da Erlang / OTP na Ericsson vem sendo, ao longo do tempo, fazer o SMP funcionar, medir seu desempenho, encontrar os gargalos e melhorar seu desempenho constantemente. Desde o lançamento da primeira versão habilitada para SMP de Erlang, esta tem sido a sua abordagem. Em versões recentes, o modelo de máquina virtual evoluiu de uma única fila de execução monolítica para uma fila de execução para cada processador, garantindo que a fila de execução não seja mais um gargalo para o sistema, conforme ilustrado na Figura 11. À medida que surgem processadores mais complexos, o sistema de tempo de execução poderá evoluir com eles sem que seja necessário reescrever seus programas (CESARINI e THOMSON, 2009).

O objetivo do trabalho em SMP pelo time de desenvolvimento Erlang é torná-lo transparente para os desenvolvedores. Estes devem desenvolver e estruturar seu código como sempre fizeram, usando a concorrência sem precisar se preocupar com o sistema operacional e hardwares subjacentes. Como resultado, os programas Erlang devem funcionar perfeitamente em qualquer sistema, independentemente do número de núcleos ou processadores (CESARINI e THOMSON, 2009).

2.5.5 Considerações finais

Como visto até aqui, Erlang não consiste apenas em uma linguagem. Erlang consiste em um apanhado de tecnologias, tais como máquinas virtuais específicas, compilador, bibliotecas e um sistema de distribuição de pacotes próprio. Vale ressaltar que, da forma como a máquina virtual foi desenvolvida, ela permite a criação de novas linguagens que possam ser executadas sobre ela. Visto tudo isto, Erlang também é conhecido como um ecossistema (CESARINI e THOMSON, 2009).

Entretanto, apesar dos avanços em *multicore* permitindo bons resultados em paralelismo, Erlang ainda carrega a má reputação de ser uma linguagem de difícil programação, principalmente para programadores oriundos da programação orientada a objetos. Erlang não comporta elementos como: metaprogramação,

polimorfismo e ferramentas de primeira classe. Na tentativa de suprir estas lacunas, surge a linguagem Elixir (THOMAS, 2018). Elixir, portanto, é tema da próxima seção.

2.6 ELIXIR

A seção 2.5 apresentou a linguagem Erlang, bem como seu ecossistema voltado à concorrência e constante melhoramentos no suporte a *multicore*. Nesta seção é apresentada a linguagem Elixir, que tem por objetivo suprir as lacunas de programação deixadas pela linguagem Erlang. A presente seção inicia com um breve histórico da linguagem Elixir e, em seguida, apresenta os novos mecanismos por ela adicionados.

2.6.1 Histórico

A história da linguagem Elixir começa quando em 2012, José Valim, criador da linguagem, percebeu como as máquinas estão cada vez mais incorporando múltiplos núcleos. Desta forma, novos softwares precisariam usar tantos núcleos quanto possível para maximizar o uso da máquina. Entretanto, esta abordagem entra em conflito diretamente com a forma como os softwares são escritos atualmente. Foi quando se deparou com Erlang e seu ecossistema (THOMAS, 2018).

No entanto, Valim sentia que ainda havia uma lacuna no ecossistema de Erlang, pois faltava o suporte a alguns dos recursos que achava necessário para seu trabalho diário. Como exemplo, podia-se citar metaprogramação, polimorfismo e ferramentas de primeira classe. A partir dessa necessidade nasceu a linguagem Elixir (THOMAS, 2018).

Elixir é uma abordagem pragmática para programação funcional. Ela valoriza suas bases funcionais e concentra-se na produtividade do desenvolvedor. Por estar dentro do ecossistema Erlang, concorrência é a espinha dorsal dos softwares escritos em Elixir. Como a coleta de lixo uma vez liberou os desenvolvedores da preocupação com gerenciamento de memória, o Elixir vem para abstrair do desenvolvedor os mecanismos de concorrência antiquados e trazer mais simplicidade nesta forma de programação (THOMAS, 2018).

2.6.2 Principais características

A linguagem de programação Elixir envolve programação funcional com imutabilidade de estados e abordagem de atores, tal como ocorre com Erlang. Contudo, em Erlang há uma lacuna na programação, principalmente para desenvolvedores oriundos de programação estruturada (como programação orientada a objetos) tais como: metaprogramação, polimorfismo, sobrecarga de funções e ferramentas de primeira classe. Elixir é uma abordagem pragmática para programação funcional. Ela valoriza seus fundamentos funcionais e foca na produtividade do desenvolvedor. (THOMAS, 2018). O Código 1 representa um exemplo de sobrecarga aplicado à função *convert_attribute_to_string*. A condição disposta após a palavra reservada *when* determina qual função será chamada de maneira dinâmica.

Código 1. Exemplo de polimorfismo em Elixir

Elixir

```
defp covert_attribute_to_string(attr_value) when is_atom(attr_value) do
  ":" <> to_string(attr_value)
end

defp covert_attribute_to_string(attr_value) when is_binary(attr_value) do
  "\"" <> to_string(attr_value) <> "\""
end
```

Fonte: Autoria própria

Elixir, portanto, capacita os desenvolvedores, fornecendo macros. O código Elixir nada mais é do que dados e, portanto, pode ser manipulado via macros como qualquer outro valor na linguagem. Finalmente, os desenvolvedores familiarizados com orientação a objetos encontram muitos dos mecanismos que consideram essenciais para escrever bons softwares, como o polimorfismo (THOMAS, 2018).

Estas características do Elixir permitem, desta forma, uma modelagem e desenvolvimento com as características estruturadas apesar de estar em um ambiente totalmente funcional. Desta forma a modelagem UML pode ser adotada e

mais facilmente materializada em função destas características, até então ausentes na linguagem Erlang (THOMAS, 2018).

2.7 PARADIGMAS DE PROGRAMAÇÃO

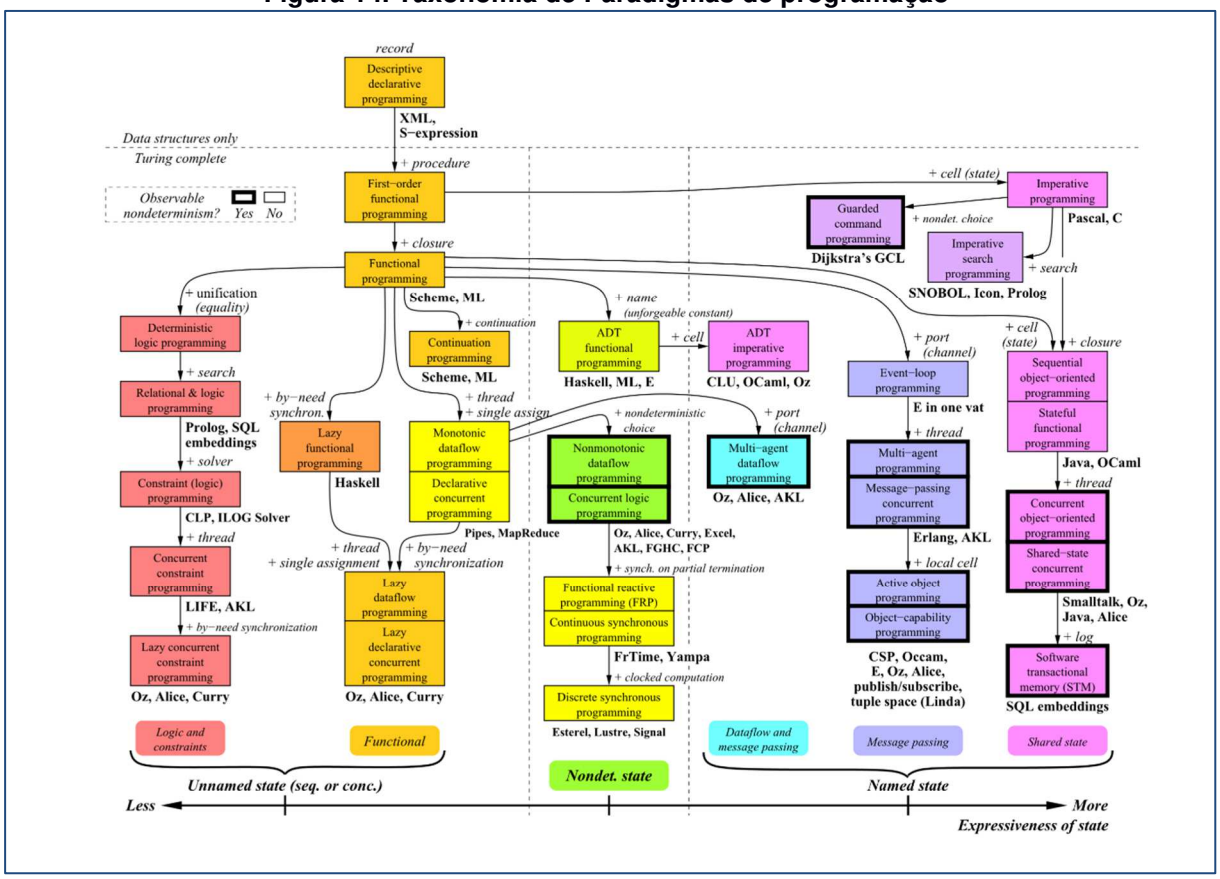
As seções 2.2 e 2.3 apresentaram, respectivamente, como as tarefas concorrem em sistemas operacionais modernos e as técnicas de comunicação entre processos. Ambos os temas estão ligados intimamente a uma programação *multicore*. Esta seção, por sua vez, traz uma conceituação sobre os paradigmas de programação. Inicia-se com uma conceituação dos paradigmas e sua taxonomia evolutiva. Por fim, são feitas considerações acerca dos principais paradigmas e suas barreiras sob a ótica de concorrência e paralelismo.

2.7.1 Taxonomia dos paradigmas de programação

Na ciência da computação, o termo paradigma é empregado como uma maneira de abstrair a forma de abstração e mesmo de raciocínio do desenvolvedor em uma determinada estrutura computacional capaz de definir o fluxo de execução de um programa. De acordo com David Watt (2004), os paradigmas se diferem em conceitos e abstrações utilizadas para representar os elementos de um programa (e.g., objetos, funções, variáveis e restrições) e a maneira com que esses interagem de maneira a ditar o fluxo de execução deste programa (e.g., atribuições, avaliações causais, repetições, empilhamento e recursividade) (WATT, 2004).

Segundo Peter Van Roy (2009), um paradigma de programação é um sistema formal que define como a programação é realizada. Ademais, cada paradigma tem o seu conjunto de técnicas e conceitos de programação que, conjuntamente, definem sua forma de estruturar o pensamento na construção de programas (VAN ROY, 2009). Van Roy desenvolveu uma taxonomia para paradigmas de programação, conforme apresenta a Figura 14 (VAN ROY, 2009).

Figura 14. Taxonomia de Paradigmas de programação



Fonte: (VAN ROY, 2009)

Conforme apresenta a Figura 14, os paradigmas de programação são organizados em um grafo que basicamente apresenta o relacionamento conceitual entre eles. Ao todo, são 27 quadros, cada qual representando um paradigma e seu respectivo conjunto de conceitos. Ademais, setas entre dois quadros representam a adição de novos conceitos, no sentido de que os quadros derivados, contemplam os conceitos dos paradigmas anteriores, acrescidos de um ou mais novos conceitos que, conjuntamente, os definem como um paradigma distinto dos demais (VAN ROY, 2009).

Os conceitos são basicamente elementos primitivos básicos que, em conjunto, dão origem aos paradigmas. Com frequência, dois paradigmas que se apresentam de formas suficientemente distintas diferem por apenas um único conceito, como por exemplo o Paradigma Funcional (PF), que é composto pelos conceitos de registros, procedimentos e escopo; assim como também o Paradigma Orientado a Objetos

(POO) é composto, com exceção de que esse último também é composto por estados nomeados com alta expressividade (VAN ROY, 2009).

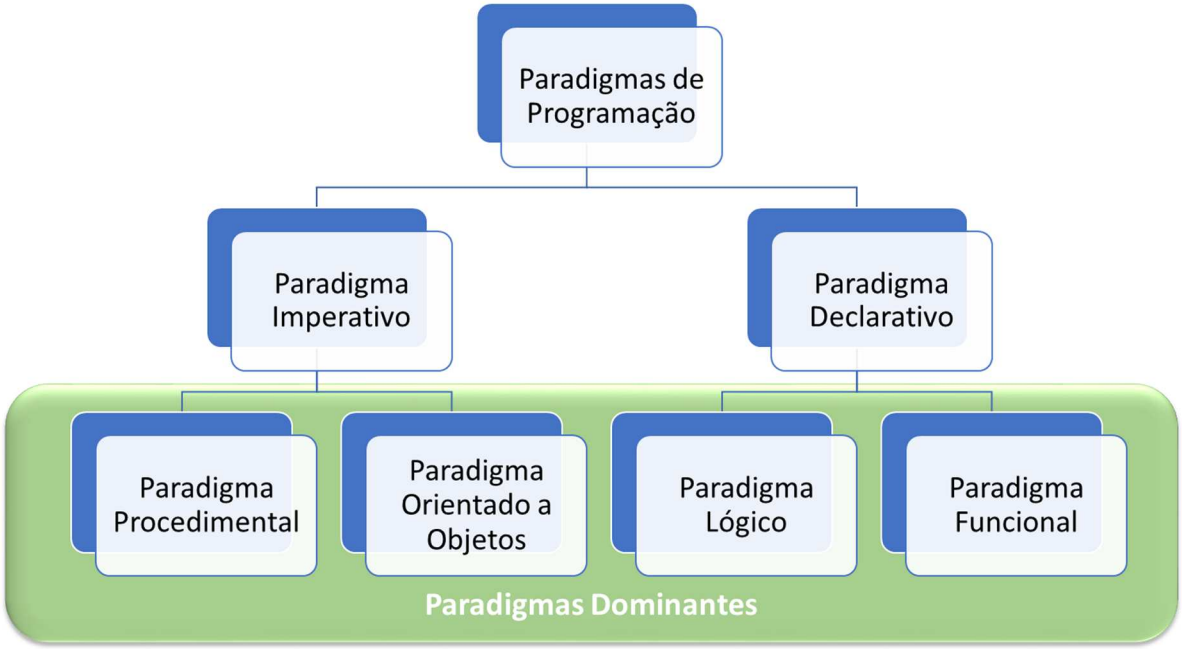
Além disso, a Figura 14 destaca a presença de duas propriedades importantes nos paradigmas: a presença de indeterminismo observável e quão fortemente eles suportam estados nomeados.

O indeterminismo observável é expresso na figura por meio de bordas espessas. O indeterminismo é observável se o mesmo programa gerar resultados diferentes para as mesmas entradas. A tendência de paradigmas que apresentam indeterminismo observável é que a programação se torna mais complexa, como é o caso da Programação Orientada a Objetos Concorrente. Em contrapartida, a Programação Declarativa Concorrente é totalmente determinística, o que tende a facilitar a programação (VAN ROY, 2009).

O suporte a estados nomeados refere-se, basicamente, à habilidade de um programa de reter informação ou, mais precisamente, de armazenar uma sequência de valores em memória (e.g., variáveis, vetores e mapas). Na Figura 14 os eixos são divididos em três níveis de expressividade, variando de estados não nomeados (mais à esquerda) a estados nomeados (mais à direita), determinísticos e não determinísticos, bem como sequencial ou concorrente. Totalizando oito combinações possíveis (VAN ROY, 2009).

Apesar da taxonomia de Van Roy apresentar relevância e definir os paradigmas de acordo com seus conceitos fundamentais, estes podem ser classificados de uma forma simplificada. Alguns autores como Banaszewski (2009), Gabbrielli e Martini (2010) e Brookshear (2012), classificam os paradigmas de programação usuais (dominantes e emergentes) como subconjuntos de dois paradigmas maiores, o Paradigma Imperativo (PI) e o Paradigma Declarativo (PD). A Figura 15 ilustra a classificação desses paradigmas de uma maneira mais resumida e simplificada em relação à Figura 14.

Figura 15. Classificação dos paradigmas de programação



Fonte: adaptado de (RONSZCKA, 2012)

Conforme ilustra a Figura 15, o PI pode ser entendido como constituído pelo Paradigma Procedimental (PP) e pelo Paradigma Orientado a Objetos (POO), os quais se diferenciam essencialmente na forma como os elementos e instruções são representados e organizados, sendo o POO considerado mais rico e supostamente estruturado em termos de expressão do código. O PD, por sua vez, pode ser entendido como constituído essencialmente pelo Paradigma Lógico (PL) e pelo Paradigma Funcional (PF). Naturalmente, esta classificação é apenas de ordem teórica, sendo que na prática os (sub)paradigmas podem apresentar intersecções.

Em tempo, conforme o ponto de vista, PP, POO, PL e PF poderiam ser considerados paradigmas e não subparadigmas ainda que com características mais próximas uns dos outros. Em todo caso, estes paradigmas se enquadram na primeira camada, à luz da Figura 15, pertencente ao grupo dos paradigmas dominantes que orienta a construção das linguagens de programação vigentes (BROOKSHEAR, 2012). Outrossim, não raro as linguagens de programação contemplam mais de um (sub)paradigma vigente, como C++ que é procedimental e orientada a objetos. Há mesmo linguagens de intersecção como LISP que possui aspectos lógico-declarativos e outros imperativo-procedimentais (VAN ROY, 2009).

2.7.2 Barreiras dos paradigmas tradicionais

Em linhas gerais, tanto o PI quanto o PD apresentam similaridades ao serem baseados em buscas/navegações sobre entidades passivas, as quais consistem em dados (e.g., fatos ou estados de variáveis ou de atributos de outras entidades computacionais) e comandos de decisão (e.g., expressões causais como se-então ou regras). Estas buscas afetam o desempenho das aplicações por gerar redundâncias de processamento e acoplamento implícito entre as entidades que compõem uma aplicação (SIMÃO, BANASZEWSKI, *et al.*, 2012).

Essencialmente, o PI impõe pesquisas orientadas a laços de repetições sobre elementos passivos, relacionando os dados (i.e., variáveis, vetores e árvores) a expressões causais (i.e., se-então ou declarações similares). Estes relacionamentos normalmente impactam negativamente no desempenho destas aplicações, devido a sua estrutura monolítica, prolixa e acoplada, o que gera a execução de código não otimizado e interdependente (GABBRIELLI e MARTINI, 2010) (BROOKSHEAR, 2012) (SIMÃO, BANASZEWSKI, *et al.*, 2012).

Por sua vez, essencialmente, o PD permite um nível maior de abstração e, portanto, maior facilidade de programação (GABBRIELLI e MARTINI, 2010). Além disso, algumas soluções declarativas podem evitar muitas das redundâncias de execução, a fim de otimizar o processamento. Dentre estas soluções, citam-se os Sistemas Baseados em Regras (SBRs), com base em algoritmos de inferência otimizados como o HAL ou o industrialmente conhecido Rete (FORGY, 1982) (CHENG e CHEN, 2000) (LEE e CHENG, 2002). No entanto, programas construídos com base em linguagens de programação usuais do PD (e.g., LISP, PROLOG e SBRs em geral) ou mesmo construídos com base em soluções otimizadas (e.g., SBRs baseados no algoritmo Rete) também apresentam desvantagens (BANASZEWSKI, SIMÃO, *et al.*, 2007) (SIMÃO e STADZISZ, 2008) (SIMÃO, 2012).

As soluções do PD são compostas por estruturas de dados de alto nível, as quais são normalmente dispendiosas em termos de processamento. Isso, de fato, agrega custos de processamento consideráveis, impactando diretamente no desempenho das aplicações. Assim, mesmo com a presença de código redundante, as soluções

do PI são normalmente melhores em desempenho do que as soluções do PD (BANASZEWSKI, 2009) (SCOTT, 2016).

Além disso, similarmente à programação no PI, a programação no PD também gera acoplamento forte entre os módulos. Isto devido ao processo de inferência ser também baseado em pesquisa sobre entidades passivas (SIMÃO e STADZISZ, 2008) (GABBRIELLI e MARTINI, 2010). Ainda, outras abordagens entre o PI e o PD, tais como a programação dirigida por eventos ou a programação funcional, não resolvem estes problemas. Em alguns casos, essas até reduzem, atenuam ou fatoram, mas não efetivamente os resolvem (BROOKSHEAR, 2012) (SCOTT, 2016).

Outrossim, o forte acoplamento dos módulos gerado pelo processo de inferência aplicado a estes paradigmas, impede a quebra do fluxo de processamento em granularidade fina. Assim, os processos e tarefas que executam esses módulos não são granulares o suficiente para serem executados de forma paralelizável (BROOKSHEAR, 2012) (SCOTT, 2016). Isto posto, segundo descrito pelo argumento de Amdahl (1967), fluxos sequenciais de processamento extensos e de diferentes tamanhos afetam negativamente no *speedup* destes tipos de aplicação.

2.8 CONCLUSÕES DO CAPÍTULO

Este capítulo iniciou apresentando fundamentações teóricas, conceitos, técnicas e avanços tecnológicos ligados ao tema *multicore*, bem como os sistemas operacionais modernos se adaptaram para fazer uso desta tecnologia cada vez mais popular. Foi também apresentado o modelo de atores e uma arquitetura baseada neste modelo, a Erlang. Foi possível observar que arquiteturas baseadas em modelos de atores mostraram-se eficientes para desenvolvimento de soluções concorrentes, pois minimizam problemas relacionados às regiões críticas.

Entretanto, mesmo com os avanços tecnológicos, os atuais paradigmas de programação ainda são uma grande lacuna em termos de programação *multicore*. Em termos mais detalhados, tanto o PI com sua estrutura monolítica, prolixa e acoplada, quanto o PD com seu processo de inferência baseado em pesquisa sobre entidades passivas, geram fluxos de processamento não paralelizáveis e de tamanhos diferenciados. Estes fluxos, mesmo que executados paralelamente, não alcançam um bom aproveitamento em termos de balanceamento em ambientes *multicore*.

Motivado por esta lacuna deixada pelos paradigmas PI e PD é apresentado o Paradigma Orientado a Notificações (PON), que será abordado no Capítulo 3. Todo o referencial teórico apresentado a seguir é oriundo do conhecimento acumulado pela equipe de pesquisadores PON e, por este motivo, decidiu-se tê-lo como um capítulo à parte.

3 O PARADIGMA ORIENTADO A NOTIFICAÇÕES (PON)

O presente capítulo tem por objetivo trazer um apanhado sobre o Paradigma Orientado a Notificações (PON). Inicia-se o capítulo com um breve histórico. Em seguida, são apresentados seus fundamentos e detalhados os mecanismos de notificação. Subsequentemente, é apresentada a NOPL com um breve histórico evolutivo até a sua versão mais recente. Após, são apresentadas as versões materializadas em hardware, bem como em software juntamente com seus avanços em *multicore.* Por último, são apresentadas considerações finais do capítulo.

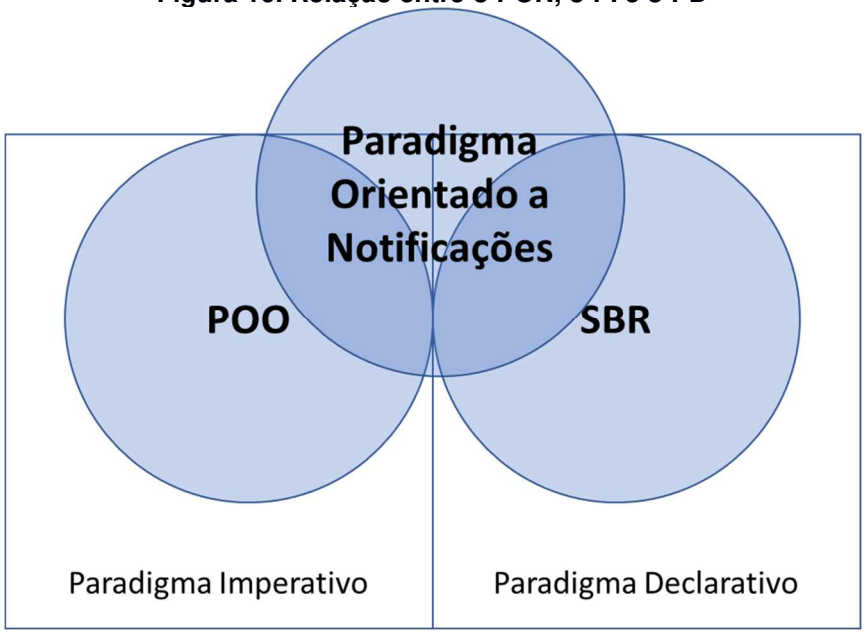
3.1 INTRODUÇÃO

O Paradigma Orientado a Notificações (PON) foi proposto como uma nova alternativa de desenvolvimento de sistemas computacionais visando diminuir problemas existentes nos paradigmas atuais, nomeadamente Paradigma Imperativo (PI) e Paradigma Declarativo (PD), descritos na seção 2.7. Exemplos destes problemas são as redundâncias de execução com o consequente mau uso de processamento e o correlato acoplamento excessivo entre entidades computacionais com a consequente dificuldade de reaproveitamento e de paralelização/distribuição (BANASZEWSKI, 2009) (SIMÃO e STADZISZ, 2008). O PON possui sua origem na dissertação de mestrado (2001) e tese de doutorado (2005) de Simão sobre Controle Orientado a Notificações (CON), ambos sob a orientação de Stadzisz, no Laboratório de Sistemas Inteligentes de Produção (LSIP) da UTFPR.

O PON até encontra algumas inspirações no PI, como flexibilidade algorítmica e a abstração em forma de classes/instâncias do POO, e algumas inspirações no PD, com a abstração do conhecimento através de regras lógico-causais e base de fatos dos SBR na forma de entidades similares a classes/instâncias. Desta forma, o PON possibilita algo de uso simbiótico de ambos os paradigmas de programação em seu modelo, mas na verdade ele faz mais que isso na medida que evoluciona ou quiçá revoluciona. Esta (r)evolução se dá no tocante ao processo de inferência ou cálculo lógico-causal e organização de entidades que permite esta inferência diferenciada. Em PON, justamente, a inferência se dá por notificações pontuais e precisas entre entidades pertinentes e colaborativas. A Figura 16 apresenta o posicionamento do

PON frente aos paradigmas Imperativo (PI) por meio do Paradigma orientado a Objetos (POO), e Declarativo (PD) por meio do Sistema Baseado em Regras (SBR) (BANASZEWSKI, 2009) (SIMÃO e STADZISZ, 2008) (SIMÃO e STADZISZ, 2009) (SIMÃO, TACLA, *et al.*, 2012b) (LINHARES, RONSZCKA, *et al.*, 2011) (SIMÃO, 2012a) (XAVIER, 2014).

Figura 16. Relação entre o PON, o PI e o PD



Fonte: adaptado de (BANASZEWSKI, 2009)

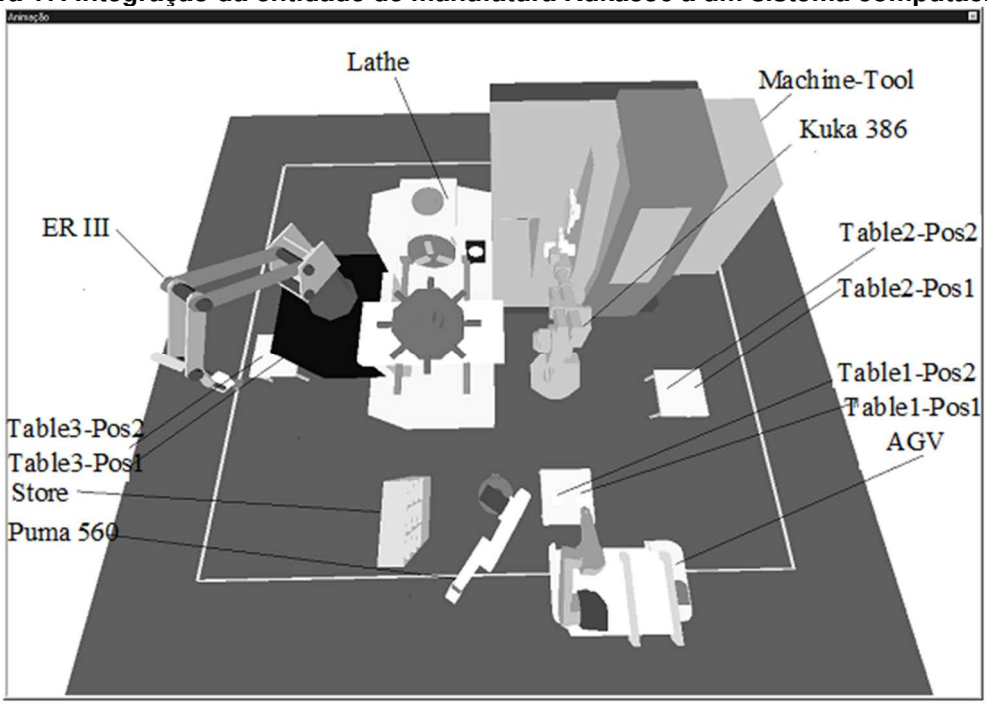
3.2 CONTEXTO HISTÓRICO

O objetivo original dos trabalhos de dissertação e tese de Simão (2001) e (2005) foi propor um novo mecanismo de controle que suprisse as necessidades relacionadas aos sistemas modernos de produção, tais como o tratamento das variações de produção e a produção customizada em massa (SIMÃO, 2005).

Simão (2005) propôs uma abordagem de controle que permite, de forma intuitiva, organizar as colaborações entre entidades de manufatura (e.g. recursos ou equipamentos) a fim de alcançar agilidade na produção. Esta abordagem refere-se a um metamodelo de controle discreto que foi aplicado à simulação de sistemas de manufatura ditos holônicos. Essa simulação ocorreu na ferramenta de projeto e

simulação de sistemas de manufatura ANALYTICE II[20], que é representada na Figura 17.

Figura 17. Integração da entidade de manufatura Kuka386 a um sistema computacional



Fonte: (SIMÃO, 2005)

As entidades de manufatura desses sistemas "inteligentes" são integradas a sistemas computacionais "comuns" (e.g. software de controle) por meio de "recursos virtuais" (i.e., drivers avançados), que permitem o acesso a dados e serviços pelos sistemas computacionais por meio de uma rede de comunicação de dados.

Para exemplificar, suponha-se a entidade de manufatura Kuka386 visto na Figura 17. Esta entidade está integrada com um componente de um sistema computacional de controle por meio do Kuka386-virtual (i.e., um *smart-driver*). Esta integração se dá por meio da rede de comunicação de dados, onde os feedbacks (e.g. comandos) entre o sistema computacional e o equipamento real são intermediados pelo recurso virtual ou driver desse equipamento.

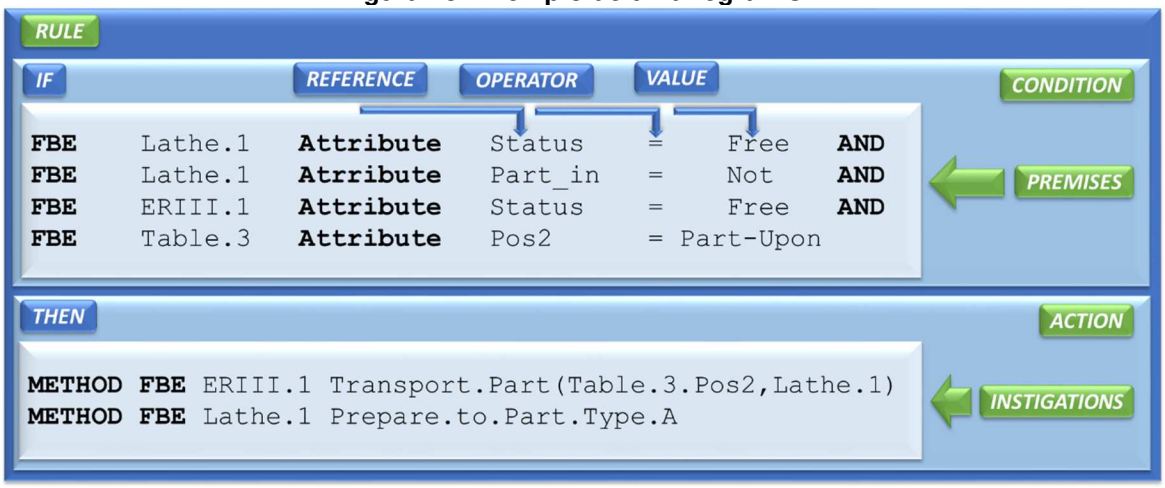
Todo e qualquer recurso virtual expressa os estados ou valores do respectivo equipamento por meio de entidades chamadas atributos, bem como disponibiliza seus serviços por meio de entidades chamadas métodos. Assim sendo, todos os recursos-

[20] Este projeto foi desenvolvido por gerações de pesquisadores do LSIP da UTFPR (KOSCIANSKI, 1999) (SIMÃO, 2005)

virtuais apresentam a mesma forma de *feedback* para com um determinado sistema computacional. No domínio da manufatura, o metamodelo em questão permite compor software de controle no qual a representação das relações causais de controle se dá por meio de regras causais sobre os atributos e métodos dos recursos-virtuais. Estes recursos-virtuais foram posteriormente renomeados “Elementos da Base de Fatos” (do inglês, *Fact Base Elements – FBE).*

A Figura 18 ilustra um exemplo de uma regra na forma de conhecimento causal (SIMÃO, 2005). A semântica dessa regra refere-se ao controle das relações entre os equipamentos Lathe.1 (i.e., um torno mecânico), ERIII.1 (i.e. um robô de transporte de peças) e Table.3 (i.e., uma mesa para armazenamento temporário de peças), os quais compõem parte da célula de manufatura simulada no ANALYTICE II.

Figura 18. Exemplo de uma regra PON



Fonte: adaptado de (SIMÃO, 2005)

Em linhas gerais, uma regra consiste em uma condição e uma ação, na qual a condição representa a avaliação causal de uma regra, enquanto uma ação consiste no conjunto de instruções (comandos) executáveis de uma regra.

No exemplo, a condição da regra verifica se os *FBE* Lathe.1 e ERIII.1 estão livres, se o *FBE* Lathe.1 não tem peça dentro de si e se há algum produto sobre o *FBE* Table.3. Se esses estados forem constatados, a ação da regra faz com que o robô ERIII.1 transporte o respectivo produto para ser manipulado pelo torno Lathe.1.

Nesses trabalhos, o autor propõe o mecanismo atualmente chamado de Controle Orientado a Notificações (CON) para sistemas de produção, mais precisamente, para sistemas de manufatura, no qual entidades computacionalmente integradas de

manufatura notificam atualizações de estados para entidades de controle cujo conteúdo seria de uma regra, como a exemplificada na Figura 18. Na verdade, essas notificações se dão por entidades menores que compõem os elementos notificantes e os elementos notificados. Este mecanismo foi, então, apresentado na forma de um metamodelo de controle discreto e holônico do CON que permite aplicar instâncias deste sobre simulação realística de sistemas de manufatura em uma ferramenta chamada ANALYTICE II (SIMÃO, 2005). Isso permitiu simular realisticamente os chamados Sistemas Inteligentes de Manufatura, também chamados de Sistemas Holônicos de Manufatura (2001) (SIMÃO e STADZISZ, 2002) (SIMÃO, 2005) (SIMÃO, STADZISZ e MOREL, 2006).

Os sistemas discretos de controle e manufatura, holônicos ou não, não foram a única aplicação deste metamodelo do CON. Os autores perceberam, subsequentemente, que poderiam aplicá-lo em várias áreas, como: concepção de controle discreto em geral; inferências discretas em geral; e mesmo software de maneira genérica. Dessa maneira, a aplicação do metamodelo de notificações à construção de programas foi, à luz do tempo, denominada de Paradigma Orientado a Notificações (PON) (SIMÃO e STADZISZ, 2008).

De maneira geral, o PON foi proposto como uma alternativa para o desenvolvimento de sistemas computacionais, visando diminuir problemas existentes nos atuais Paradigma Imperativo (PI) e Paradigma Declarativo (PD). Exemplos de problemas são as redundâncias de execução com o consequente mau uso de processamento e o correlato acoplamento excessivo entre entidades computacionais com a consequente dificuldade de reaproveitamento e de paralelização/distribuição das entidades computacionais de um sistema (SIMÃO e STADZISZ, 2008) (BANASZEWSKI, 2009) (PORDEUS, 2017) (OLIVEIRA, ROTH, *et al.*, 2018).

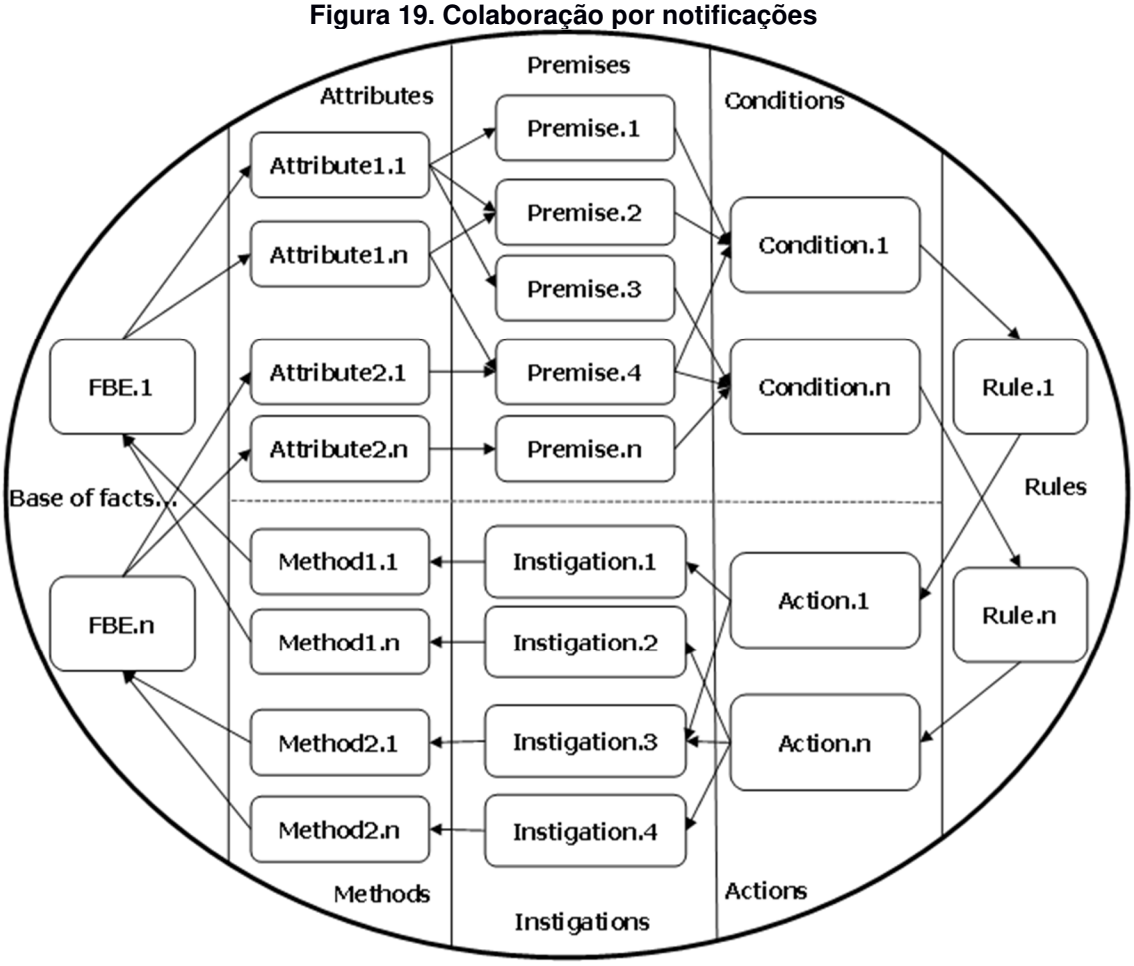
3.3 FUNDAMENTOS

No PON, cada regra causal é computacionalmente representada por um conjunto de entidades relacionadas. Dentre elas, o cerne é a entidade *Rule*. Uma *Rule* é decomposta em uma entidade *Condition* e uma entidade *Action*. De maneira similar, uma *Condition* é decomposta em uma ou mais entidades *Premises* e uma *Action* é

decomposta em uma ou mais entidades *Instigations*. Cada *Fact Base Element (FBE)* também é decomposto em entidades menores, os *Attributes* e os *Methods*.

Todas as entidades colaboram por meio de inferências baseadas em notificações, a fim de ativar as *Rules* pertinentes para execução (SIMÃO e STADZISZ, 2008). Essa abordagem é explicitada pelo esquema ilustrado na Figura 19 (SIMÃO, 2005). Nessa abordagem, cada *FBE* notifica o seu conhecimento factual por meio de capacidades reativas incorporadas às suas entidades *Attributes* às demais entidades envolvidas.

Figura 19. Colaboração por notificações



Fonte: adaptado de (SIMÃO, 2005)

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

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

Portanto, por meio do mecanismo de inferência, o PON provê uma solução efetiva para compor e executar software de controle no domínio de sistemas de manufatura modernos. Este fato foi confirmado pela implementação e análise do metamodelo sobre diversas perspectivas no simulador ANALYTICE II (SIMÃO, 2005) (SIMÃO, 2008). Nessas análises, o metamodelo cumpriu as expectativas (SIMÃO, 2001) (SIMÃO, 2005) (SIMÃO, 2008) (SIMÃO e STADZISZ, 2002) (SIMÃO, STADZISZ e MOREL, 2006).

3.4 METAMODELO E MECANISMO DE NOTIFICAÇÕES

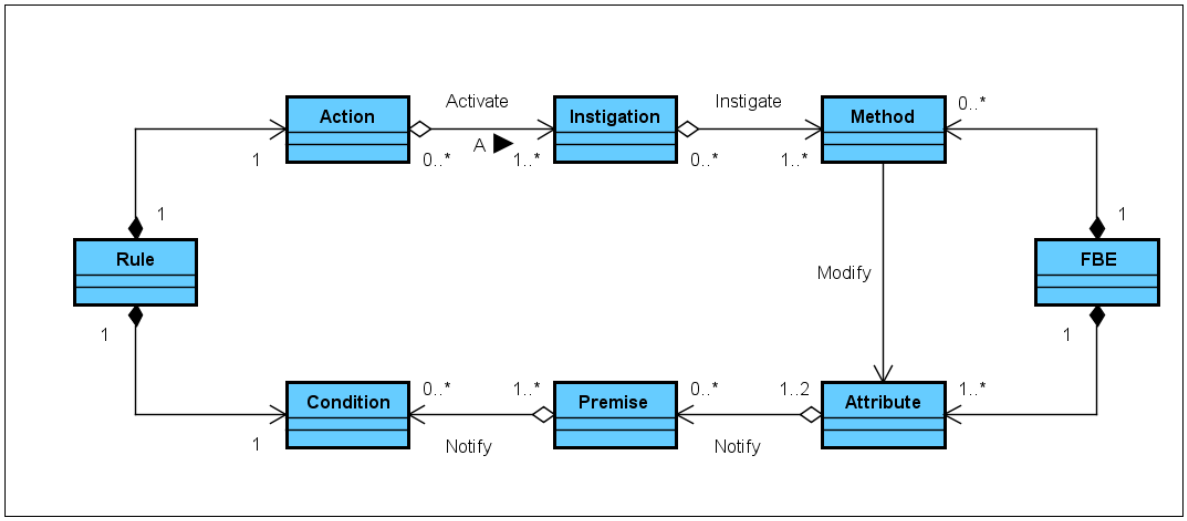
A Figura 20 exemplifica o metamodelo proposto para o PON. Estruturalmente, uma aplicação segundo o modelo PON é representada na forma de Base de Fatos (*FBE – Fact Base Element*) e de Regras (*Rules*). As entidades *FBE* são utilizadas para representar objetos do mundo (reais ou abstratos) em um sistema computacional, por meio de conjuntos de estados mutuamente exclusivos que são tratados por entidades chamadas de Atributos (*Attributes*) e por meio de correlatos serviços tratados por entidades chamadas de Métodos (*Methods*). As entidades *Rules*, por sua vez, definem o cálculo lógico-causal a ser efetuado sobre os estados dos *FBE*s, controlando a execução dos seus serviços, justamente à luz dos *Attributes* e dos *Methods*.

O mecanismo de notificações consiste na estrutura interna de execução das instâncias do PON, que determina o fluxo de execução das aplicações. Por meio desse, as responsabilidades de uma aplicação são divididas entre seus objetos, que cooperam por meio de notificações para formar o fluxo de execução da aplicação. As relações pelas quais os objetos colaboram, também estão ilustradas na Figura 20.

Os objetos das classes *Rule* e *FBE* (*Fact Base Element*) se apresentam em extremidades opostas e se relacionam com o auxílio dos objetos colaboradores conforme as conexões modeladas (BANASZEWSKI, 2009). Estas conexões são estabelecidas em tempo de execução à medida que os objetos são criados. Por exemplo, na criação de um objeto *Premise* pelo menos um objeto *Attribute* é considerado como referência para si.

Na Figura 20, pode-se constatar a modelagem de dois fluxos opostos de notificações: o fluxo ativo e o passivo em relação aos objetos da extremidade. Por exemplo, o fluxo de notificações originado no *FBE* é ativo em relação ao *FBE* e passivo em relação à *Rule*. De maneira oposta, o fluxo de notificações originado na *Rule* é ativo em relação à *Rule* e passivo em relação ao *FBE*. Esses fluxos são formados pela cooperação entre os objetos colaboradores dessas extremidades e objetivam reproduzir, em forma de metamodelo, a dinâmica já abordada na seção 3.3.

Figura 20. Relação entre objetos PON



Fonte: (BANASZEWSKI, 2009)

O momento exato da execução de uma *Rule* é determinado após a resolução de conflito entre essa e as demais regras aprovadas, caso haja conflitos. Um conflito ocorre quando duas ou mais regras referenciam um mesmo *FBE* e demandam exclusividade de acesso a esse *FBE*.

Em seguida, as regras concorrem para adquirir acesso exclusivo a esse *FBE*, sendo que somente uma dessas regras em conflito pode executar por vez (que obteve o acesso exclusivo). Com os conflitos solucionados, a respectiva *Rule* está pronta

para executar o conteúdo da sua *Action*. Uma *Action* é conectada a um ou vários *Instigations*. Os *Instigations* colaboram com as atividades das *Actions*, acionando a execução de algum serviço de um objeto *FBE* por meio dos seus objetos *Methods*. Comumente, as chamadas para os *Methods* mudam os estados dos *Attributes* e o ciclo de notificação recomeça (BANASZEWSKI, 2009).

Por fim, é possível concluir que objetos colaboradores se apresentam desacoplados ou “minimamente acoplados” devido à comunicação realizada por meio de notificações. Estes comportamentos (i.e., comunicações por notificações) e estruturas (i.e., desacoplamento) favorecem a aplicação do mecanismo de notificações para ambientes multiprocessados, pois se faz necessário que um objeto notificante conheça o endereço do objeto notificado para que uma notificação ocorra.

3.5 MATERIALIZAÇÕES EM HARDWARE

A partir dos esforços para materializar o PON em software, vislumbrou-se também a materialização do PON através de implementações em hardware, de forma que permitam a execução de aplicações concebidas seguindo o modelo PON de maneira mais fidedigna no tocante ao paralelismo. Desta forma, foram propostas técnicas, modelos de circuitos e estruturas arquiteturais de hardware que permitissem construir ambientes nos quais o mecanismo de notificações pudesse executar, de fato, em paralelo. Isto com o objetivo de minimizar questões de *overhead* presente nas soluções propostas por meio de software.

A primeira proposta de materialização de uma aplicação PON em hardware foi feita por (WITT, SIMAO, *et al.*, 2011), no qual foi proposto um conjunto de circuitos sequenciais/combinacionais que implementam os elementos do metamodelo de notificações em hardware (WITT, SIMAO, *et al.*, 2011) (SIMÃO, 2012a). Estes circuitos podem ter suas interconexões sintetizadas em FPGAs, de forma a implementar a cadeia de notificações do PON para uma aplicação específica (PORDEUS, KERSCHBAUMER, *et al.*, 2016).

Segundo este modelo, a execução da cadeia de notificações depende exclusivamente da propagação de sinais digitais entre os circuitos que representam cada elemento do metamodelo. Esta propagação pode ocorrer de forma paralela, portanto permitindo a ocorrência do paralelismo intrínseco previsto no PON. Além

disso, as propagações das notificações ocorrem em um nível de abstração muito baixo (propagação de sinais síncronos no circuito gerado), fazendo com que o desempenho nesta abordagem seja superior ao desempenho nas abordagens na forma de software.

Em um primeiro experimento, os elementos do metamodelo PON de uma aplicação para simulação de um terminal telefônico foram implementados por meio de blocos em lógica reconfigurável. Neste caso, o PON não apresentou resultados satisfatórios, em relação à mesma aplicação desenvolvida por meio de máquinas de estados para hardware (abordagem tradicional). Ainda, para o nível de abstração empregado para a construção para a máquina de estados, a abordagem em PON não trouxe vantagens no tocante à facilidade de desenvolvimento (PORDEUS, KERSCHBAUMER, *et al.*, 2016).

Jasinski (2012) propôs um *framework* que permite fazer uma descrição de uma aplicação PON e compilar esta descrição para um código VHDL, que descreve um circuito sequencial/combinacional e que implementa a aplicação PON em hardware. Os elementos PON suportados pelo *framework* (*Attributes*, *Premises*, *Conditions* e *Instigations*) são descritos através da linguagem YAML (*YAML Ain't Markup Language*)[21]. Após a compilação, são gerados circuitos que implementam a aplicação baseados nos elementos do metamodelo PON desenvolvidos por Witt et al. (2011) (LINHARES, 2015).

No âmbito de PON-HD, outro experimento subsequente tratou de uma aplicação para ordenação de números inteiros em lógica reconfigurável. Primeiramente, esta aplicação foi desenvolvida em uma abordagem tradicional de desenvolvimento puramente em VHDL (sem os conceitos do PON) e através de máquinas de estado. Em seguida, essa mesma aplicação foi desenvolvida em PON-HD, mas agora em uma nova versão, na qual os elementos do metamodelo PON foram implementados por meio de componentes VHDL. Com isso, foi possível comparar a implementação em abordagem tradicional para com a nova abordagem de PON-DH. Neste experimento, a abordagem PON-HD apresentou desempenho e quantidade de elementos lógicos semelhantes à abordagem puramente em VHDL. Porém, o desenvolvimento segundo o modelo PON apresenta maior nível de abstração em relação à abordagem de desenvolvimento puramente em VHDL, o que facilita o desenvolvimento

[21] YAML permite descrever estrutura hierárquica de maneira mais legível do que em *eXtensible Markup Language* (XML).

(KERSCHBAUMER, SIMÃO, *et al.*, 2015) (PORDEUS, KERSCHBAUMER, *et al.*, 2016).

Porém, como desvantagem, essa abordagem apresenta limitação de escalabilidade, pois esta mapeia os elementos da cadeia de notificações de determinada aplicação para instâncias dos circuitos digitais correspondentes em hardware. Como consequência, cada uma dessas instâncias consome uma determinada quantidade de elementos lógicos disponível na FPGA; desta forma a quantidade de instâncias é limitada pela capacidade do dispositivo FPGA. Ainda, cada mudança na aplicação requer uma reconfiguração da FPGA, o que diminui a flexibilidade no uso desta abordagem em relação ao modelo em software o qual tem seu código binário simplesmente carregado da memória para a plataforma em execução (LINHARES, 2015).

Outrossim, Peters (2012) propôs a implementação em lógica reconfigurável de um coprocessador PON (CoPON). CoPON é uma solução híbrida na qual a parte da aplicação responsável pelo processamento factual é executada em um núcleo von Neumann e a parte da aplicação responsável pelo processamento ou cálculo lógico-causal e propagação de notificações é executada por meio de um coprocessador baseado nos princípios do PON.

O hardware do processador é configurável de acordo com o projeto, em relação à quantidade de instâncias de elementos PON suportados (*Attributes*, *Premises*, *Conditions* e *Rules*), que são utilizados para a construção da aplicação PON. Já os elementos *Methods* do PON são codificados em software, utilizando o *framework* PON C++ desenvolvido por Banaszewski (2009) e são executados por um núcleo von Neumann sintetizado em FPGA (PETERS, 2012) (LINHARES, 2015). Ainda, essa solução poderia ser evoluída de maneira a ser sinérgica à Linguagem e Compilador do PON.

Linhares (2015) propôs e avaliou em seu doutoramento um novo modelo de arquitetura de processador que visava ser mais adaptada à execução de programas desenvolvidos seguindo o PON do que o modelo de arquitetura von Neumann, mas mantendo sua flexibilidade e generalidade. Deste trabalhou surgiu a NOCA ou ArqPON. A NOCA se diferencia do PON-HD permitindo executar e recarregar diferentes aplicações em software, sendo agora o processador e memória um conjunto de elementos que se orientam por notificações (LINHARES, 2015).

3.6 MATERIALIZAÇÕES EM SOFTWARE

Esta seção apresenta as materializações do PON no tocante ao estado da técnica para as implementações em software.

3.6.1 Arquétipo ou *Framework* PON

A primeira implementação dos conceitos PON foi realizada sobre um arquétipo ou *framework* desenvolvido em linguagem de programação C++. O objetivo deste *framework* foi oferecer uma interface de programação (do inglês, *Application Programing Interface* - API) para desenvolvimento de aplicações seguindo o modelo PON, com abstrações necessárias para compor os *FBEs* e *Rules* (BANASZEWSKI, 2009) (VALENÇA, 2012).

O Framework PON C++ foi proposto na forma de um protótipo por Simão em 2007 (SIMÃO e STADZISZ, 2008) (SIMÃO, 2012a), remontando ao framework que implementa o metamodelo do Controle Orientado a Notificações (CON) (SIMÃO, 2001) (SIMÃO, 2005) sobre a ferramenta ANALYTICE II (SIMÃO, 2005). Este framework foi refeito no trabalho de Banaszewski (2009) gerando o chamado *Framework* PON C++ 1.0 (SIMÃO, BANASZEWSKI, *et al.*, 2017a), sendo otimizado e melhorado nos trabalhos de Valença (2012) e Ronszcka (2012). Esta versão foi posteriormente chamada de Framework PON C++ 2.0 (SIMÃO, RONSZCKA, *et al.*, 2017b).

De fato, as implementações em forma de *frameworks* são implementações compostas por um conjunto de classes cujos métodos são materializados em PI/POO, sendo que sua implementação se baseia em percorrer estruturas de dados (originalmente fornecidas pela STL– *Standard Template Library* na versão 1.0 do *Framework* – e subsequentemente por biblioteca própria na versão 2.0 do *Framework*) para avaliação das relações lógico-causais e fazer o envio de notificações na forma de chamada de métodos. Esta abordagem é desvantajosa à filosofia do PON, pois há uma enorme dependência de estruturas de dados e a execução sobre estas estruturas se dá de forma sequencial, decorrendo assim em degradação de desempenho (BANASZEWSKI, 2009) (VALENÇA, 2012) (RONSZCKA, 2012).

Ainda assim, segundo Ronszcka (2012), o *Framework* PON C++ 2.0 permite uma rápida prototipação de aplicações PON para teste sobre plataformas de computação convencionais. Entretanto, mesmo após otimizações feitas no âmbito do *framework*, os experimentos conduzidos em (SIMÃO et al, 2012b; 2012c) concluíram que os resultados ainda não se mostraram satisfatórios em termos de desempenho, conforme o cálculo assintótico do PON, cálculo este apresentado em (SIMÃO, 2005) (BANASZEWSKI, 2009). Este fato se deve principalmente ao uso de estruturas de dados dispendiosas em termos de desempenho para o processo de inferência.

Desta forma, Valença (2012) e Ronszcka (2012) efetuaram uma série de otimizações com objetivo de melhorar o desempenho de execução das aplicações sobre *framework*. O resultado deste trabalho é uma segunda versão do *framework* baseado em uma variedade de estruturas de dados mais otimizadas do que as equivalentes fornecidas pela STL, por exemplo, vetores (PONVECTOR), listas (PONLIST) e tabelas *hash* (PONHASH). Estas otimizações fizeram o *framework* apresentar ganhos de desempenho em diversas aplicações quando comparado à versão original do *framework* (VALENÇA, 2012) (LINHARES, 2015).

Mesmo após otimizações com relação ao *framework*, foram observados em experimentos (SIMÃO, 2012a) que os resultados ainda não eram satisfatórios em termos de desempenho, conforme o cálculo assintótico do PON, cálculo este feito em (BANASZEWSKI, 2009). Essa degradação de desempenho se deve principalmente ao uso de estruturas de dados assaz “caras” ao executar o processo de inferência. Por este motivo, motivou-se a criação de uma linguagem e compilador específicos ao modelo PON, com o objetivo de eliminar ou minimizar estes problemas apresentados (FERREIRA, 2015). Adicionalmente, o PON necessitava de uma linguagem própria para melhorar em termos facilidade de desenvolvimento e desempenho. Isto dito, a Linguagem e Compilador para o PON é objeto da próxima subseção.

3.6.2 Linguagem e Compilador PON

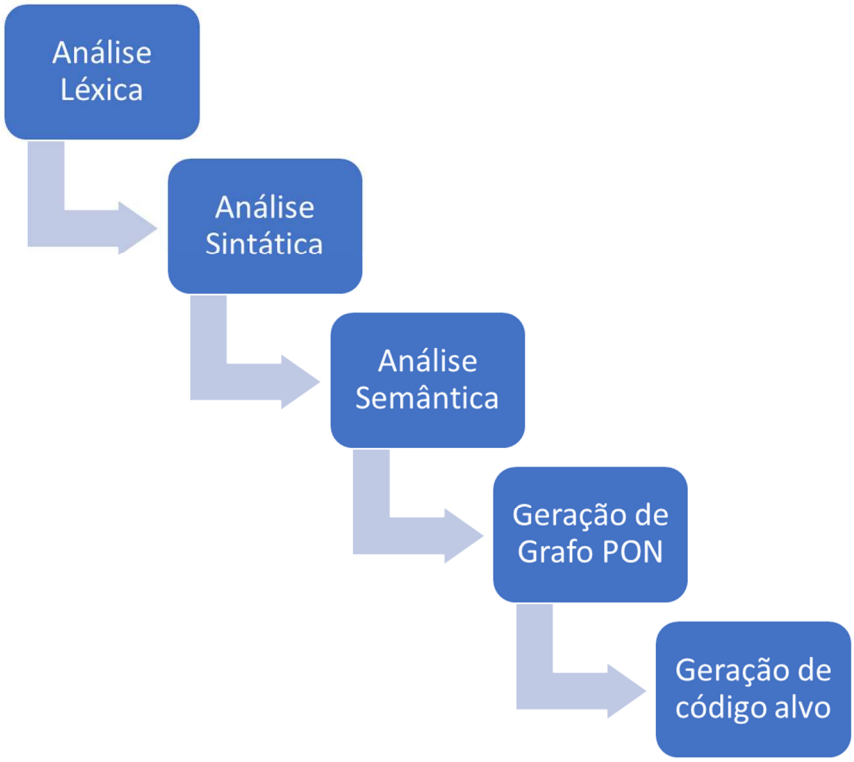
Um compilador é um programa que recebe como entrada um programa escrito em uma linguagem de programação (linguagem fonte) e o traduz para um programa equivalente em outra linguagem (linguagem alvo), a qual não é necessariamente

“linguagem de máquina”. Independente da linguagem fonte ou alvo, o processo de compilação é dividido em duas partes: análise e síntese (AHO, LAM, *et al.*, 2008).

Na etapa de análise, são compreendidas as etapas de análise léxica, análise sintática e análise semântica. A análise léxica, a partir do código fonte, lê cada caractere e produz uma sequência de símbolos chamados símbolos-léxicos que são então manipulados pelo analisador sintático. A análise sintática, por sua vez, verifica se os símbolos estão encadeados de acordo com a especificação sintática da linguagem. Por fim, a análise semântica verifica outras consistências, por exemplo, ela verifica se as atribuições de variáveis são dos tipos corretos. Ainda nessa etapa, os dados coletados são armazenados em uma estrutura de dados chamada tabela de símbolos. Na etapa de síntese, é realizada a conversão intermediária em um código alvo desejado, utilizando as informações armazenadas na tabela de símbolos (AHO, LAM, *et al.*, 2008).

Como linguagem alvo na tecnologia NOPL, a partir da linguagem específica ao PON, podem ser geradas inúmeras opções, pois o processo final apresenta um grafo PON (RONSZCKA, 2019) que é então usado para geração do correspondente código alvo. A produção de novas materializações também é facilitada em função desta plataforma genérica de abstração de modelagem. A Figura 21 apresenta as etapas da compilação da linguagem PON.

Figura 21. Etapas de Compilação de NOPL



Fonte: (RONSZCKA, 2019)

Atualmente as seguintes materializações já estão disponíveis para NOPL em plataforma sequencial (von Neumann):

- Código específico para *framework* PON 2.0;
- Código específico PON para C++ e código específico PON para C (FERREIRA, 2015). Isto encontra algum paralelo, a título de curiosidade, com as primeiras versões do C++ que eram baseadas em um código C com classes (STROUSTRUP, 2013)
- Código C++ *Namespaces* (ATHAYDE e NEGRINI, 2016)

3.6.3 Materializações do PON em *multicore*

Quanto à materialização do PON em *multicore*, alguns esforços são apresentados nesta seção. Weber et al. (2010) especializaram o *Framework* PON C++ 1.0 (BANASZEWSKI, 2009) por meio da proposta de um ambiente *multithread* para execução concorrente dos elementos presentes no modelo de notificações do CON.

Nesta versão as entidades CON (*Rule, Condition, Premise* etc.) podem executar cada qual em uma *thread* em particular. Contudo, esta solução não foi adicionada nas materializações atuais do PON de maneira explicita. Conquanto, Belmonte et al. (2012) utilizaram este ambiente para decisões tomadas em desenvolvimento subsequente.

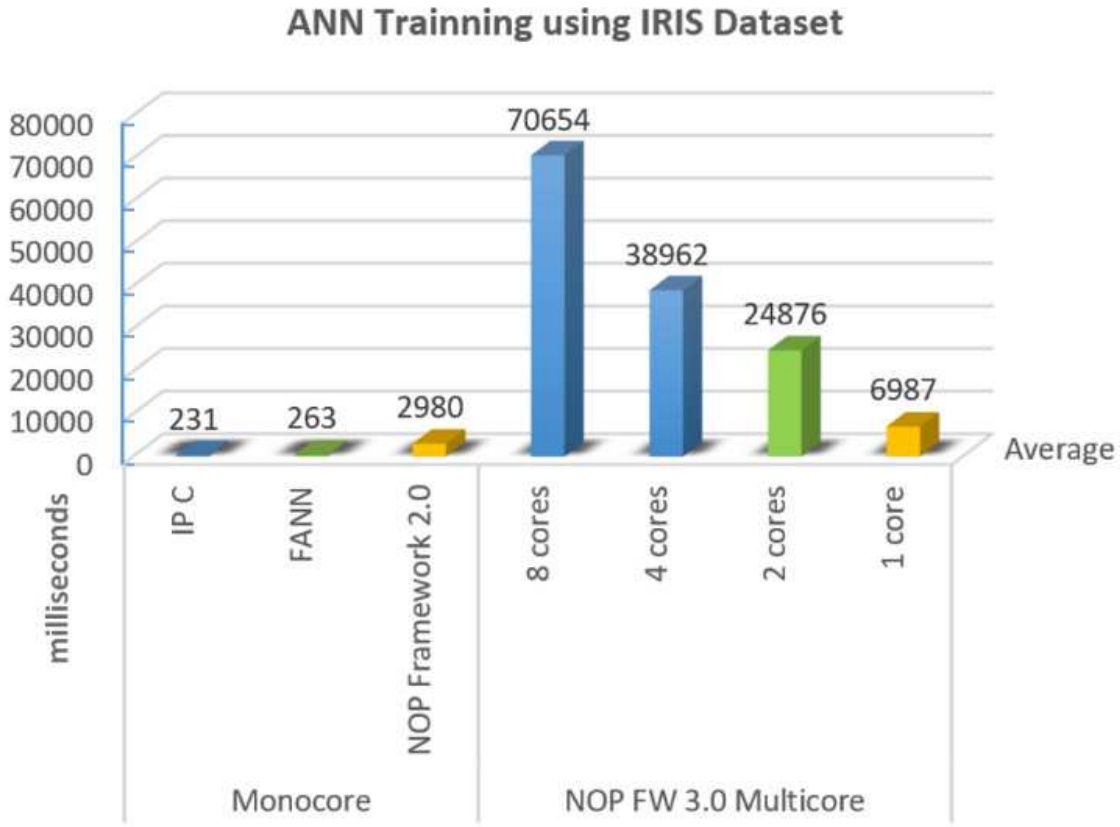
Neste trabalho, Belmonte et al. (2012) estenderam a implementação do *Framework* PON C++ 2.0 C++ (2012), incluindo a execução de *Rules* por meio de *threads* independentes (BELMONTE, 2012). Neste contexto, a extensão do *Framework* PON C++ 2.0 por meio de *threads*, intitulada de *Framework* PON C++ 3.0, fornece uma maneira (via uma camada adicional) de paralelizar elementos PON em múltiplos núcleos, de forma transparente (BELMONTE, LINHARES, *et al.*, 2016).

As principais classes que implementam os elementos PON no *Framework* PON C++ 2.0 foram mantidas, sendo que algumas classes foram criadas para suportar o mecanismo *multithread* quando disponível. O *Framework* PON C++ 3.0 usa uma API que implementa o modelo de execução Posix Threads (*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 múltiplas *threads* criadas (BELMONTE, LINHARES, *et al.*, 2016) (SCHÜTZ, FABRO, *et al.*, 2018).

O *Framework* PON C++ 3.0 possui os métodos necessários, conforme apresentado em Belmonte (2012), que visa organizar o balanceamento da execução de um software de forma automática, sem a intervenção explícita do desenvolvedor. Ainda, o processo de alocação de entidades em múltiplos núcleos é transparente para o desenvolvedor. A única mudança no código usado em entidades PON, em relação ao código fonte do *Framework* PON C++ 2.0, é a especificação do núcleo de execução para cada entidade a ser utilizada (BELMONTE, LINHARES, *et al.*, 2016).

Entretanto, resultados obtidos no *Framework* PON C++ 3.0 não apresentaram resultados satisfatórios, tendo seus tempos de execução consideravelmente superiores em comparação com versões monoprocessadas. Estes valores podem ser vistos na Figura 22, que apresenta os resultados do experimento NeuroPON para o treinamento da rede RNA MLP com método BP, apresentado por Schutz em sua tese de doutorado (SCHÜTZ, 2019). Os resultados demonstram que, conforme se aumenta o número de núcleos disponíveis, maiores são os tempos de execução para o *Framework* PON C++ 3.0.

Figura 22. Tempos de execução (em milissegundos) do treinamento de RNA MLP com método BP, utilizando o modelo computacional NeuroPON em materializações de software do PON, para a base de dados da flor de Iris



Fonte: (SCHÜTZ, 2019)

3.7 ESTADO DA ARTE DO PON EM SOFTWARE

Até aqui foi visto um apanhado dos esforços dos pesquisadores acerca do tema PON nas mais diversas áreas. Mais precisamente na área de software, recentemente Ronszcka (2019) consolidou e propôs novas tecnologias. O resultado deste trabalho pode ser considerado o atual estado da arte em software para PON. Os componentes desta tecnologia são, então, apresentados em detalhes a seguir.

3.7.1 MCPON

Ronszcka (2019) identificou que os métodos e técnicas utilizados na teoria de compilação tradicional não apresentam as características desejadas e efetivamente

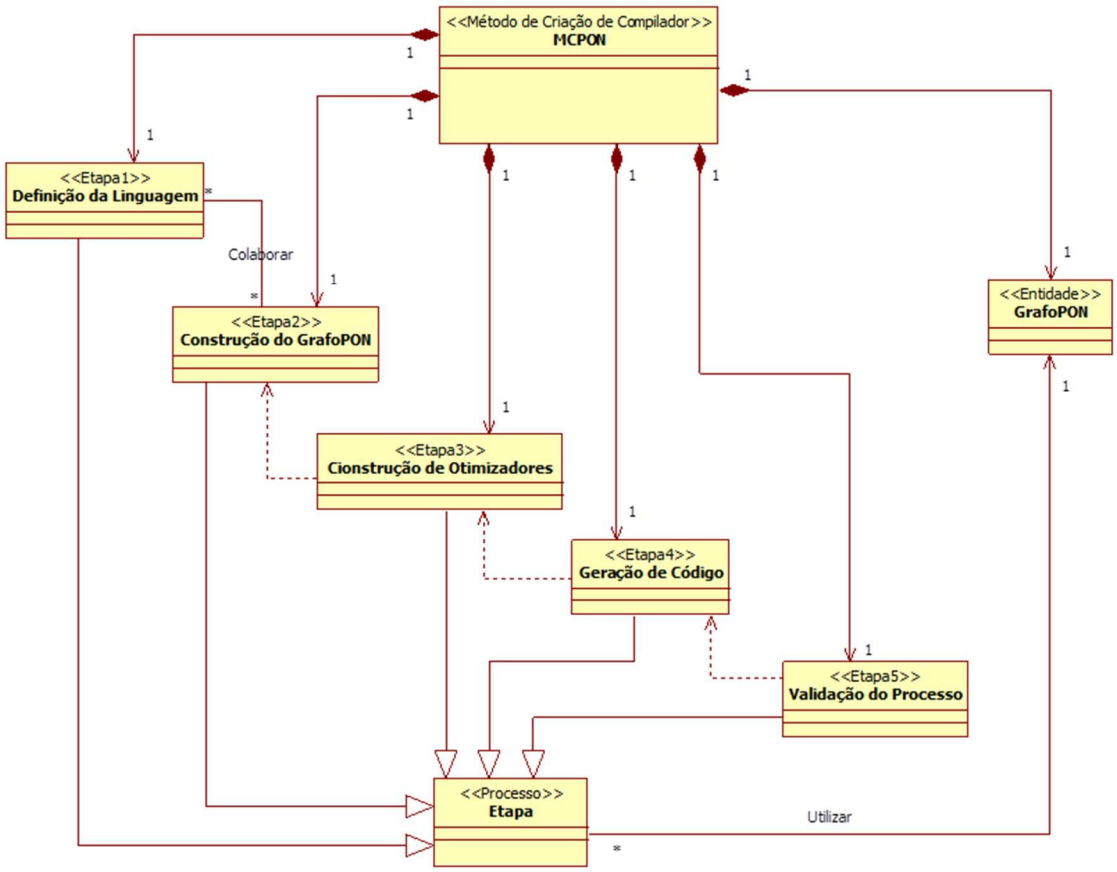
adequadas para a criação de compiladores próprios para o PON. Isso porque o PON se baseia em uma nova forma de construir e conectar as colaborações de um programa por meio de suas entidades notificantes. Neste âmbito, ele propôs um novo método para uniformizar o processo de construção de linguagens e compiladores específicos para o PON em plataformas distintas. Para esse método foi dado o nome de MCPON (RONSZCKA, 2019).

O primeiro passo para estruturação do MCPON foi a definição de um conjunto de diretrizes e regras para a construção de uma representação intermediária adequada para programas PON. Esta representação visava substituir apropriadamente as representações baseadas em árvore e mesmo as representações codificadas em linguagens de mais baixo nível, dos métodos de compilação tradicionais, os quais são pouco adequados ao PON. Nesse âmbito, vislumbrou-se que uma estrutura de dados em forma de grafo poderia representar apropriadamente um programa PON (RONSZCKA, 2019).

Em suma, o advento do Grafo PON tornou-se um elemento balizador 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).

Isto posto, o Grafo PON se constitui naturalmente em um dos elementos fundamentais do método proposto, nomeadamente MCPON. A Figura 23 apresenta o Grafo PON em relação aos demais constituintes do método em questão. Na figura, os constituintes do PON são modelados em termos de um diagrama de classes em UML sendo que, naturalmente neste caso, as instâncias não são objetos usuais, mas sim elementos estereotipados (segundo suas naturezas) que permitem compor todo o MCPON (RONSZCKA, 2019).

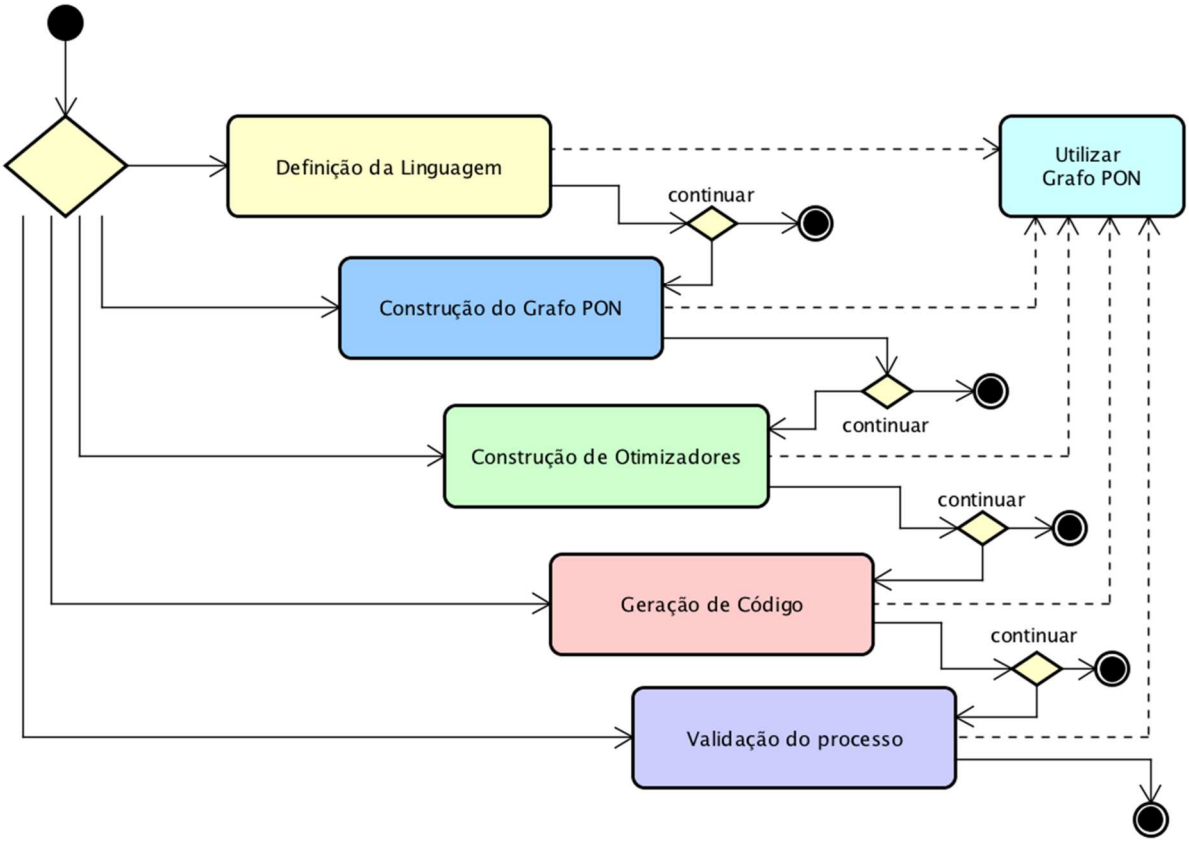
Figura 23. Visão geral estrutural do MCPON



Fonte: (RONSZCKA, 2019)

De maneira geral, conforme o modelo da Figura 23, o MCPON institui um sistema completo para o processo de compilação, sendo seus constituintes o já introduzido Grafo PON e cinco etapas que se especializam cada qual conforme seu papel. Nesse âmbito, de modo a ilustrar as principais etapas do método, a Figura 24 esboça a visão geral das atividades delas no MCPON (RONSZCKA, 2019).

Figura 24. Visão geral das etapas do MCPON



Fonte: (RONSZCKA, 2019)

Conforme apresenta a Figura 24, o método MCPON é constituído por cinco etapas, as quais se apoiam no Grafo PON. Em linhas gerais, a primeira etapa do método visa essencialmente construir linguagens particulares para o PON. A segunda etapa visa nortear o processo de construção do Grafo PON, inter-relacionando-se com a primeira etapa para este fim. A terceira etapa visa a construção de otimizadores, com o objetivo de eliminar possíveis redundâncias nos grafos gerados. A quarta etapa visa a transformação/tradução dos grafos em códigos-alvo tanto em linguagens quanto em plataformas distintas. Por fim, a quinta etapa visa a construção de validadores, com o objetivo de verificar a completude de cada compilador arquitetado (RONSZCKA, 2019).

3.7.2 Grafo PON

De maneira geral, a estrutura do PON, definida por meio das entidades notificantes que compõem seu modelo de programação, possibilitou o advento de uma inovação no processo de compilação. Esta inovação permite traduzir programas distintos escritos em linguagens próprias ao PON em uma única estrutura de dados uniforme. Esta estrutura possui um formato de grafo, no qual é possível acomodar representações de todas as entidades extraídas de um programa PON. Estas entidades, em conjunto, formam a definição completa de um programa. Para essa estrutura foi dado o nome de Grafo PON (RONSZCKA, 2019).

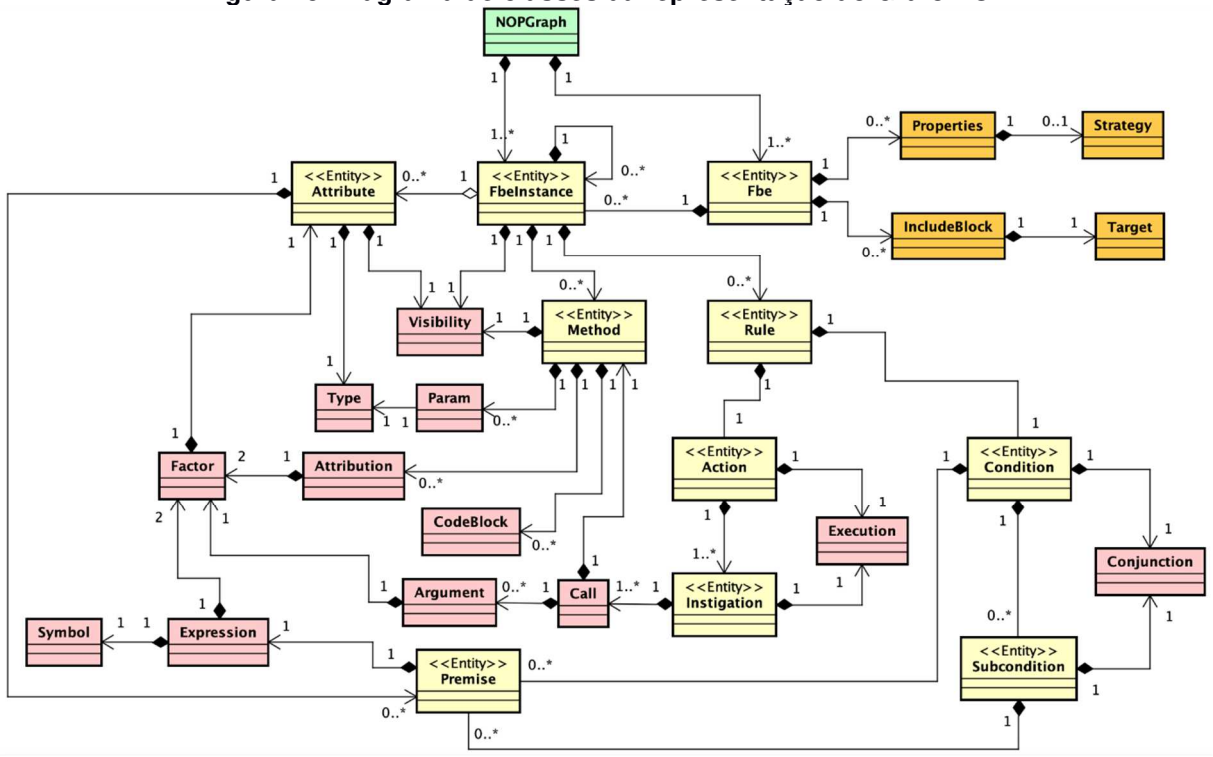
Diferente dos métodos tradicionais que mapeiam as características dos elementos de um programa em uma tabela de símbolos, bem como a execução algorítmica do programa em um conjunto de árvores sintáticas abstratas (Abstract Syntax Trees – AST, do inglês), no MCPON o mapeamento das entidades é feito unicamente no Grafo PON. De maneira geral, seguindo as diretrizes para a composição dos grafos especializados, estes grafos armazenariam cada um dos elementos de um dado programa de forma distinta, permitindo mapear tanto as particularidades das entidades que compõe o programa, quanto as conexões de toda a cadeia de notificações. Esta característica particular do Grafo PON possibilita a reconstrução de qualquer programa escrito sob os conceitos do PON em uma representação intermediária especializada (RONSZCKA, 2019).

Ademais, a forma com que os elementos são conectados uns aos outros, definindo relações por notificações entre eles, elimina a necessidade de mapear o fluxo de execução algorítmica das instruções em uma árvore sintática abstrata, como ocorre em um programa regido pelo estilo tradicional de execução imperativa. Na verdade, a utilização destas árvores não se encaixa perfeitamente para o PON, uma vez que a construção de um programa PON não segue os princípios de programação imperativa (algorítmica e baseada em pesquisa), mas sim, um modelo de interconexão de entidades notificantes, o que, de fato, motiva a criação de uma estrutura particular e própria (RONSZCKA, 2019).

Em suma, esse grafo seria uma representação completa de todas as entidades participantes do processo de inferência por notificações, criado especificamente para representar as ligações entre os elementos de um dado programa. Na prática, todo e

qualquer programa desenvolvido para o PON poderia ter sua representação mapeada no Grafo PON, desde que sejam seguidas algumas diretrizes para a construção desta estrutura. Neste âmbito, o diagrama de classes ilustrado na Figura 25 apresenta o modelo genérico de entidades e suas dependências, bem como possíveis conexões (RONSZCKA, 2019).

Figura 25. Diagrama de classes da representação do Grafo PON



Fonte: (RONSZCKA, 2019)

Conforme ilustra o diagrama de classes na Figura 25, a classe principal *NOPGraph* representa o ponto central do mapeamento das entidades de um programa em PON. Basicamente, essa classe possui relações com a classe *FBE* e com a classe *FBEInstance*. Na prática, um programa PON pode conter de *1* a *N* definições de *FBE*, assim como pode conter de *1* a *N* instâncias de *FBE*, os quais são acomodados nessas relações (RONSZCKA, 2019).

A classe *FBE* possui algumas relações que definem, de maneira genérica, uma entidade *FBE* em um programa PON. Nesta classe são definidas as particularidades compartilhadas entre todas as instâncias de cada *FBE*, como suas propriedades e demais características próprias para a geração de código (RONSZCKA, 2019).

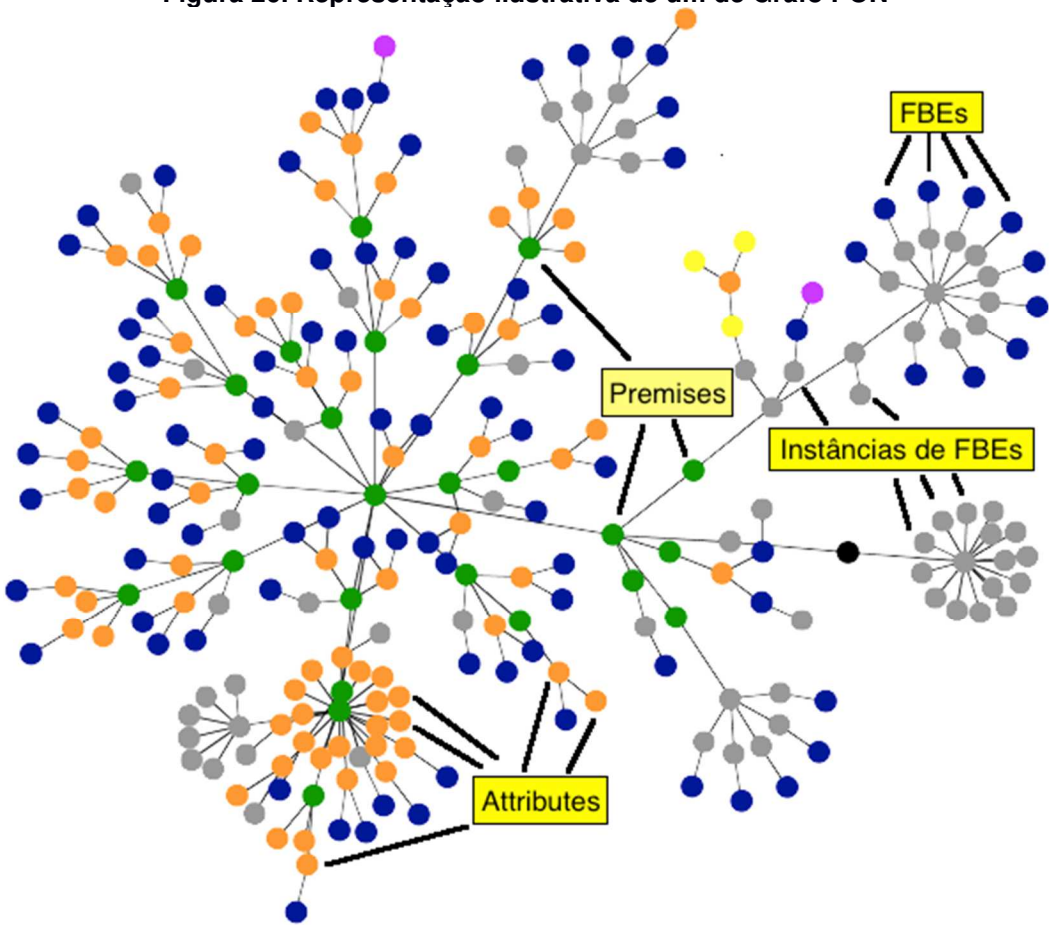
A classe *FBEInstance*, por sua vez, representa as instâncias de um *FBE* em um programa PON. Esta classe é tida como elemento central, uma vez que interconecta, em sua essência, todos os elementos que, em conjunto, formam a lógica de um programa desenvolvido em PON. De maneira geral, uma instância de um *FBE* é constituída pela parte facto-execucional de sua representação, com base em seus respectivos *Attributes* e *Methods* e pela parte lógico-causal, a qual é representada por *Rules* e demais entidades colaboradoras (*Condition, Premise, Action, Instigation*) (RONSZCKA, 2019).

Além disso, instâncias de *FBE* podem agregar *N* outras instâncias de *FBE*, criando assim, uma relação hierárquica entre elas. Esta hierarquia de instâncias de *FBE*, bem como suas respectivas partes facto-execucionais e lógico-causais, estabelece um grafo de entidades interconectadas que, em conjunto, formam um programa completo em PON (RONSZCKA, 2019).

Ainda, conforme ilustra o diagrama, as demais classes auxiliares, representadas pela cor vermelha, permitem fazer a interconexão desse conjunto de classes ‘principais’, mapeando particularmente as referências para os elementos da cadeia de notificação. Além disso, estas classes possuem o objetivo de complementar a estrutura do grafo com informações particulares e pontuais de cada uma das entidades que as agregam (RONSZCKA, 2019).

É importante ressaltar que o diagrama de classes expõe as classes fundamentais da estrutura, as quais devem ser mapeadas pontualmente e individualmente, de acordo com a estrutura do programa a ser representado. Neste sentido, de modo a elucidar o entendimento do Grafo PON, a Figura 26 é uma representação análoga do que seria a estrutura de um programa qualquer (RONSZCKA, 2019).

Figura 26. Representação ilustrativa de um de Grafo PON



Fonte: (RONSZCKA, 2019)

A Figura 26 ilustra como seria a composição de um Grafo PON. Basicamente, cada recorrência de um elemento significativo em um dado programa (*e.g. FBE,* Instâncias de *FBE, Attributes* de instâncias de *FBE* etc.) seria representado por um nó no Grafo PON. Ademais, cada relação notificante entre uma entidade e outra é mapeada com ligações (*i.e.,* arestas) no grafo. Desta maneira, qualquer programa escrito com os princípios do PON poderia ser mapeado de forma completa em uma única estrutura de dados.

Em suma, a construção do Grafo PON tem por objetivo auxiliar no processo de compilação, mapeando de forma completa a essência de um programa. Com base neste mapeamento fidedigno é possível realizar a tradução deste mesmo programa para uma outra representação especializada. De fato, este processo constitui em uma das principais etapas de um compilador e, por este motivo, precisou do advento do chamado Grafo PON para possibilitar um processo de compilação efetivo.

3.8 CONSIDERAÇÕES SOBRE O CAPÍTULO

O presente capítulo apresenta o PON, que possui desacoplamento natural dos elementos viabilizando a construção distribuída de programas. Deste paradigma surge a linguagem NOPL, uma ferramenta que permite padronizar a expressão de softwares PON. Com o MCPON, o compilador PON, é alcançado o atual estado da arte da tecnologia NOPL no qual é possível viabilizar materialização em diversas plataformas. Foram expostos também, os esforços no tocante às materializações *multicore* em software PON as quais, apesar de esforços conjuntos, não obteve resultados satisfatórios.

À luz destes resultados insatisfatórios em software PON *multicore*, refletiu-se sobre a hipótese destes resultados serem em função da forma de como foram implementados. Sendo mais preciso, da implementação do balanceamento dos elementos PON nesta tecnologia. Partindo, então, desta premissa, refletiu-se sobre soluções e arquiteturas atuais que possuíam resultados comprovados em distribuição e paralelismo *multicore*. Desta premissa, deriva, então o trabalho que é apresentado no capítulo a seguir.

4 DESENVOLVIMENTO DO TRABALHO

Este capítulo apresenta o desenvolvimento do trabalho. Conforme descrito na seção 1.2 o objetivo principal é a proposta de uma solução via Tecnologia NOPL que traduza os elementos PON, naturalmente desacoplantes, em microatores NOPL Erlang-Elixir permitindo assim, sua execução em plataforma Erlang com suporte multicore. Com esta tecnologia, espera-se possibilitar a execução concorrente e paralela com balanceamento apropriado entre os núcleos. Com isto, alcançar-se-ia a programação multicore apropriada e transparente, ou seja, sem conhecimento anterior de concorrência e paralelismo por parte do desenvolvedor. Desta forma, para alcançar o objetivo principal, este trabalho é distribuído em etapas, as quais são descritas a seguir:

- PON e Modelo de atores: nesta etapa objetiva-se uma proposta de tradução dos elementos PON em modelo de passagem de mensagens assíncronas, compatível com atores.
- Modelagem UML: nesta etapa, tem-se por objetivo específico, uma modelagem UML baseada na tradução dos elementos PON para microatores, os quais foram propostos na etapa anterior.
- *Framework* NOP Elixir: esta etapa tem como objetivo a construção dos elementos PON na forma de um *framework* na linguagem Elixir, conforme modelado em UML na etapa anterior.
- NOPL para o framework NOP Elixir: nesta etapa objetiva-se, por fim, a construção de um compilador da linguagem NOPL que traduz, através do Grafo PON, os elementos PON em instâncias do *Framework* NOP Elixir.

4.1 PON E MODELO DE ATORES

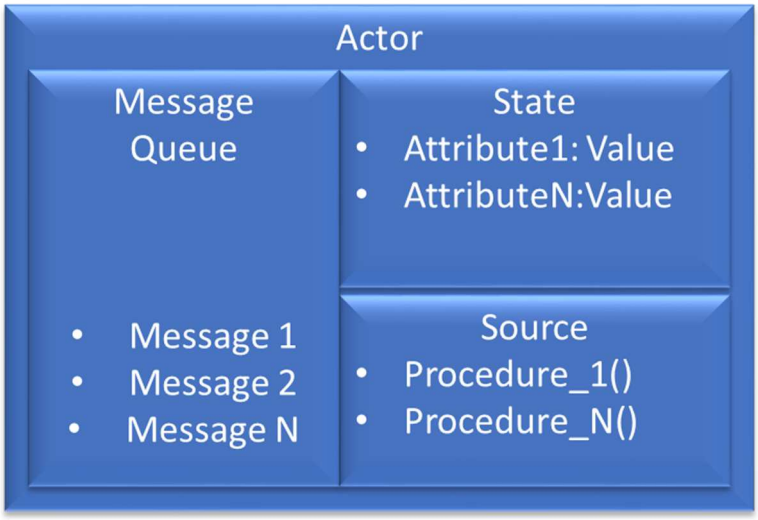
Como visto na seção 2.4.5, Van Roy define o modelo de atores como uma extensão do modelo declarativo. Esta extensão é determinada pela passagem assíncrona de mensagens entre os agentes computacionais elaborados declarativamente. PON, por sua vez, é um paradigma o qual carrega os atributos do

modelo declarativo concorrente como visto na seção 3.3. Isto posto, o caminho para o aproveitamento da arquitetura concorrente Erlang envolve a adequação do atual modelo a um modelo do PON de passagem de mensagens assíncronas que atenda os princípios semânticos do modelo de atores, descrito na seção 2.4. Desta forma, os princípios básicos da semântica do modelo de atores descrita por Hewitt (1973) (1977) pode ser expressa de maneira simplificada como:

1. Enviar um número finito de mensagens para outros atores;
2. Criar um número finito de novos atores;
3. Designar o comportamento a ser usado para a próxima mensagem que receber;
4. As informações em um modelo de ator devem ser transmitidas por, e somente por, mensagens.
5. O ator só poderá se comunicar com outro ator se este lhe for conhecido por meio de Nome, endereço ou ID.

Assim, o aproveitamento do Erlang como plataforma para código a ser gerado via Tecnologia NOPL é possível por meio da tradução dos elementos do paradigma PON em elementos que satisfaçam todas as condições descritas acima e respeitem o modelo genérico dado, conforme apresentado na Figura 27. Nesta figura observa-se três elementos que constituem o modelo de atores: a fila de mensagens (*Message Queue)*, que permite a organização de mensagens a serem processadas pelo ator de forma a permitir um ambiente concorrente; o estado (*State*), que armazena informações pertinentes ao ator; e o código fonte (*Source)*, onde estão dispostos os algoritmos para o tratamento da fila de mensagens.

Figura 27. Modelo genérico de ator com separação entre estado e processo

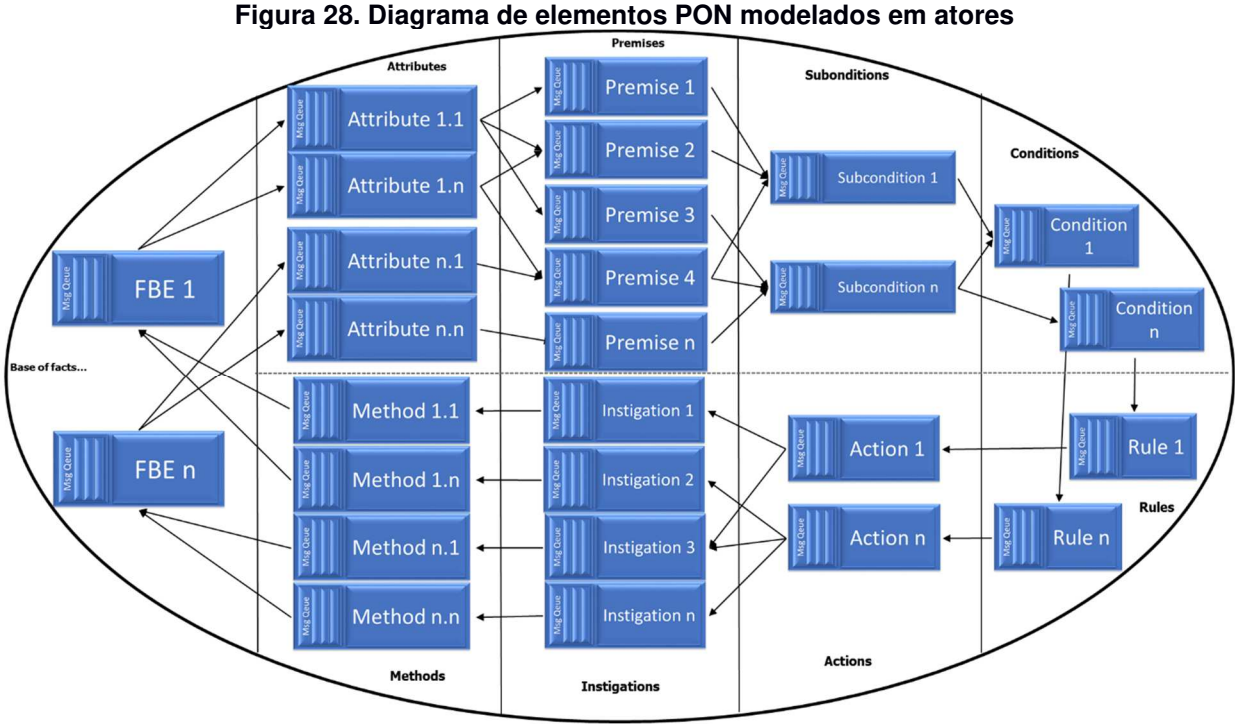


Fonte: adaptado de (HEWITT, 1973)

4.1.1 Elementos PON definidos como atores

De maneira geral, para a devida modelagem dos elementos PON em atores, cada elemento é apresentado na forma descrita na Figura 27. Em seguida, todas as notificações são modeladas como mensagens assíncronas. Desta forma tem-se um novo diagrama de relacionamento dos elementos PON agora modelados para atores. Este diagrama é apresentado na Figura 28.

Figura 28. Diagrama de elementos PON modelados em atores



Fonte: Autoria própria

A modelagem clássica de atores sugere que um ator seja a representação completa de um modelo. Porém, a proposta apresentada que traz a tradução dos elementos PON para atores, apresenta uma diferente abordagem em relação à abordagem clássica. Cita-se duas principais diferenças:

- **A tradução de programa PON representa uma parte desacoplada**. Em função do PON ser desacoplado, tanto os elementos da base de fatos (*FBEs, Attributes* e *Methods*), quanto suas condições lógico-causais (*Rule, Condition, Subcondition, Premise)* e ativadores (*Action e Instigation)* são desmembradas em vários atores.
- **Os fluxos de processamento executados nestes atores são menores**. Em função deste desmembramento do elemento lógico, cada elemento PON modelado para ator carrega uma pequena parte do fluxo do processamento, permitindo um maior paralelismo. Este fato permite, em teoria, um ganho de *speedup* máximo conforme proposto por Amdahl (1967).

Em virtude destes diferenciais de modelagem, foi decidido usar o termo “Microator” para diferenciar o produto da tradução de um elemento PON em ator de um ator clássico. Desta forma, facilita-se o entendimento de quando se faz referência

a um elemento PON traduzido para atores de uma referência usual de atores da literatura.

A seguir, são apresentados cada um dos elementos PON definidos na Figura 28 e formatados na modelagem definida na Figura 27, sendo que cada modelagem é submetida aos princípios da semântica de atores de forma a demonstrar sua adequação a este modelo. Uma vez que todos os elementos PON são expressos em microatores, é possível então uma materialização destes elementos em soluções fundamentadas neste modelo para atores, em tecnologias como o Erlang.

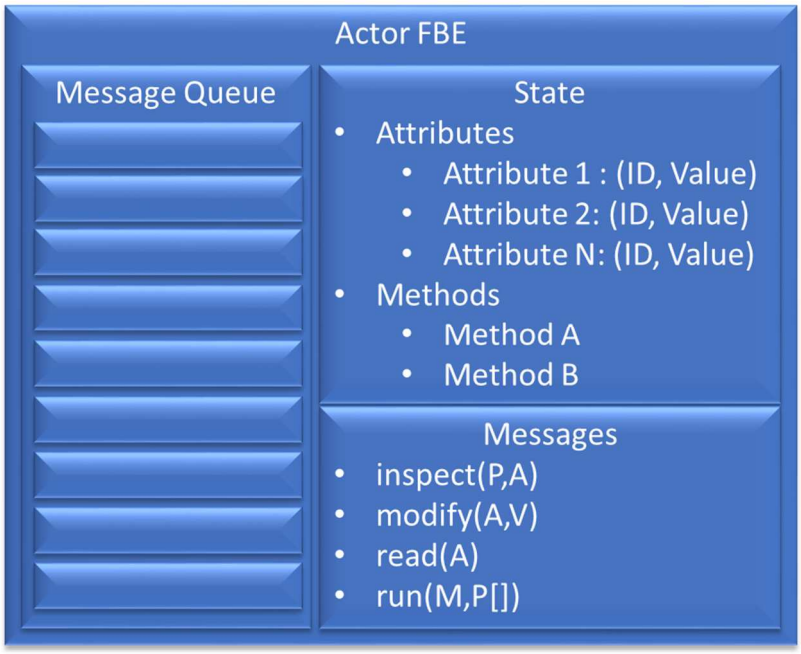
FBE como Microator

Um *FBE* é definido como um microator cujo estado contém uma lista de *Attributes* em forma de tupla, cada tupla contendo o endereço (ID) e o valor atual do *Attribute.* Ainda no estado, uma lista de *Methods* também é disponibilizada. No evento de criação, o microator FBE cria os microatores *Method,* bem como cria e inicializa os microatores *Attribute* correspondentes. O microator FBE responde às seguintes mensagens:

- **modify(A,V):** a mensagem *modify* é responsável por modificar o valor do *Attribute,* identificado no parâmetro *A*, para o valor identificado no parâmetro *V*. Esta mensagem identifica o *Attribute A* e envia uma mensagem *Attribute.modify(V).* Esta mensagem é a modelagem do mecanismo de notificação para modificação do valor de um *Attribute.*
- **inspect(P,A):** a mensagem *inspect* identifica o Attribute A e envia uma mensagem *Attribute.inspect(P)* solicitando ao microator que inclua o microator *Premise,* identificado pelo parâmetro *P,* em sua lista de *Premises* interessadas na mudança do valor do *Attribute.*
- **read(A):** A mensagem *read* identifica o *Attribute A* e envia uma mensagem *Attribute.read()* para recuperar o valor atual do *Attribute,* passando também o endereço do remetente da mensagem o qual será posteriormente notificado com uma mensagem contendo o valor do *Attribute.*
- **run(M,P):** A mensagem *run* identifica o Method e envia uma mensagem *Method.run(P)*, onde P são os parâmetros para a execução.

Uma representação do FBE em microator é apresentada na Figura 29.

Figura 29. Modelo alto nível de ator *FBE*



Fonte: Autoria própria

A validação do modelo com passagem de mensagem assíncrona para *FBE* com as características necessárias para modelo de atores pode ser verificada na Tabela 3.

Tabela 3. Aderência do elemento *FBE* em modelo de atores

Característica	Sit.	Justificativa
1. Número finito de mensagens	✓	• Envio de notificações, tanto para seus *Attributes*, quanto para seus *Methods* são limitados aos elementos previamente conhecidos – sempre um conjunto finito.
2. Número finito de atores	✓	• *FBE* gera *Attributes* e *Methods*. Um número sempre finito de atores
3. Definir comportamento para próxima mensagem.	✓	• Ao ser solicitado, o *FBE* pode consultar valores de seus *Attributes*, alterando o estado e permitindo um novo comportamento para a próxima mensagem.

4. Comunicação somente por mensagem		• O fluxo de notificação de maneira geral pode ser apresentado como um conjunto mensagens em modelagem de atores.
5. Conhecer o ID do ator para enviar mensagem		• Envio de notificações está limitado aos *Attributes e Methods conhecidos pelo fato de ser o ator criador destes microatores*, portanto o ator *FBE* conhece os endereços de todos os microatores os quais deseja se comunicar por meio de mensagens.

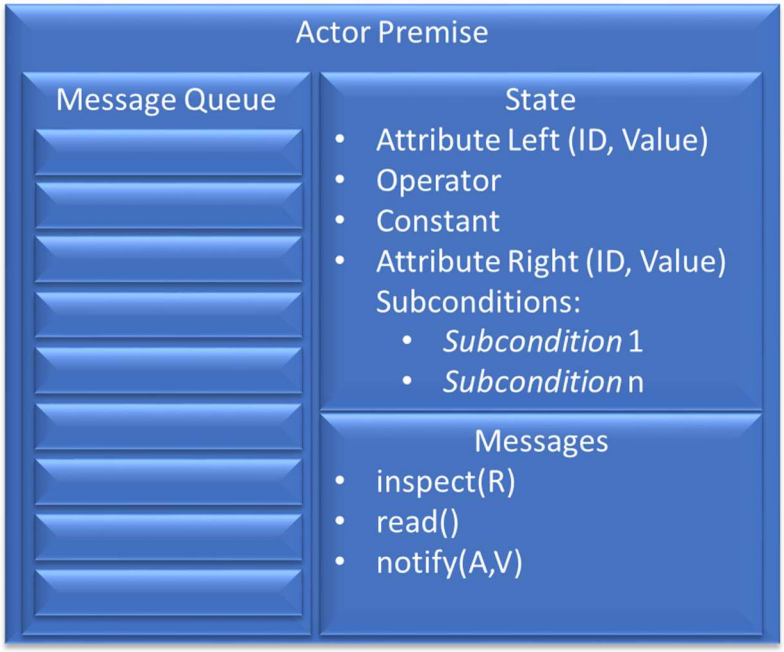
Attribute como Microator

Um *Attribute* é definido como um microator cujo estado contém o endereço (ID) do FBE que o criou, o Valor atual e uma lista de *Premises* interessadas nas modificações de seu valor. O microator *Attribute* responde às seguintes mensagens:

- **modify(V):** a mensagem *modify* é responsável por modificar o valor do *Attribute.* Neste momento, o microator *Attribute* envia mensagens *Premise.notify(V)* à todas as *Premises* interessadas na mudança de seu valor. Esta mensagem é a modelagem do mecanismo de notificação para modificação do valor de um *Attribute.*
- **inspect(P):** a mensagem *inspect* faz com que o microator *Attribute* reconheça a *Premise,* identificada em *P,* como interessada nas mudanças de seus valores. Neste momento então, a *Premise P* é inserida em sua lista de *Premises.* Uma mensagem com o valor atual é enviada à *Premise P* contendo seu valor atual para sua inicialização e/ou avaliação.
- **read():** A mensagem *read* faz com que o microator envie uma mensagem ao remetente com seu valor atual.

Uma representação do *Attribute* em microator é apresentada na Figura 30.

Figura 30. Modelo alto nível de ator *Attribute*



Fonte: Autoria própria

A validação do modelo com passagem de mensagem assíncrona para *Attribute* com as características necessárias para modelo de atores pode ser verificada na Tabela 4.

Tabela 4. Aderência do elemento *Attribute* em modelo de atores

Característica	Sit.	Justificativa
1. Número finito de mensagens	✔	• Envio de notificações é limitado ao número de *Premises* conhecidas – sempre um conjunto finito.
2. Número finito de atores	N/A	• *Attributes* não geram outros atores
3. Definir comportamento para próxima mensagem.	✔	• Quando solicitado, o *Attribute* pode adicionar *Premises* à sua lista, fato que pode interferir em seu comportamento.
4. Comunicação somente por mensagem	✔	• O fluxo de notificação de maneira geral pode ser apresentado como um conjunto mensagens em modelagem de atores.
5. Conhecer o ID do ator para enviar mensagem	✔	• Envio de notificações está limitado às *Premises conhecidas*.

Method como Microator

Um *Method* é definido como um microator cujo estado contém o endereço (ID) do FBE que o criou, o algoritmo para execução e a lista de parâmetros. O microator *Method* responde somente à seguinte mensagem:

- **run(P[]):** a mensagem *run* é responsável por disparar a execução do algoritmo. Neste momento, os parâmetros são recuperados e transmitidos para o algoritmo que é executado. Os parâmetros podem ser constantes ou valores de *Attributes* de FBEs que são passadas para a função *Instigation.run()*, como será visto a seguir. Esta mensagem é a modelagem do mecanismo de notificação execução um *Method.*

Uma representação do *Method* em microator é apresentada na Figura 31.

Figura 31. Modelo alto nível de ator *Method*

Fonte: Autoria própria

A validação do modelo com passagem de mensagem assíncrona para *Attribute* com as características necessárias para modelo de atores pode ser verificada na Tabela 5.

Tabela 5. Aderência do elemento *Method* em modelo de atores

Característica	Sit.	Justificativa
1. Número finito de mensagens	✓	• O envio de notificações está limitado às notificações de mudança de valores de *Attributes* contidas no algoritmo.
2. Número finito de atores	N/A	• *Method* não geram outros atores
3. Definir comportamento para próxima mensagem.	N/A	• *Method* não possui mudança de comportamento ao longo de sua vida.
4. Comunicação somente por mensagem	✓	• O fluxo de notificação de maneira geral pode ser apresentado como um conjunto mensagens em modelagem de atores.
5. Conhecer o ID do ator para enviar mensagem	✓	• O *Method* conhece o ID do *FBE* que o criou. Outros FBEs que possam receber notificações de mudança de atributos também podem ser recuperados através dos parâmetros.

Premise como Microator

Uma *Premise* é definida como um microator cujo estado contém seu resultado lógico atual, um *Attribute* como operando à esquerda da expressão, este em forma de tupla contendo o endereço (ID) e o valor atual. Ainda no estado é armazenado o operador, um valor constante ou um segundo *Attribute* como operando à direita em forma de tupla, este também contendo o ID e o valor atual. Uma lista de *Subconditions* interessadas nas mudanças de seu resultado lógico também consta em seu estado. O microator *Premise* responde às seguintes mensagens:

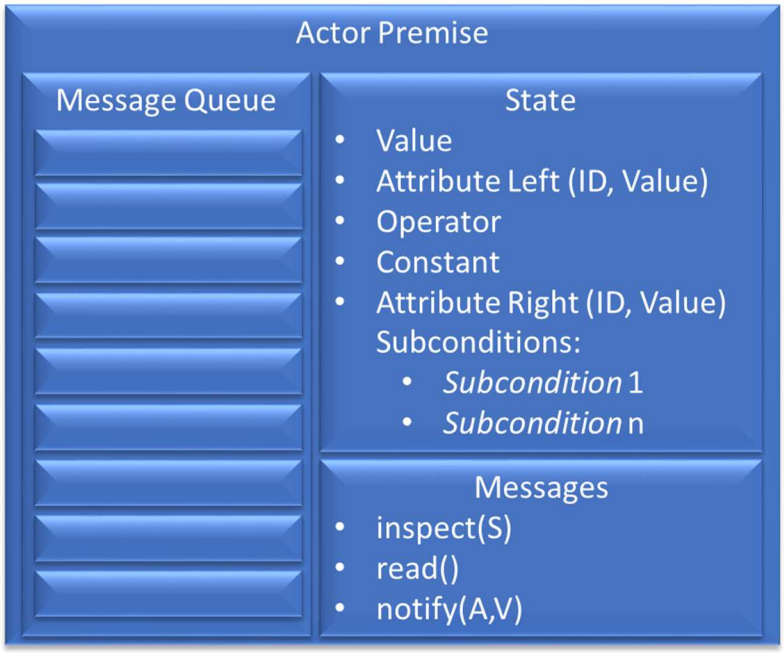
- **notify(A,V):** a mensagem *notify* é responsável por reavaliar o resultado lógico em função da notificação da mudança do valor do *Attribute,* identificado em *A,* para o novo valor *V.* Nesta mensagem, a *Premisse* armazena o valor atual e reavalia seu resultado lógico. Caso o resultado lógico tenha sido alterado, mensagens *Subcondition.notify(V)* são enviadas para as *Subconditions* interessadas nas mudanças de seu resultado lógico.

Esta mensagem é a modelagem do mecanismo de notificação para modificação do resultado lógico de uma *Premise.*

- **inspect(S):** a mensagem *inspect* faz com que o microator *Premise* reconheça a *Subcondition*, identificada em S, como interessada nas mudanças de seu resultado lógico. Neste momento então, a *Subcondition S* é inserida em sua lista de *Subconditions*. Uma mensagem com o resultado lógico atual é enviada à *Subcondition S* para sua inicialização e/ou avaliação.
- **read():** A mensagem *read* faz com que o microator envie uma mensagem ao remetente com seu resultado lógico atual.

A Figura 32 apresenta uma modelagem alto nível de um microator *Premise*.

Figura 32. Modelo alto nível de ator *Premise*



Fonte: Autoria própria

A validação do modelo com passagem de mensagem assíncrona para *Premise* com as características necessárias para modelo de atores pode ser verificada na Tabela 6.

Tabela 6. Aderência do elemento *Premise* em modelo de Atores

Característica	Sit.	Justificativa
1. Número finito de mensagens	✓	• Envio de notificações de mudança do valor lógico de sua expressão é limitado às *Subconditions* previamente cadastradas para receber estas notificações – sempre um conjunto finito.
2. Número finito de atores	N/A	• Elemento *Premise* não gera novos atores.
3. Definir comportamento para próxima mensagem.	✓	• Ao ser notificado por um *Attribute,* a *Premise* pode alterar seu valor lógico, bem como cadastrar ou descadastrar *Subconditions* de sua lista de notificações que influenciarão em seus comportamentos futuros.
4. Comunicação somente por mensagem	✓	• O fluxo de notificação de maneira geral pode ser apresentado como um conjunto mensagens em modelagem de atores.
5. Conhecer o ID do ator para enviar mensagem	✓	• Envio de notificações está limitado às *Subconditions* previamente cadastradas, portanto é pré-requisito que o ator *Premise* conheça os endereços de todos os atores relacionados para o envio das notificações por meio das mensagens.

Subcondition como Microator

Uma *Subcondition* é definida como um microator cujo estado contém seu resultado lógico atual, a expressão lógica a ser analisada e uma lista de *Premises* em forma de tupla contendo o endereço (ID) e o resultado lógico atual. Ainda no estado encontra-se uma lista de *Conditions* interessadas nas mudanças de seu resultado lógico. O microator *Subcondition* responde somente às seguintes mensagens:

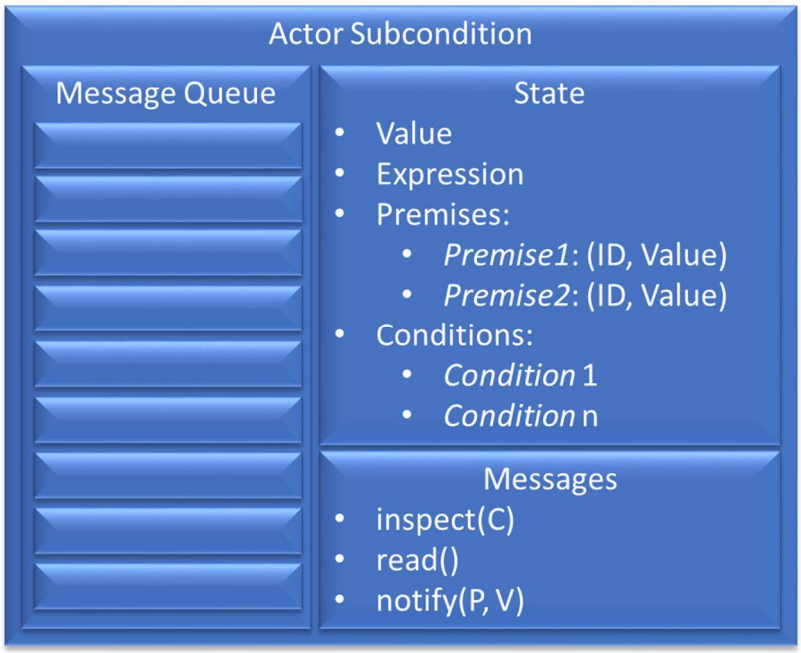
- **notify(P,V):** a mensagem *notify* é responsável por reavaliar o resultado lógico em função da notificação da mudança do valor de uma *Premise,*

identificado em *P,* para o novo resultado lógico *V.* Nesta mensagem, a *Subcondition* armazena o valor atual e reavalia seu resultado lógico. Caso o resultado lógico tenha sido alterado, mensagens *Condition.notify(V)* são enviadas para as *Conditions* interessadas nas mudanças de seu resultado lógico. Esta mensagem é a modelagem do mecanismo de notificação para modificação do resultado lógico de uma *Subondition.*

- **inspect(C):** a mensagem *inspect* faz com que o microator *Subcondition* reconheça a *Condition*, identificada em C, como interessada nas mudanças de seu resultado lógico. Neste momento então, a *Condition C* é inserida em sua lista de *Conditions*. Uma mensagem com o resultado lógico atual é enviada à *Condition C* para sua inicialização e/ou avaliação.
- **read():** A mensagem *read* faz com que o microator envie uma mensagem ao remetente com seu resultado lógico atual.

A Figura 33 representa um modelo alto nível de microator para o elemento *Subcondition*.

Figura 33. Modelo alto nível de ator *Subcondition*



Fonte: Autoria própria

A validação do modelo com passagem de mensagem assíncrona para *Subcondition* com as características necessárias para modelo de atores pode ser verificada na Tabela 7.

Tabela 7. Aderência do elemento *Subcondition* em modelo de Atores

Característica	Sit.	Justificativa
1. Número finito de mensagens	✓	• Envio de notificações de mudança do valor lógico de sua expressão é limitado às *Conditions* previamente cadastradas para receber estas notificações – sempre um conjunto finito.
2. Número finito de atores	N/A	• Elemento *Condition* não gera novos atores.
3. Definir comportamento para próxima mensagem.	✓	• Ao ser notificado por uma *Premise,* a *Subcondition* pode alterar seu valor lógico, bem como cadastrar ou descadastrar *Conditions* de sua lista de notificações que influenciarão em seus comportamentos futuros.
4. Comunicação somente por mensagem	✓	• O fluxo de notificação de maneira geral pode ser apresentado como um conjunto mensagens em modelagem de atores.
5. Conhecer o ID do ator para enviar mensagem	✓	• Envio de notificações está limitado às *Conditions* previamente cadastradas, portanto é pré-requisito que o ator *Subcondition* conheça os endereços de todos os atores relacionados para o envio das notificações por meio de mensagens.

Condition como Microator

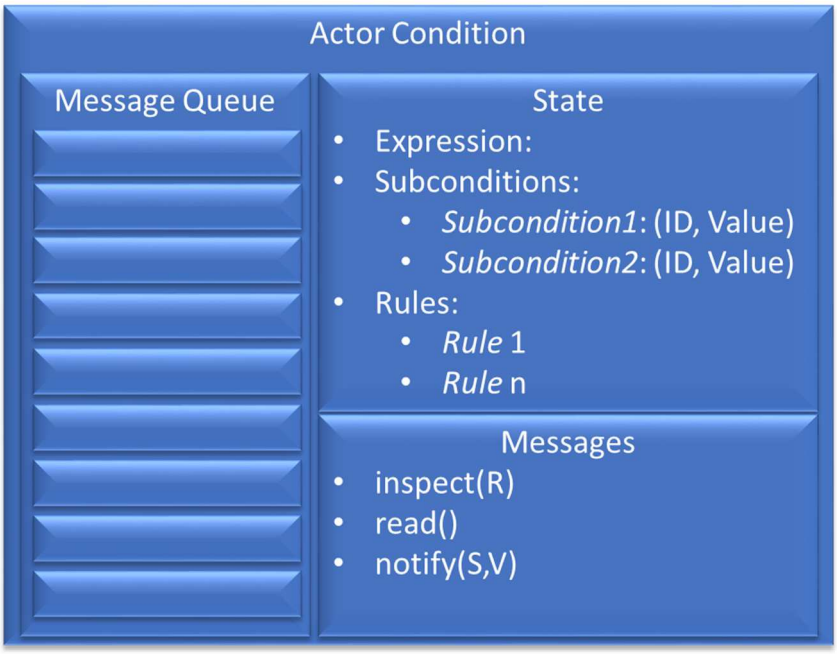
Uma *Condition* é definida como um microator cujo estado contém seu resultado lógico atual, a expressão lógica a ser analisada e uma lista de *Subconditions* em forma de tupla contendo o endereço (ID) e o resultado lógico atual. Ainda no estado

encontra-se uma lista de *Rules* interessadas nas mudanças de seu resultado lógico. O microator *Subcondition* responde somente às seguintes mensagens:

- **notify(S,V):** a mensagem *notify* é responsável por reavaliar o resultado lógico em função da notificação da mudança do valor de uma *Subcondition,* identificado em *S,* para o novo resultado lógico *V.* Nesta mensagem, a *Condition* armazena o valor atual e reavalia seu resultado lógico. Caso o resultado lógico tenha sido alterado, mensagens *Rule.notify(V)* são enviadas para as *Rules* interessadas nas mudanças de seu resultado lógico. Esta mensagem é a modelagem do mecanismo de notificação para modificação do resultado lógico de uma *Condition.*
- **inspect(R):** a mensagem *inspect* faz com que o microator *Condition* reconheça a *Rule*, identificada em R, como interessada nas mudanças de seu resultado lógico. Neste momento então, a *Rule R* é inserida em sua lista de *Rules*. Uma mensagem com o resultado lógico atual é enviada à *Rule R* para sua inicialização e/ou avaliação.
- **read():** A mensagem *read* faz com que o microator envie uma mensagem ao remetente com seu resultado lógico atual.

A Figura 34 apresenta um modelo de ator alto nível de microator para o elemento *Condition.*

Figura 34. Modelo alto nível de ator *Condition*



Fonte: Autoria própria

A validação do modelo com passagem de mensagem assíncrona para *Condition* com as características necessárias para modelo de atores pode ser verificada na Tabela 8.

Tabela 8. Aderência do elemento *Condition* em modelo de Atores

Característica	Sit.	Justificativa
1. Número finito de mensagens	✓	• Envio de notificações de mudança do valor lógico de sua expressão é limitado às *Rules* previamente cadastradas para receber estas notificações – sempre um conjunto finito.
2. Número finito de atores	N/A	• Elemento *Condition* não gera novos atores.
3. Definir comportamento para próxima mensagem.	✓	• Ao ser notificado por uma *Subcondition,* a *Condition* pode alterar seu valor lógico, bem como cadastrar ou descadastrar *Rules* de sua lista de notificações que influenciarão em seus comportamentos futuros.

4. Comunicação somente por mensagem		• O fluxo de notificação de maneira geral pode ser apresentado como um conjunto mensagens em modelagem de atores.
5. Conhecer o ID do ator para enviar mensagem		• Envio de notificações está limitado às *Rules* previamente cadastradas, portanto é pré-requisito que o ator *Condition* conheça os endereços de todos os atores relacionados para o envio das notificações por meio das mensagens.

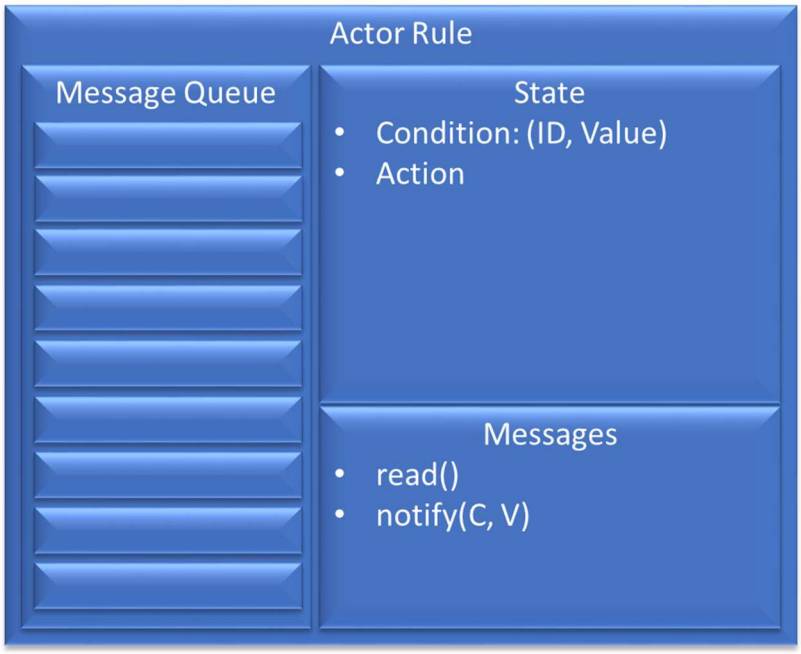
Rule como Microator

Uma *Rule* é definida como um microator cujo estado contém uma *Condition* a ser analisada em forma de tupla, contendo seu endereço (ID) e seu resultado lógico. Uma *Action* também é encontrada dentro do estado de um microator *Rule.* No evento de criação, o microator *Rule* cria os microatores *Condition* e *Action* guardando em seu estado os endereços correspondentes. O microator *Rule* responde às seguintes mensagens:

- **notify(C,V):** a mensagem *notify* é responsável por reavaliar o resultado lógico em função da notificação da mudança do valor de uma *Condition,* identificado em *C,* para o novo resultado lógico *V.* Nesta mensagem, a *Rule* armazena o valor atual e verifica o resultado lógico da *Condition C*. Caso o resultado lógico seja *verdadeiro*, uma mensagem *Action.run(F[])* é enviada para a *Action* que foi pela *Rule* criada. Esta mensagem é a modelagem do mecanismo de notificação para modificação do resultado lógico de uma *Rule.*
- **read():** A mensagem *read* faz com que o microator envie uma mensagem ao remetente resultado lógico atual da *Condition.*

A Figura 35 apresenta um modelo alto nível de microator para o elemento *Rule.*

Figura 35. Modelo alto nível de ator *Rule*



Fonte: Autoria própria

A validação do modelo com passagem de mensagem assíncrona para *Rule* com as características necessárias para modelo de atores pode ser verificada na Tabela 9.

Tabela 9. Aderência do elemento *Rule* em modelo de Atores

Característica	Sit.	Justificativa
1. Número finito de mensagens	✓	• O ator *Rule* envia mensagens de cadastro para a *Condition* a qual lhe interessa ser notificada, bem como para *Action por ela criada*. Sempre um conjunto finito de atores.
2. Número finito de atores	✓	• O ator *Rule* é o criador do microator *Action,* sempre limitado a um microator.
3. Definir comportamento para próxima mensagem.	✓	• Ao ser notificado por uma *Condition,* a *Rule* pode alterar seu valor lógico influenciando seu comportamento em mensagens futuras.
4. Comunicação somente por mensagem	✓	• O fluxo de notificação de maneira geral pode ser apresentado como um conjunto mensagens em modelagem de atores.

5. Conhecer o ID do ator para enviar mensagem		• Envio de notificações está limitado à *Condition* a qual a *Rule* tem interesse em ser notificada e à *Action* que irá notificar quando seu valor lógico se tornar verdadeiro.

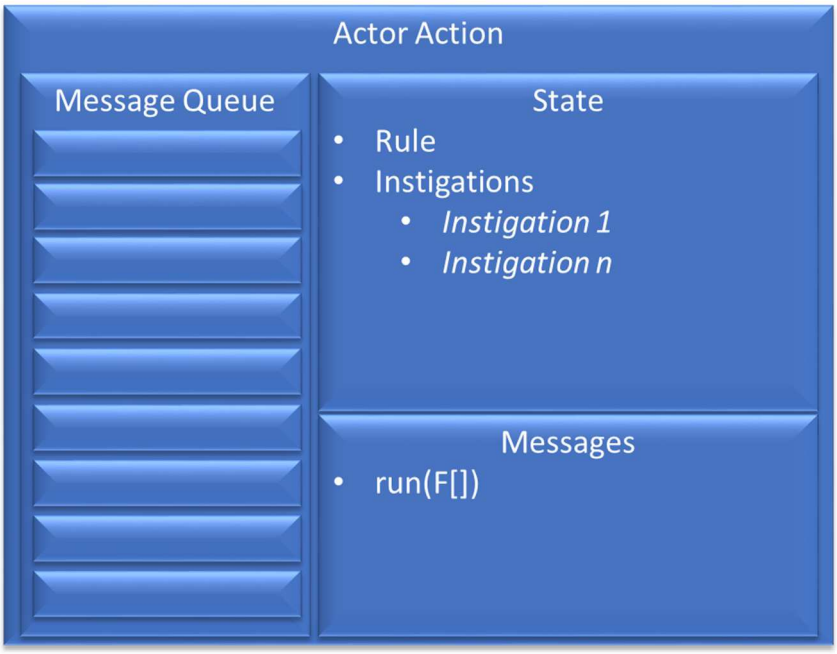
***Action* como Microator**

Um *Action* é definido como um microator cujo estado contém o endereço (ID) da Rule que o criou. Uma lista de *Instigations* também estão no estado do microator. No evento de criação, o microator *Action* cria os microatores *Instigation* guardando em seu estado os endereços correspondentes. O microator *Action* responde apenas à seguinte mensagem:

- **run(F[]):** a mensagem *run* é responsável por enviar mensagens *Instigation.run(F[])* para as *Instigations* que foram por ela criadas. Esta mensagem é a modelagem do mecanismo de notificação execução um *Method.*

A Figura 36 apresenta um modelo alto nível de microator para o elemento *Action.*

Figura 36. Modelo alto nível de ator Action



Fonte: Autoria própria

A validação do modelo com passagem de mensagem assíncrona para *Action* com as características necessárias para modelo de atores pode ser verificada na Tabela 10.

Tabela 10. Aderência do elemento *Action* em modelo de Atores

Característica	Sit.	Justificativa
1. Número finito de mensagens	✓	• O ator *Action* envia mensagens para as *Instigations* por ela criada. Sempre um conjunto finito de atores.
2. Número finito de atores	✓	• O ator *Action* cria microatores *Instigation,* sempre um conjunto finito de atores.
3. Definir comportamento para próxima mensagem.	N/A	• O ator *Action* não altera seu comportamento durante seu ciclo de vida.
4. Comunicação somente por mensagem	✓	• O fluxo de notificação de maneira geral pode ser apresentado como um conjunto mensagens em modelagem de atores.

5. Conhecer o ID do ator para enviar mensagem		• O ator *Action* gera os atores *Instigation*, portanto conhece seus IDs por ser seu criador.

Instigation como Microator

Uma *Instigation* é definida como um microator cujo estado contém o endereço (ID) da Action que a criou. Uma lista de *Methods* também estão no estado do microator. O microator *Instigation* responde apenas à seguinte mensagem:

run(F[]): a mensagem *run* é responsável por enviar mensagens *Method.run(P[])* para os *Methods* que foram por ela criadas traduzindo os valores disponíveis na lista de *FBE,* identificada em *F[]*. Esta mensagem é a modelagem do mecanismo de notificação execução um *Method.*

A Figura 37 apresenta um modelo alto nível de ator *Instigation*.

Figura 37. Modelo alto nível de ator *Instigation*

Fonte: Autoria própria

A validação do modelo com passagem de mensagem assíncrona para *Instigation* com as características necessárias para modelo de atores pode ser verificada na Tabela 11.

Tabela 11. Aderência do elemento *Instigation* em modelo de Atores

Característica	Sit.	Justificativa
1. Número finito de mensagens	✓	• O ator *Instigation* é responsável por executar os *Methods* dos *FBE,* portanto o envio de mensagens está subordinado à lógica do referido *Method*.
2. Número finito de atores	N/A	• O ator *Instigation* não gera novos elementos.
3. Definir comportamento para próxima mensagem.	N/A	• O ator *Instigation* não altera seu comportamento durante seu ciclo de vida.
4. Comunicação somente por mensagem	✓	• O fluxo de notificação de maneira geral pode ser apresentado como um conjunto mensagens em modelagem de atores.
5. Conhecer o ID do ator para enviar mensagem	✓	• O ator *Instigation* é responsável por disparar execução de *Method*, para isto necessita do ID do agente original para a execução.

4.1.2 Considerações sobre PON como Microatores

Como foi possível observar no detalhamento da subseção 4.1.1, o PON apresenta uma sinergia com o modelo de atores, pois é possível observar uma tradução de seus elementos para atores de uma forma bastante harmônica e intuitiva. Desta analogia entre as duas abordagens destaca-se a natureza holônica[22] em ambas. O paradigma PON possui sua origem em modelos holônicos, tal como o é em modelo de atores.

[22] Um holon é algo que é simultaneamente um todo e uma parte. A palavra foi usada por Arthur Koestler em seu livro O Fantasma na Máquina e palavra holon é uma tradução grega da palavra latina *universum*, no sentido de totalidade, um todo.

Outra sinergia também presente entre ambas abordagens é o método de comunicação entre suas entidades. Destaque para a comunicação através de notificações do PON, sinérgica à comunicação de mensagens do modelo de atores. A necessidade de conhecimento prévio dos destinatários com os quais se deseja comunicar também merece destaque na sinergia entre as abordagens.

4.2 MODELAGEM UML

Concluído o processo de conformação de cada elemento do paradigma PON com seus comportamentos (código-fonte) e atributos (estado), passa-se então para a etapa de materialização destes dentro da arquitetura Erlang. Esta seção, portanto, propõe uma modelagem UML fundamentada nas definições dispostas na seção 4.1.

4.2.1 Modelagem

Como atores são essencialmente objetos (HEWITT, 1973), é possível utilizar-se dos padrões da UML (*Unified Modeling Language*) para modelar, então, os elementos do *framework*. Outrossim, a modelagem em atores proposta na seção 4.1 é pouco detalhada, desta forma ela não possui informação suficiente para servir como base de documentação e apoiar um posterior desenvolvimento. Desta forma, uma modelagem mais completa e detalhada se faz necessária. Portanto, para atender à esta demanda, uma modelagem UML é apresentada tendo como base a modelagem de atores.

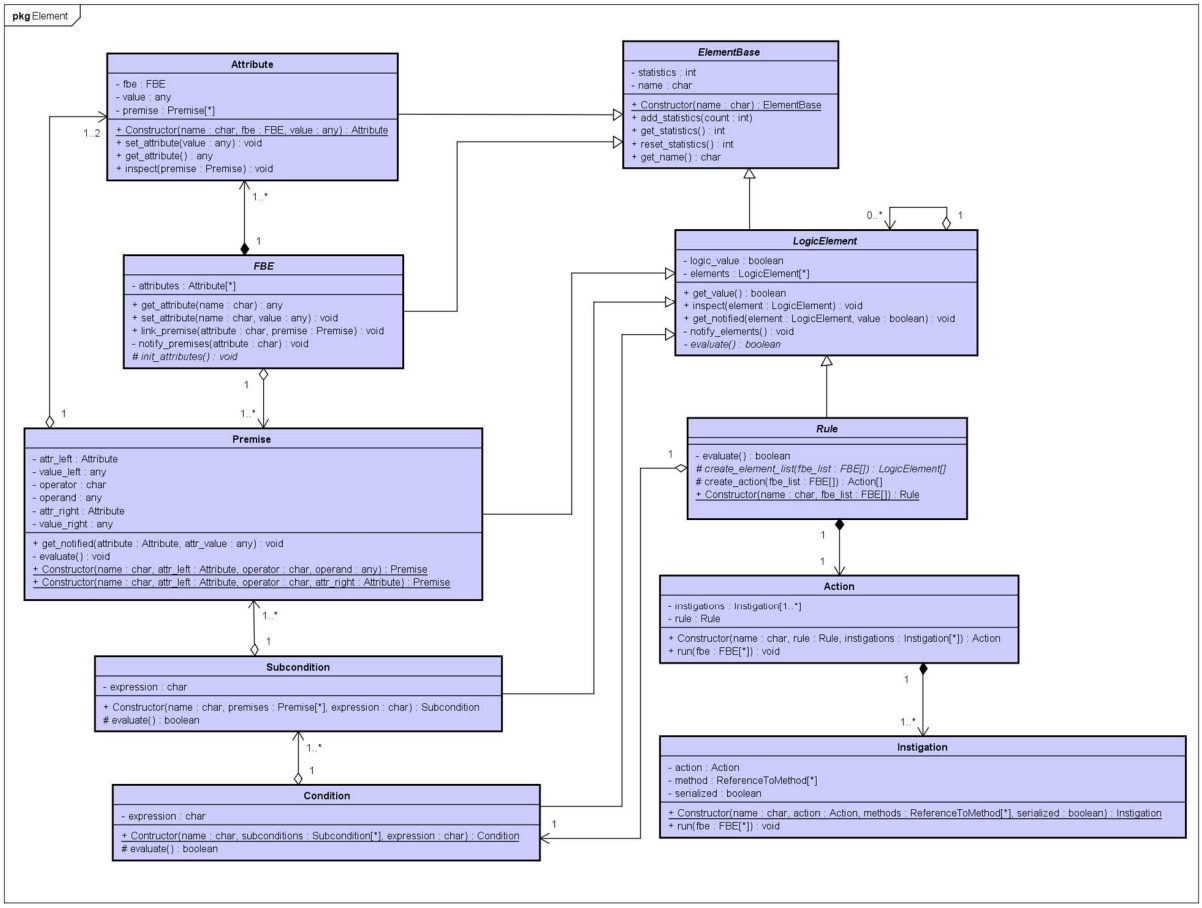
A modelagem UML é concebida visando não somente a execução de software PON com sua lógica e comportamento, mas também visa agregar ferramentas auxiliares para apoio como, por exemplo, funções para cálculos de tempo de execução e contagem de notificações que permitam analisar o comportamento dos modelos aplicáveis ao *framework*. Por este motivo, a modelagem está distribuída em dois grupos distintos de elementos:

- Elementos do paradigma: microatores que reproduzem o comportamento dos elementos do paradigma
- Elementos complementares: atores que auxiliam no controle e geração de informação complementar e para análise.

4.2.2 Elementos do paradigma PON

A modelagem dos elementos segue como inspiração a definição base de descrita na seção 4.1. O seu resultado pode ser visto na Figura 38.

Figura 38. Modelagem UML dos elementos do paradigma PON



Fonte: Autoria própria

A modelagem UML tem como proposta apresentar classes abstratas para os elementos PON *FBE* e *Rule* de forma que estes elementos sejam incorporados ao desenvolvimento de software PON como *framework*. Desta forma, o desenvolvimento consiste em criar classes específicas destes em seu projeto de software PON. As demais classes que modelam elementos PON (*Attribute, Premise, Subcondition, Condition, Action* e *Instigation)* são modeladas de forma a serem utilizadas sem a necessidade de especializações.

Conforme já mencionado, a modelagem proposta teve como orientação a modelagem de atores apresentada na seção 4.1, portanto é possível observar classes

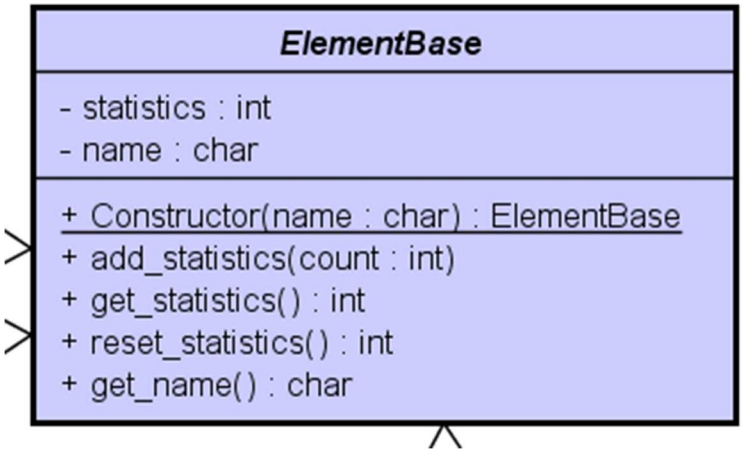
homônimas a quase todos os elementos PON, com a única exceção sendo o microator *Method.* O microator *Method* foi suprimido da modelagem, pois o modelo propõe que suas atribuições sejam construídas como métodos de classes FBE. Outra diferença ao modelo de atores é a inclusão de classes abstratas. Estas classes são apresentadas a fim de concentrar comportamentos comum, como é o caso da classe *ElementBase* e a classe *LogicElement.* A seguir, são detalhados cada um dos elementos dispostos nesta modelagem.

ElementBase

A classe *ElementBase* é a classe básica para todos os elementos do paradigma. Nela estão construídos os comportamentos comuns a todos elementos, a saber, atribuição de nome que pode ser verificado pelo atributo *name.* O *Constructor*, por sua vez, recebe o nome do objeto e o atribui ao atributo *name* que permanece imutável durante todo o ciclo de vida do objeto.

Também é possível verificar a quantidade de notificações geradas pelo atributo *statistics* e os métodos *reset_statistics, add_statistics e get_statistics* que respectivamente iniciam, incrementam e recuperam o valor de notificações acumulado no atributo privado *statistics.* A Figura 39 apresenta a modelagem UML do elemento *ElementBase.*

Figura 39. Modelagem UML para *ElementBase*

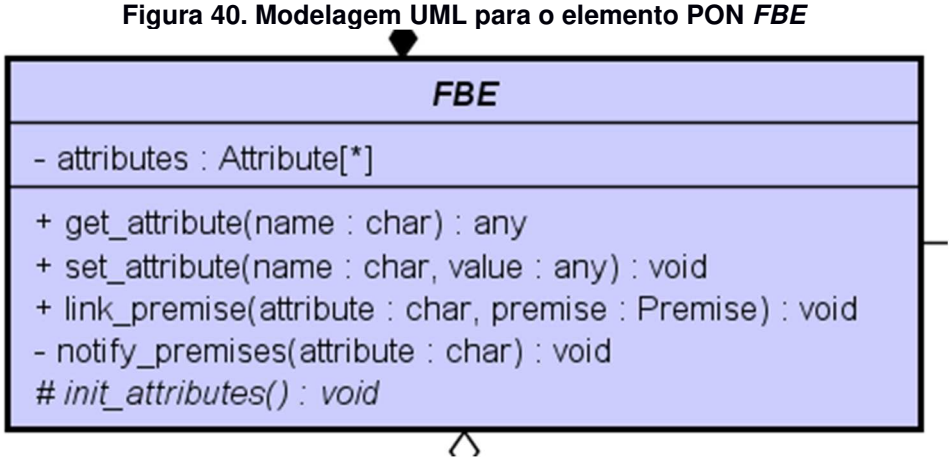


Fonte: Autoria própria

FBE

A classe *FBE* tem o objetivo de reproduzir os comportamentos do *FBE* do paradigma PON. Esta classe se propõe a ser uma classe abstrata para que as materializações possam produzir classes especializadas a partir dela. No processo de criação de classes específicas, a classe especializada precisará redefinir o método abstrato *ini_attributes().* Neste método deverá ser codificada a criação de objetos do tipo *Attribute* que serão armazenados no atributo *attributes.* Estes atributos podem ser lidos e escritos através dos métodos *get_attribute()* e *set_attribute()* respectivamente. Com o método *link_premise()* é possível cadastrar objetos da classe *Premise* como interessados nas mudanças de um atributo. Importante lembrar que os elementos PON *Method* devem ser então construídos como métodos da própria classe. A Figura 40 apresenta a modelagem detalhada da classe FBE.

Figura 40. Modelagem UML para o elemento PON *FBE*



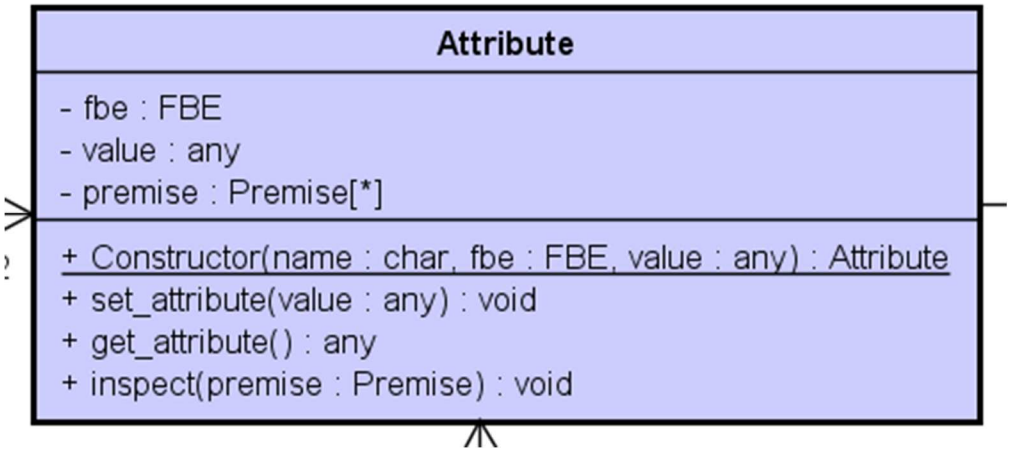
Fonte: Autoria própria

Attribute

A classe *Attribute* tem o objetivo de reproduzir os comportamentos do elemento PON *Attribute.* O método *Constructor* recebe como parâmetros seu nome, o objeto FBE e o valor inicial, estes armazenados nos atributos privados. Objetos do tipo *Premise* podem se cadastrar como interessadas nas alterações de valores a partir do método *inspect().* O valor pode ser lido e alterado pelos métodos *get_attribute()* e *set_attribute()* respectivamente, este segundo dispara as notificações às *Premises* por

meio do método *Premise.get_notified().* A Figura 41 apresenta o detalhamento da modelagem da classe *Attribute*.

Figura 41. Modelagem UML para o elemento PON *Attribute*



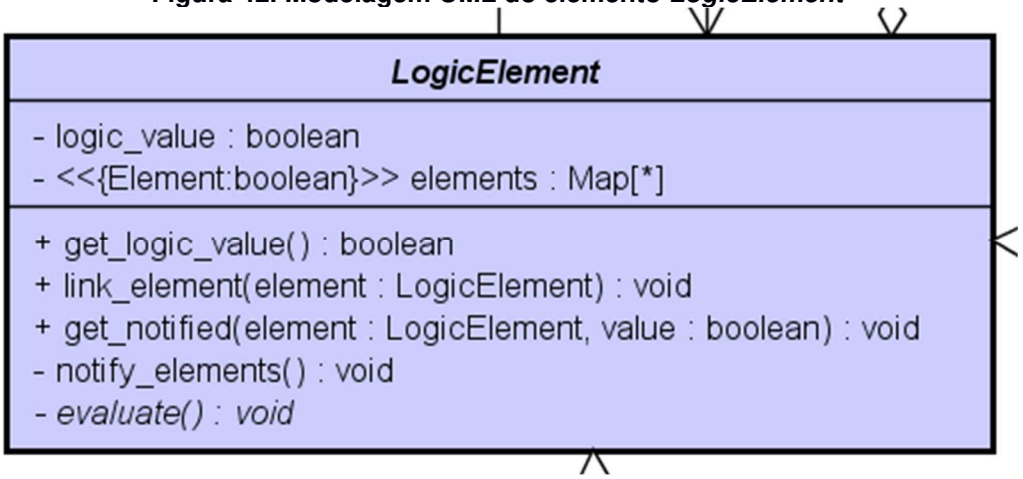
Fonte: Autoria própria

LogicElement

Com o objetivo de construir comportamentos comuns entre os elementos lógicos *Premise, Condition, Subcondition e Rule* a classe abstrata *LogicElement* se encarrega de criar todos os controles de ativação e notificação comuns a estas classes. O Atributo *logic_value* contém o valor lógico atual do elemento. Este valor lógico é atualizado sempre que for notificado pelo método *get_notified()*. Este, por sua vez, irá acionar o método abstrato *evaluate(),* que terá seu comportamento definido de forma específica para cada elemento como será visto a seguir.

O método *link_element()* permite que outros elementos lógicos se cadastrem para serem notificados quando da mudança de seu valor lógico. Os elementos conhecidos são armazenados no atributo *elements*. O valor lógico atual pode ser recuperado pelo método *get_logic_value()*. A Figura 42 apresenta a modelagem UML para o elemento *LogicElement.*

Figura 42. Modelagem UML do elemento *LogicElement*



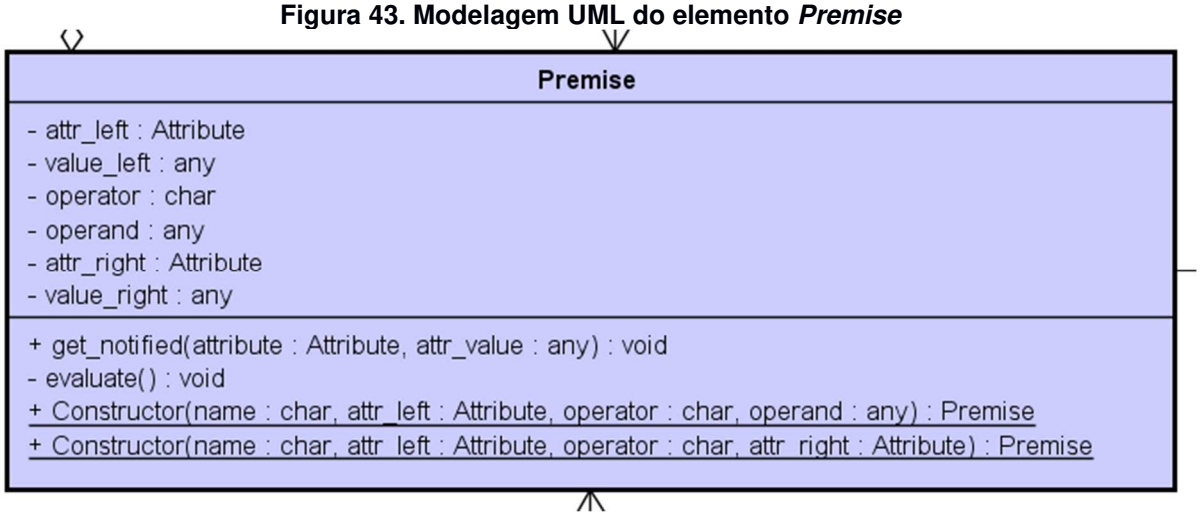
Fonte: Autoria própria

Premise

Como uma classe especializada de *LogicElement*, a *Premise* pode ser construída de duas formas: comparativo entre um *Attribute* a um valor constante ou comparativo entre dois *Attributes*. O método construtor *Constructor* é sobrecarregado de forma a permitir a criação de objetos baseando-se apenas na lista de atributos.

Nos atributos *atr_left,* e *value_left* ficam armazenadas as informações do *Attribute* à esquerda da expressão lógica. O Operador representado pelo atributo *operator*, que pode receber os valores *EQ, NE, GE, GT, LE e LT* respectivamente para operadores lógicos igual, diferente, maior ou igual, maior, menor ou igual e menor. O *operand* é usado quando da comparação entre um *Attribute* e um valor constante que fica nele armazenado. Os atributos *attr_right, e value_right* são responsáveis por armazenar as informações quando da comparação entre dois *Attributes*. Na criação do objeto, este se notifica aos elementos *Attribute* já recuperando destes os valores atuais envolvidos na sua expressão, permitindo assim, que já consiga definir seu valor lógico. É feita a sobrecarga no método público *get_notified()* de forma a ser utilizado pelos objetos *Attribute* quando estes necessitarem enviar uma notificação de mudança de valor. A Figura 43 apresenta a modelagem UML para o elemento *Premise*.

Figura 43. Modelagem UML do elemento *Premise*

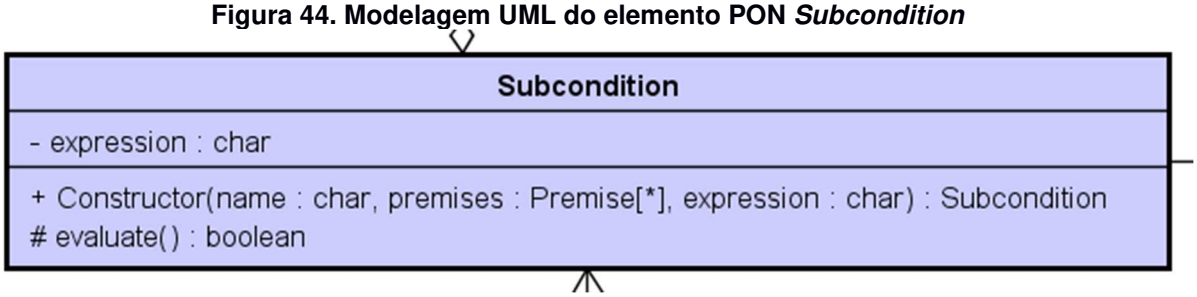


Fonte: Autoria própria

Subcondition

A classe *Subcondition* é apresentada como a abstração do elemento PON homônimo. Especializada de *LogicElement* esta classe redefine o método *evaluate()* e apresenta um método construtor com as informações necessárias à sua inicialização que são o nome, a expressão lógica e uma lista de objetos *Premise* os quais ela irá se cadastrar por meio do método *Premise.inspect().* A Figura 44 apresenta a modelagem UML detalhada da classe *Subcondition.*

Figura 44. Modelagem UML do elemento PON *Subcondition*



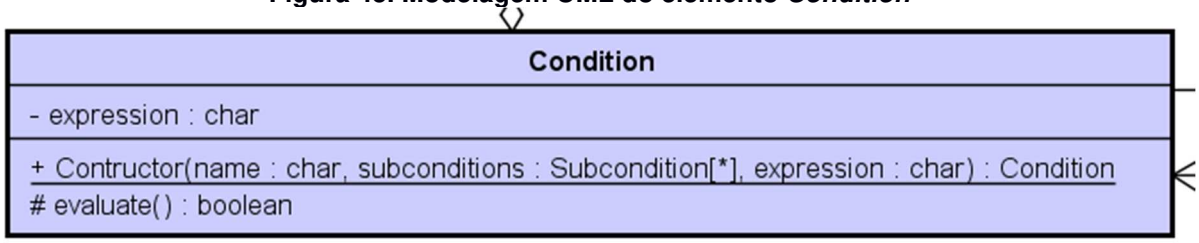
Fonte: Autoria própria

Condition

De maneira bastante análoga à classe *Subcondition*, a classe *Condition* também redefine o método *evaluate()* e recebe um construtor próprio com os parâmetros necessários à sua inicialização, que são o nome, a expressão e uma lista de objetos

Subcondition os quais o objeto se cadastra por meio do método *Subcondition.inspect().* A Figura 45 apresenta a modelagem UML para o elemento *Condition*.

Figura 45. Modelagem UML do elemento *Condition*

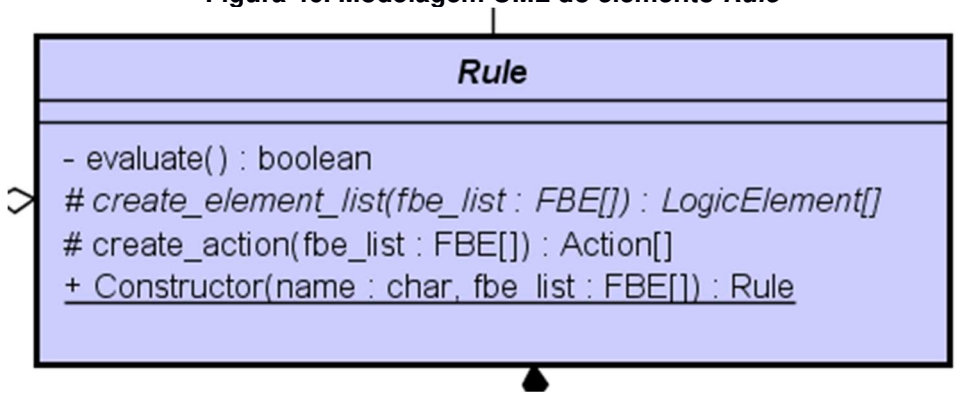


Fonte: Autoria própria

Rule

A classe *Rule* é criada para reproduzir os comportamentos do elemento de homônimo no paradigma. De forma análoga ao que ocorreu com a classe *FBE*, esta também é uma classe abstrata visando a criação de classes especializadas a partir desta. Estas especializações são encarregadas de redefinir os métodos *create_element_list()* e *create_action()* que criam todos os elementos lógicos (*Premise*, *Condition*, etc.) e o elemento *Action* respectivamente. Como uma classe especializada de *LogicElement*, o método *evaluate()* é especializado para observar o valor lógico do objeto *Condition.* O construtor recebe uma lista de FBE a fim de construir as ligações corretas para os elementos lógicos, como a *Premise*, bem como as ligações dos métodos, necessários na construção da *Instigation.* Uma vez que seu valor lógico se torne verdadeiro, o objeto dispara a execução da *Action* por meio do método *Action.run().* A Figura 46 apresenta a modelagem UML para o elemento *Rule*.

Figura 46. Modelagem UML do elemento *Rule*

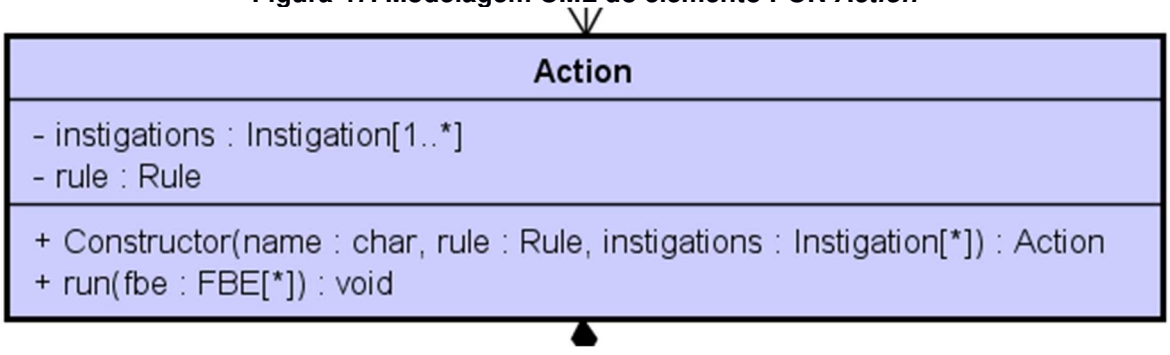


Fonte: Autoria própria

Action

A classe *Action* se encarrega de reproduzir os comportamentos do elemento PON de mesmo nome. O método construtor recebe os parâmetros necessários para sua inicialização, a saber a *Rule* e uma lista de objetos *Insigation.* O método *run()* é usado por objetos do tipo *Rule* para dispararem a execução dos objetos *Instigation* por meio do método *Instigation.run().* A Figura 47 apresenta o detalhamento da classe *Action.*

Figura 47. Modelagem UML do elemento PON *Action*



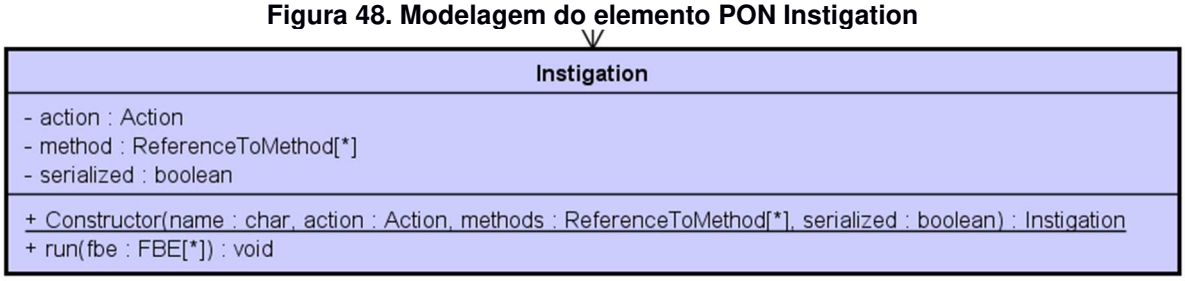
Fonte: Autoria própria

Instigation

A classe *Instigation* se encarrega de reproduzir os comportamentos do elemento PON *Instigation*. O método construtor recebe os parâmetros necessários para sua inicialização, a saber o nome, a *Action,* e uma lista de referências a métodos de

objetos *FBE* e a informação se os métodos devem ser executados sequencialmente. O método *run()* é usado por objetos do tipo *Action* para dispararem a execução dos métodos de *FBE.* A Figura 47 apresenta o detalhamento da classe *Action.*

Figura 48. Modelagem do elemento PON Instigation

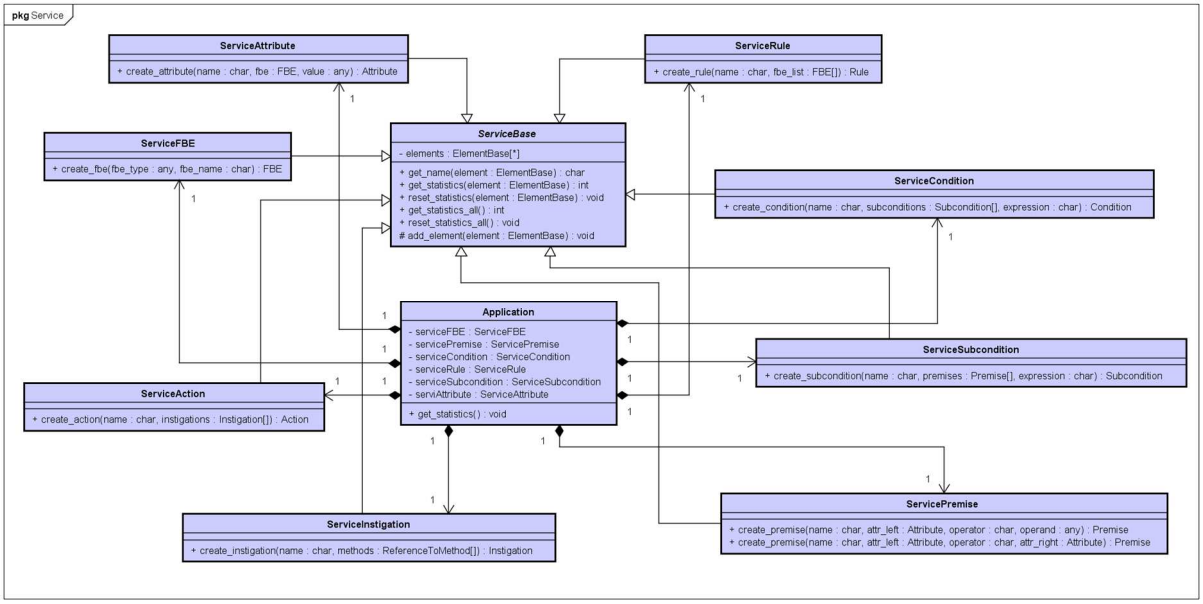


Fonte: Autoria própria

4.2.3 Elementos complementares

A modelagem dos elementos complementares tem por objetivo uma interface mais amigável de utilização, bem como um controle de estatísticas de execução visando facilitar tanto o uso do *framework* quanto da coleta de informações. O resultado desta modelagem pode ser visto na Figura 49.

Figura 49. Modelagem UML dos elementos auxiliares



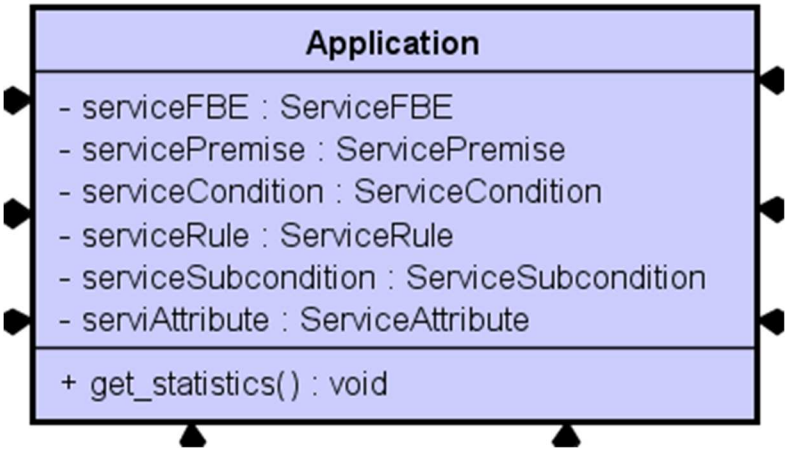
Fonte: Autoria própria

A modelagem dos elementos complementares é feita visando a aplicação de processos supervisores do Erlang/OTP (CESARINI e THOMSON, 2009, p. 174). Por este motivo estes elementos foram modelados por um conjunto de objetos *singleton*[23]. Desta forma eles ficam disponíveis durante todo o ciclo de vida da aplicação. A seguir cada objeto é apresentado em detalhes.

Application

A classe *Application* é a classe raiz da aplicação PON. Sua função principal é criar os demais processos e supervisioná-los[24] a fim de garantir a estabilidade da aplicação no caso de algum processo venha a falhar. A Figura 50 apresenta a modelagem UML para o elemento *Application*

Figura 50. Modelagem UML da classe Application



Fonte: Autoria própria

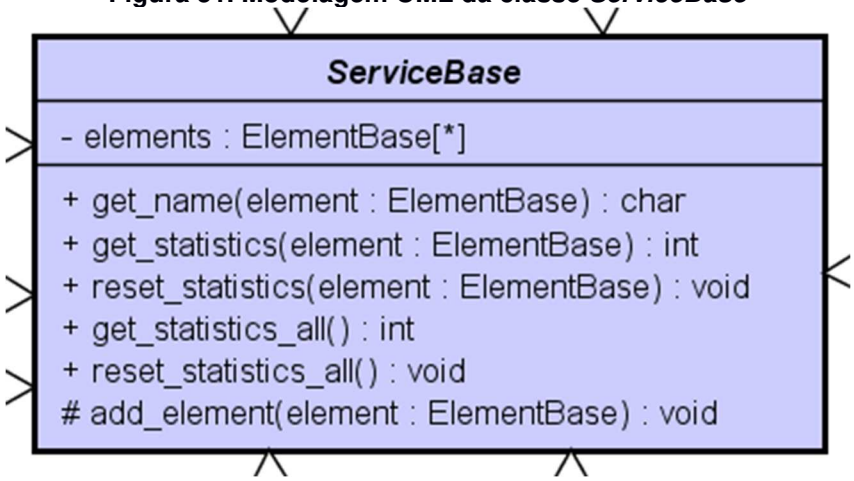
A classe *Application*, em sua criação cria os demais objetos *singleton* que controlam os elementos do *framework*. Os objetos *singleton ServiceFBE, ServicePremise, ServiceCondition e ServiceRule* são criados como supervisores para os elementos do *framework* respectivamente *FBE*, *Premise, Condition e Rule.*

[23] *Singleton* é um padrão de projeto de software (do inglês, *Design Pattern*). Este padrão garante a existência de apenas uma instância de uma classe, mantendo um ponto global de acesso ao seu objeto.
[24] Um processo Supervisor é um processo que supervisiona outros processos chamados processos filhos. Um supervisor tem, entre outras atribuições, o rastreamento e relatório de erros. Os supervisores são usados para construir uma estrutura de processo hierárquica chamada de árvore de supervisão, de forma a estruturar um aplicativo tolerante a falhas.

ServiceBase

A classe *ServiceBase* apresenta-se como uma classe abstrata das demais classes de serviço concentrando as funcionalidades comuns a elas. A Figura 51 apresenta a modelagem UML para o elemento *ServiceBase*.

Figura 51. Modelagem UML da classe *ServiceBase*



Fonte: Autoria própria

A esta classe estão atribuídos métodos criados como simplificadores de sintaxe para os métodos dos elementos, a saber *get_name, get_statistics, reset_statistics* para os métodos homônimos definidos na subseção 4.2.2.

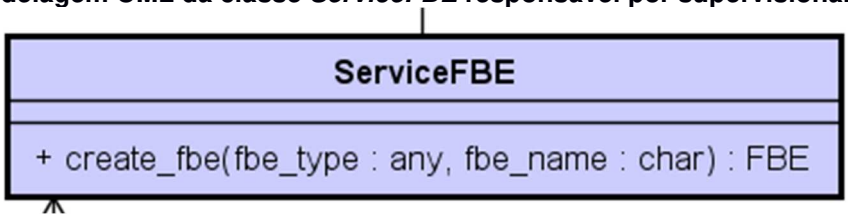
Também existem métodos que disparam funções para todos os elementos conhecidos em *elements* como *get_statistics_all* que recupera a soma de todos os resultados do método *get_statistics;* e *reset_statistics_all* que dispara o método *reset_statistics* para todos os elementos conhecidos pelo serviço. O método *add_element* é de visibilidade protegida e é usado em especializações da classe para interagir com o atributo *elements.*

Supervisores

Uma vez construído o ambiente de aplicação, são propostas classes supervisoras que se encarregarão da criação e supervisão do ciclo de vida de cada objeto apresentado na subseção 4.2.2. Estas classes também auxiliam na recuperação de estatísticas, pois preservam uma lista de todos os objetos ativos.

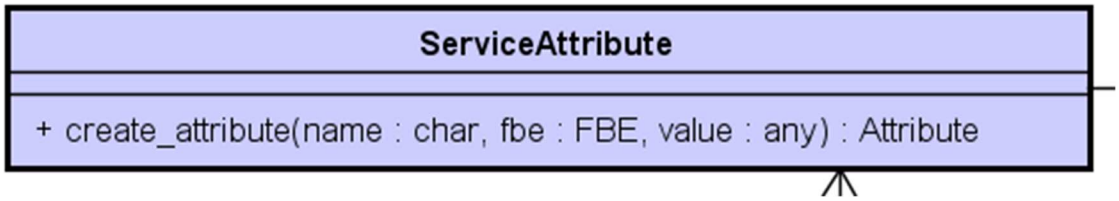
Desta forma as classes são construídas como *singleton* no início da aplicação. De forma geral, estas classes são especializações da classe *ServiceBase* no qual são criados métodos específicos para a construção de objetos os quais irá supervisionar. A seguir são apresentadas cada uma das classes de supervisão.

Figura 52. Modelagem UML da classe *ServiceFBE* responsável por supervisionar objetos *FBE*



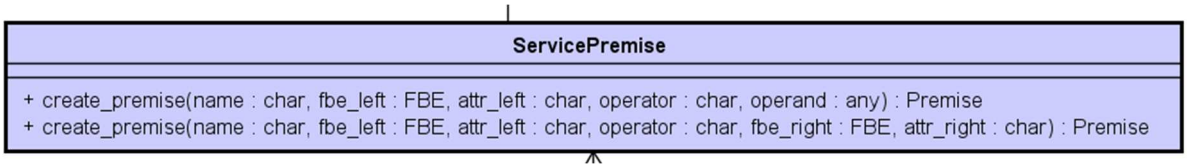
Fonte: Autoria própria

Figura 53. Modelagem UML da classe *ServiceAttribute* responsável por supervisionar objetos *Attribute*



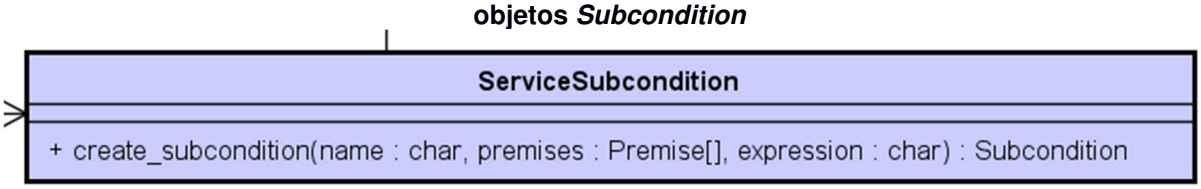
Fonte: Autoria própria

Figura 54. Modelagem UML da classe *ServicePremise* responsável por supervisionar objetos *Premise*



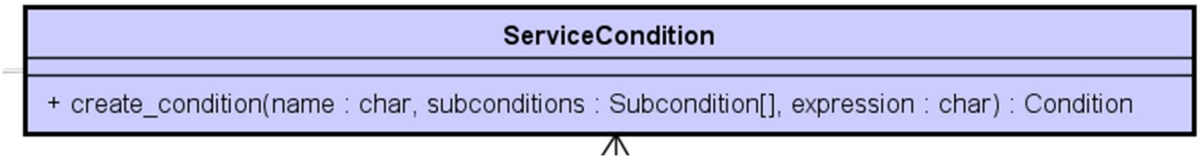
Fonte: Autoria própria

Figura 55. Modelagem UML da classe *ServiceSubcondition* responsável por supervisionar objetos *Subcondition*



Fonte: Autoria própria

Figura 56. Modelagem UML da classe *ServiceCondition* responsável por supervisionar objetos *Condition*



Fonte: Autoria própria

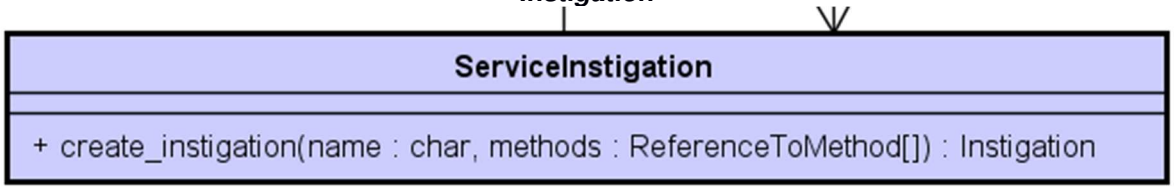
Figura 57. Modelagem UML da classe *ServiceRule* responsável por supervisionar objetos *Rule*

Fonte: Autoria própria

Figura 58. Modelagem UML da classe *ServiceAction* responsável por supervisionar objetos *Action*

Fonte: Autoria própria

Figura 59. Modelagem UML da classe *ServiceInstigation* responsável por supervisionar objetos *Instigation*



Fonte: Autoria própria

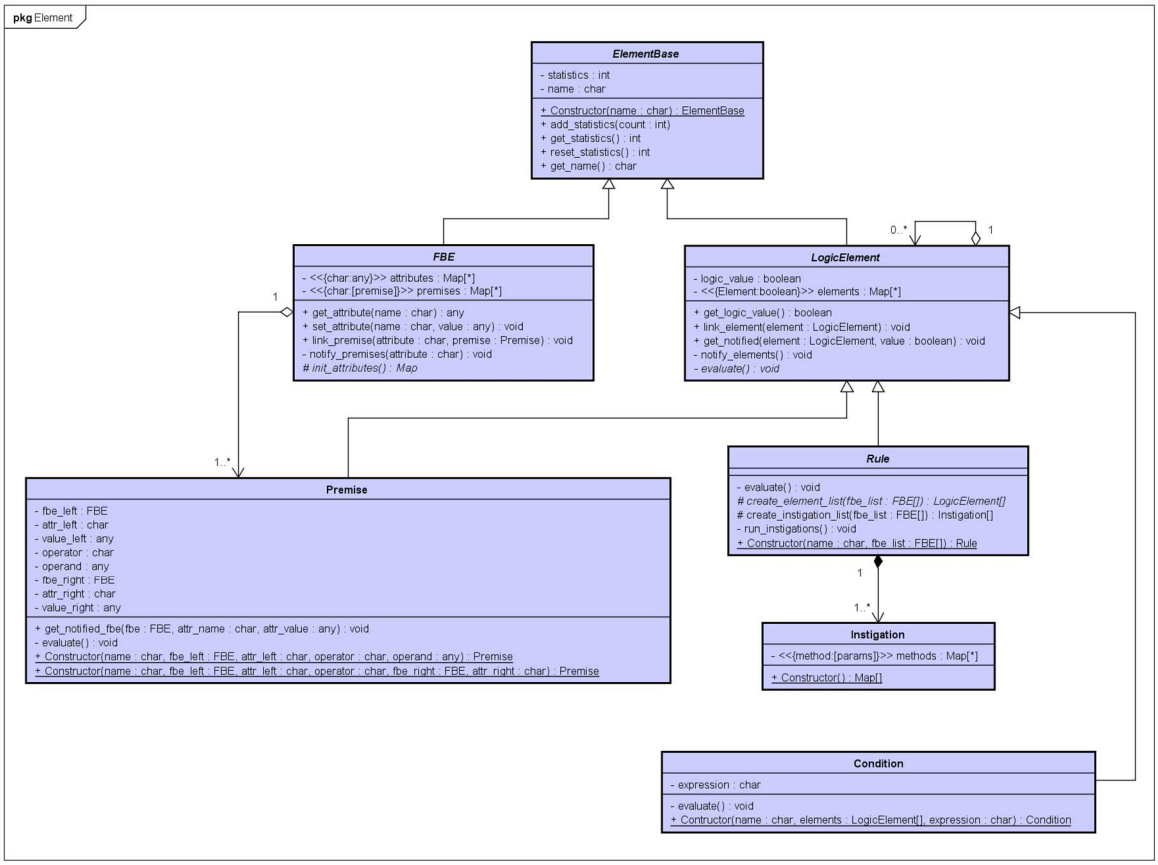
4.3 *FRAMEWORK* NOP ELIXIR

O desenvolvimento do *Framework* NOP Elixir tem por objetivo construir, baseado na modelagem descrita na seção 4.2, um *framework* em linguagem Erlang/Elixir conforme proposto no capítulo introdutório deste documento. A seguir são descritos em detalhes os componentes deste *Framework*.

4.3.1 Adaptação da modelagem UML dos elementos PON

Tendo então como base a modelagem UML proposta na seção 4.2, observou-se que o desenvolvimento do framework poderia sofrer algumas adaptações com o objetivo de promover otimizações e/ou simplificações ao desenvolvimento. Em virtude disto, é proposta uma modelagem UML adaptada dos elementos PON que é apresentada na Figura 60. Esta modelagem proposta apresenta as diferenças que são detalhadas a seguir.

Figura 60. Modelagem UML simplificada para desenvolvimento do *framework* NOP Elixir



Fonte: Autoria própria

Fusão das classes *FBE* e *Attribute*

Os comportamentos classes *FBE* e *Attribute* foram concentrados na classe *FBE*. Desta forma, envios de mensagens entre estes dois elementos foram eliminadas do

framework, colaborando assim com uma redução considerável do fluxo de mensagens de notificação.

Fusão das classes *Subcondition* e *Condition*

Em uma análise mais detalhada da modelagem descrita na seção 4.2, foi possível observar que os comportamentos das classes *Subcondition* e *Condition* eram bastante sinérgicos. Desta forma, optou-se por consolidar os comportamentos das duas classes em uma única classe nominada *Condition*. Com esta remodelagem, houve um maior reaproveitamento de códigos e uma simplificação considerável no desenvolvimento do *framework.*

Fusão das classes *Rule* e *Action*

Também objetivando uma redução no número de mensagens sem comprometimento da execução de software PON, as classes *Rule* e *Action* foram fundidas em uma única classe nominada de *Rule*. Desta forma, as mensagens entre estes atores foram suprimidas sem comprometimento do comportamento esperado do *framework*.

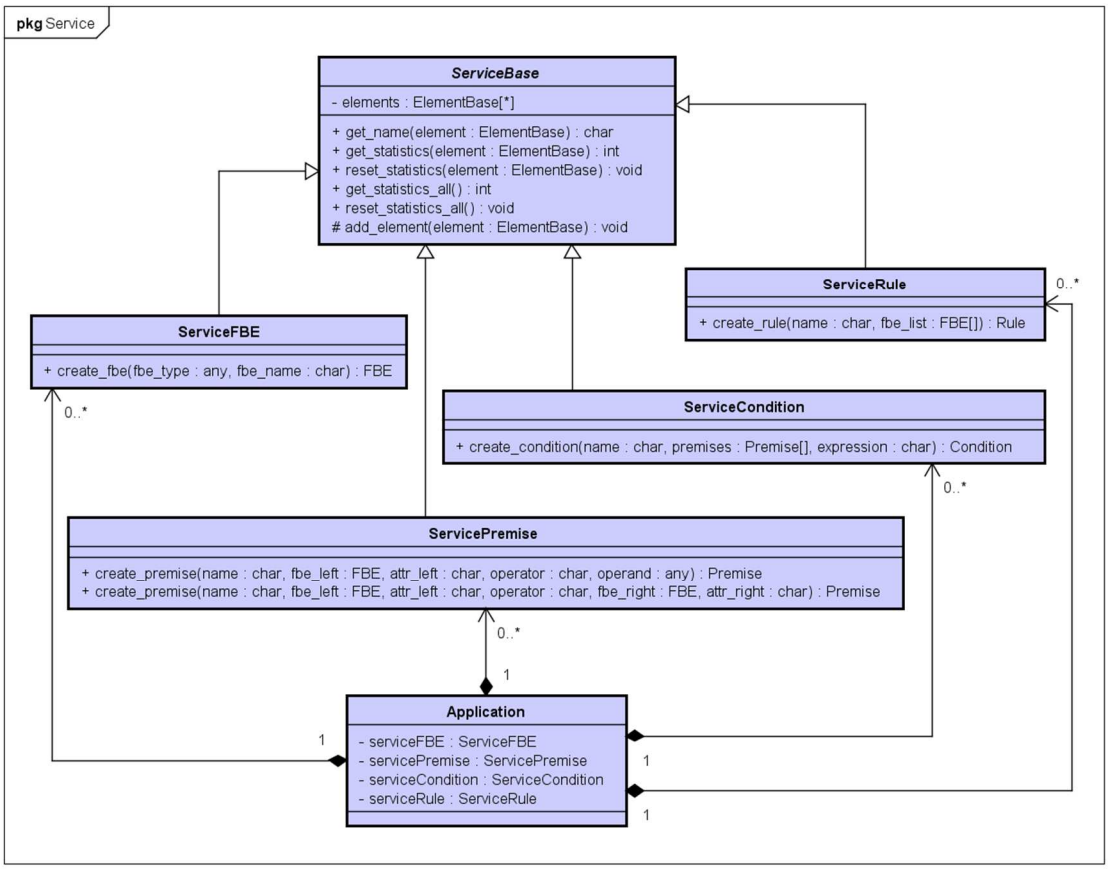
Criação dinâmica de objetos *Instigation*

Esta mudança não afetou a modelagem de forma geral, mas sim em sua execução. Como a construção de processos em Erlang não é custosa, processos do tipo *Instigation* são criados no momento em que algum método precisa ser executado. Uma vez finalizadas todas as execuções aguardadas, o processo é encerrado. Este dinamismo é possível, pois como foi apresentado na subseção 4.1.1, atores *Instigation* não modificam seu estado ao longo de sua vida, não sendo necessário manter o processo ativo para preservar mudanças em seu estado. Desta forma houve uma otimização na ocupação de memória do ambiente Erlang.

4.3.2 Adaptação da modelagem UML dos elementos complementares

Uma vez que algumas classes foram fundidas, houve a necessidade de adequação do diagrama UML dos elementos complementares. A adequação refere-se à supressão das classes supervisoras dos objetos que foram fundidos em uma única classe, visto que não são mais necessários. O diagrama UML adaptado para o desenvolvimento do *framework* é apresentado na Figura 60.

Figura 61. Diagrama UML dos elementos complementares adaptado para o *framework* NOP Elixir



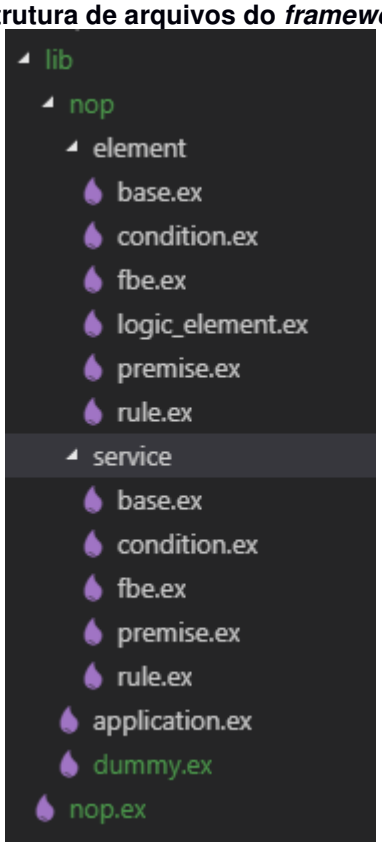
Fonte: Autoria própria

4.3.3 Estrutura de arquivos

A estrutura de arquivos do *framework* é apresentada da seguinte forma: na raiz, o arquivo *nop.ex* que possui as definições de módulo. Em seguida, dentro da pasta *lib* estão os arquivos *application.ex* e *dummy.ex* que apresentam, respectivamente, as

definições da aplicação e uma demonstração de utilização do *framework* para uso em TDD (*Test Driven Development).* A Figura 62 apresenta a estrutura de arquivos do *framework* NOP Elixir.

Figura 62. Estrutura de arquivos do *framework* NOP Elixir



Fonte: Autoria própria

A seguir, as subpastas *element* e *service* separam os desenvolvimentos dos elementos do *framework* dos elementos auxiliares conforme definido na seção 4.2.1. Desta forma, os arquivos carregam as implementações dos módulos conforme Tabela 12.

Tabela 12. Organização dos módulos desenvolvidos por pasta e arquivo

Pasta	**Arquivo**	**Módulo**
service	base.ex	*NOP.Service.ServiceBase*
service	condition.ex	*NOP.Service.Condition*
service	fbe.ex	*NOP.Service.FBE*
service	premixe.ex	*NOP.Service.Premise*
service	rule.ex	*NOP.Service.Rule*

element	base.ex	*NOP.Element.ElementBase*
element	condition.ex	*NOP.Element.Condition*
element	fbe.ex	*NOP.Element.FBE*
element	logic_element.ex	*NOP.Element.LogicElement*
element	premise.ex	*NOP.Element.Premise*
element	rule.ex	*NOP.Element.Rule*

4.3.4 Elementos do paradigma PON

Os elementos do PON são encontrados na pasta *element* dentro da estrutura de arquivos do *framework*. Encontrado no arquivo *base.ex,* o primeiro módulo do qual derivam todos os demais módulos é o *NOP.Element.ElementBase* que implementa o elemento UML *ElementBase*.

Em seguida, o apresenta-se o módulo *NOP.Element.FBE*, que é encontrado no arquivo *fbe.ex* e implementa o elemento UML *FBE.* Este módulo e é uma especialização[25] de *NOP.Element.ElementBase* juntamente com *GenServer*[26] (THOMAS, 2018, p. 229-242). O Código 2 apresenta um trecho do código deste módulo. Este exemplo é o processamento da mensagem *set_attr*, que corresponde à implementação do método *set_attribute* definido na modelagem UML para a classe *FBE.*

[25] Em Elixir, uma forma de se representar a especialização é através do comando *use* (THOMAS, 2018)
[26] O módulo GenServer implementa um servidor genérico de forma a incorporar comportamentos padrões como, facilidade para tratamento de mensagens e rastreamento e relatório de erros.

Código 2. Implementação de trata mento da mensagem *set_attribute* em microator *FBE* na linguagem Elixir

Elixir

```elixir
def handle_cast({:set_attr, attr_name, attr_value}, state) do

  fbe_attrs = get_attributes_from_state(state)

  fbe_attrs = set_attribute(fbe_attrs, attr_name, attr_value, state)

  state = set_attributes_to_state(state, fbe_attrs)

  #Notification count
  state = add_statistics_to_state(state, 1)

  {:noreply, state}
end
```

Fonte: Autoria própria

Quando da declaração de um *FBE*, portanto, é necessário implementar a função *init_attributes()* com a inicialização dos *Attributes*. Também é neste momento que são implementados os demais métodos inerentes ao *FBE*. O Código 3 apresenta um exemplo de uma implementação de um *FBE* completo com a definição dos *Attributes* definida na função *init_attributes(),* bem como uma função *change_value_to3()* representando um *Method* específico do *FBE.*

Código 3. Exemplo de implementação de *FBE* como especialização do módulo *NOP.Element.FBE*

Elixir

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

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

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

end
```

Fonte: Autoria própria

Na sequência, o módulo *NOP.Element.LogicElement* é o módulo base que serve de ancestral comum de todos os elementos lógicos, a saber (*Premise, Condition e Rule)*. Também especializado de *NOP.ElementBase* está declarado no arquivo *logic_element.ex* e é responsável por implementar o elemento UML *LogicElement.*

O elemento UML *Premise* é implementado no *framework* pelo módulo *NOP.Element.Premise* e encontra-se no arquivo *premise.ex e é e*specializado de *Genserver* e *PON.Element.LogicElement*.

Conforme descrito na seção 4.2.1, os elementos PON *Condition* e *Subcondition* foram condensados em um único elemento UML *Condition.* Este, por sua vez, é implementado no módulo *NOP.Element.Condition* e se encontra no arquivo *condition.ex* trazendo todos os comportamentos de ambos os elementos PON.

Finalizando a lista de especializações de *NOP.Element.LogicElement* está o módulo *NOP.Element.Rule* que implementa o elemento UML *Rule* e se encontra no arquivo *rule.ex*. O Código 4 apresenta um exemplo de implementação de uma *Rule* completa.

Código 4. Exemplo de implementação de Rule como especialização de *NOP.Element.Rule*

Elixir

```
defmodule NOP.Element.Rule_dummy3 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: Autoria própria

O elemento UML *Instigation* não necessitou da construção de um módulo próprio, pois todo o comportamento esperado pode ser aproveitado por meio de *Tasks* (THOMAS, 2018, p. 293).

4.3.5 Elementos complementares

Dos elementos complementares, o primeiro módulo que é apresentado é o módulo *NOP.Application*. Especializado de *Application*[27] (THOMAS, 2018, p. 277-278) este módulo é responsável por supervisionar os demais módulos de serviço que serão apresentados a seguir. Este módulo também encapsula algumas funções de tratamento de conjunto de atores a fim de trazer mais simplicidade na utilização do *framework*. Este módulo encontra-se no arquivo application.ex e é o único dos listados nesta seção que não se encontra dentro da pasta *service.* O Código 5 apresenta o

[27] O módulo *Application* permite o empacotamento de softwares desenvolvidos na plataforma Erlang. Estes são semelhantes ao conceito de biblioteca, comuns em outras linguagens de programação, mas com características próprias desta plataforma. Neste módulo implementam-se funcionalidades como ciclo de vida da aplicação, que pode então ser carregada, iniciada e interrompida.

trecho de inicialização deste módulo. Nesta etapa os serviços complementares são, então, iniciados e ficam sob sua supervisão.

Código 5. Trecho de código de inicialização do módulo Application do *framework*

Elixir

```elixir
def start(_type, nil) do
  import Supervisor.Spec, warn: false

  children = [
    worker(NOP.Service.FBE,  [NOP.Service.FBE]),
    worker(NOP.Service.Premise, [NOP.Service.Premise]),
    worker(NOP.Service.Condition, [NOP.Service.Condition]),
    worker(NOP.Service.Rule, [NOP.Service.Rule]),
  ]

  opts = [strategy: :one_for_one, name: NOP.Supervisor]

  Supervisor.start_link(children, opts)

end
```

Fonte: Autoria própria

O próximo módulo é o módulo *NOP.Service.ServiceBase*. Este módulo encontra-se no arquivo *base.ex* dentro da subpasta *service*. Este módulo é responsável por implementar o elemento UML *ServiceBase* e traz as funcionalidades comuns a todos os serviços.

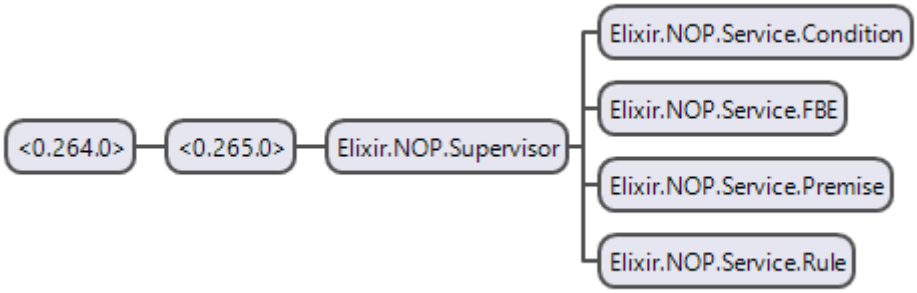
O módulo *NOP.Service.FBE* é encontrado no arquivo *fbe.ex*. Especializado de *NOP.Service.ServiceBase* e *GenServer* é o primeiro *singleton* inicializado por *NOP.Application*. Além de trazer facilidades de programação simplificando chamadas para os elementos do paradigma, este se comporta como um supervisor de todos os elementos *NOP.Element.FBE* criados durante uma execução do *framework*.

A seguir, o módulo NOP*.Service.Premise,* também *singleton,* responsável por supervisionar e trazer simplificações de chamadas aos elementos *NOP.Element.Premise.* Localizado em *premise.ex* e especializado de *NOP.Service.ServiceBase* e *GenServer*, também é inicializado e supervisionado por *NOP.Application* na inicialização da aplicação.

Da mesma forma, o módulo *NOP.Service.Condition* é responsável por supervisionar e trazer simplificações de chamadas aos elementos *NOP.Element.Condition*. Localizado em *condition.ex* e especializado de *NOP.Service.ServiceBase* e *GenServer* é o terceiro supervisor *singleton* inicializado por *NOP.Application*.

Por fim, o último *singleton* inicializado por *NOP.Application* encontra-se no módulo *NOP.Service.Rule*. Este é codificado no arquivo *rule.ex* e também, como os demais supervisores, é especializado por *NOP.Service.ServiceBase* e *GenServer*. Desta forma, uma aplicação que utiliza o *Framework* NOP Elixir terá sempre, e no mínimo, seis processos supervisores iniciados para gerenciar os processos relacionados ao paradigma, são eles: *NOP.Application*, *NOP.Supervisor*, *NOP.Service.FBE*, *NOP.Service.Premise*, *NOP.Service.Condition* e *NOP.Service.Rule*. A Figura 63 apresenta a aplicação sendo executada com o *Framework* NOP Elixir conforme hierarquia de supervisão. O primeiro nó (ID <0.264.0>) é o processo raiz, criado pela VM para todo aplicativo Erlang; o segundo nó (ID <0.265.0>) corresponde ao módulo inicial da aplicação disponibilizada pelo *Framework* NOP Elixir, ou seja, *NOP.Application*; em seguida, o processo Supervisor e os processos de gerenciamento dos demais elementos PON.

Figura 63. Estrutura de aplicações supervisoras em uma aplicação NOP Elixir



Fonte: Autoria própria

4.3.6 Funcionalidades complementares

Com o objetivo de construir um conjunto de soluções que além de executar software PON, algumas funcionalidades foram inseridas no *framework* NOP Elixir. Uma primeira funcionalidade foi a contagem de notificações gerada para cada elemento e a possibilidade de recupera estas informações de maneira individual, por tipo de elemento ou da aplicação como um todo. O Código 6 apresenta um código exemplo no qual utiliza o controle de estatísticas de notificação e os apresenta na tela.

Código 6. Exemplo de utilização de estatísticas do *framework* NOP Elixir

Elixir

stat_2000.exs

statistics > stat_2000.exs

```
1   # Run in iex:
2   # c "./statistics/stat_2000.exs"
3
4   NOP.Application.reset_element_list_total()
5   {time_in_microseconds, _ret_val} =  :timer.tc(
6     fn ->
7       CTA.Init.run_10x10(2000)
8       NOP.Application.wait_up_to_end_all_process()
9     end)
10
11  IO.puts("The liquid time is #{div(time_in_microseconds,1000)}")
12  IO.puts("The notification count is #{NOP.Application.get_statistics_from_total()}")
13
```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL 1: bash

```
fnegrini@FHMOB2250 MINGW64 /e/GIT/nop_elixir/samples/cta (master)
$ iex -S mix
Interactive Elixir (1.7.4) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> c "./statistics/stat_2000.exs"
The liquid time is 3140
The notification count is 1430683
[]
iex(2)>
```

Fonte: Autoria própria

Outra funcionalidade disponibilizada é a geração de *log* para os eventos PON. Esta funcionalidade permite facilitar o acompanhamento do fluxo de notificações por parte do desenvolvedor. Com isto facilita-se a análise e depuração de um código PON. A Figura 64 apresenta um exemplo de saída do log de execução de um programa PON em *Framework* NOP Elixir.

Figura 64. Log de execução de um programa PON em *Framework* NOP Elixir

Fonte: Autoria própria

4.3.7 Melhorias

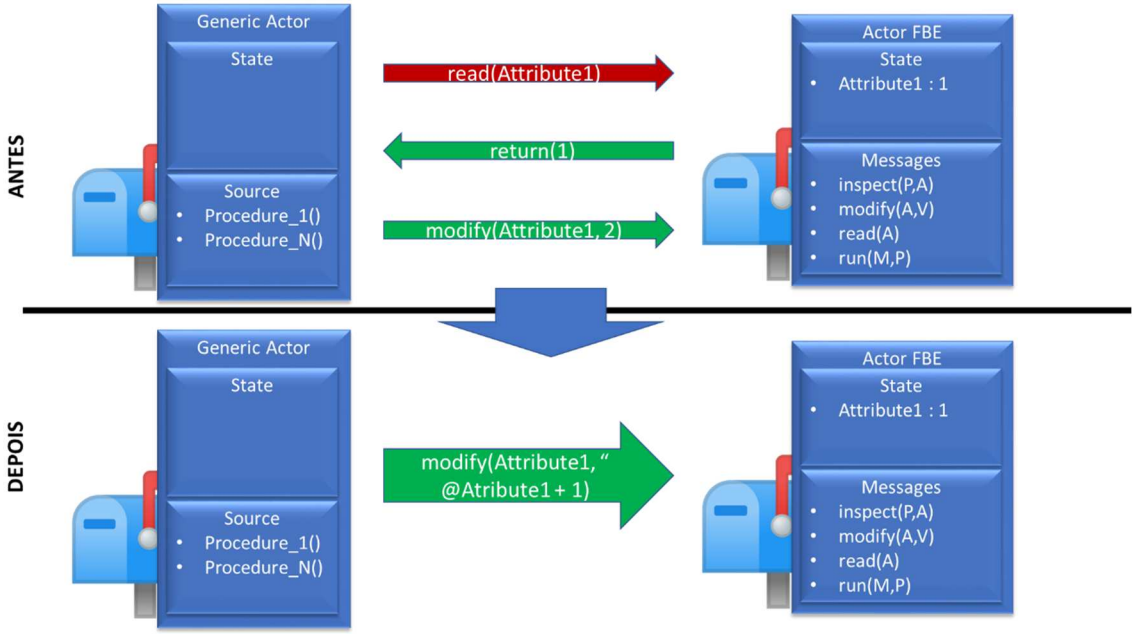
A primeira versão estável do *Framework* NOP Elixir já possuía a parte de comunicação entre os agentes lógicos (i.e., *Premise, Condition e Rule)* bem como o mecanismo de instigação (*Action* e *Instigation*). Contudo, a leitura e escrita dos *Attributes* dos *FBEs* era bastante monolítica contando apenas com mensagens que permitiam apenas uma operação por vez. A evolução dos casos de estudos iniciais que serviram de experimentação e testes, demandou melhorias, principalmente no sentido de evitar bloqueios de mensagens permitindo uma comunicação sem gargalos. As principais melhorias são descritas a seguir.

Atribuição de valor por expressão

Como visto na subseção 4.1.1, a leitura de um *Attribute* de um *FBE* é feita por meio de mensagem síncrona *read(A).* Contudo, mensagens síncronas são onerosas ao fluxo de notificações, pois causam uma interrupção além de gerarem um grande

risco de impasses ou *deadlocks*[28] (CESARINI e THOMSON, 2009, p. 137). Portanto a necessidade de se incrementar o valor de um determinado *Attribute* passou a ser um impasse e demandou esta melhoria. Como o *FBE* já possui o valor em seu estado, a melhoria consiste em sobrecarregar a mensagem *modify(A, V)* para que possa receber uma expressão Elixir válida. Esta expressão substitui os nomes dos *Attributes* pelos valores de seus respectivos *Attributes* homônimos e, então, atribui o valor calculado ao *Attribute* destino. Esta melhoria permite utilizar não só o valor do *Attribute* a ser modificado, mas qualquer outro *Attribute* do *FBE* disponível bastando referenciá-lo pelo nome na expressão.

Figura 65. Demonstração alto nível da melhoria: atribuição por expressão



Fonte: Autoria própria

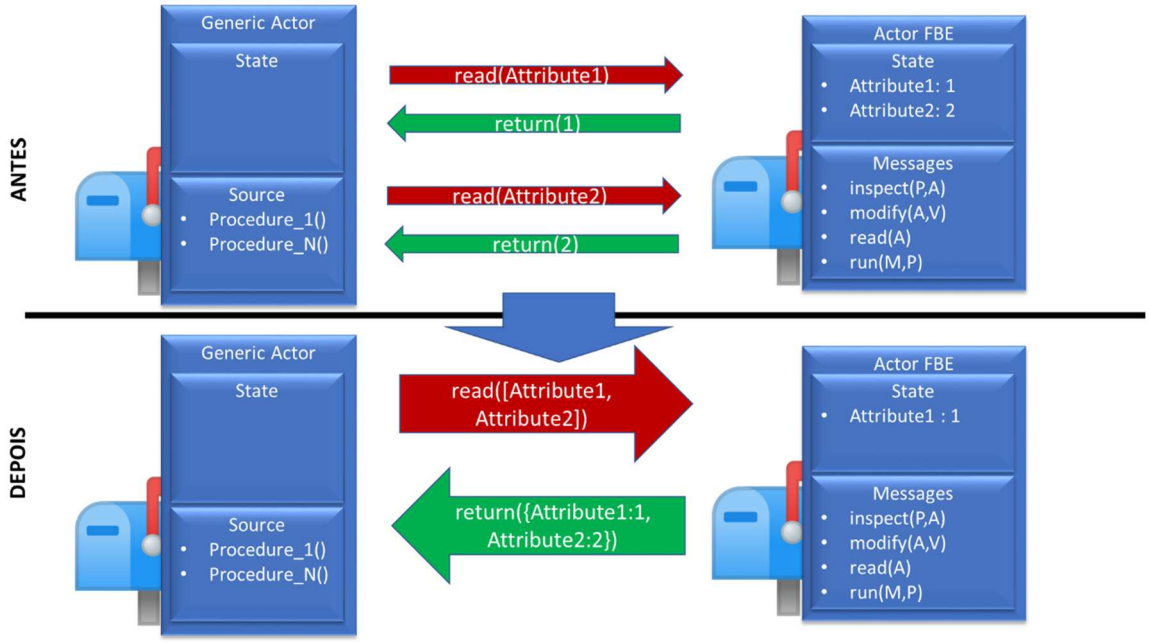
A Figura 65 faz uma apresentação alto nível da melhoria. Neste exemplo, a intenção é incrementar *Attribute1* em uma unidade. Antes da modificação, portanto, era necessária uma leitura do *Attribute* para posterior cálculo do valor. Após a melhoria, a mensagem *modify* aceita uma expressão que é calculada diretamente no microator *FBE*. As flechas vermelhas representam mensagens síncronas e as flechas verdes representam mensagens assíncronas.

[28] Um impasse ou *deadlock* é uma situação em que dois programas ou processos diferentes dependem um do outro para conclusão, porque ambos estão usando os mesmos recursos ou devido a erros de programação.

Leitura múltipla de *Attributes*

Outra melhoria feita no *Framework,* no sentido de minimizar mensagens síncronas, foi a sobrecarga da mensagem *read* para a leitura de múltiplos *Attributes* em uma única mensagem. Desta forma, se o ator solicitante precisar de vários *Attributes* do *FBE*, poderá fazê-los todos em uma só comunicação. A Figura 66 faz uma apresentação alto nível da melhoria na qual o ator solicitante necessita ler os valores *Attribute1 e* Attribute2 do *FBE*. Antes da melhoria, eram necessárias duas comunicações (uma para cada *Attribute*). Após a melhoria, portanto, é possível solicitar os dados de um conjunto de *Attributes* em uma única mensagem. As flechas vermelhas representam mensagens síncronas e as flechas verdes representam mensagens assíncronas.

Figura 66. Demonstração alto nível da melhoria: leitura múltipla de atributos



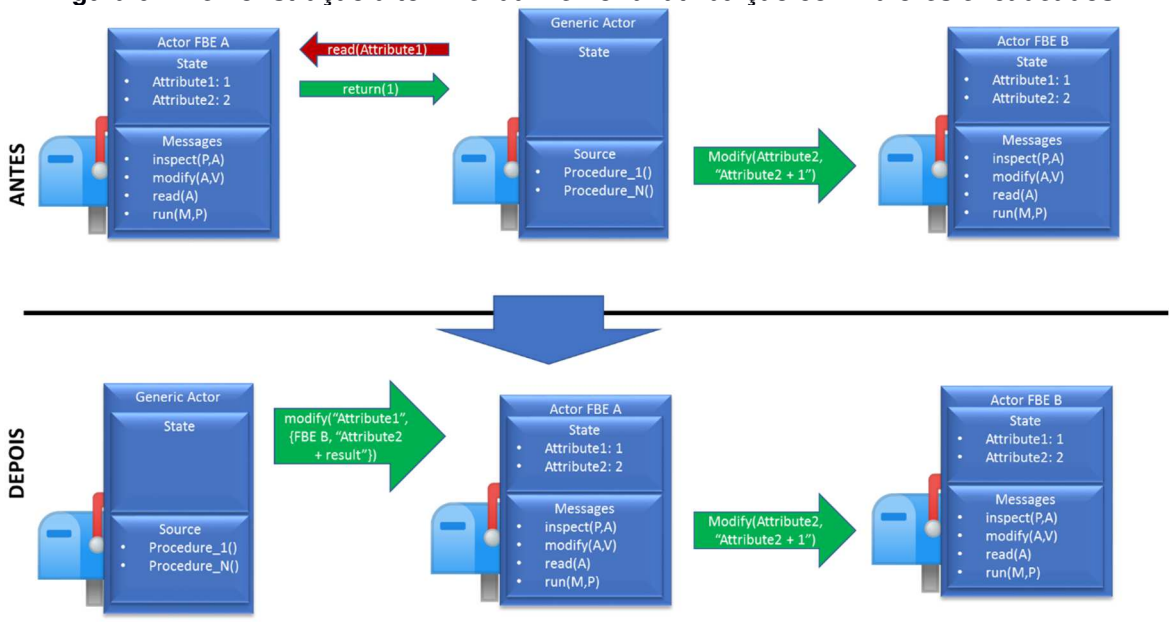
Fonte: Autoria própria

Atribuição com valores encadeados

Outro problema de impasse no *framework* foi quando a atualização do valor de um *Attribute* de um determinado *FBE* dependia de *Attributes* contidos em outro *FBE*. Como o valor agora está em dois microatores distintos, a modificação necessária para evitar mensagens assíncronas demandou um encadeamento de mensagens. Desta

forma, a primeira mensagem é enviada para o *FBE* que possui o valor a ser usado e esta, por sua vez, envia o valor para o *FBE* final passando o valor necessário para a consolidação. A melhoria consiste em sobrecarregar a mensagem *modify* de forma a enviar uma corrente de modificação informando que a modificação deverá ser propagada para o próximo *FBE* contido na mensagem e o valor calculado é apresentado em forma de uma expressão.

Figura 67. Demonstração alto nível da melhoria: atribuição com valores encadeados



Fonte: Autoria própria

A Figura 67 apresenta um exemplo no qual o valor de *FBE A.Attribute1* deverá ser somado em *FBE B.Attribute2* com os resultados antes e após a melhoria. Antes da melhoria, portanto, era necessária uma leitura do *Attribute1* na *FBE A* para, então, atualizar o valor no *FBE B.* Após a melhoria, a mensagem *modify* informa que o valor a ser alterado é, na verdade, de um terceiro *FBE* que, neste exemplo, é o *FBE B.* O *FBE A,* portanto, processa a mensagem atualizando com os valores dos *Attributes* nela contidos e envia o comando de modificação para o *FBE B.* As flechas vermelhas representam mensagens síncronas e as flechas verdes representam mensagens assíncronas.

4.3.8 Empacotamento e distribuição

Aplicativos são a maneira idiomática de empacotar software em Erlang. Como Elixir está imerso neste ecossistema, este também se aproveita destes recursos de forma natural. Portanto, os aplicativos são semelhantes ao conceito de biblioteca comum em outras linguagens de programação, mas com algumas características adicionais (HEXDOCS (APPLICATION), 2019).

Um aplicativo é um componente que implementa algumas funcionalidades específicas, com uma estrutura de diretórios padronizada, configuração e ciclo de vida. Os aplicativos são carregados, iniciados e interrompidos (HEXDOCS (APPLICATION), 2019).

A distribuição do *Framework* NOP Elixir através de aplicação permite uma utilização rápida e simples. Além de promover a fácil atualização de versões evolutivas. No APÊNDICE G desta dissertação está disponível o guia de utilização com todas as informações necessárias para a utilização deste *Framework*.

4.4 COMPILADOR NOPL ERLANG-ELIXIR

Nesta seção é apresentada a construção de um novo compilador intitulado NOPL Erlang-Elixir a partir das diretrizes dispostas no MCPON. Este compilador gera, a partir do Grafo PON, elementos para o *Framework* NOP Elixir apresentado na seção 4.3., permitindo assim desenvolver aplicações em alto nível em LingPON para tal framework, bem como complementando a demonstração do MCPON com uma nova linguagem alvo.

4.4.1 Compilador e o Grafo PON

De acordo com Ronszcka (2019), a estrutura do PON, definida por meio das entidades notificantes que compõem seu modelo de programação, possibilitou o advento de uma inovação no processo de compilação. Esta inovação permite traduzir programas distintos escritos em linguagens próprias ao PON em uma única estrutura de dados uniforme. Esta estrutura possui um formato de grafo, no qual é possível

acomodar representações de todas as entidades extraídas de um programa PON. Estas entidades, em conjunto, formam a definição completa de um programa. Para essa estrutura foi dado o nome de Grafo PON, conforme detalhado na subseção 3.7.2.

De maneira mais clara, no âmbito desta dissertação, o trabalho consiste em traduzir o Grafo gerado na compilação de um programa PON em estruturas do *Framework* NOP Elixir. Para a construção do compilador, portanto, são feitas extensões específicas para o novo compilador a partir do que é disponibilizado pelo MCPON. Estas extensões necessárias ao novo compilador se classificam em duas partes distintas: (1) Extensão dos analisadores léxico e sintático; e (2) construção de uma classe de compilação responsável pela interpretação do Grafo PON.

4.4.2 Extensão dos analisadores léxico e sintático

O MCPON disponibilizou analisadores léxico e sintático para interpretação dos programas PON. Entretanto, para a construção do novo compilador, foi necessário estender estas ferramentas. A extensão do analisador léxico consiste na interpretação de um novo token para reconhecimento da palavra reservada *ELIXIR*. Desta forma, o compilador passa a reconhecer esta palavra reservada para o tratamento de código específico em linguagem alvo Elixir. O Código 7 apresenta a extensão feita no analisador léxico disponibilizado pelo MCPON.

Código 7. Extensão do analisador léxico para reconhecer a palavra reservada *ELIXIR*

Analisador Lexico

```
ELIXIR                          {

                                    if (code_block == 0) {
                                        return ELIXIR;
                                    } else if (code_block == 1) {
                                        code_block = 2;
                                        return ELIXIR;
                                    } else {
                                        code += strdup(yytext);
                                    }

                                }
```

Fonte: Autoria própria

Com o reconhecimento de código específico e utilização do token *ELIXIR,* o analisador sintático também necessitou de devida extensão de forma a reconhecê-lo

como uma fonte de códigos específica. O Código 8 apresenta a extensão feita no analisador sintático para este novo compilador destacada em azul.

Código 8. Extensão do analisador sintático para reconhecimento de código específico em Elixir

Analisador Sintático

```
target                  : CODE_GENERATION_EXAMPLE {
                            $$ = graph->createTarget(Target::CODE_GENERATION_EXAMPLE_TARGET);
                          }
                          | NAMESPACES {
                            $$ = graph->createTarget(Target::NAMESPACES_TARGET);
                          }
                          | FRAMEWORK_CPP_2_0 {
                            //$$ = graph->createTarget(Target::FRAMEWORK_CPP_2_0_TARGET);
                            $$ = graph->createTarget(Target::CODE_GENERATION_EXAMPLE_TARGET);
                          }
                          | FRAMEWORK_CPP_3_0 {
                            //$$ = graph->createTarget(Target::FRAMEWORK_CPP_3_0_TARGET);
                            $$ = graph->createTarget(Target::CODE_GENERATION_EXAMPLE_TARGET);
                          }
                          | ELIXIR {
                            $$ = graph->createTarget(Target::ELIXIR_TARGET);
                          }
                          ;
```

Fonte: Autoria própria

Ainda no analisador sintático também é feita a extensão para reconhecimento do target *ELIXIR_TARGET* de forma a reconhecer como uma nova opção de saída na execução do compilador NOPL via linha de comando. Desta forma, a análise do grafo é direcionada para classe de compilação *CodeGenerationElixir,* que será detalhada a seguir. O Código 9 apresenta a ampliação do analisador sintático para direcionamento da classe específica.

Código 9. Extensão no analisador sintático para integração com a classe de compilação para framework Elixir.

Analisador Sintático

```
case Target::ELIXIR_TARGET: {

    compiler = new CodeGenerationElixir();
    graph = compiler->graph;

} break;

default: {

    cout << "Target is undefined" << endl;
    return EXIT_FAILURE;

} break;

}
```

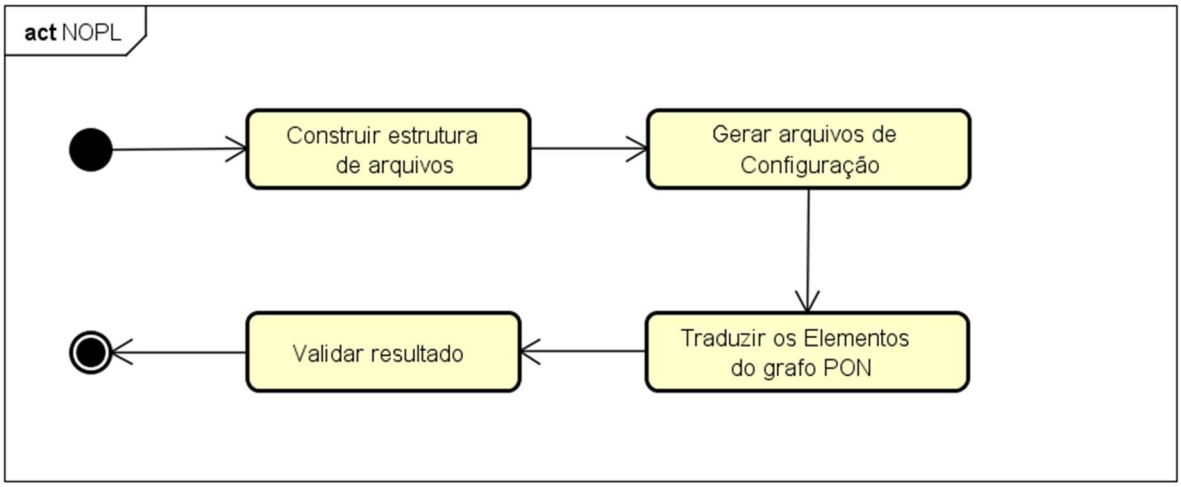
Fonte: Autoria própria

4.4.3 Classe de compilação

A construção da classe de compilação *CodeGenerationElixir*[29] consiste no maior esforço no trabalho de construção do compilador, pois nela são codificadas as etapas descritas na Figura 68. O desenvolvimento consiste na criação uma classe especializada da classe original *Compiler* (disponibilizada pelo MCPON) e redefinição o método *generateCode()*. Neste método são executadas todas as etapas necessárias para a construção de um código-alvo. Estas etapas são apresentadas no fluxograma disposto na Figura 68. A seguir, cada etapa deste fluxo é apresentada de maneira detalhada.

[29] O código fonte em sua integralidade está disponível no APÊNDICE K..

Figura 68. Fluxograma das etapas de compilação



Fonte: Autoria própria

Construção da estrutura de arquivos

Para a construção da estrutura de arquivos é utilizado o comando *mix new.* Este comando é disponibilizado após a instalação do compilador Elixir e se encarrega da construção de toda uma estrutura de arquivos de um projeto Elixir. A adoção do comando em detrimento a uma construção manual de estrutura de arquivos permite que o compilador evolua com as distribuições da plataforma Erlang/Elixir. O comando *mix new* é então chamado pelo compilador. O Código 10 apresenta esta etapa de construção da estrutura de arquivos, codificada no método *createFileStructure().*

Código 10. Código para a construção da estrutura de arquivos

Compilador

```
void CodeGenerationElixir::createFileStructure(){

    std::cout << "\nGenerating file structure..." << std::endl;

    std::stringstream commandStream;
    commandStream << "rmdir /Q /S " << MAIN_FOLDER;
    // Eliminate old structure
    system(commandStream.str().c_str());

    // Create project system
    std::stringstream commandStream2;
    commandStream2 << "mix new  " << MAIN_FOLDER;
    system(commandStream2.str().c_str());

}
```

Fonte: Autoria própria

Geração do arquivo de configuração

Uma vez construída a estrutura de arquivos, parte-se para a etapa de geração do arquivo de configuração *mix.exs*. Neste arquivo é inserida a dependência para o módulo HEX *nop_elixir*. Após a construção do arquivo de configuração, o comando *mix deps.get* é executado a fim de recuperar e instalar o referido módulo. O Código 11 apresenta um exemplo de arquivo *mix.exs* gerado pelo compilador.

Código 11. Exemplo de arquivo de configuração mix.exs

Elixir

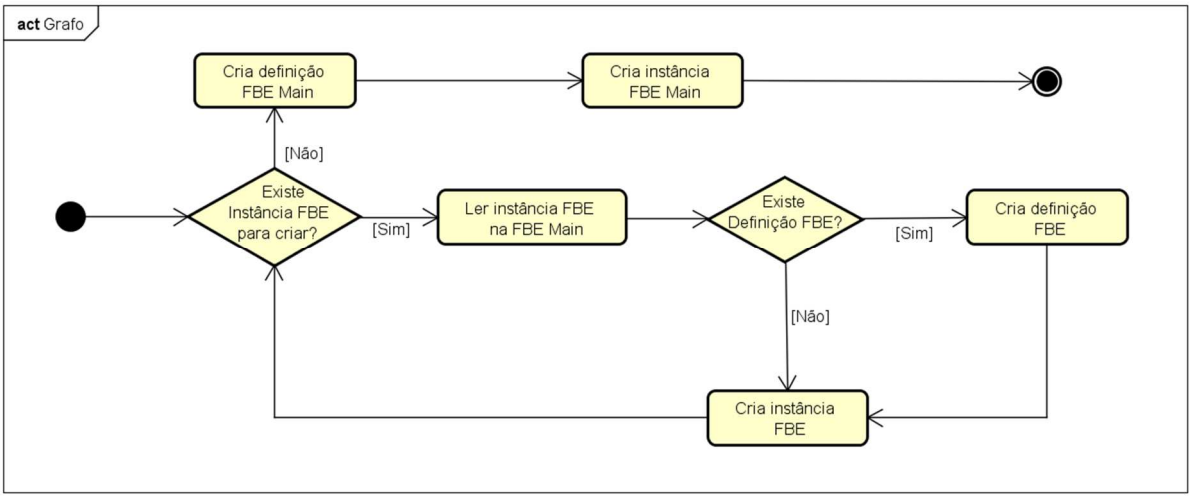
```elixir
defmodule NopEx.MixProject do
  use Mix.Project

  def project do
    [
      app: :nop_ex,
      version: "0.1.0",
      elixir: "~> 1.7",
      start_permanent: Mix.env() == :prod,
      deps: deps()
    ]
  end

  # Run "mix help compile.app" to learn about applications.
  def application do
    [
      extra_applications: [:logger, :nop_elixir]
    ]
  end

  # Run "mix help deps" to learn about dependencies.
  defp deps do
    [
      {:nop_elixir, "~> 0.0.22"}
    ]
  end
end
```

Fonte: Autoria própria

Tradução dos elementos do grafo PON

A tradução do Grafo PON consiste na interpretação das instâncias e seus *FBE*s e na geração de arquivos da linguagem Elixir, conforme apresentado na subseção 3.7.2. O fluxo de leitura do Grafo PON pode ser visualizado no fluxograma apresentado na Figura 69.

Figura 69. Fluxograma para tradução do Grafo PON



Fonte: Autoria própria

O fluxo consiste em navegar em todas as instâncias declaradas no código PON e construir estruturas correspondentes para cada *FBE*. Ao final, a definição FBE Main é criada e, dentro desta definição, uma função Elixir é construída para a declaração de todas as instâncias anteriormente identificadas. Todo este código de inicialização é colocado no arquivo *nop_ex.ex.* A Figura 70 apresenta o código que verifica se a definição FBE já está criada e cria a definição caso ainda não exista.

Figura 70. Código para criação de definição de FBE

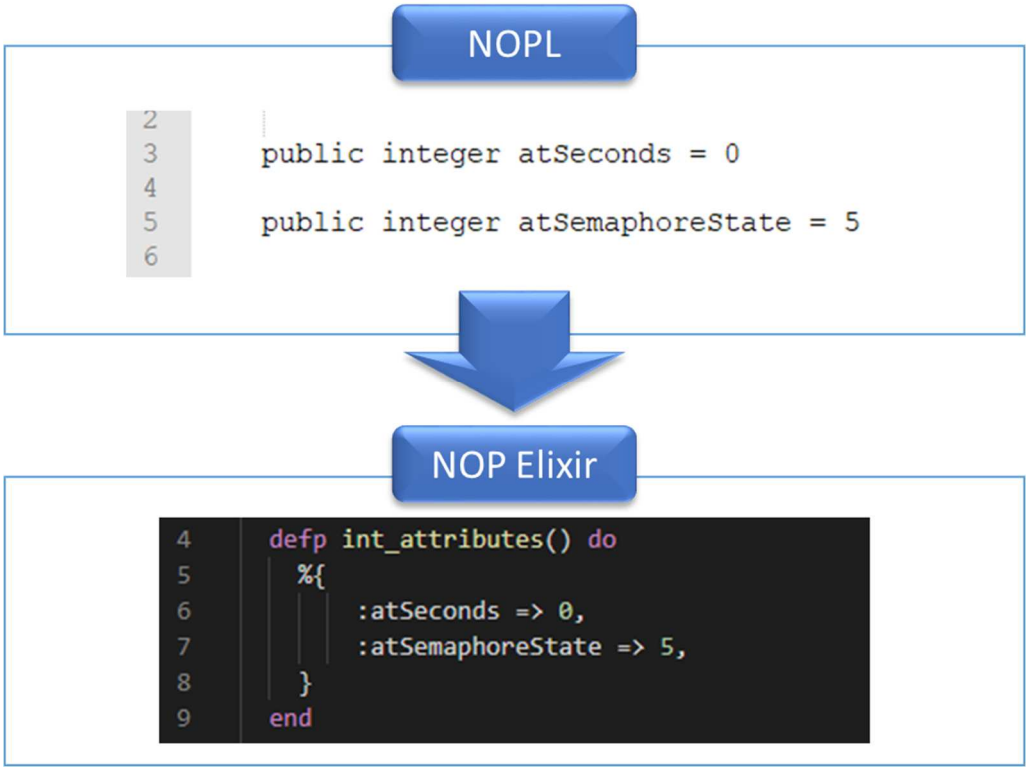
Compilador

```
void CodeGenerationElixir::checkFBE(Fbe *fbe){

    std::stringstream ss;
    ss << MAIN_FOLDER << "/lib/" << fbe->getName();
    std::string path = ss.str();

    //Skip Main FBE Creation
    if(!strcmp(fbe->getName().c_str(),"Main"))
        return;

    //Check if FBE implemented: check if respective folder exists
    if(!directoryExists(path)){
        createFBE(fbe, path);
    }

}
```

Fonte: Autoria própria

Para cada FBE é criada uma pasta com o seu nome. Nesta pasta é criado o arquivo *fbe.ex* com a definição do elemento em formato de módulo Elixir. Em seguida os *Methods* do *FBE* são traduzidos para funções Elixir dentro deste módulo. Os *Attributes* são traduzidos em estados dentro do método *init_attributes*. O Código 12 apresenta um exemplo da tradução de *Attributes* para o *framework* feita pelo compilador.

Código 12. Exemplo de tradução de um Attribute para *Framework* NOP Elixir



Fonte: Autoria própria

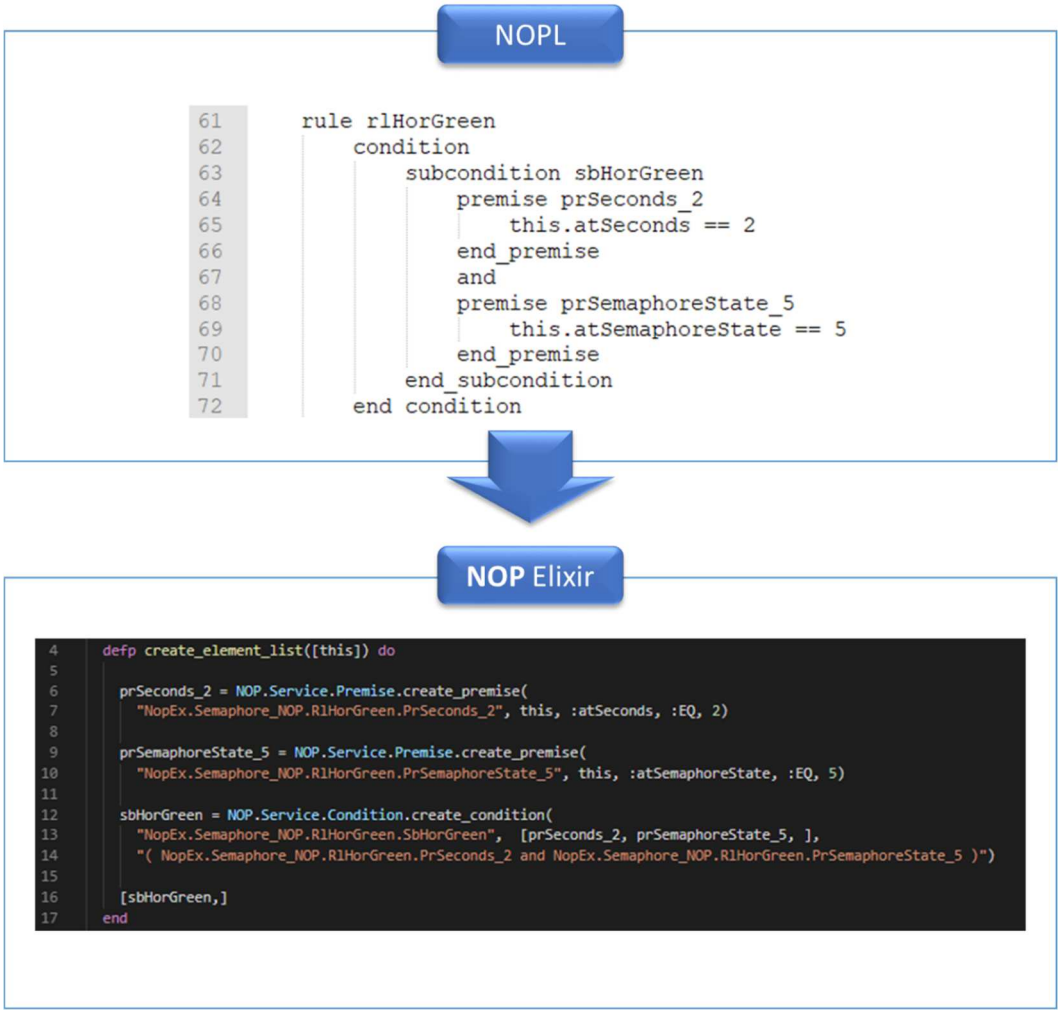
Os *Methods* de atribuição são traduzidos em funções dentro do módulo e recebem o mesmo nome dado no Grafo PON. O Código 13 apresenta o exemplo da tradução de um *Method* do tipo atribuição para o *Framework*.

Código 13. Exemplo de tradução de um *Method* para *Framework* NOP Elixir

Fonte: Autoria própria

Após a tradução do *FBE*, as *Rules* da *FBE* são traduzidas para módulos específicos em arquivos cujo nome é o nome da *Rule,* ficando estes arquivo*s* também na pasta do *FBE*. O Código 14 apresenta o exemplo da tradução de uma lógica de uma *Rule* para o *Framework* NOP Elixir.

Código 14. Exemplo da tradução de lógica de uma Rule para *framework* NOP Elixir



Fonte: Autoria própria

Dentro do módulo principal em *fbe.*ex é criada uma função *initialize().* Nesta função se encontra a inicialização do *FBE*, bem como a inicialização e vinculação de cada módulo gerado a partir da tradução das *Rule* do *FBE.* Esta função de inicialização garante a correta instanciação de todo o ambiente PON com o *FBE* e suas *Rules.* O Código 15 apresenta um exemplo de uma função *initialize()* gerada a partir do compilador.

Código 15. Exemplo de função *initialize* declarada em todos os módulos FBE

Elixir

```elixir
def initialize(name) do

  this = NOP.Service.FBE.create_fbe(NopEx.Semaphore_NOP, name)

  NOP.Service.Rule.create_rule(NopEx.Semaphore_NOP.RlHorGreen, "NopEx.Semaphore_NOP.RlHorGreen", [this])

  NOP.Service.Rule.create_rule(NopEx.Semaphore_NOP.RlHorRed, "NopEx.Semaphore_NOP.RlHorRed", [this])

  NOP.Service.Rule.create_rule(NopEx.Semaphore_NOP.RlHorYellow, "NopEx.Semaphore_NOP.RlHorYellow", [this])

  NOP.Service.Rule.create_rule(NopEx.Semaphore_NOP.RlVerGreen, "NopEx.Semaphore_NOP.RlVerGreen", [this])

  NOP.Service.Rule.create_rule(NopEx.Semaphore_NOP.RlVerRed, "NopEx.Semaphore_NOP.RlVerRed", [this])

  NOP.Service.Rule.create_rule(NopEx.Semaphore_NOP.RlVerYellow, "NopEx.Semaphore_NOP.RlVerYellow", [this])

  this
end
```

Na construção da definição do *FBE Main*, a tradução ocorre de maneira um pouco diferente da construção dos demais *FBEs*. Nela, a função *initialize()* contém a inicialização das instâncias definidas no grafo PON. A construção das *Main Rules* também ocorre de maneira um diferente. Ao contrário de uma *Rule* de um *FBE* o qual apenas recebe a própria FBE como parâmetro de criação, as *Main Rules* são criadas recebendo como parâmetros todos os *FBEs* envolvidos na *Rule.* Para localizar todos estes *FBEs*, é feita uma navegação em profundidade em todos os elementos de definição lógico-causal (*Subconditions* e *Premises*) e de execução (*Instigations* e *Methods*). Todos estes *FBEs* localizados são apresentados como parâmetros de criação da *Rule.* O Código 16 apresenta um exemplo de uma *Main Rule* traduzida pelo compilador.

Código 16. Exemplo de compilação de uma *Main Rule*

NOPL

```
rule rlOpeningGate
    condition
        premise prRemoteControlOpenOn
            event.atEventState == 1
        end_premise
        and
        premise prGateIsClosed
            gate.atGateState == 0
        end_premise
    end_condition
    action sequencial
        instigation
            call event.mtReset()
            call gate.mtOpened()
        end_instigation
    end_action
end_rule
```

NOP Elixir

```
defmodule NopEx.Main.RlOpeningGate do
  use NOP.Element.Rule

  defp create_element_list([[event, gate, ]]) do

    prGateIsClosed = NOP.Service.Premise.create_premise("NopEx.Main.RlOpeningGate.PrGateIsClosed",
      gate, :atGateState, :EQ, 0)

    prRemoteControlOpenOn = NOP.Service.Premise.create_premise("NopEx.Main.RlOpeningGate.PrRemoteControlOpenOn",
      event, :atEventState, :EQ, 1)

    [prGateIsClosed,prRemoteControlOpenOn,]
  end

  defp create_instigation_list([[event, gate, ]]) do

    [
      {NopEx.Event, :mtReset, [event]},
      {NopEx.Gate, :mtOpened, [gate]},
    ]

  end

end
```

Fonte: Autoria própria

O Código 17 apresenta um exemplo de uma função *initialize()* e uma função *create_main_rules()* criada pelo compilador para um *FBE Main*.

Código 17. Exemplo de função *initialize* e *create_main_rules* para o *FBE Main*

Elixir

```
def initialize() do
  instances = %{}
  instances = Map.put(instances, :event, NopEx.Event.initialize("event"))

  instances = Map.put(instances, :gate, NopEx.Gate.initialize("gate"))

  create_main_rules(instances)

  instances
end

def create_main_rules(instances) do

  NOP.Service.Rule.create_rule(NopEx.Main.RlClosingGate, "NopEx.Main.RlClosingGate",
    [[instances[:event], instances[:gate], ]])

  NOP.Service.Rule.create_rule(NopEx.Main.RlOpeningGate, "NopEx.Main.RlOpeningGate",
    [[instances[:event], instances[:gate], ]])


end
```

Fonte: Autoria própria

Validação do resultado

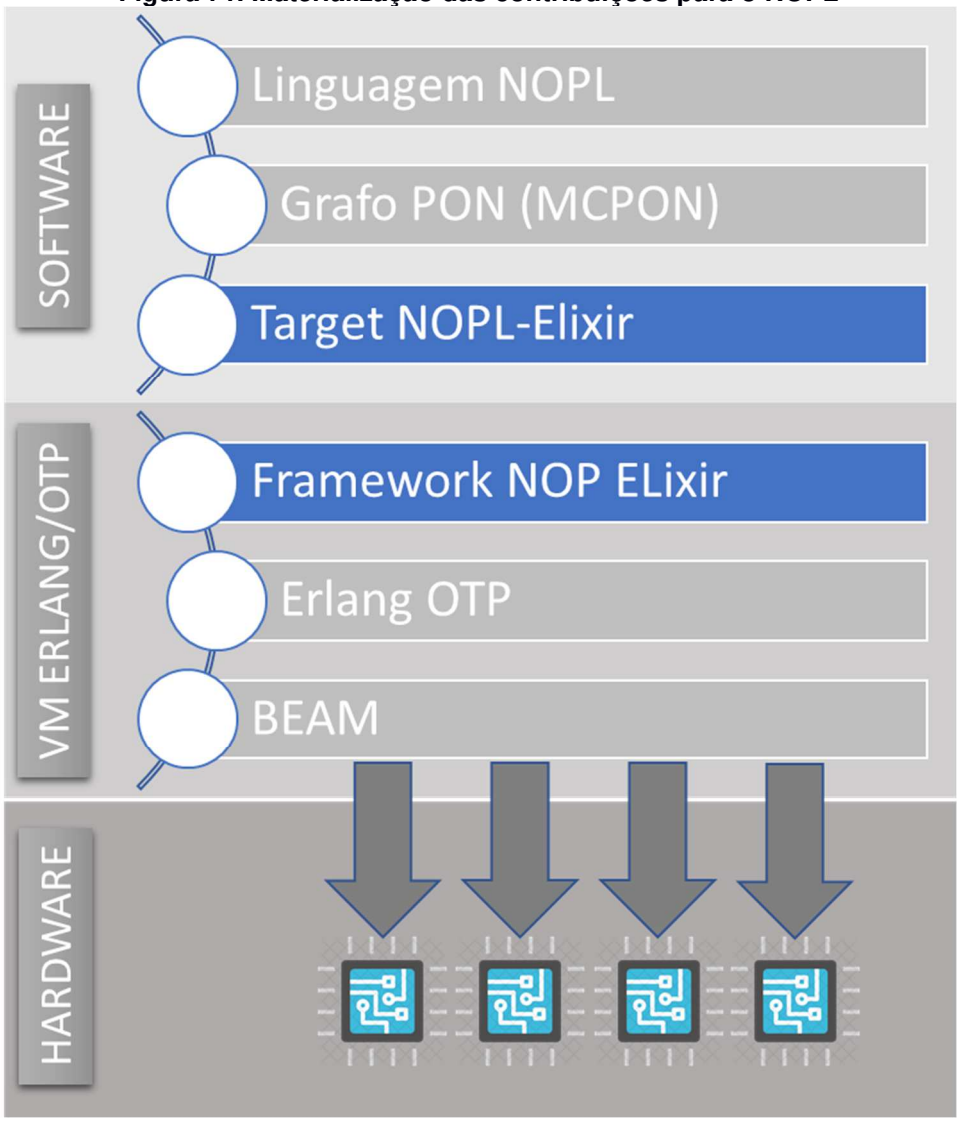
A etapa de validação do resultado consiste em verificar se todo o código gerado pelo compilador possui algum erro de sintaxe. Para esta validação é executado um comando *mix test.* Este comando dispara o processo de compilação do código-alvo gerado. Neste momento, caso algum erro de sintaxe tenha sido encontrado, este é apresentado ao final do processo de compilação. Desta forma é possível, por exemplo, a correção do código fonte e recompilação do programa PON.

4.5 CONSIDERAÇÕES FINAIS

Este capítulo iniciou apresentando uma proposta de modelagem dos elementos do paradigma PON em atores. Como consequência, demonstrou-se a sinergia entre estas duas abordagens. Em seguida, a modelagem de atores foi a base para uma modelagem UML e posterior materialização dos elementos na linguagem Elixir por meio de um *framework* nomeado de *Framework* NOP Elixir. Por último, foi apresentado um compilador para MCPON de forma a gerar elementos PON via NOPL no *framework* NOP Elixir.

O diagrama disposto na Figura 71 apresenta a materialização deste trabalho sob o viés de software e hardware demonstrando a integração do PON através da linguagem NOPL com a arquitetura Erlang/OTP possível a partir das contribuições aqui apresentadas.

Figura 71. Materialização das contribuições para o NOPL



Fonte: Autoria própria

O que foi visto até aqui foi uma proposta de implementação dos elementos PON em modelo de microatores. Atendendo, assim, parte dos objetivos específicos apresentados no Capítulo 1. A seguir, o Capítulo 5 traz os casos de estudos e os resultados obtidos dos experimentos realizados.

5 CASOS DE ESTUDO

Este capítulo tem como objetivo descrever os casos de estudos utilizados para avaliar resultados das contribuições apresentadas no Capítulo 4. São apresentados os materiais e métodos utilizados, a descrição de cada caso de estudo e das diferentes configurações utilizadas, os resultados obtidos e a discussão sobre estes resultados. Ainda são apresentadas reflexões sobre possíveis melhorias a serem aplicadas à solução dada em si, em particular ao *Framework* NOP Elixir.

5.1 CONSIDERAÇÕES INICIAIS

A escolha de casos de uso que apresentassem resultados comparativos concretos foi ponto de várias reflexões e discussões junto ao grupo de pesquisa do PON até se chegar ao que será apresentado a seguir. A discussão se dá basicamente pela singularidade do experimento necessário para a área de estudo, a qual dificulta um comparativo quantitativo que demonstre com clareza os ganhos alcançados com a solução apresentada neste presente trabalho.

Comparativos com outro software pensado em *multicore* não se mostraram uma forma clara de experimentação. Isto se dá porque dois dos dados experimentais que podem ser obtidos, a saber: o tempo gasto no desenvolvimento do respectivo software e o desempenho desta solução, dependem essencialmente do nível de experiência do desenvolvedor.

Por outro lado, se o caso de estudo a ser experimentado fosse software não concebido para *multicore*, ter-se ia inevitavelmente uma solução com pouca ou nenhuma característica para este tipo de ambiente, o que tornaria o comparativo de pouca representatividade[30].

Por último, observados os resultados preliminares, chegou-se à conclusão que apresentar a evolução do programa em ambientes controlados variando-se apenas o número de núcleos foi a forma mais concreta de demonstração. Esta forma é possível observar e calcular o *speedup*, bem como observar o balanceamento de carga entre os núcleos. Ambas as informações obtidas são objetivas e mensuráveis.

[30] Experimentos comparativos entre software PON não concebido para *multicore* e o *Framework* NOP Elixir podem ser vistos no APÊNDICE H**.**

5.2 MATERIAIS E MÉTODOS

Esta seção apresenta os detalhes com relação às ferramentas e equipamentos utilizados para os experimentos, além do método empregado para obtenção e análise dos resultados.

5.2.1 Recursos Materiais e Métodos

Materiais

Como plataforma de execução dos experimentos foram utilizados quatro ambientes virtualizados M5ad Amazon EC2 (AMAZON, 2019a). As instâncias M5ad oferecem processadores AMD EPYC série 7000 com uma velocidade de clock turbo de 2,5 GHz em todos os núcleos. Além disto, estas instâncias disponibilizam a opção de hardware dedicado (AMAZON, 2019), garantindo o mínimo de ruído nos experimentos executados. Todas as máquinas são instaladas com sistema operacional Ubuntu Server 18.04 LTS[31] e armazenamento do tipo SSD[32].

Para ambiente de execução Erlang foi utilizada a distribuição Erlang/OTP 21 (erts-10.3.1) e Eshell V10.3.1. Para ambiente de execução Elixir foi utilizada a versão 1.8.1. Estas versões foram replicadas para todas as instâncias a fim de manter o máximo de equivalência entre software e sistema operacional.

Finalmente, salienta-se que foi optado por este tipo de ambiente primeiramente pela facilidade de acesso e relativamente baixo custo para utilização, além do fato de manterem uma boa equivalência de hardware entre as instâncias, variando-se essencialmente o número de núcleos disponíveis. Há uma variação na memória disponível entre os modelos utilizados, contudo será falado acerca da influência da memória com mais detalhes em momento oportuno ainda neste capítulo. A Tabela 13 apresenta os tipos utilizados nos experimentos e suas respectivas quantidades de núcleos e memória.

---

[31] Distribuição: *Ubuntu Server 18.04 LTS (HVM), SSD Volume Type - ami-02c8813f1ea04d4ab*

[32] SSD (do inglês, *Solid-State Drive*) ou unidade de estado sólido é um tipo de dispositivo, sem partes móveis, para armazenamento não volátil de dados digitais.

Tabela 13. Detalhamentos dos tipos de máquinas usadas no experimento

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

Fonte: Autoria própria

Método Aplicado

O método utilizado para realizar os experimentos envolveu os seguintes passos:

- Desenvolvimento de programa em linguagem NOPL.
- Compilação da linguagem NOPL, via compilador NOPL para *Framework* NOP Elixir, para geração de produto executável em ambiente Elixir.
- Execução de produto da compilação em todos os ambientes virtualizados.
- Tabulação dos valores médios de cinco execuções para os valores: tempo de execução, taxa de ocupação de cada núcleo e consumo de memória.
- Tabelamento dos dados, geração dos gráficos e apresentação e análise dos resultados.

5.2.2 Cálculo de tempo de execução

Para os processamentos *single thread* o cálculo de tempo de execução é uma tarefa relativamente simples, ou seja, em linhas gerais grava-se um *timestamp* em um momento imediatamente anterior ao processo que deseja computar e um mesmo *timestamp* imediatamente após o final deste. O resultado é a diferença destes dois valores. Para o caso do *Framework* NOP Elixir que é paralelizado em vários processos, o cálculo do tempo de execução torna-se mais complexo.

Após várias investidas e consultas às comunidades de desenvolvedores Erlang e Elixir, verificou-se que a forma mais assertiva de avaliar se a execução de um processo foi finalizada, é observando a fila de mensagens e o status de todos os processos relacionados aos elementos PON, ou seja, os microatores. Quando todos

os processos estiverem com sua fila de mensagens vazia e seu status de processamento for *aguardando*, entende-se que o fluxo de trabalho foi finalizado.

Um efeito colateral desta forma de avaliação do tempo em um processo *busy-waiting*[33] é a sobrecarga (do inglês, *overhead*) causado por este processamento. Portanto, todos os cálculos de tempo apresentados neste capítulo são influenciados por esta sobrecarga, de forma que é possível afirmar que, se não houvesse a contagem de tempo, o tempo real de execução seria menor do que os apresentados.

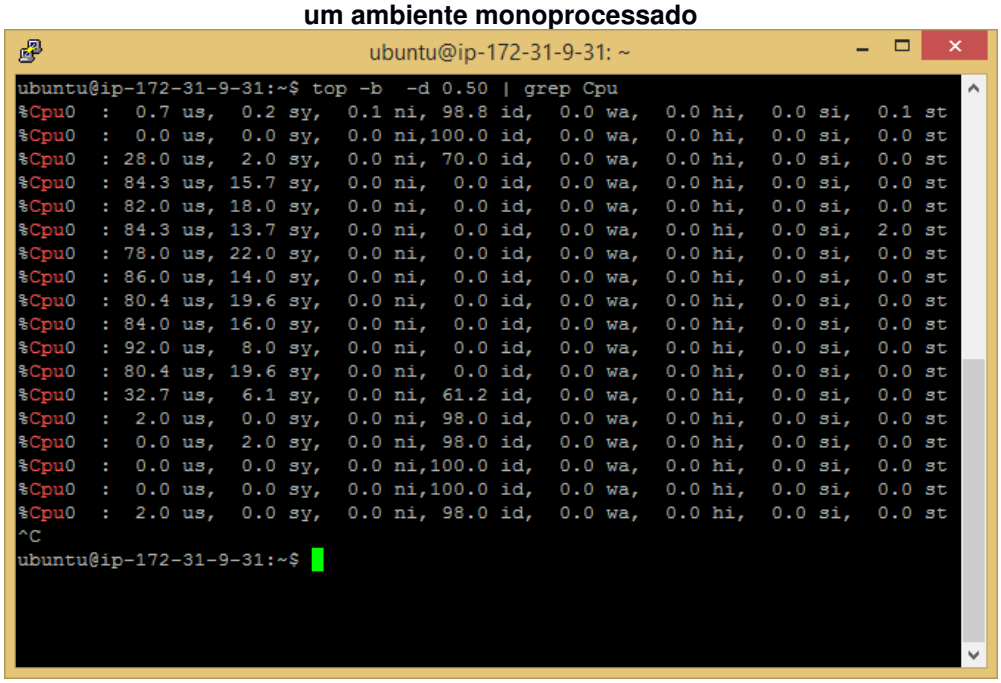
Como até o presente momento não é possível estimar o quanto de sobrecarga este processo de contagem interfere no resultado, os valores finais foram mantidos com esta sobrecarga.

5.2.3 Cálculo da taxa de ocupação dos núcleos

Para identificar a taxa de ocupação de cada núcleo foi utilizado o comando *top* do Ubuntu. Este comando foi acionado em modo *batch* (opção -b) com intervalos variados conforme cada núcleo (opção -d). Em seguida estes valores foram direcionados para arquivo a fim de serem preservados durante as execuções, posteriormente estes valores foram, então, tabulados para serem devidamente apresentados. A Figura 72 apresenta um exemplo da saída do comando *top* em modo *batch* usado para armazenar os valores dos experimentos.

[33] *busy-waiting*, *busy-looping* ou *spinning* é uma técnica na qual um processo verifica repetidamente se uma condição seja verdadeira, como se a entrada do teclado ou um bloqueio está disponível.

Figura 72. Exemplo de saída do comando top para acompanhamento de taxa de ocupação de um ambiente monoprocessado



Fonte: Autoria própria

5.2.4 Cálculo de ocupação da memória

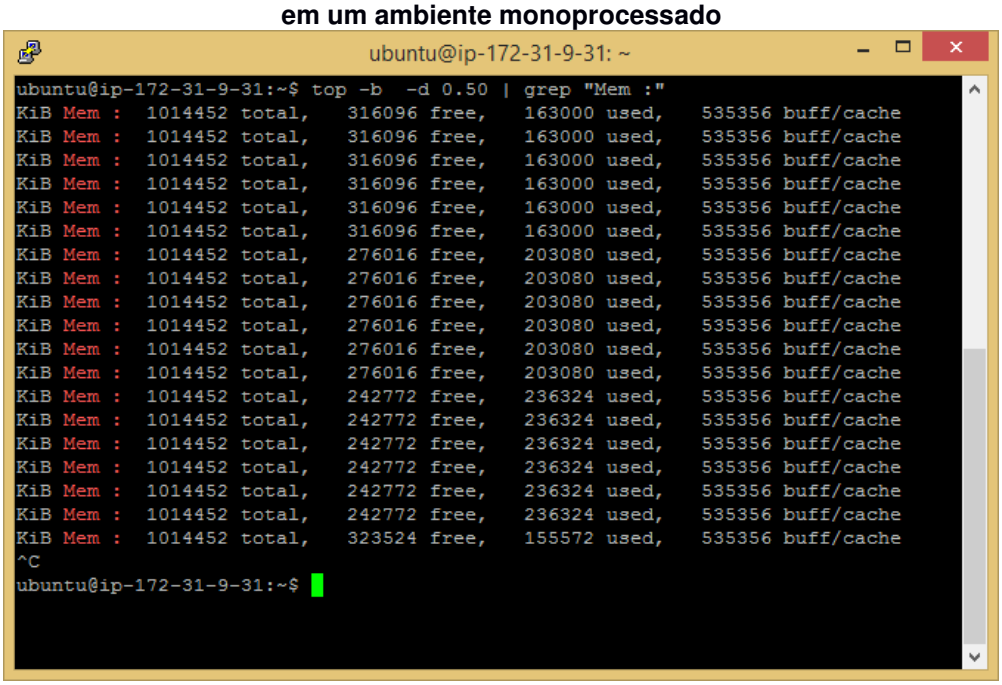
De maneira geral, os exercícios aqui apresentados são bastante modestos no consumo de memória em relação à sua disponibilidade mesmo no ambiente com menor disponibilidade, como o MONO, que possui apenas 2GB de memória (vide Tabela 13). O objetivo principal deste cálculo é demonstrar que a memória não atuou como um limitante nos resultados de forma a garantir o mínimo ou nenhuma interferência nos resultados. Desta forma, o valor auferido durante os experimentos foi o consumo total de memória do SO. Com isto, objetivou-se demonstrar que:

1. O consumo total de memória do sistema operacional não atinge valores elevados a ponto de ser necessário ativar memória virtual ou swap[34].
2. O consumo de memória total médio é equivalente em todas as simulações independente do ambiente virtualizado.

[34] Memória *Swap* é uma memória virtual que se utiliza área de memória do disco como memória complementar. A utilização deste recurso normalmente interfere na performance dos programas por fazer paginação ou *swap* entre a memória RAM e o disco.

Para identificar a memória total ocupada pelos experimentos o método foi muito parecido com o do cálculo de ocupação dos núcleos apresentado na seção 5.2.3. Foi utilizado o comando top do Ubuntu em formato batch (opção -b) com intervalos variados conforme cada núcleo (opção -d). Em seguida estes valores foram direcionados para arquivo a fim de serem preservados durante as execuções, posteriormente estes valores foram tabulados para serem então apresentados. A Figura 73 apresenta um exemplo da saída do comando top em modo batch usado para armazenar os valores dos experimentos.

Figura 73. Exemplo de saída do comando top para acompanhamento de ocupação de memória em um ambiente monoprocessado



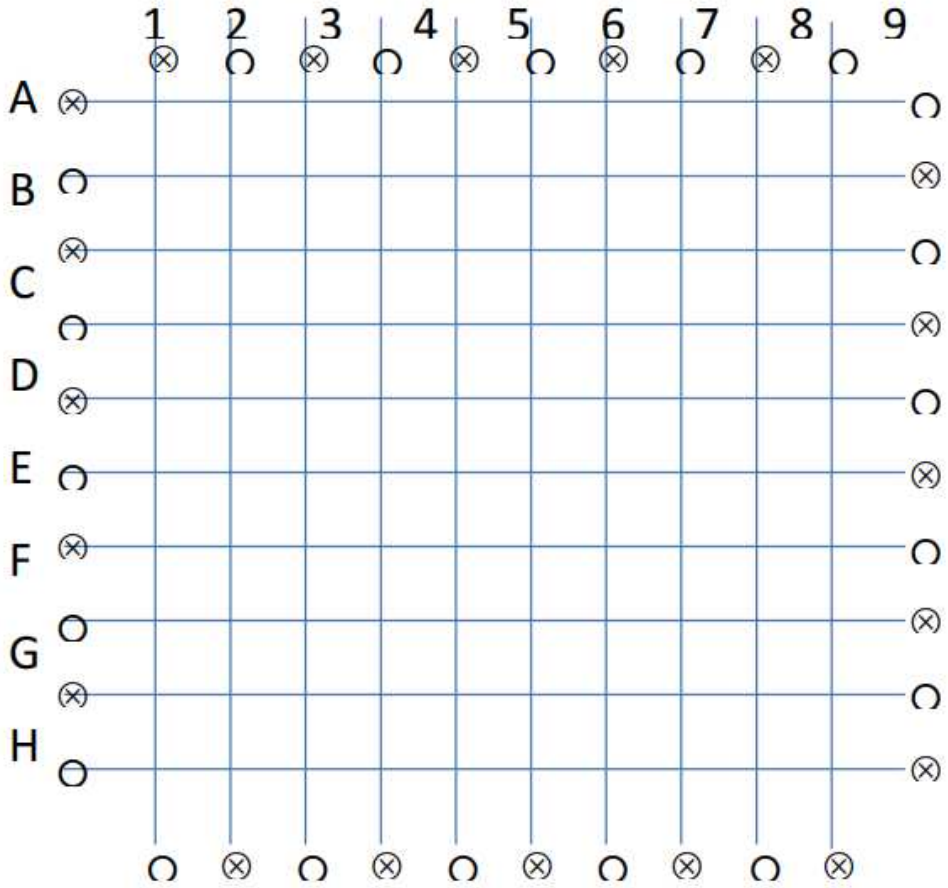
Fonte: Autoria própria

5.2.5 Ambiente de Simulação

Todos os casos de estudo apresentados neste trabalho são executados em um mesmo ambiente de simulação para Controle de Tráfego Automatizado (CTA). Todos os detalhes e lógicas apresentados se encontram em (RENAUX, R., *et al.*, 2015). O objetivo do CTA é criar um ambiente de simulação de tráfego de uma área urbana aplicando-se diferentes estratégias de controle a fim de comparar o desempenho computacional conforme aumenta a complexidade da estratégia.

O simulador representa elementos do mundo real, como veículos, ruas, pistas, quadras, sinaleiros, sensores e cruzamentos em uma região de simulação matricial composta por dez ruas verticais e 10 ruas horizontais de mão única, formando uma matriz 10x10 com um total de 100 interseções como pode ser visto na Figura 74.

Figura 74. Ambiente de simulação CTA: 10 ruas verticais - 10 ruas horizontais



Fonte: (RENAUX, R., *et al.*, 2015)

Cada quadra possui o comprimento de 100 metros e capacidade de 25 veículos por pista, sendo que cada rua pode possuir de 1 a 4 pistas. Os veículos são criados de forma constante, no intervalo de 0.1, 0.2, 0.3, 0.4 ou 0.5 veículos por segundo. Quando um veículo é criado em uma entrada, ele começa a se movimentar a uma velocidade de 20m/s, e deve parar de se movimentar quando o sinal da quadra em que se encontra estiver vermelho ou no caso de a próxima quadra estar cheia. Em cada cruzamento, uma porcentagem de veículos poderá virar para uma outra quadra. Esta porcentagem pode variar de 5% até 35%.

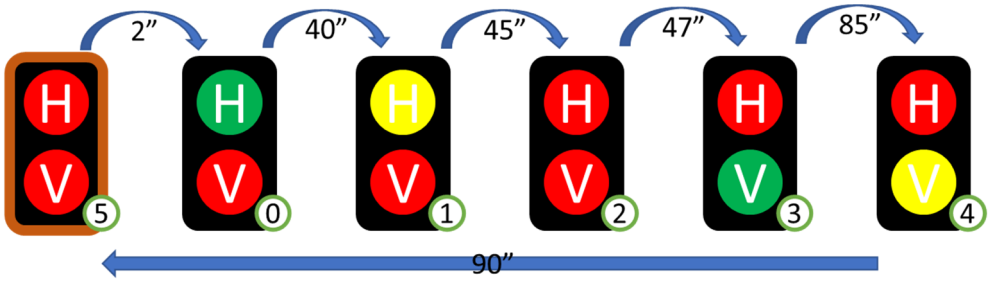
Em todas as quadras são instalados sensores que monitoram a quantidade de veículos que estão parados num sinal vermelho. Esses sensores apresentam três estados: *FEW*, *MANY* e *FULL*. Para o sensor estar no estado *FEW* a taxa de ocupação da rua deve ser menor do que 60%, para o estado *MANY*, a taxa de ocupação deve estar entre 60% e 99% e para o estado *FULL*, a taxa de ocupação deve ser 100%.

5.3 CASO DE ESTUDO I: ESTRATÉGIA INDEPENDENTE

Este primeiro caso de estudo se deu pela aplicação da estratégia de controle Independente (IND). Na estratégia independente, cada semáforo possui tempos fixos para cada estado do sinaleiro, não sendo considerados os sensores de quantidade de veículos e tempos de semáforos vizinhos. A execução deste caso de estudo ocorre em um número de 4000 iterações, em que cada iteração representa a passagem de uma unidade de tempo no qual os estados do semáforo são avaliados e suas ações executadas, de acordo com o resultado lógico causal. Para isso, a lógica de funcionamento deste controle de semáforos é descrita pelo diagrama de estados da Figura 75.

Os círculos *H* representam a cor do semáforo para a rua disposta horizontalmente, os círculos *V* representam a cor do semáforo para a rua disposta verticalmente. Os números abaixo das setas azuis representam o intervalo em segundos até o próximo estado, este apontado pela flecha. Os números no canto inferior direito são o identificador de cada estado. O código fonte em NOPL deste caso de uso está disponível em sua integralidade no APÊNDICE E.

Figura 75. Diagrama de estados da aplicação "Controle de Semáforos"



Fonte: adaptado de (RENAUX, R., *et al.*, 2015)

O elemento semáforo, então, é traduzido em um *FBE* e possui os *Attributes* segundos (*atSeconds*) e estado (*atSemaphoreState*) para representar sua situação. Esta modelagem é então traduzida para NOPL conforme detalhado no Código 18.

Código 18. Exemplo de um *FBE* da aplicação "Controle de semáforos"

NOPL

```
1   fbe Semaphore_NOP
2
3       public integer atSeconds = 0
4
5       public integer atSemaphoreState = 5
6
7   end_fbe
```

Fonte: Autoria própria

Methods também são acrescentados à definição do *FBE* como ações que podem ser implementadas pela instância dela. O Código 19 representa a definição de um método dentro do *FBE* semáforo seguindo a definição da linguagem NOPL.

Código 19. Exemplo de um Método de um *FBE* da aplicação "Controle de semáforos"

NOPL

```
7       private method mtResetTimer
8           attribution
9               this.atSeconds = 0
10          end_attribution
11      end_method
```

Fonte: Autoria própria

Cada transição do diagrama da Figura 75 é traduzida para uma *Rule*, totalizando seis *Rules* para esta estratégia de controle. Por sua vez, cada *Rule* contém

duas *Premises*, sendo a avaliação do estado e o tempo do *FBE* Semáforo. O Código 20 apresenta um exemplo de uma *Rule* desenvolvida para a sintaxe do NOPL.

Código 20. Exemplo de uma Rule de aplicação de controle de semáforos

NOPL

```
49      rule rlHorGreen
50          condition
51              subcondition sbHorGreen
52                  premise prSeconds_2
53                      this.atSeconds == 2
54                  end_premise
55                  and
56                  premise prSemaphoreState_5
57                      this.atSemaphoreState == 5
58                  end_premise
59              end_subcondition
60          end_condition
61          action acHTLG
62              instigation inHTLG
63                  call this.mtHorGreen()
64              end_instigation
65          end_action
66      end_rule
```

Fonte: Autoria própria

A dinâmica de execução deste experimento consiste em iterar 4000 vezes, na qual a cada interação, uma matriz 10x10 de “semáforos” (totalizando 100 semáforos) tem seu *Attribute* de tempo incrementado e seu estado reavaliado.

O resultado da compilação NOPL para o *Framework* NOP Elixir resultou em um conjunto de 1 (um) microator *FBE,* 6 (seis) microatores *Rules,* 6 (seis) microatores *Condition,* 12 (doze) microatores *Subcondition* e 6 (seis) microatores *Instigation* para cada semáforo. Como o experimento gerou uma matriz de 10x10, cada um dos elementos acima descritos foi reproduzido 100 (cem) vezes. Desta forma, foi criado um total de 3.100 (três mil e cem) microatores.

5.3.1 Configurações dos experimentos

O programa, após compilado foi transferido para cada um dos ambientes virtualizados, foi então executado trinta vezes e os valores foram capturados e tabulados. Durante o período de execução houve o monitoramento da taxa de ocupação dos núcleos e do número de notificações. As execuções geraram uma quantidade média[35] de 2.861.271 notificações. Todos os valores obtidos podem ser verificados em detalhes no APÊNDICE B - Tabulação de resultados do caso de estudo I.

5.3.2 Resultados e discussões

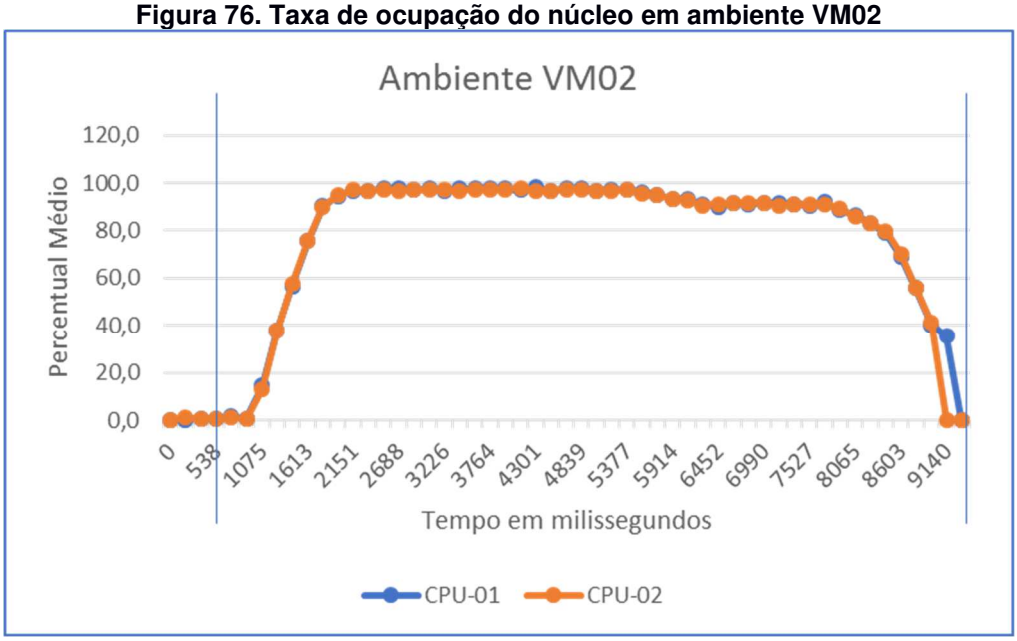
A seguir são apresentados os resultados quantitativos da execução da estratégia independente em cada um dos ambientes virtuais apresentados na seção 5.2. Nesta subseção são apresentados os valores tabulados e, sempre que pertinente, cada ambiente virtual é comparado à luz dos demais ambientes virtuais de maneira objetiva.

Ambiente VM02 (dois núcleos)

O ambiente VM02 teve um tempo médio de 8499 milissegundos após as trinta execuções. O desvio padrão foi de 232,579 – equivalente à 2,74%. A Figura 76 apresenta a taxa de ocupação média de cada núcleo nas trinta execuções. As linhas verticais azuis representam os momentos em que a execução é iniciada e finalizada.

[35] O número varia minimamente ainda em função de indeterminismo causado pela concorrência. Mais detalhes sobre indeterminismo será apresentado na seção 6.3

Figura 76. Taxa de ocupação do núcleo em ambiente VM02

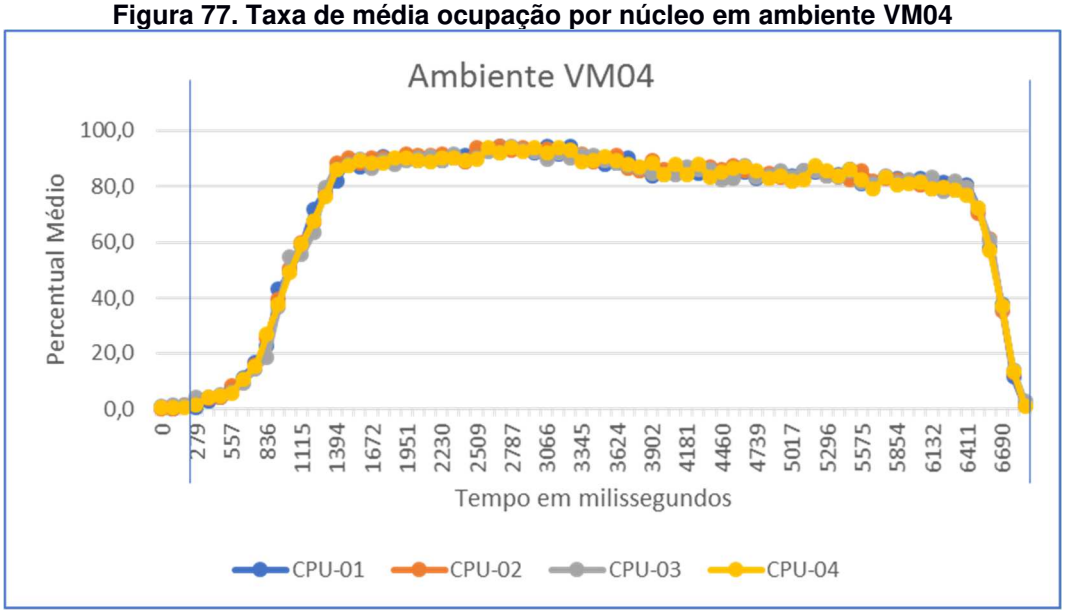


Fonte: Autoria própria

Ambiente VM04 (dois núcleos)

O ambiente VM04 teve um tempo médio de 6469 milissegundos. O desvio padrão nas trinta execuções foi de 231,952 – equivalente a 3,59%. Este valor médio representa uma redução de 23,89% em relação ao ambiente VM02. Sua taxa de ocupação entre os núcleos, como pode ser visto na Figura 77, se manteve bem balanceado entre todas as execuções. As linhas verticais azuis representam os momentos em que a execução é iniciada e finalizada.

Figura 77. Taxa de média ocupação por núcleo em ambiente VM04

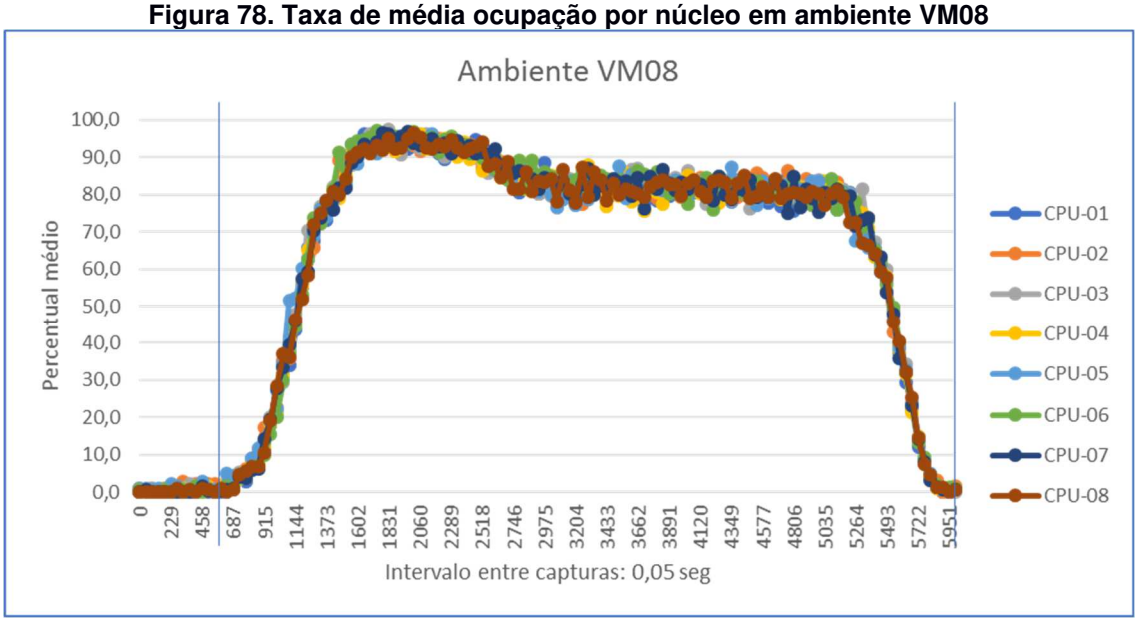


Fonte: Autoria própria

Ambiente VM08 (oito núcleos)

O ambiente VM08 teve um tempo médio de 5542 milissegundos. O desvio padrão nas trinta execuções foi de 198,868 – equivalente a 3,59%. Este valor médio representa uma redução de 34,79% em relação ao ambiente VM02 e 14,32% em relação ao ambiente VM04. Sua taxa de ocupação entre os núcleos também se apresenta bem balanceada durante a execução das simulações como pode ser visto na Figura 78. As linhas verticais azuis representam os momentos em que a execução é iniciada e finalizada.

Figura 78. Taxa de média ocupação por núcleo em ambiente VM08

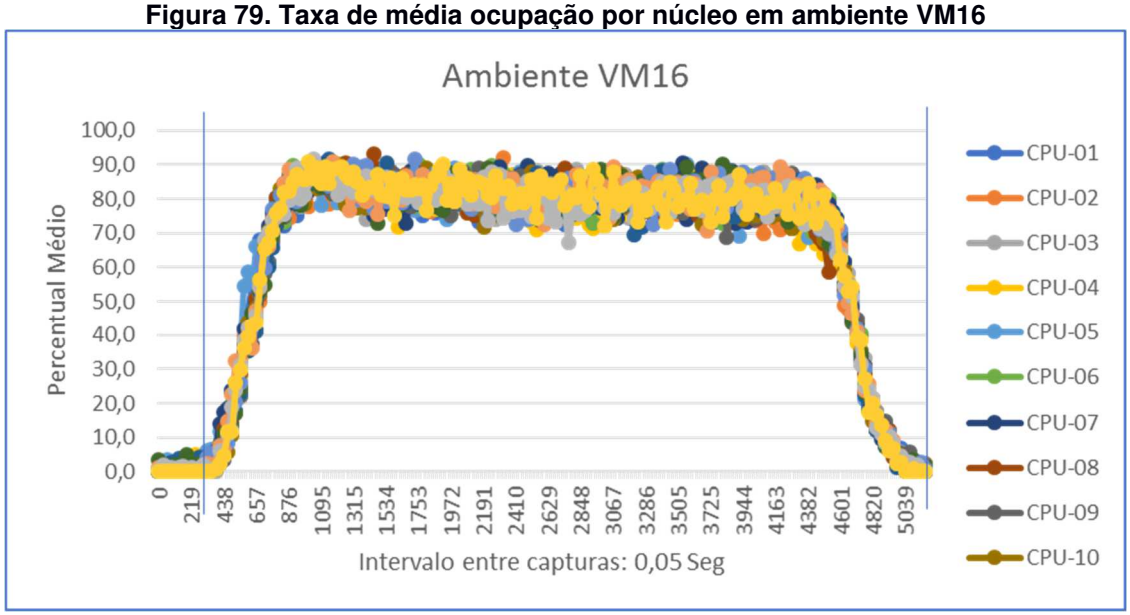


Fonte: Autoria própria

Ambiente VM16 (dezesseis núcleos)

O ambiente VM16 teve um tempo médio de 4703 milissegundos. O desvio padrão nas trinta execuções foi de 187,964 – equivalente a 4,00%. Este valor médio representa uma redução de 44,66% em relação ao ambiente VM02 e 15,13% em relação ao ambiente VM08. Sua taxa de ocupação entre os núcleos não diferiu dos demais ambientes apesar de maior número de núcleos como é apresentado na Figura 79. As linhas verticais azuis representam os momentos em que a execução é iniciada e finalizada.

Figura 79. Taxa de média ocupação por núcleo em ambiente VM16

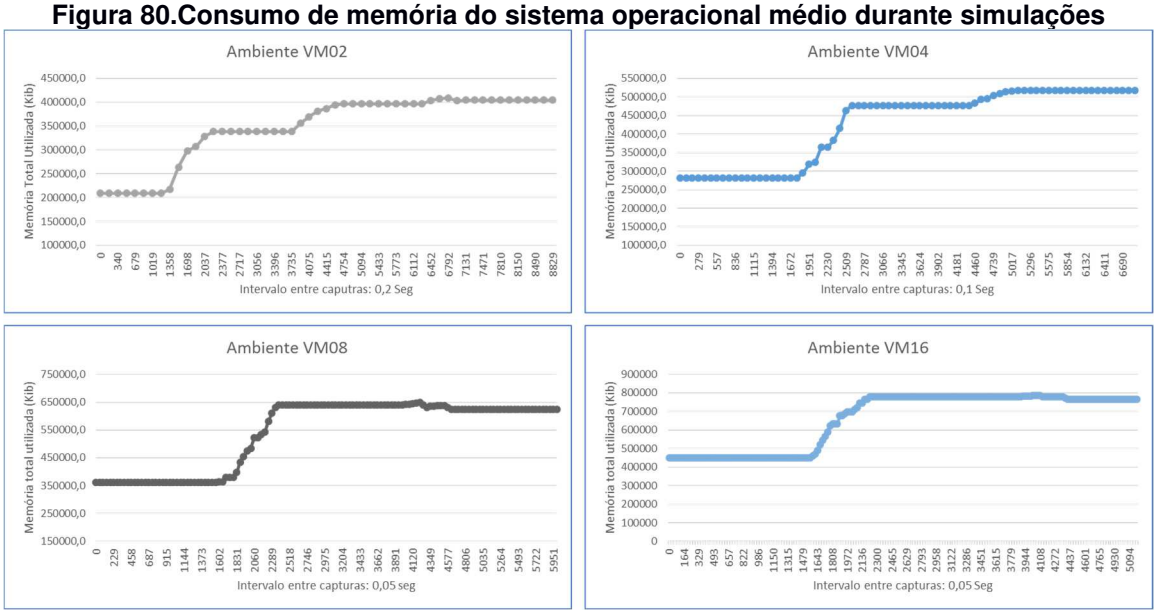


Fonte: Autoria própria

5.3.3 Consumo de memória

Com é possível observar na Figura 80, o consumo de memória total, ou seja, consumo de tarefas do sistema operacional somado ao consumo de memória da simulação, ficou longe do disponível mesmo na instância VM02 que possuía 8 GB de memória RAM. As simulações neste ambiente não ultrapassaram a 450 MB em nenhuma simulação neste ambiente. Desta forma, a memória não teve impactos negativos na simulação, pois não houve acesso à memória virtual (*swap*). Isto posto, é possível concluir que a melhora no desempenho da simulação é, majoritariamente, em função do aumento do número de núcleos.

Figura 80.Consumo de memória do sistema operacional médio durante simulações

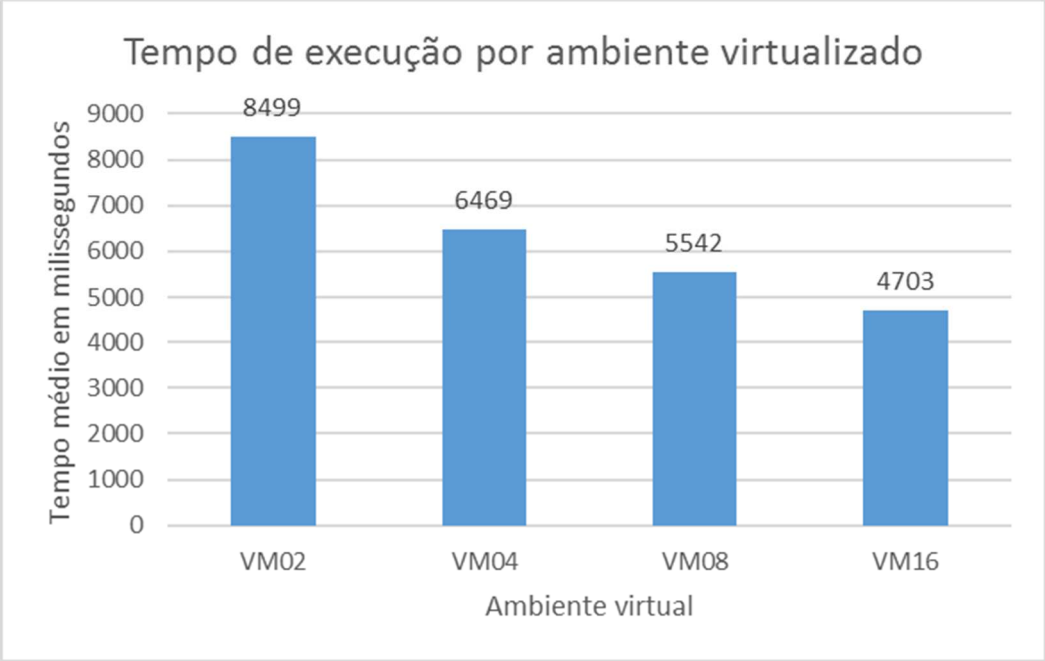


Fonte: Autoria própria

5.3.4 Considerações finais

Os resultados apresentados nas seções anteriores apresentam a execução do programa NOPL materializado em um ambiente Erlang através do *Framework* NOP Elixir. A execução deste programa nos diversos ambientes cuja principal diferença é a disponibilidade de núcleos apresenta uma expressiva redução conforme são acrescidos os núcleos como pode ser visto na Figura 81 ao passo que, em todos os ambientes, a taxa de ocupação dos núcleos se manteve balanceada em todo o tempo de simulação. Estes dados demonstram que a execução lógico-causal foi distribuída entre os núcleos e com um bom nível de balanceamento.

Figura 81. Tempo médio de execução por ambiente virtualizado



Fonte: Autoria própria

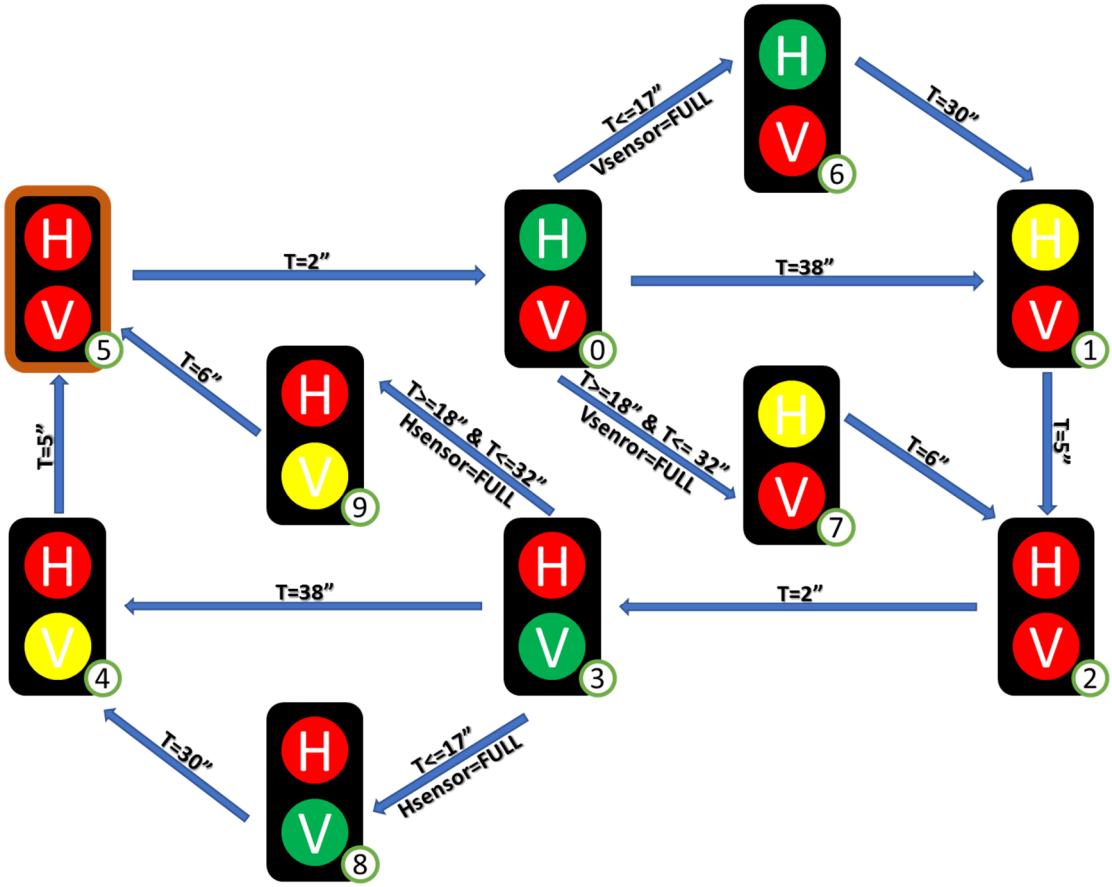
5.4 CASO DE ESTUDO II: CONTROLE BASEADO EM CONGESTIONAMENTO FACILITADO

Na estratégia de controle baseada em congestionamento facilitado (CBCF), diferente da estratégia independente, são considerados dois atributos para mudança de estado dos semáforos: o tempo de cada semáforo e o estado referente à quantidade de veículos parados. Se o sensor detecta que a porcentagem de veículos parados está entre 60% até 100%, e o tempo do sinaleiro em vermelho é menor do que 24 segundos, então o tempo total do sinaleiro no estado vermelho é ajustado para 30 segundos.

Caso o sensor detecte que a taxa de ocupação está entre 60% e 100% e o tempo do semáforo vermelho está entre 25 segundos e 39 segundos, o sinaleiro oposto altera imediatamente para o estado amarelo e o tempo restante do sinaleiro no estado vermelho é ajustado para 6 segundos. Se o sensor detectar que a ocupação está entre 60% e 100% (*MANY* ou *FULL*), e o tempo do sinaleiro em vermelho for maior do que 39 segundos, não é realizada nenhuma alteração no tempo do sinaleiro. A Figura 82 apresenta um diagrama de estados da estratégia de controle CBCF.

Os círculos *H* representam a cor do semáforo para a rua disposta horizontalmente, os círculos *V* representam a cor do semáforo para a rua disposta verticalmente. O texto descrito em cada seta representa a lógica para o próximo estado, este apontado pela flecha. Os números no canto inferior direito são o identificador de cada estado. O código fonte em NOPL deste caso de uso está disponível em sua integralidade no APÊNDICE F.

Figura 82. Diagrama de estados da estratégia CBCF



Fonte: adaptado de (RENAUX, R., *et al.*, 2015)

O elemento semáforo é então traduzido para o *FBE* que além dos *Attributes* segundos (*atSeconds*) e estado (*atSemaphoreState),* também passa a ter *Attributes* para representar o estado dos sensores de ocupação vertical *(atVVSS*) e horizontal (*atHVSS*). Convertido, então, em linguagem NOPL conforme pode ser visto no Código 21.

Código 21. *FBE* para estratégia CBCF em NOPL

NOPL

```
1   fbe Semaphore_NOP
2
3       public integer atSeconds = 0
4
5       public integer atSemaphoreState = 5
6
7       public integer atHVSS = 0
8
9       public integer atVVSS = 0
10
```

Fonte: Autoria própria

A quantidade de *Methods* para atribuição de estados também aumentou em comparação à estratégia independente. Passando a contar agora com dez diferentes *Methods* para este fim (controle independente possuía apenas seis). Mas o maior incremento de complexidade está no incremento do número de *Rules*, que passou de seis na estratégia independente para dezoito. O Código 22 apresenta um exemplo de uma *Rule* em estratégia CBCF traduzido para NOPL.

Código 22. Rule para estratégia CBCF em NOPL

NOPL

```
86      rule rlCBCL1
87          condition
88              subcondition sbCBCL1
89                      premise prSeconds this.atSeconds == 2 end_premise
90                      and
91                      premise prSemaphoreState this.atSemaphoreState == 5 end_premise
92              end_subcondition
93          end_condition
94          action actCBCL1
95              instigation inCBCL1
96                  call this.mtHTLG()
97                  call this.mtRT()
98              end_instigation
99          end_action
100     end_rule
```

Fonte: Autoria própria

A dinâmica de execução deste experimento segue a mesma de sua antecessora, ou seja, iterar 4000 vezes, na qual a cada interação, uma matriz 10x10 de “semáforos” (totalizando 100 semáforos) tem seu *Attribute* de tempo incrementado e seu estado reavaliado.

O resultado da compilação em NOPL Erlang-Elixir resultou em um conjunto de 1 (um) microator *FBE,* 18 (dezoito) microatores *Rules,* 18 (dezoito) microatores *Condition,* 36 (trinta e seis) microatores *Subcondition* e 18 (dezoito) microatores *Instigation* para cada semáforo. Como o experimento gerou uma matriz de 10x10, cada um dos elementos acima descritos foi reproduzido 100 (cem) vezes. Desta forma, foram criados um total de 9.100 (nove mil e cem) microatores.

5.4.1 Configurações dos experimentos

O programa, após compilado foi transferido para cada um dos ambientes virtualizados, foi então executado trinta vezes e os valores foram capturados e tabulados. Durante o período de execução houve o monitoramento da taxa de ocupação dos núcleos e do número de notificações. As execuções geraram uma quantidade média[36] de 9.217.687 notificações. Todos os valores obtidos podem ser verificados em detalhes no APÊNDICE C.

5.4.2 Resultados e discussões

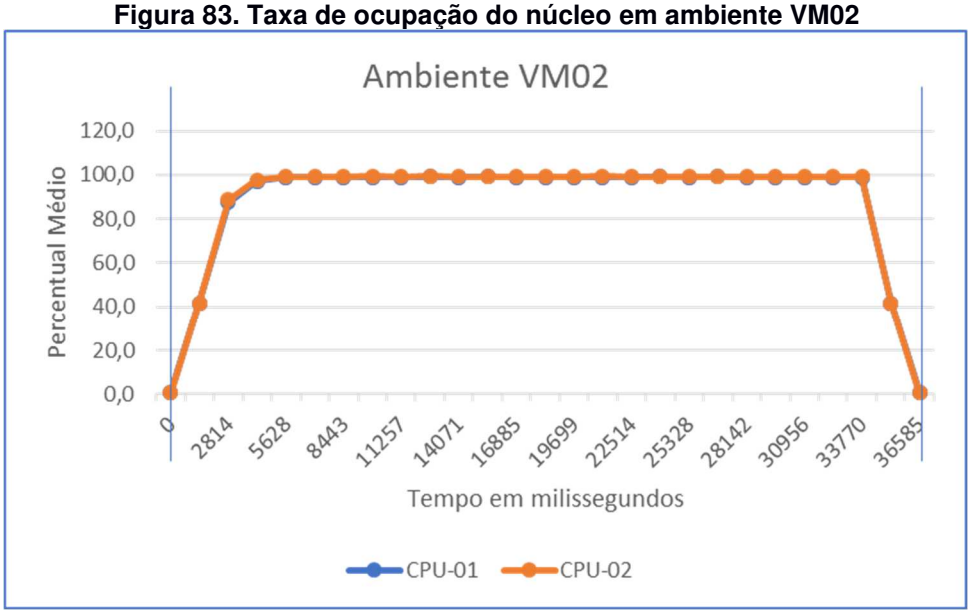
A seguir são apresentados os resultados quantitativos da execução da estratégia CBCF em cada um dos ambientes virtuais apresentados na seção 5.2. Nesta subseção são apresentados os valores tabulados e, sempre que pertinente, cada ambiente virtual é comparado à luz dos demais ambientes virtuais de maneira objetiva.

[36] O número varia minimamente ainda em função de indeterminismo causado pela concorrência. Mais detalhes sobre indeterminismo será apresentado na seção 6.3

Ambiente VM02 (dois núcleos)

O ambiente VM02 teve um tempo médio de 36492 milissegundos após as trinta execuções. O desvio padrão foi 457,496. A Figura 83 apresenta a taxa de ocupação média de cada núcleo nas trinta execuções. As linhas verticais azuis representam os momentos em que a execução é iniciada e finalizada.

Figura 83. Taxa de ocupação do núcleo em ambiente VM02

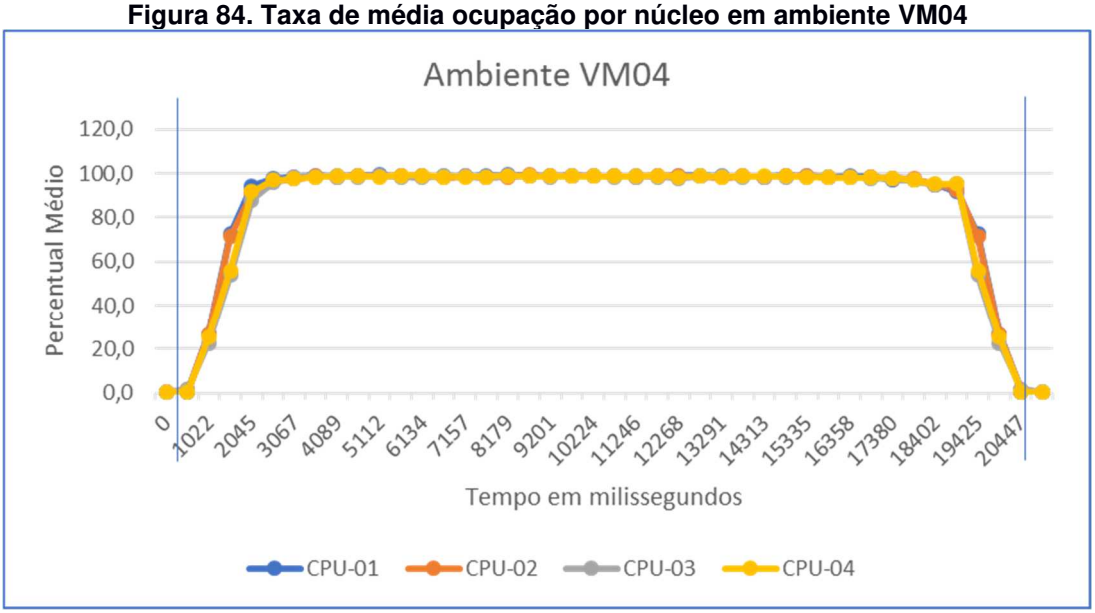


Fonte: Autoria própria

Ambiente VM04 (quatro núcleos)

O ambiente VM04 teve um tempo médio de 19470 milissegundos. O desvio padrão nas trinta execuções foi de 379,405. Este valor médio representa uma redução de 46,65% em relação ao ambiente VM02. Sua taxa de ocupação entre os núcleos, como pode ser visto na Figura 84, se manteve bem balanceado entre todas as execuções. As linhas verticais azuis representam os momentos em que a execução é iniciada e finalizada.

Figura 84. Taxa de média ocupação por núcleo em ambiente VM04

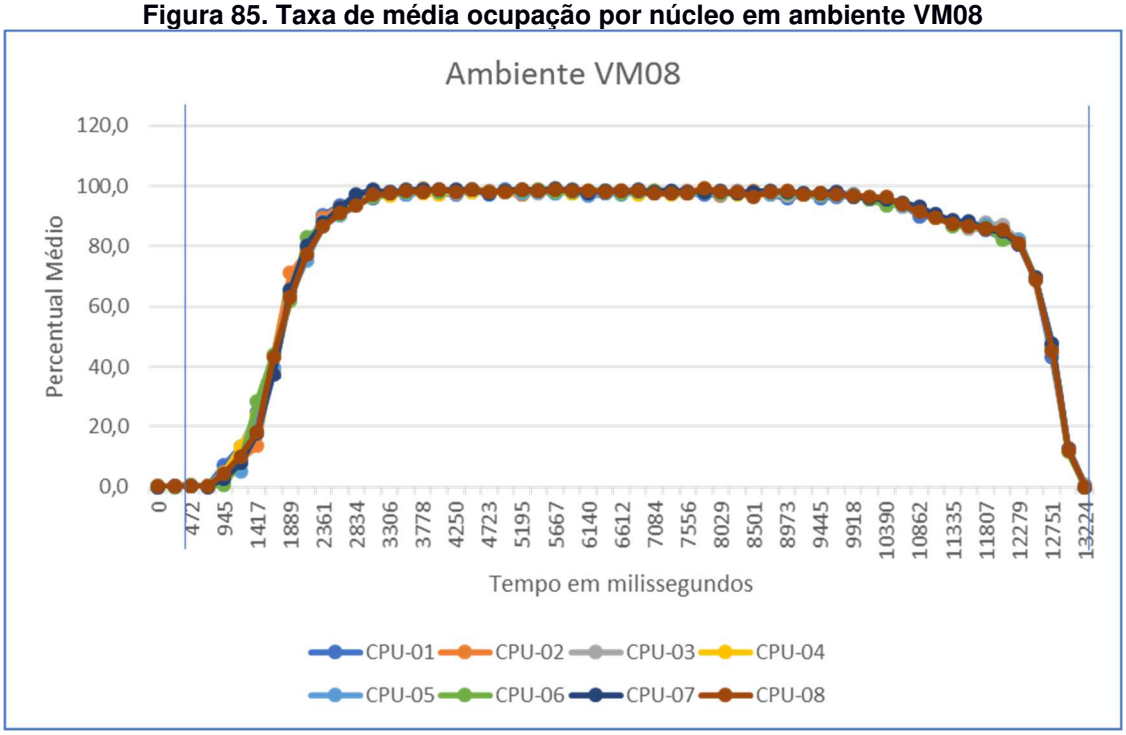


Fonte: Autoria própria

Ambiente VM08 (oito núcleos)

O ambiente VM08 teve um tempo médio de 12960 milissegundos. O desvio padrão nas trinta execuções foi de 437,761 – equivalente a 3,38%. Este valor médio representa uma redução de 64,49% em relação ao ambiente VM02 e 33,44% em relação ao ambiente VM04. Sua taxa de ocupação entre os núcleos também se apresenta bem balanceada durante a execução das simulações como pode ser visto na Figura 85. As linhas verticais azuis representam os momentos em que a execução é iniciada e finalizada.

Figura 85. Taxa de média ocupação por núcleo em ambiente VM08

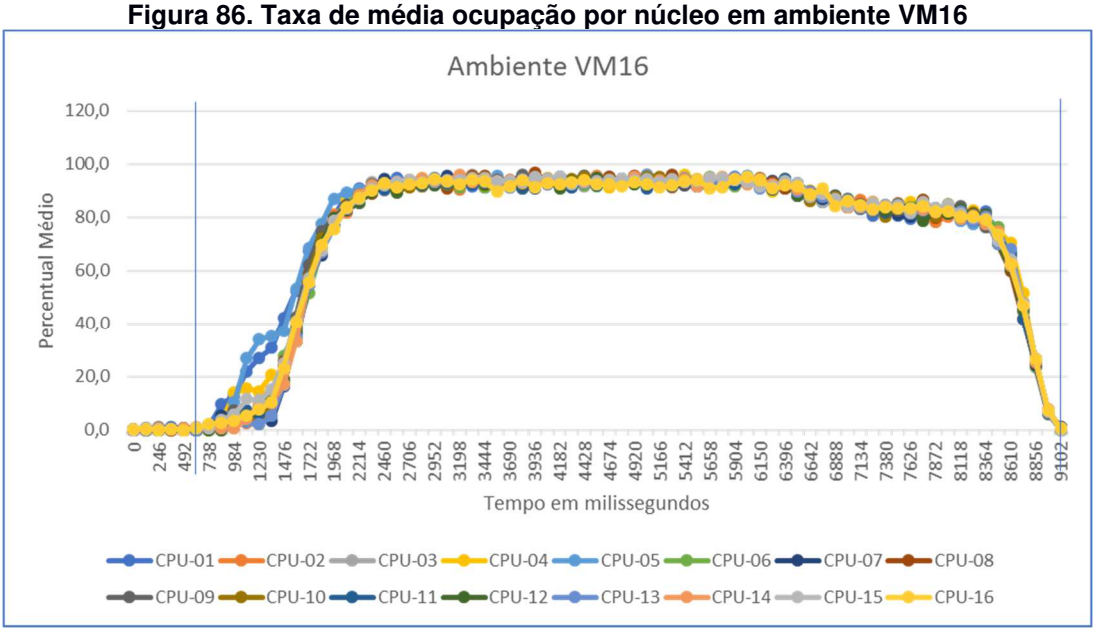


Fonte: Autoria própria

Ambiente VM16 (dezesseis núcleos)

O ambiente VM16 teve um tempo médio de 8525 milissegundos. O desvio padrão nas trinta execuções foi de 133,273 – equivalente a 1,56%. Este valor médio representa uma redução de 76,64% em relação ao ambiente VM02 e 34,22% em relação ao ambiente VM08. Sua taxa de ocupação entre os núcleos não diferiu dos demais ambientes apesar de maior número de núcleos como é apresentado na Figura 86. As linhas verticais azuis representam os momentos em que a execução é iniciada e finalizada.

Figura 86. Taxa de média ocupação por núcleo em ambiente VM16

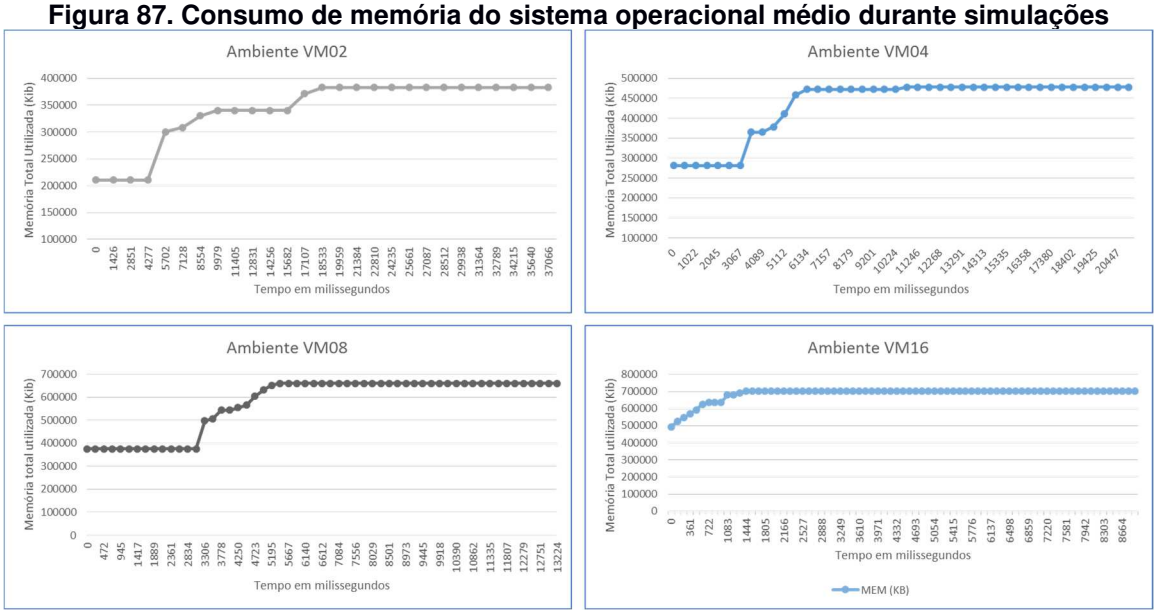


Fonte: Autoria própria

5.4.3 Consumo de memória

Como é possível observar na Figura 87, tal como no caso de estudo I, o consumo de memória total, ou seja, consumo de tarefas do sistema operacional somado ao consumo de memória da simulação também ficou longe do disponível mesmo na instância VM02 que possuía 8 GB de memória RAM. As simulações neste ambiente não ultrapassaram a 440 MB em nenhuma simulação neste ambiente. Portanto, tal como ocorreu no estudo de caso anterior, também não houve acesso à memória virtual (swap). Da mesma forma, então, a melhora no desempenho da simulação é, majoritariamente, em função do aumento do número de núcleos.

Figura 87. Consumo de memória do sistema operacional médio durante simulações

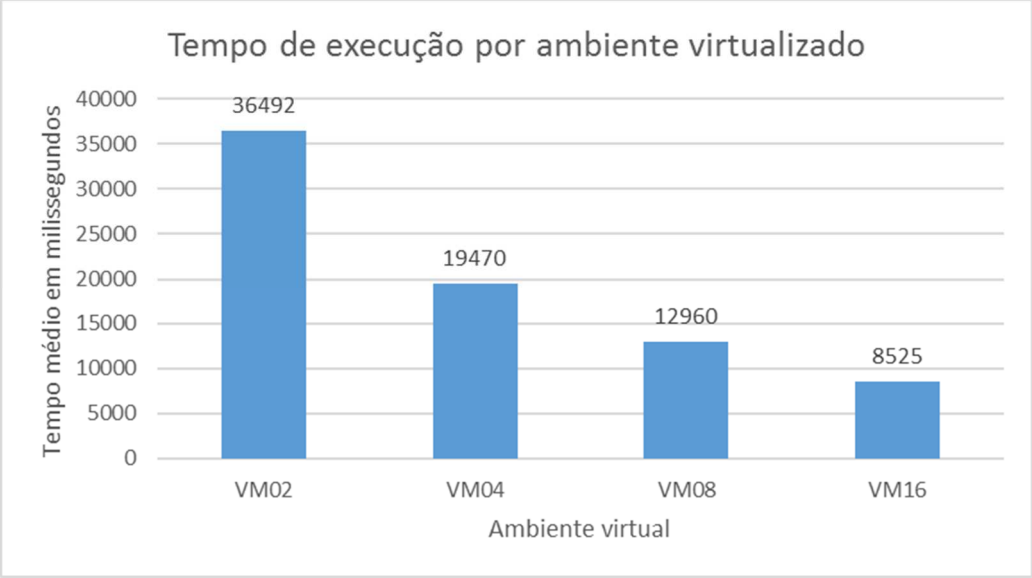


Fonte: Autoria própria

5.4.4 Considerações finais

Os resultados apresentados nas sessões anteriores demonstram a execução do programa NOPL materializado em um ambiente Erlang através do *Framework* NOP Elixir. A execução deste programa nos diversos ambientes cuja principal diferença é a disponibilidade de núcleos também apresenta uma expressiva redução conforme são acrescidos os núcleos. Fato este comprovado no gráfico da Figura 88 ao passo que, em todos os ambientes, a taxa de ocupação dos núcleos se manteve balanceada em todo o tempo de simulação. Estes dados sugerem que a execução lógico-causal foi distribuída entre os núcleos e com um bom nível de balanceamento.

Figura 88. Tempo médio de execução por ambiente virtualizado



Fonte: Autoria própria

5.5 COMPARATIVOS ENTRE OS ESTUDOS DE CASO

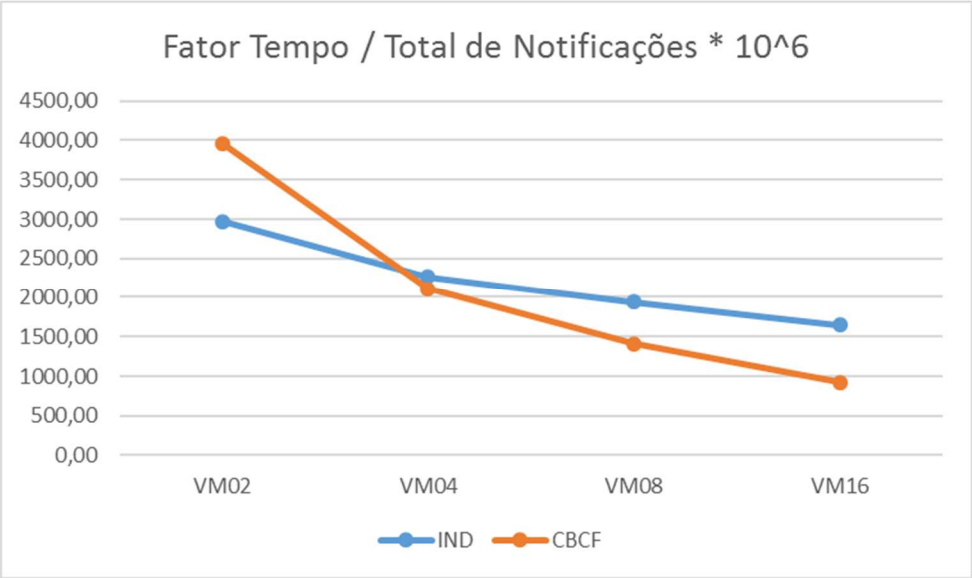
Nas sessões anteriores deste capítulo foram apresentados dois estudos distintos demonstrando melhoras no tempo de execução ao passo que sempre se mantinham os núcleos balanceados. Contudo, ao se cruzar as informações tabuladas entre os experimentos pode-se extrair algumas informações que sugerem que o NOPL Erlang-Elixir tem seu desempenho melhorado para algoritmos mais complexos conforme o aumento de núcleos como pode ser visto a seguir.

Como um indicador de desempenho, é usado um fator relacionado ao tempo por notificação. Este fator é definido como o tempo de execução em cada processador dividido pelo número total de notificações de cada estudo. Como o número resultante foi um número muito pequeno, foi criado um multiplicador no valor de 10^6 para que possa ser feita a devida comparação. O resultado da comparação entre os fatores de cada caso de estudo pode ser visto na Figura 89.

Como é possível verificar, a estratégia baseada em congestionamento facilitado (CBCF) obteve um fator inicial visivelmente superior à estratégia independente (IND), ou seja, gastou mais tempo para processar cada notificação considerando mesmo ambiente virtual. Porém, à medida que o ambiente virtual dispõe de mais núcleos, o desempenho da estratégia CBCF melhora a ponto de ser ligeiramente mais rápida

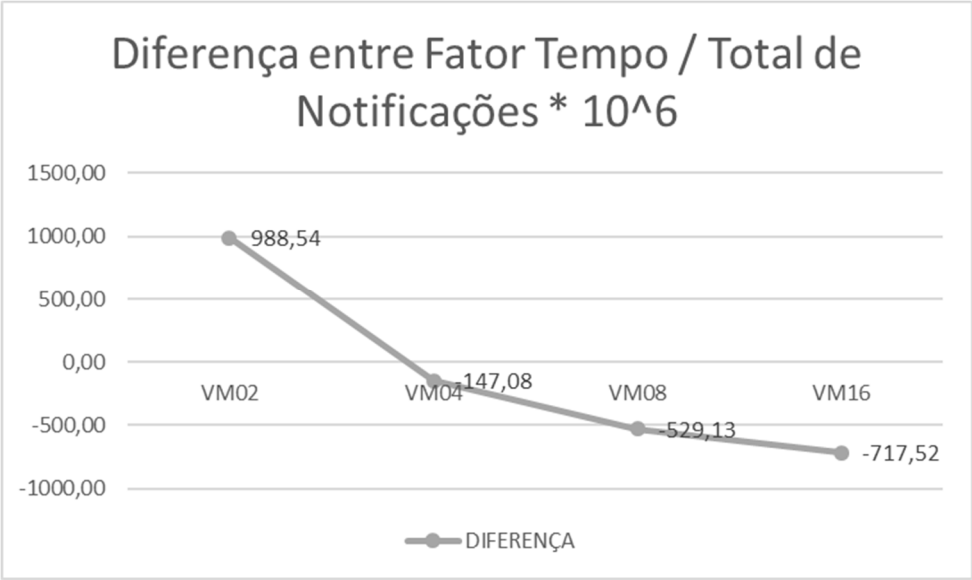
que a estratégia IND no ambiente VM04 (quatro núcleos). Esta diferença aumenta à medida que se disponibilizam mais núcleos. Este é mais bem evidenciado na Figura 90, esta apresenta apenas a diferença entre os respectivos fatores.

Figura 89. Comparativo de fator tempo sobre total de notificações



Fonte: Autoria própria

Figura 90. Diferença de fator tempo sobre total de notificações

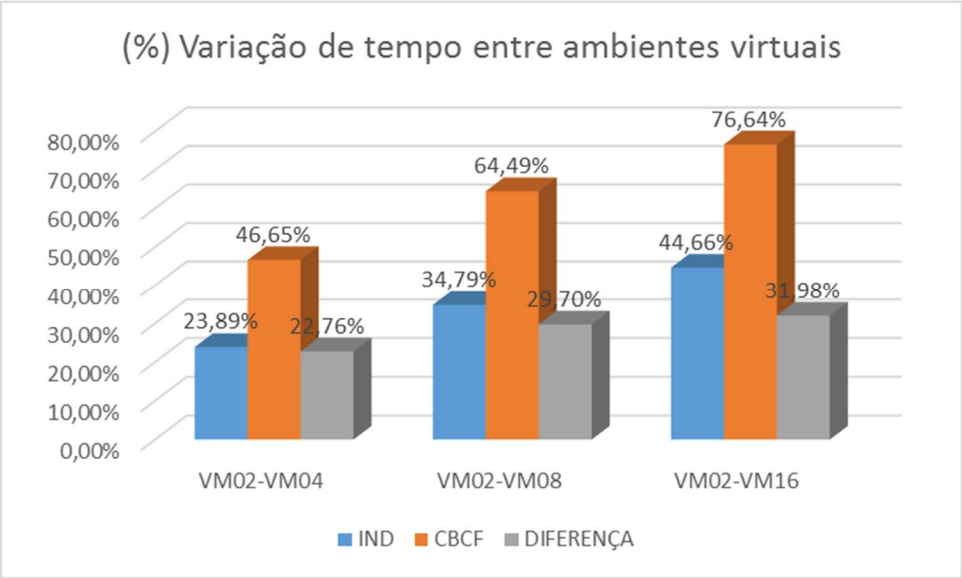


Fonte: Autoria própria

Outra informação que reforça este fato é a variação de ganho percentual de tempo de cada estudo de caso total entre o ambiente VM02 (dois núcleos) e o ambiente VM16 (dezesseis núcleos). Enquanto a estratégia IND obteve uma melhora

total de 44,66% entre estes ambientes, a estratégia CBCF conseguiu uma melhora de 76,64%. A Figura 91 apresenta um comparativo mais completo das melhoras de tempo bem como a diferença entre cada uma delas.

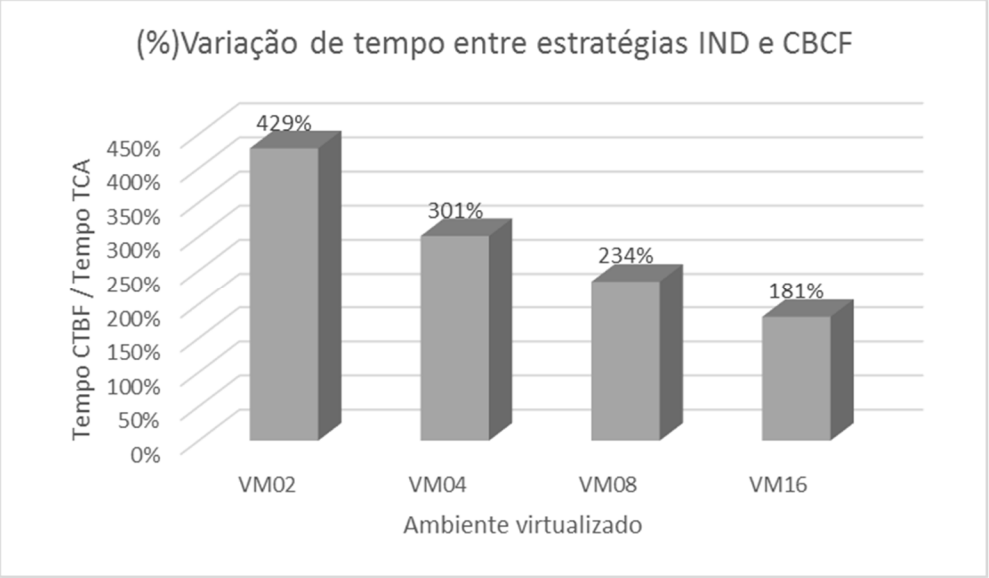
Figura 91. Comparativo de variação dos estudos entre ambientes virtuais



Fonte: Autoria própria

Uma última informação que corrobora a relação positiva entre desempenho e complexidade com o aumento de núcleos é a criação de um fator entre o tempo total da estratégia CBCF com a estratégia IND para cada ambiente. Este fator demonstra o quanto a estratégia CBCF foi mais demorada em relação à estratégia IND em cada ambiente. Ao se fazer este comparativo, percebe-se que o fator diminui à medida que se aumentam os núcleos como é apresentado na Figura 92.

Figura 92. Percentual de variação de tempo da estratégia CBCF em função da estratégia CTA



Fonte: Autoria própria

Em ambos os experimentos é possível observar um crescimento da taxa ao início da execução. Este período de ativação durante o qual o escalonador da máquina virtual Erlang ainda não alcançou a máxima ocupação das filas de execução Erlang (*threads)*. É possível observar, portanto que o experimento CBCF se manteve com 81,8% de taxa de ocupação média após o período de ativação, enquanto o experimento CTA perfez uma taxa de ocupação média de 90,8% neste mesmo período.

Uma hipótese de o segundo experimento (CBCF) alcançar uma taxa de ocupação média superior à do primeiro experimento se deve ao fato deste segundo possuir um número superior de *Rules* independentes entre si. Estas, por sua vez, geram grupos de microatores de elementos lógicos (*Condition, Subcondition e Premise)* com menor interdependência. Desta forma, o escalonador da máquina virtual Erlang consegue ocupar melhor as filas de processamento em função de menor dependência entre os processos. Todas as tabelas e dados brutos desta seção estão disponíveis no APÊNDICE D.

5.6 CONSIDERAÇÕES FINAIS

O presente capítulo apresentou um comparativo de dois casos de estudo sendo executados em várias máquinas virtuais com diferentes números de núcleos. Estas diferentes execuções do mesmo software puderam comprovar o aproveitamento do ambiente *multicore* de forma transparente ao desenvolvedor. Ainda, algumas análises puderam ser feitas a partir de um comparativo entre os resultados dos dois casos de estudo.

Apesar de os casos de estudos se limitarem a uma comparação interna da tecnologia proposta, ou seja, demonstrar os avanços da tecnologia sem maiores comparativos com outras tecnologias existentes, alguns comparativos externos foram feitos durante a evolução desta tecnologia. Estes comparativos e seus resultados podem ser vistos nos APÊNDICE H e APÊNDICE I respectivamente para comparativos com uma versão monoprocessada e *multicore* respectivamente. Para ambos os comparativos, o *Framework* NOP Elixir obteve resultados inferiores às técnicas comparadas.

Não obstante, um comparativo com várias técnicas de desenvolvimento, a saber, C++ clássico, *Framework* LingPON C++ 2.0 e biblioteca *ReactiveX* apresentado em (NEGRINI, PORDEUS e SIMÃO, 2019). pode ser visto no APÊNDICE J. Neste comparativo é possível observar melhora considerável do *Framework* NOP Elixir frente às demais técnicas. Ainda, no APÊNDICE A é apresentado um comparativo à luz das mesmas métricas apresentadas nos casos de uso deste trabalho para o NeuroPON (SCHÜTZ, 2019). Trabalho apresentado por Fernando Schütz com tema de sua tese de doutorado. Neste trabalho é possível observar o *speedup* para oito núcleos, ou seja, S(8) de 15,7883.

Em tempo, o experimento detalhado na seção 5.3 e seus resultados foi apresentado no congresso Escola Regional de Alto Desempenho (ERAD-RJ) ocorrido no CEFET-RJ entre os dias 04 e 06 de setembro de 2019 (NEGRINI, RONSZCKA, *et al.*, 2019) recebendo o prêmio de melhor artigo do evento.

6 CONCLUSÕES E TRABALHOS FUTUROS

Neste capítulo são apresentadas as conclusões deste trabalho. Inicialmente, resume-se a pesquisa e seus desdobramentos. Em seguida, são detalhadas as principais contribuições e os resultados obtidos neste trabalho de mestrado. Por fim, são demostrados trabalhos futuros relacionados ao tema de pesquisa.

6.1 RESUMO DA PESQUISA

A arquitetura *multicore* teve seus primeiros exemplares na década de 1960, ainda sem muita expressividade, porém. Foi quando, no final da década de 1990, fabricantes de microprocessadores passaram a adotar esta arquitetura em seus produtos, pois viram nela uma alternativa à limitação física dos materiais que começavam a encontrar barreiras para o aumento da frequência de processamento (*clock*) (DEBENEDICTIS, 2017).

A arquitetura básica dos processadores *multicore* baseia-se na arquitetura SMP (*Symmetric MultiProcessors*). Nessa arquitetura os diferentes núcleos são capazes de executar fluxos de instruções de forma independente e compartilham uma área de memória centralizada (WANG e WANG, 2006). Entretanto, o incremento de núcleos de processamento tornaram o desenvolvimento de software ainda mais complexo, pois para o correto aproveitamento desta tecnologia, exigem-se softwares criados com sub-rotinas desacopladas para serem executadas nesta arquitetura (DEBENEDICTIS, 2017).

Tecnologias têm sido criadas a fim de diminuir este esforço de desenvolvimento de software concorrente e paralelizável. Neste âmbito, o presente trabalho de mestrado foi inicialmente apresentado o modelo de atores, proposto por Hewitt (1973) que, motivado pela perspectiva de máquinas de computação altamente paralelas, propôs um o modelo com troca de mensagens assíncronas aliada ao estado isolado de cada elemento (HEWITT, 1973) (HEWITT, 1977) (HEWITT e BAKER, 1977) (AGHA, 1985). Este modelo, então, serviu de base para tecnologias futuras. A Erlang é uma destas tecnologias, tema visitado neste trabalho de mestrado.

Erlang surgiu em meados da década de 1980 nos laboratórios de pesquisa da Ericsson. Seus idealizadores buscavam solução para problemas de telefonia, os quais

são altamente concorrentes por natureza. Sem tecnologia disponível à época que atendesse às necessidades em questão, criaram, então, a linguagem Erlang baseada em modelo de atores (CESARINI e THOMSON, 2009) (ARMSTRONG, 2007). Entretanto, apesar de Erlang se apresentar concorrente e paralelizável, Erlang ainda carrega a má reputação de ser uma linguagem de difícil programação. Na tentativa de suprir estas lacunas, surge a linguagem Elixir. Elixir é uma abordagem pragmática para programação funcional. Ela valoriza seus fundamentos funcionais e foca na produtividade do desenvolvedor (THOMAS, 2018). Todavia, nem Elixir, tão pouco Erlang conseguiu isentar os desenvolvedores de projetarem seus códigos pensando em concorrência.

Mesmo com os avanços tecnológicos até aqui apresentados em termos de linguagens de programação e balanceamento de fluxos de processos, os atuais paradigmas de programação ainda são uma grande lacuna em termos de programação *multicore*. Mais detalhadamente, tanto o PI com sua estrutura monolítica, prolixa e acoplada, quanto o PD com seu processo de inferência baseado em pesquisa sobre entidades passivas, geram fluxos de processamento não paralelizáveis e de tamanhos diferenciados. Neste sentido, mesmo que despendidos esforços técnicos para o desacoplamento, permitindo sua execução paralela, não alcançam um bom aproveitamento em termos de balanceamento em ambientes *multicore* (AMDAHL, 1967).

Motivado por esta lacuna deixada pelos paradigmas PI e PD, que afeta naturalmente as linguagens de programação delas decorrentes, é apresentado o Paradigma Orientado a Notificações (PON). PON possui desacoplamento explícito (ou acoplamento mínimo, conforme o ponto de vista) das entidades que compõem o modelo e mesmo de suas partes devido à organização baseada em notificações. Esta característica o viabiliza, dentre outros, para a construção de programas com execução paralela e/ou distribuída tão fina quanto a arquitetura computacional permitir (PETERS, 2012) (LINHARES, 2015) (BELMONTE, LINHARES, *et al.*, 2016) (BARRETO, VENDRAMIN e SIMÃO, 2018) (OLIVEIRA, ROTH, *et al.*, 2018) (SCHÜTZ, FABRO, *et al.*, 2018) (KERSCHBAUMER, LINHARES, *et al.*, 2018).

Visando demonstrar o PON, houve previamente implementações dele na forma de *frameworks* sobre outras linguagens de programação usuais orientando-as, portanto, a trabalhar por notificações (BANASZEWSKI, 2009) (VALENÇA, 2012). Subsequentemente, visando estabelecer o PON, houve naturalmente a necessidade

de desenvolvimento linguagem própria ao PON. Ocorreu, portanto, a definição da chamada tecnologia NOPL (Notification Oriented Programing Language). Esta pesquisa foi liderada pelo pesquisador A. F. Ronszcka, envolvendo desenvolvimento de linguagens e tecnologia de compiladores (FERREIRA, 2015) (RONSZCKA, 2019).

Nesta sua última versão, a Tecnologia NOPL apresenta um Método de Compilação do PON (MCPON) que comporta uma solução de compilação de linguagens próprias do PON em uma única estrutura de dados uniforme intermediária em formato de grafo de entidades notificantes – o Grafo PON. A partir do Grafo PON, por meio de navegações neste grafo, pode-se então compor compiladores para geração de código final para plataformas distintas (RONSZCKA, 2019).

Entretanto, os esforços na utilização da Tecnologia NOPL ainda não haviam contemplado a construção de compiladores no tocante à plataforma *multicore*. Mesmo os esforços prévios à Tecnologia NOPL no que ser refere à uma materialização *multicore* em *software* PON, no formato de um *Framework* PON C++ 3.0, não obteve resultados satisfatórios (SCHÜTZ, 2019). Entretanto, a pesquisa em questão permitiu verificar que o uso de PON em *multicore* era possível e, se bem articulado, promissor (BELMONTE, 2012) (BELMONTE, LINHARES, *et al.*, 2016).

À luz destes resultados insatisfatórios, porém com potencial de pesquisa em software PON *multicore*, refletiu-se sobre a hipótese destes resultados serem em função da implementação ter se dado em tecnologias antiquadas para esta arquitetura, além do *framework* dado não ter sido efetivamente findado conforme relata (SCHÜTZ, 2019). A partir deste ponto, refletiu-se sobre soluções e arquiteturas atuais que possuíam resultados comprovados em concorrência e paralelismo *multicore,* particularmente que permitissem implementação implícita do balanceamento dos elementos PON. Desta premissa, deriva-se, então este trabalho de mestrado que se apoia na linguagem Elixir sobre a plataforma Erlang como elemento de sinergia para o PON.

Neste contexto, a contribuição deste trabalho consiste primeiramente na elaboração de um *framework* de microatores do PON sobre a linguagem Elixir, o que foi chamado de *Framework* NOP Elixir. Este *framework* possibilitou a execução concorrente e paralela de *software* PON em ambiente *multicore* com balanceamento fino de carga entre os núcleos. Ainda, como contribuição suplementar, houve a elaboração e implementação de um compilador via tecnologia NOPL, que permitiu traduzir de linguagem de desenvolvimento do PON diretamente em *Framework* NOP

Elixir. Desta forma, alcançando o desenvolvimento para *multicore* em alto nível, sem preocupações com desacoplamentos, concorrência ou paralelismos. Esta tecnologia foi então nominada Tecnologia NOPL Erlang-Elixir.

6.2 PRINCIPAIS CONTRIBUIÇÕES E RESULTADOS

A modelagem dos elementos do paradigma PON em atores, permitiu um novo ponto de vista à luz da implementação de software PON. Esta modelagem também demonstrou a forte sinergia entre estas abordagens, as quais citou-se principalmente a natureza holônica e o método de comunicação. Esta sinergia permitiu, então, a tradução de elemento do PON em ator, conforme o desacoplamento implícito de suas entidades. Entretanto, estes atores resultantes desta modelagem à luz do PON, apresentaram características exclusivas como a decomposição tanto dos elementos fato-execucionais, quanto dos fluxos lógicos-causais em pequenas partes. Por este motivo, deu-se a definição de microatores. Esta modelagem do PON e seus elementos à luz do modelo de atores permitiu intuir a viabilidade de uma implementação subsequente de PON em arquiteturas que suportariam atores, como Erlang/Elixir e mesmo outros como Akka e Go.

Partindo-se da modelagem de microatores, ainda assaz abstrata, viu-se a necessidade de uma modelagem mais completa, a fim de servir de base documental para a construção do *Framework* NOP Elixir. Desta necessidade, e orientada pela modelagem de atores, foi proposta a modelagem UML. Portanto, a modelagem UML foi um avanço sobre à modelagem de atores à luz da Engenharia de *Software*. A modelagem UML permitiu expressar os elementos de forma mais completa e detalhada, tornando-se a documentação base completa para a construção do *Framework* NOP Elixir.

O *Framework* NOP Elixir foi concebido desde o início com o objetivo principal de reproduzir o comportamento de software PON em Elixir. Entretanto, várias ferramentas adicionais foram desenvolvidas e utilizadas visando apoiar o ambiente de pesquisa e/ou depuração. No *Framework* NOP Elixir todos os eventos relativos ao comportamento de software PON são logados de forma a facilitar o acompanhamento do fluxo de notificações por parte do desenvolvedor. Ainda, ferramentas estatísticas como tempo de execução do programa e contagem de notificações foram incluídas

durante o desenvolvimento. Por fim, a fácil distribuição do *Framework* é garantida por meio de pacotes HEX Erlang.

O compilador NOPL Erlang-Elixir, por sua vez, faz a consolidação de todas as contribuições expostas individualmente, promovendo a integração da tecnologia NOPL, a partir do método de compilação MCPON e o Grafo PON, à plataforma concorrente paralela Erlang. Esta solução como um todo foi então nominada de Tecnologia NOPL Erlang-Elixir.

A partir da Tecnologia NOPL Erlang-Elixir elaborada como contribuição maior deste trabalho de pesquisa enquanto conjunto harmônico de artefatos, foi elaborado um conjunto de estudos de caso. Nos casos de estudo apresentados, foram feitas execuções em ambientes que variaram basicamente pelo número de núcleos. Nos resultados foi possível verificar uma expressiva redução do tempo de execução conforme o número de núcleos aumentou, ao passo que, em todos os ambientes, os núcleos se mantiveram com um bom nível de balanceamento. Estes resultados demonstraram a execução concorrente paralela esperada como proposto no objetivo inicial.

Uma análise comparativa mais detalhada entre os casos de estudo também permitiu observar que, entre os casos estudados, houve uma relação positiva entre desempenho e o incremento de complexidade lógico-causal. Em outras palavras, quando se disponibilizaram mais núcleos para a arquitetura proposta neste trabalho, melhor foi o desempenho do algoritmo PON com mais exemplares de microatores lógico-causais (*Rule, Condition, Subcondition* e *Premise)* comparativamente ao algoritmo PON com menos representantes desta classe de microatores. Este fato foi observado nos casos estudados verificando-se fatores de melhora sensivelmente maiores para o segundo caso de estudo cuja lógica-causal é mais complexa. Esta análise sugeriu que a arquitetura tende a melhorar o uso dos núcleos conforme aumentam os números de microatores lógico-causais independentes entre si, até que alcancem um ponto de saturação ainda não atingido nestes experimentos. Neste quadro dado, o escalonador Erlang conseguiu trabalhar com mais fluidez nos processos e aumentar a carga nas filas de processamento da máquina virtual.

Os experimentos apresentados neste trabalho demonstraram que o paradigma PON possui potencial para aproveitamento em arquiteturas *multicore* em virtude da natureza desacoplante de seus elementos. Esta afirmação é sustentada por dois resultados apresentados neste trabalho. O primeiro resultado foi a concorrência e o

paralelismo transparente para desenvolvedor, alcançados com a tradução dos elementos PON em microatores e posterior desenvolvimento da Tecnologia NOPL Erlang-Elixir. O segundo resultado foi o bom nível de balanceamento e elevada carga de ocupação dos núcleos alcançado nas execuções, em função de cada microator carregar apenas um pedaço pequeno (e consequentemente leve) do fluxo de processamento. Isto posto, conforme apresentado na lei de Amdahl (1967), softwares PON possuem potencial para grande aumento de *speedup* quando dispostos em ambientes *multicore*.

Como última reflexão a este trabalho de mestrado, uma contribuição suplementar se deu em virtude da forma de como as contribuições foram apresentadas. Neste formato, permitir-se-á, em trabalhos complementares futuros, evoluções e contribuições de forma independente. Como exemplo, enquanto uma equipe se ocupa de corrigir ou ampliar alguma interpretação do compilador NOPL Erlang-Elixir, outra equipe pode atuar otimizando o *Framework* NOP Elixir. De outra forma, uma terceira equipe pode aproveitar a modelagem de atores e a modelagem UML para construir software PON em uma tecnologia baseada em atores ainda não explorada.

6.3 TRABALHOS FUTUROS

Esta seção apresenta por meio de tópicos um conjunto de trabalhos futuros vislumbrados a partir das contribuições trazidas pelos esforços de pesquisa do presente trabalho.

Materialização PON em manycore

Conforme apresentado na seção 3.3, a natureza desacoplante do PON tem potencial para produção de software altamente granularizável e paralelizável. Estes atributos corroboram com os tipos de software que são favorecidos em arquiteturas *manycore.* Isto posto, materializações neste tipo de arquitetura possuem potencial para ganhos em termos de velocidade de processamento.

Framework NOP Elixir Distribuído

Como a materialização proposta neste trabalho está ancorada na arquitetura Erlang, vários recursos deste não foram explorados neste trabalho. Um destes recursos é a facilidade de distribuição dos processos entre máquinas distintas, ou mais precisamente, processamento distribuído. De fato, em Erlang, suporte a distribuição entre máquinas é anterior ao suporte *multicore.* Portanto, a união de distribuição e aproveitamento *multicore* em uma mesma arquitetura se faz pertinente e plausível.

Tratamento de *Attributes* compostos

O *Framework* NOP Elixir já foi desenvolvido pensando em comunicação entre microatores *FBE*, ou seja, está preparado para a relação de agregação entre estes elementos. Contudo esta funcionalidade não foi aplicada ao compilador NOPL Erlang-Elixir. Portanto um trabalho futuro é a integração do compilador para esta funcionalidade do *Framework* NOP Elixir.

Testes com mais variações de núcleos

Todos os exercícios apresentados nos casos de estudo foram feitos em máquinas virtuais que variavam entre um e oito núcleos. Contudo, mais testes se fazem pertinentes com a adição de mais núcleos a fim de se observar o comportamento da tecnologia, quiçá até um ponto de saturação que o paralelismo em PON permitiria. Dito de outra forma, tais testes servirão para verificar se a relação positiva entre complexidade e eficiência é preservada e, se sim, até quando.

Resolução de indeterminismos

O trabalho apresentado não fez nenhum controle à luz do indeterminismo, inerente a processos paralelos. Estes controles são propostos por Simão et ali em (SIMÃO e STADZISTZ, 2010). Por este motivo, um aprimoramento no *Framework*

NOP Elixir para incorporar estes controles se apresenta como um trabalho desejável e importante por definição.

REFERÊNCIAS

AGHA, G. A. **Actors:** A Model of concurrente computation in distributed systems. Cambridge, Massachusetts: MIT - Artificial inteligence laboratory, 1985. 198 p.

AHO, A. V. et al. **Compiladores:** Princípios, técnicas e ferramentas. [S.l.]: Addison Wesley, 2008.

AMAZON. Instâncias dedicadas do Amazon EC2, 23 abr. 2019. Disponivel em: <https://docs.aws.amazon.com/pt_br/AWSEC2/latest/UserGuide/dedicated-instance.html>. Acesso em: 19 nov. 2019.

AMAZON. Instâncias M5 do Amazon EC2. **Instâncias M5 do Amazon EC2**, 2019a. Disponivel em: <https://aws.amazon.com/pt/ec2/instance-types/m5/>. Acesso em: 05 dez. 2019.

AMD. AMD Ryzen™ Threadripper™ 2990WX Processor. **AMD**, 26 maio 2019. Disponivel em: <https://www.amd.com/pt/products/cpu/amd-ryzen-threadripper-2990wx>. Acesso em: 26 maio 2019.

AMDAHL, G. M. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. **AFIPS Conference Proceedings**, 1967. 483–485.

ARMSTRONG, J. A History of Erlang. **Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages**, San Diego, California, 2007. 6-1--6-26.

ARMSTRONG, J. **Programming Erlang:** Software for a Concurrent World. Raleigh, NC, USA: Pragmatic Bookshelf, 2013.

ATHAYDE, E. B.; NEGRINI, F. Implementação de Compilação para C++ Namespaces para a LingPON e Otimizações no Tratamento de Premissas. **Trabalho realizado na disciplina Tópicos Avançados em Engenharia de Software (CAES101 – PPGCA/UTFPR) publicado no Anexo E da Dissertação de Mestrado de Leonardo Araújo Santos**, Curitiba, Brasil, 2016.

BANASZEWSKI, R. **Paradigma Orientado a Notificações:** Avanços e Comparações. Curitiba: Dissertação (Mestrado em Engenharia Elétrica e Informática Industrial): Universidade Tecnológica Federal do Paraná, 2009.

BANASZEWSKI, R. F. et al. Notification Oriented Paradigm (NOP) - A Software Development Approach based on Artificial Intelligence Concepts. **VI Congress of Logic Applied to Technology – LAPTEC 2007**, Santos, 2007.

BARRETO, R. M. W. **Notification Oriented Paradigm in the Context of Distributed Systems**. Curitiba, Brasil: Relatório da disciplina de Tópicos Especiais Em Paradigma Orientado A Notificações. CPGEI-PPGCA/UTFPR, 2016.

BARRETO, W. R. M.; VENDRAMIN, A. C. B. K.; SIMÃO, J. M. Notification Oriented Paradigm for Distributed Systems. **Computer on the Beach 2018**, Florianópolis - SC, 2018.

BAUDE, F.; VIDAL-NAQUET, G. Actors as a parallel programming model. **Choffrut C., Jantzen M. (eds) STACS 91**, Springer, Berlin, Heidelberg, 480, 1991.

BAUTSCH, M. Cycles of Software Crises. **ENISA Quarterly on Secure Software**, 3, n. 4, 2007. 3-5.

BELMONTE, D. **Método para Distribuição da Carga de Trabalho dos Softwares PON em Multicore**. Curitiba, Brasil: Trabalho de Qualificação de Doutorado, CPGEI, UTFPR, 2012.

BELMONTE, D. et al. A new Method for Dynamic Balancing of Workload and Scalability in Multicore Systems. **IEEE Latin America Transactions**, 2016.

BLAKE, G.; DRESLINSKI, R. G.; MUDGE, T. A survey or multicore processors. **Signal Processing Magazine, IEEE**, 26, n. October/2009, Outubro 2009. 26-37.

BROOKSHEAR, G. **Computer Science:** An Overview. [S.l.]: Addison Wesley, 2012.

BRYAN, S. **Multicore Processors – A Necessity**. [S.l.]: ProQuest, 2008.

CESARINI, F.; THOMSON, S. **Erlang Programing**. Cambridge: O'Reilly, 2009.

CHENG, A. M. K.; CHEN, J. R. Response Time Analysis of OPS5 Production Systems. **IEEE Transactions on Knowledge and Data Engineering**, 12, n. 3, 2000. 391-409.

CLINGER, W. **Foundations of Actor Semantics**. [S.l.]: Mathematics Doctoral Dissertation. MIT, 1981.

DEBENEDICTIS, E. P. It's Time to Redefine Moore's Law Again, 50, 2017.

DIJKSTRA, E. The Humble Programmer. **Communications of the ACM**, 1972. 859–866.

DIJKSTRA, E. W. Cooperating sequential processes. **Technological University**, 1965.

FERREIRA, C. A. **Linguagem e compilador para o paradigma orientado a notificações (PON):** Avanços e comparações. Curitiba, Brasil: Dissertação de Mestrado, PPGCA - UTFPR., 2015.

FORGY, C. RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, 19, 1982. 17-37.

GABBRIELLI, M.; MARTINI, S. **Programming Languages:** Principles and Paradigms. 1st ed. [S.l.]: Springer Publishing Company, Incorporated, 2010.

GARCIA, J. A. et al. A Comparative Performance Evaluation of Different Implementations of the SOAP Protocol. **Fifth European Conference on Web Services**, 2007. 109-118.

GREIF, I. **Semantics of Communicating Parallel Processes**. [S.l.]: EECS Doctoral Dissertation. MIT, 1975.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture, Fifth Edition:** A Quantitative Approach. [S.l.]: Morgan Kaufmann Publishers Inc, 2011.

HEWITT, C. . B. P. . S. R. **A Universal Modular Actor Formalism for Artificial Intelligence**. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc, 1973.

HEWITT, C. **Viewing Control Structures as Patterns of Passing Messages**. [S.l.]: Journal of Artificial Intelligence, 1977.

HEWITT, C.; BAKER, H. **Laws for Communicating Parallel Processes**. Toronto, Canada: IFIP, 1977.

HEXDOCS (APPLICATION). Application (Hexdocs). **Hexdocs**, 24 jan. 2019. Disponivel em: <https://hexdocs.pm/elixir/Application.html>. Acesso em: 24 jan. 2019.

HUGHES, C. **Parallel and Distributed Programming Using C++.** Boston: Addison-Wesley, 2003.

IMAM, S.; SARKAR, V. Selectors: Actors with Multiple Guarded Mailboxes. **Proceedings of the 4th International Workshop on Programming Based on Actors Agents - Decentralized Control**, New York, NY, USA, 2014. 1-14.

JASINSKI, R. P. **Framework para Geração de Hardware em VHDL a Partir de Modelos em PON (Paradigma Orientado a Notificações)**. UTFPR. Curitiba. 2012.

KERSCHBAUMER, R. et al. **Paradigma Orientado a Notificações para a Síntese de Lógica Reconfigurável**. Curitiba, Paraná, Brasil: LA-CCI/CBIC, 2015.

KERSCHBAUMER, R. et al. Notification Oriented Paradigm to Implement Digital Hardware. **Journal of Circuits Systems and Computers**, 2018.

KOSCIANSKI, A. E. A. FMS Design and Analysis: Developing a Simulation Env. **XV International Conference on CAD/CAM: Robotics and Factories of the Future,** 1999.

LEE, P.-Y.; CHENG, A. M. K. HAL: a faster match algorithm. **IEEE Transactions on Knowledge and Data Engineering**, 5, 2002. 1047-1058.

LINHARES, R. R. **Contribuição para o desenvolvimento de uma arquitetura de computação própria ao paradigma orientado a notificações**. Brasil. 2015.

LINHARES, R. R. et al. Comparações entre o Paradigma Orientado a Objetos e o Paradigma Orientado a Notificações sob o contexto de um simulador de sistema telefônico. **III Congreso Internacional de Computación y Telecomunicaciones**, Lima, Perú, 2011. 103-106.

LOVE, R. **Linux Kernel Development**. Indianapolis: Sams, 2004.

MATTSON, T. **The Future of Many Core Computing:** A tale of two processors. [S.l.]: Intel Labs, 2010.

MAZIERO, C. **Sistemas Operacionais**. [S.l.]: DINF-UFPR, 2017. Disponivel em: <http://wiki.inf.ufpr.br/maziero/lib/exe/fetch.php?media=so:so-livro.pdf>. Acesso em: 03 jan. 2019.

NEGRINI, F. et al. NOPL-Erlang: Programação multicore transparente em linguagem de alto nível. **ERAD-RJ - Escola Regional de Alto Desempenho**, Rio de Janeiro, 04 set. 2019.

NEGRINI, F.; PORDEUS, L.; SIMÃO, J. M. Linguagem do Paradigma Orientado a Notificações: comparativos via simulador de tráfego. **XLI International Sodebras Congress**, Maceió, Brasil, 15, n. 169, Novembro 2019.

OLIVEIRA, R. N. **Uso do Paradigma Orientado a Notificações em Sistemas Sencientes**. Curitiba, Brasil: Relatório da disciplina de Tópicos Especiais Em Ec: Paradigma Orientado A Notificações. CPGEI-PPGCA/UTFPR, 2016.

OLIVEIRA, R. N. et al. Notification Oriented Paradigm Applied to Ambient Assisted Living Tool. **IEEE Latin America Transactions**, 2018.

PETERS, E. **Coprocessador para Aceleração de Aplicações Desenvolvidas Utilizando Paradigma Orientado a Notificações**. Curitiba, Brasil: Dissertação de Mestrado. CPGEI, UTFPR, 2012.

PETERSON, G. L. Myths About the Mutual Exclusion Problem. **Information Processing Letters**, 3, 1981. 115-116.

PILLA, M.; SANTOS, R.; CAVALHEIRO, G. **Introdução à programação em arquiteturas**. Porto Alegre: In: Anais da IX Escola Regional de Alto Desempenho, 2009. 73-102 p.

PORDEUS, L. F. **SIMULAÇÃO DE UMA ARQUITETURA DE COMPUTAÇÃO PRÓPRIA AO PARADIGMA ORIENTADO A NOTIFICAÇÕES**. Curitiba: UTFPR, 2017.

PORDEUS, L. F. et al. **NOTIFICATION ORIENTED PARADIGM TO DIGITAL HARDWARE**. [S.l.]: Revista SODEBRAS, v. 11, 2016. 116-122 p.

REACTIVEX. Reactive Extensions for C++. **Reactive Extensions for C++**, 2019. Disponivel em: <https://github.com/Reactive-Extensions/RxCpp>. Acesso em: 22 set. 2019.

RENAUX, D. P. B. et al. CONOPS. **CTA CONOPS**, 2015. Disponivel em: <http://www.dainf.ct.utfpr.edu.br/~douglas/CTA_CONOPS.pdf>. Acesso em: 01 set. 2019.

RONSZCKA, A. **Contribuição para a Concepção de Aplicações no Paradigma Orientado a Notificações (PON) sob o viés de Padrões**. Curitiba: Dissertação (Mestrado em Engenharia Elétrica e Informática Industrial): Universidade Tecnológica Federal do Paraná, 2012.

RONSZCKA, A. F. **Método para a criação de linguagens e compiladores voltados a plataformas distintas baseado no PON**. Curitiba: UTFPR, 2019.

RONSZCKA, A. F. et al. Notification-Oriented and Rete Network Inference: A Comparative Study. **IEEE International Conference**, n. Systems, Man, and Cybernetics (SMC), 2015. 807–814.

RONSZCKA, A. F. et al. Notification-Oriented Programming Language and Compiler. **SBESC – VII - Brazilian Symposium on Computing Systems Engineering**, 2017.

RUSSINOVICH, M.; SOLOMON, D. **Windows Internals:** Covering Windows Server 2008 and Windows Vista. New York: Microsoft Press, 2008.

SATISH, N. et al. Can traditional programming bridge the ninja performance gap for parallel computing applications? **ACMSIGARCH Computer Architecture News**, v. 40, p. 440–451, 2012.

SAVAGE, J. E.; ZUBAIR, M. A unified model for multicore architectures. **IFMT '08**, New York, 2008. 1-12.

SCHÜTZ, F. **NeuroPON:** Uma abordagem para o desenvolvimento de Redes Neurais Artificiais utilizando o Paradigma Orientado a Notificações. Texto de tese (Doutorado em Engenharia Elétrica e Informática Industrial). ed. Curitiba - Brasil: Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI), 2019. 271 p.

SCHÜTZ, F. et al. Proposal of a declarative and parallelizable artificial neural network using the notification-oriented paradigm. **Neural Computing and Applications**, 2018. 1-12.

SCOTT, M. L. **Programming Language Pragmatics**. [S.l.]: Elsevier Science & Technology, 2016.

SIMÃO, J. **Proposta de uma Arquitetura de Controle para Sistemas Flexíveis de Manufatura Baseada em Regras e Agentes**. Curitiba: Dissertação (Mestrado em Engenharia Elétrica e Informática Industrial): Universidade Tecnológica Federal do Paraná, 2001.

SIMÃO, J. **A Contribution to the Development of a HMS (Holonic Manufacturing System) Simulation Tool and Proposition of a Meta-Model for Holonic Control**. Curitiba: Tese (Doutorado em Engenharia Elétrica e Informática Industrial): Universidade Tecnológica Federal do Paraná, 2005.

SIMÃO, J. E. A. Holonic Manufacturing Execution Systems for Customised and Agile Production: Manufacturing Plant Simulation, Belo Horizonte, 2008.

SIMÃO, J. E. A. **Notification Oriented and Object Oriented Paradigms comparison via Sale System.** [S.l.]: Journal of Software Engineering and Applications, 2012. 5: 695-710.

SIMÃO, J. E. A. **Comparações entre duas materializações do Paradigma Orientado a Notificações (PON):** Framework PON Prototipal versus Framework PON Primário. Lima: In: IV, 2012a.

SIMÃO, J. M. Consolidação do Paradigma Orientado a Notificações (PON). **Chamada Pública 15/2017. Programa de Bolsas de Produtividade em Pesquisa e Desenvolvimento Tecnológico / Extensão. Projeto Aprovado. Fundação Araucária**, 2018.

SIMÃO, J. M. et al. Notification Oriented Paradigm (NOP) and Imperative Paradigm: A Comparative Study. **Journal of Software Engineering and Applications**, 2012. 5: 402-416, 2012.

SIMÃO, J. M. et al. Notification Oriented Paradigm (NOP) and Imperative Paradigm : A Comparative Study. **Journal of Software Engineering and Applications**, 6, n. 5, 2012b. 402-416.

SIMÃO, J. M. et al. **Framework PON C++ 1.0**. BR512017001015-3, 03 ago. 2017a.

SIMÃO, J. M. et al. **Framework PON C++ 2.0.** BR51201700149-5, 01 dez. 2017b.

SIMÃO, J. M.; STADZISTZ, P. C. Mecanismo de resolução de conflito e garantia de determinismo para o paradigma orientado a notificações (PON), 2010.

SIMÃO, J. M.; STADZISZ, P. C. **Paradigma Orientado a Notificações (PON) – Uma Técnica de Composição e Execução de Software Orientado a Notificações**. Nº INPI PI08055181, 2008. Pedido de Patente submetida ao INPI/Brasil (Instituto Nacional de Propriedade Industrial) em 2008 e a Agência de Inovação/UTFPR em 2007.

SIMÃO, J. M.; STADZISZ, P. C. Inference Based on Notifications: A Holonic Metamodel Applied to Control Issues. **IEEE Transactions on Systems, Man and Cybernetics. Part A, Systems and Humans**, 39, 2009. 238-250. 39 (1): 238-250, 2009.

SIMÃO, J.; STADZISZ, P. An Agent-Oriented Inference Engine applied for Supervisory Control of Automated Manufacturing Systems, Amsterdam, 2002. 234-241.

SIMÃO, J.; STADZISZ, P. **Paradigma Orientado a Notificações (PON):** Uma Técnica de Composição e Execução de Software Orientado a Notificações. Brasil: Patente submetida ao INPI (Instituto Nacional de Propriedade Industrial). Nº Provisório: 015080004262, 2008.

SIMÃO, J.; STADZISZ, P. Inference Based on Notifications: A Holonic Metamodel Applied to Control Issues. **IEEE Transactions on Systems, Man and Cybernetics**, 39, 2009. 238-250. 39 (1): 238-250, 2009.

SIMÃO, J.; STADZISZ, P.; MOREL, G. Manufacturing Execution System for Customized Production, Amsterdam, 2006.

STROUSTRUP, B. **The C ++ Programming.** [S.l.]: Addison-Welsey, 2013.

TALAU, M. **PONIP:** Uso do Paradigma Orientado a Notificações em Redes IP. Curitiba, Brasil: Relatório da disciplina de Tópicos Especiais Em Ec: Paradigma Orientado a Notificações. CPGEI-PPGCA/UTFPR, 2016.

TANENBAUM, A. S. **Sistemas Operacionais Modernos - 3a Edição**. São Paulo. 2010. (978-85-7605-237-1).

TANENBAUM, A.; WOODHULL, A. **Operating Systems Design and Implementation**. New Jersey: Prentice Hall, 2006.

THOMAS, D. **Programing Elixir**. [S.l.]: The Pragmatic Programers, 2018. ISBN 978-1-68050-299-2.

VAJDA, A. **Programming Many-Core Chips**. New York: Springer, 2011. ISBN 978-1-4419-9739-5.

VALENÇA, G. **Contribuição para a Materialização do Paradigma Orientado a Notificações(PON)**. Curitiba: Dissertação (Mestrado em Computação Aplicada): Universidade Tecnológica Federal do Paraná, 2012.

VAN ROY, P. **Concepts, Techniques, and Models of Computer Programming**. London, England: The MIT Press, 2004. ISBN 0-262-22069-5.

VAN ROY, P. **Programming Paradigms for Dummies:** What Every Programmer Should Know. [S.l.]: New Computational Paradigms for Computer Music, 2009.

VINOSKI, S. Concurrency with Erlang. **IEEE Internet Computing**, 11, n. 5, 2007. 90-93.

WANG, S.; WANG, L. Thread-Associative Memory for Multicore and Multithreaded Computing. **Proceedings of the International Symposium on Low Power Electronics and Design**, Tegernsee, 2006. 139-142.

WATT, D. A. **Programming Language Design Concepts**. [S.l.]: J. Willey & Sons, 2004.

WEBER, L. E. A. **Viabilidade de Controle Orientado a Notificações (CON) em ambiente concorrente baseado em threads**. Cornélio Procópio: [s.n.], 2010.

WITT, F. A. et al. **Comparação entre o Paradigma Orientado a Objetos (POO) e o Paradigma Orientado a Notificações (PON) em um Controle Discreto em Lógica Reconfigurável**. [S.l.]: XVI SICITE - Seminário de Iniciação Científica e Tecnológica da UTFPR, 2011.

APÊNDICE A - Estudo de caso NeuroPON

NeuroPON foi apresentado por Schütz (2019) como uma abordagem para o desenvolvimento de Redes Neurais Artificiais (RNA) utilizando o Paradigma Orientado a Notificações (PON). Este experimento, inicialmente apresentado em materialização via hardware, teve uma segunda compilação para a arquitetura proposta neste trabalho. Esta materialização permitiu, então, a plausibilidade deste modelo em NOPL Erlang-Elixir. Importante reforçar que não houve uma formalização e projeto teóricos para a execução, visto que o projeto do *Framework* se encontrava em suas primeiras versões quando da execução destas simulações.

O código do experimento NeuroPON foi de uma RNA MLP para a função XOR. Em boa hora, salienta-se que esse código foi escrito espelhando-se em (i.e., traduzindo de) outros códigos em NeuroPON previamente elaborados por Schütz. Após estas traduções e ajustes, o programa resultante foi testado nas mesmas máquinas utilizadas nos casos de estudo apresentados neste trabalho (vide subseção 5.2.1).

Tabela 14. Resultados da execução do experimento NeuroPON de uma RNA MLP para a função XOR em Erlang

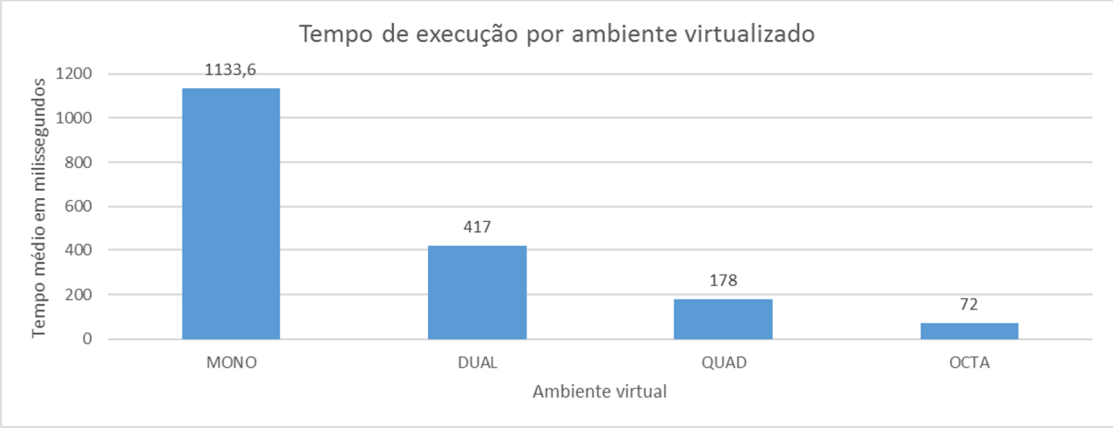
	Média	SPEEDUP	Diferença com MONO	Diferença com anterior	Execução 1	Execução 2	Execução 3	Execução 4	Execução 5
MONO	1133,6		0,00%		473	593	1684	1804	1114
DUAL	417	2,7159	63,18%	63,18%	50	612	489	580	356
QUAD	178	6,3757	84,32%	57,40%	190	160	158	235	146
OCTA	72	15,7883	93,67%	59,62%	103	37	84	62	73

Fonte: (SCHÜTZ, 2019)

Conforme apresentado na Tabela 14, foram efetuadas trinta execuções do código, em cada uma das máquinas. A segunda coluna apresenta a média do tempo de execução em milissegundos. Pode-se observar, portanto, uma queda de, aproximadamente, 63% no tempo de execução entre a máquina monoprocessada (MONO) e a máquina de dois núcleos (DUAL), de aproximadamente 84% entre a máquina de dois núcleos (DUAL) e a máquina de quatro núcleos (QUAD), e de

aproximadamente 94% entre a máquina de quatro núcleos (QUAD) e a máquina de oito núcleos (OCTA). A Figura 93 apresenta o gráfico destas médias de tempo de processamento.

Figura 93. Representação gráfica dos tempos médios de execução NeuroPON de uma RNA MLP para a função XOR em Erlang



Fonte: (SCHÜTZ, 2019)

APÊNDICE B - Tabulação de resultados do caso de estudo I

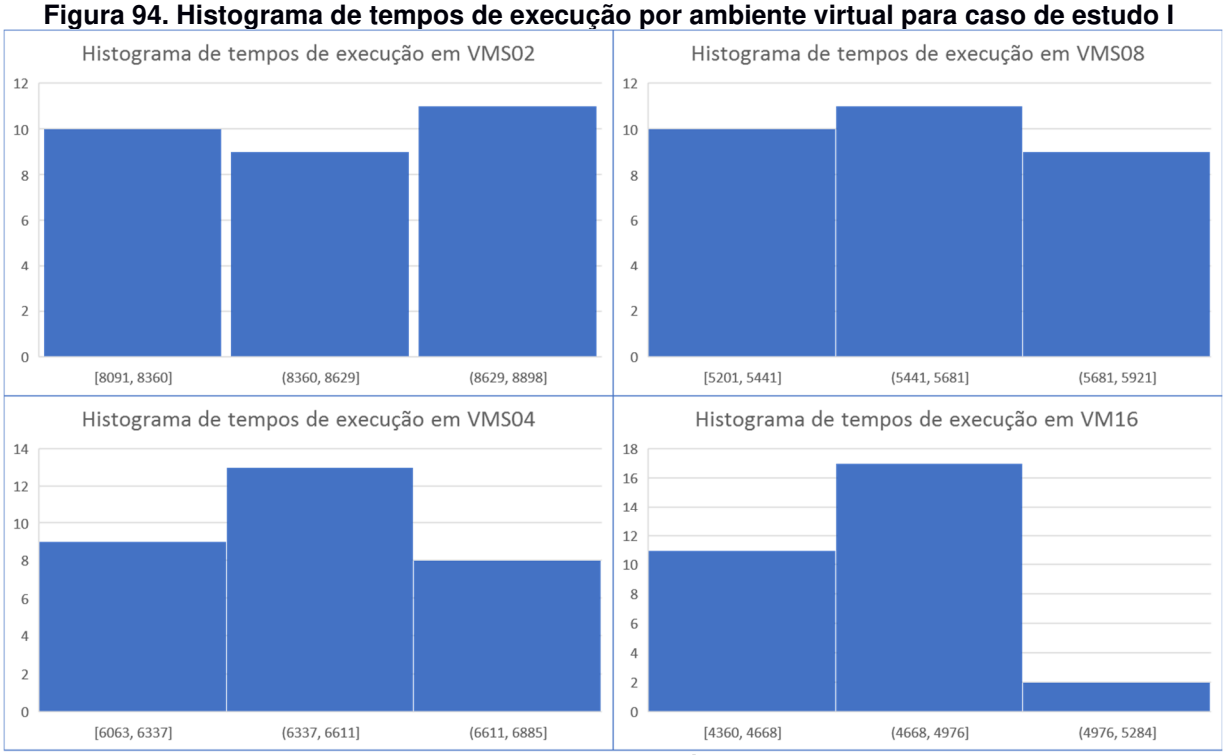
A seguir são apresentados na Tabela 15 os valores recuperados do caso de estudo apresentado na seção 5.3. A Tabela 15 apresenta, respectivamente, os tempos e o número de notificações de cada execução para todos os ambientes virtuais. Nela também estão disponíveis os valores estatísticos média, desvio padrão, mínimo e máximo.

Tabela 15. Resultados das execuções do caso de estudo I

Execução	VM02		VM04		VM08		VM16	
	Tempo (ms)	Notificações	Tempo (ms)	Notificações	Tempo (ms)	Notificações	Tempo (ms)	Notificações
01	8423	2861574	6539	2862469	5228	2863046	4399	2862661
02	8760	2861708	6181	2862545	5364	2863238	4978	2863364
03	8681	2861025	6501	2862374	5497	2863468	4838	2863536
04	8168	2860653	6063	2861670	5337	2863268	4742	2862666
05	8474	2861018	6192	2862197	5796	2864236	4915	2862870
06	8626	2861345	6360	2862684	5849	2864017	4802	2862722
07	8724	2860914	6134	2863097	5750	2863789	4975	2862796
08	8567	2861321	6261	2861308	5842	2864421	4901	2863315
09	8895	2861318	6840	2864019	5456	2862832	4395	2862102
10	8443	2861906	6553	2863326	5776	2863655	4810	2862760
11	8527	2860824	6495	2862838	5510	2863667	4771	2862829
12	8218	2861595	6436	2863009	5571	2863817	4534	2862617
13	8740	2861916	6399	2863582	5492	2863623	4596	2862780
14	8356	2860978	6568	2862343	5743	2863505	4671	2862694
15	8730	2861658	6398	2862438	5672	2864143	4702	2862483
16	8826	2860926	6155	2862221	5497	2863537	4739	2862679
17	8100	2860994	6469	2863480	5535	2863253	4645	2862207
18	8449	2861540	6789	2863291	5650	2863493	4489	2861953
19	8786	2861531	6537	2862225	5751	2863047	4360	2861768
20	8650	2861756	6745	2862800	5201	2862691	4985	2862763
21	8666	2860951	6745	2863053	5270	2863356	4726	2862914
22	8790	2860866	6229	2861558	5611	2863533	4489	2862150
23	8456	2861923	6829	2863392	5386	2863327	4922	2863068
24	8213	2861042	6293	2862509	5273	2863955	4740	2862562
25	8091	2860856	6713	2863398	5744	2863915	4927	2863070
26	8329	2861279	6138	2862529	5410	2862829	4774	2862656
27	8166	2861400	6873	2862918	5862	2864212	4411	2862334
28	8322	2861250	6561	2862362	5526	2863732	4656	2862489
29	8526	2861765	6419	2862029	5326	2863471	4760	2862647
30	8266	2860311	6644	2862784	5343	2862787	4452	2862774
Média	**8499**	**2861271**	**6469**	**2862682**	**5542**	**2863529**	**4703**	**2862674**
Mínimo	**8091**	**2860311**	**6063**	**2861308**	**5201**	**2862691**	**4360**	**2861768**
Máximo	**8895**	**2861923**	**6873**	**2864019**	**5862**	**2864421**	**4985**	**2863536**
Variação	**804**	**1612**	**810**	**2711**	**661**	**1730**	**625**	**1768**
Desvio Padrão	**232,5789**	**406,4655**	**231,9521**	**613,8810**	**198,8683**	**445,2003**	**187,9636**	**386,4450**
Desv. Padrão (%)	**2,7366%**	**0,0142%**	**3,5858%**	**0,0214%**	**3,5882%**	**0,0155%**	**3,9963%**	**0,0135%**

Fonte: Autoria própria

Figura 94. Histograma de tempos de execução por ambiente virtual para caso de estudo I



Fonte: Autoria própria

APÊNDICE C - Tabulação de resultados do caso de estudo II

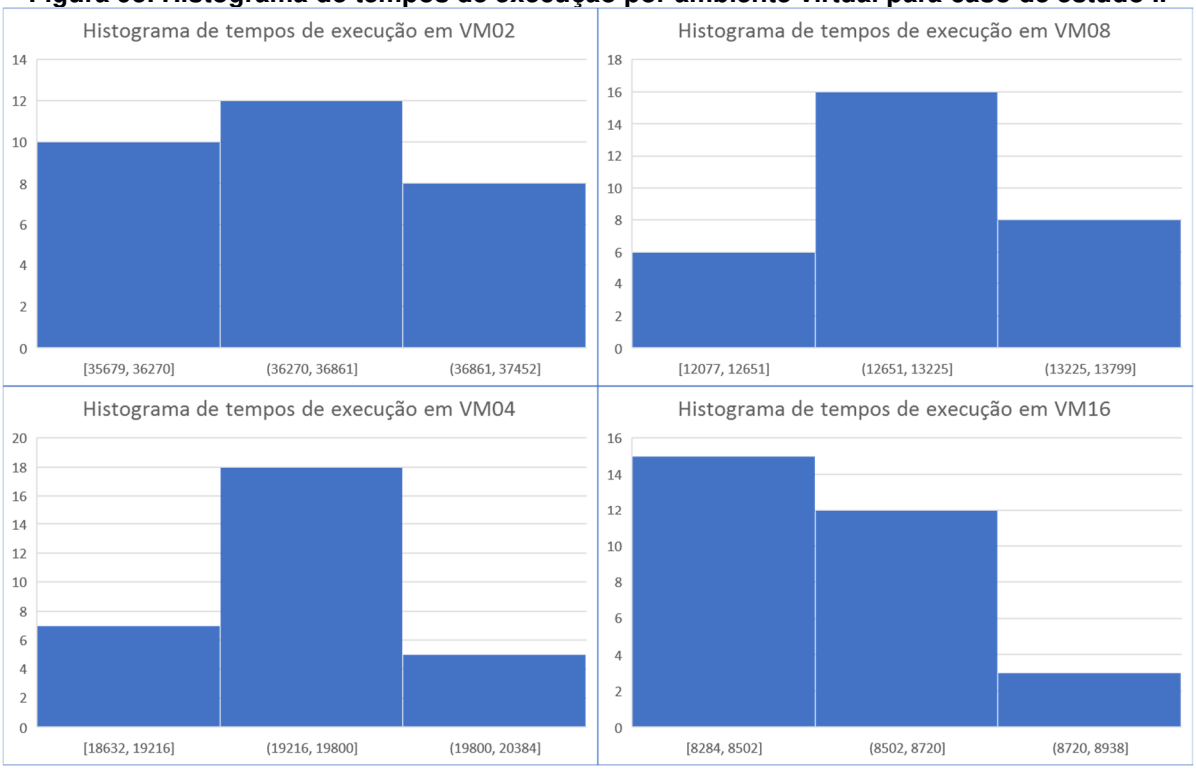
A seguir são apresentados na Tabela 15 os valores recuperados do caso de estudo apresentado na seção 5.4. A Tabela 15 apresenta, respectivamente, os tempos e o número de notificações de cada execução para todos os ambientes virtuais. Nela também estão disponíveis os valores estatísticos média, desvio padrão, mínimo e máximo.

Tabela 16. Resultados das execuções do caso de estudo II

Execução	VM02		VM04		VM08		VM16	
	Tempo (ms)	Notificações	Tempo (ms)	Notificações	Tempo (ms)	Notificações	Tempo (ms)	Notificações
01	35679	9211754	19954	9209400	13633	9209400	8661	9209400
02	35831	9218724	19695	9214478	13582	9216206	8322	9209592
03	36223	9217784	19226	9214188	12378	9217894	8619	9211124
04	35890	9209400	19678	9218476	12923	9215440	8382	9210062
05	36584	9217494	19721	9212970	12095	9209494	8498	9210438
06	36149	9218988	19648	9215142	13061	9218680	8636	9209588
07	36515	9217584	19351	9215324	12077	9212508	8476	9211688
08	37181	9218816	20050	9217482	13199	9210450	8576	9213588
09	36706	9217448	19822	9218570	13204	9218548	8350	9209906
10	36533	9217926	18934	9214490	13279	9217074	8819	9210450
11	36561	9217396	19249	9216540	12962	9216886	8318	9209494
12	36986	9218886	19262	9214802	13798	9214900	8435	9213878
13	36924	9217632	19389	9218866	12855	9213122	8551	9209588
14	36648	9216002	18632	9213220	13076	9217478	8484	9216578
15	36895	9217702	19007	9218740	13128	9218682	8573	9209878
16	36329	9218144	19621	9218576	12554	9217642	8488	9214246
17	35935	9218672	19033	9215688	13382	9216264	8434	9210066
18	37177	9218882	20185	9218690	12657	9218614	8632	9209400
19	36464	9218706	20239	9215770	12477	9213968	8284	9211672
20	36232	9218886	19168	9215374	13326	9218352	8670	9211958
21	36518	9218838	18938	9214086	12982	9217266	8574	9211018
22	37109	9218842	19357	9217126	12859	9216656	8428	9212926
23	35697	9217510	19545	9217208	12839	9212040	8769	9211892
24	36931	9218830	19795	9217956	13563	9217834	8473	9209686
25	35797	9218964	19291	9216586	12863	9210058	8545	9209400
26	37452	9217796	19178	9214388	12862	9209776	8393	9216014
27	36084	9218878	19246	9218760	13381	9214960	8605	9209400
28	36497	9217922	19562	9217548	12687	9218480	8434	9211104
29	36659	9218838	19660	9218728	12189	9216444	8729	9212872
30	36564	9217362	19650	9213502	12921	9211892	8594	9209588
Média	**36492**	**9217687**	**19470**	**9216089**	**12960**	**9215234**	**8525**	**9211216**
Mínimo	**35679**	**9209400**	**18632**	**9209400**	**12077**	**9209400**	**8284**	**9209400**
Máximo	**37452**	**9218988**	**20239**	**9218866**	**13798**	**9218682**	**8819**	**9216578**
Variação	**1773**	**9588**	**1607**	**9466**	**1721**	**9282**	**535**	**7178**
Desvio Padrão	**465,0585**	**2050,0418**	**379,4051**	**2257,9673**	**437,7611**	**3100,4239**	**133,2729**	**1977,1108**
Desv. Padrão (%)	**1,2744%**	**0,0222%**	**1,9487%**	**0,0245%**	**3,3779%**	**0,0336%**	**1,5633%**	**0,0215%**

Fonte: Autoria própria

Figura 95. Histograma de tempos de execução por ambiente virtual para caso de estudo II



Fonte: Autoria própria

APÊNDICE D - Tabulação dos comparativos entre os estudos de caso

Este apêndice apresenta os valores apresentados nos gráficos apresentados na seção 5.5.

Tabela 17. Percentual de variação de tempo entre estratégias IND e CBCF

	VM02	VM04	VM08	VM16
IND	8499	6469	5542	4703
CBCF	36492	19470	12960	8525
DIF(%)	429%	301%	234%	181%

Fonte: Autoria própria

Tabela 18. Comparativo de melhora de desempenho entre os ambientes

	MONO-DUAL	MONO-QUAD	MONO-OCTA	DUAL-QUAD	QUAD-OCTA
CTA	37,13%	60,13%	73,27%	36,59%	32,95%
CBTF	42,66%	69,53%	82,15%	46,87%	41,41%
DIFERENÇA	5,53%	9,40%	8,88%	10,28%	8,46%

Fonte: Autoria própria

Tabela 19. Comparativo de fator tempo dividido por número de notificações

	MONO	DUAL	QUAD	OCTA
CTA	3745,75	2355,04	1493,34	1001,24
CBTF	5451,77	3126,18	1660,95	973,19
DIFERENÇA	1706,02	771,14	167,61	-28,06

Fonte: Autoria própria

APÊNDICE E - Códigos fonte em linguagem NOPL do caso de estudo I

Este apêndice apresenta os códigos fonte em linguagem NOPL utilizados no caso de estudo apresentado na seção 5.3.

Arquivo cta.nop

```
fbe Semaphore_PON

      public integer atSeconds = 0

      public integer atSemaphoreState = 5

      private method mtRUN
            params
                  Integer iterations
            end_params
            code ELIXIR
                  Enum.map(1..iterations,
                    fn(x) ->
                        PON.Service.FBE.set_attribute(this,
                          :atSeconds, rem(x,90))
                     end)
            end_code
      end_method

      private method mtResetTimer
            attribution
                  this.atSeconds = 0
            end_attribution
      end_method

      private method mtHorGreen
            attribution
                  this.atSemaphoreState = 0
            end_attribution
      end_method

      private method mtHorYellow
            attribution
                  this.atSemaphoreState = 1
            end_attribution
      end_method

      private method mtHorRed
            attribution
                  this.atSemaphoreState = 2
            end_attribution
      end_method

      private method mtVerGreen
            attribution
                  this.atSemaphoreState = 3
            end_attribution
      end_method
```

```
      private method mtVerYellow
            attribution
                  this.atSemaphoreState = 4
            end_attribution
      end_method

      private method mtVerRed
            attribution
                  this.atSemaphoreState = 5
            end_attribution
      end_method

      rule rlHorGreen
            condition
                  subcondition sbHorGreen
                        premise prSeconds_2
                              this.atSeconds == 2
                        end_premise
                        and
                        premise prSemaphoreState_5
                              this.atSemaphoreState == 5
                        end_premise
                  end_subcondition
            end_condition
            action acHTLG
                  instigation inHTLG
                        call this.mtHorGreen()
                  end_instigation
            end_action
      end_rule

      rule rlHorYellow
            condition
                  subcondition sbHorYellow
                        premise prSeconds_40
                              this.atSeconds == 40
                        end_premise
                        and
                        premise prSemaphoreState_0
                              this.atSemaphoreState == 0
                        end_premise
                  end_subcondition
            end_condition
            action acHTLY
                  instigation inHTLY
                        call this.mtHorYellow()
                  end_instigation
            end_action
      end_rule

      rule rlHorRed
            condition
                  subcondition sbHorRed
                        premise prAtSeconds_45
                              this.atSeconds == 45
                        end_premise
                        and
                        premise prSemaphoreState_1
                              this.atSemaphoreState == 1
                        end_premise
                  end_subcondition
            end_condition
            action acHTLR
```

```
                instigation inHTLR
                      call this.mtHorRed()
                end_instigation
          end_action
    end_rule

    rule rlVerGreen
          condition
                subcondition sbVerGreen
                      premise prAtSeconds_47
                            this.atSeconds == 47
                      end_premise
                      and
                      premise prSemaphoreState_2
                            this.atSemaphoreState == 2
                      end_premise
                end_subcondition
          end_condition
          action acVTLG
                instigation inVTLG
                      call this.mtVerGreen()
                end_instigation
          end_action
    end_rule

    rule rlVerYellow
          condition
                subcondition sbVerYellow
                      premise prAtSeconds_85
                            this.atSeconds == 85
                      end_premise
                      and
                      premise prSemaphoreState_3
                            this.atSemaphoreState == 3
                      end_premise
                end_subcondition
          end_condition
          action acVTLY
                instigation inVTLY
                      call this.mtVerYellow()
                end_instigation
          end_action
    end_rule

    rule rlVerRed
          condition
                subcondition sbVerRed
                      premise prAtSeconds_90
                            this.atSeconds == 90
                      end_premise
                      and
                      premise prSemaphoreState_4
                            this.atSemaphoreState == 4
                      end_premise
                end_subcondition
          end_condition
          action acVTLR
                instigation inVTLR
                      call this.mtVerRed();
                end_instigation
                instigation inResetTime
                      call this.mtResetTimer()
                end_instigation
```

```
            end_action
      end_rule

end_fbe
```

Arquivo Main.nop

```
fbe Main

      public Semaphore_PON Semaphore_01_01
      public Semaphore_PON Semaphore_01_02
      public Semaphore_PON Semaphore_01_03
      public Semaphore_PON Semaphore_01_04
      public Semaphore_PON Semaphore_01_05
      public Semaphore_PON Semaphore_01_06
      public Semaphore_PON Semaphore_01_07
      public Semaphore_PON Semaphore_01_08
      public Semaphore_PON Semaphore_01_09
      public Semaphore_PON Semaphore_01_10

      public Semaphore_PON Semaphore_02_01
      public Semaphore_PON Semaphore_02_02
      public Semaphore_PON Semaphore_02_03
      public Semaphore_PON Semaphore_02_04
      public Semaphore_PON Semaphore_02_05
      public Semaphore_PON Semaphore_02_06
      public Semaphore_PON Semaphore_02_07
      public Semaphore_PON Semaphore_02_08
      public Semaphore_PON Semaphore_02_09
      public Semaphore_PON Semaphore_02_10

      public Semaphore_PON Semaphore_03_01
      public Semaphore_PON Semaphore_03_02
      public Semaphore_PON Semaphore_03_03
      public Semaphore_PON Semaphore_03_04
      public Semaphore_PON Semaphore_03_05
      public Semaphore_PON Semaphore_03_06
      public Semaphore_PON Semaphore_03_07
      public Semaphore_PON Semaphore_03_08
      public Semaphore_PON Semaphore_03_09
      public Semaphore_PON Semaphore_03_10

      public Semaphore_PON Semaphore_04_01
      public Semaphore_PON Semaphore_04_02
      public Semaphore_PON Semaphore_04_03
      public Semaphore_PON Semaphore_04_04
      public Semaphore_PON Semaphore_04_05
      public Semaphore_PON Semaphore_04_06
      public Semaphore_PON Semaphore_04_07
      public Semaphore_PON Semaphore_04_08
      public Semaphore_PON Semaphore_04_09
      public Semaphore_PON Semaphore_04_10

      public Semaphore_PON Semaphore_05_01
      public Semaphore_PON Semaphore_05_02
      public Semaphore_PON Semaphore_05_03
      public Semaphore_PON Semaphore_05_04
      public Semaphore_PON Semaphore_05_05
      public Semaphore_PON Semaphore_05_06
      public Semaphore_PON Semaphore_05_07
      public Semaphore_PON Semaphore_05_08
      public Semaphore_PON Semaphore_05_09
```

```
      public Semaphore_PON Semaphore_05_10

      public Semaphore_PON Semaphore_06_01
      public Semaphore_PON Semaphore_06_02
      public Semaphore_PON Semaphore_06_03
      public Semaphore_PON Semaphore_06_04
      public Semaphore_PON Semaphore_06_05
      public Semaphore_PON Semaphore_06_06
      public Semaphore_PON Semaphore_06_07
      public Semaphore_PON Semaphore_06_08
      public Semaphore_PON Semaphore_06_09
      public Semaphore_PON Semaphore_06_10

      public Semaphore_PON Semaphore_07_01
      public Semaphore_PON Semaphore_07_02
      public Semaphore_PON Semaphore_07_03
      public Semaphore_PON Semaphore_07_04
      public Semaphore_PON Semaphore_07_05
      public Semaphore_PON Semaphore_07_06
      public Semaphore_PON Semaphore_07_07
      public Semaphore_PON Semaphore_07_08
      public Semaphore_PON Semaphore_07_09
      public Semaphore_PON Semaphore_07_10

      public Semaphore_PON Semaphore_08_01
      public Semaphore_PON Semaphore_08_02
      public Semaphore_PON Semaphore_08_03
      public Semaphore_PON Semaphore_08_04
      public Semaphore_PON Semaphore_08_05
      public Semaphore_PON Semaphore_08_06
      public Semaphore_PON Semaphore_08_07
      public Semaphore_PON Semaphore_08_08
      public Semaphore_PON Semaphore_08_09
      public Semaphore_PON Semaphore_08_10

      public Semaphore_PON Semaphore_09_01
      public Semaphore_PON Semaphore_09_02
      public Semaphore_PON Semaphore_09_03
      public Semaphore_PON Semaphore_09_04
      public Semaphore_PON Semaphore_09_05
      public Semaphore_PON Semaphore_09_06
      public Semaphore_PON Semaphore_09_07
      public Semaphore_PON Semaphore_09_08
      public Semaphore_PON Semaphore_09_09
      public Semaphore_PON Semaphore_09_10

      public Semaphore_PON Semaphore_10_01
      public Semaphore_PON Semaphore_10_02
      public Semaphore_PON Semaphore_10_03
      public Semaphore_PON Semaphore_10_04
      public Semaphore_PON Semaphore_10_05
      public Semaphore_PON Semaphore_10_06
      public Semaphore_PON Semaphore_10_07
      public Semaphore_PON Semaphore_10_08
      public Semaphore_PON Semaphore_10_09
      public Semaphore_PON Semaphore_10_10

      public method mtSIMULATE
            params
                  Integer iterations
            end_params
            code ELIXIR
                  instances = initialize()
```

```
Task.start(NopEx.Semaphore_PON, :mtRUN,
      [instances[:Semaphore_01_01], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
      [instances[:Semaphore_01_02], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_01_03], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_01_04], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_01_05], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_01_06], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_01_07], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_01_08], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_01_09], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_01_10], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_02_01], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_02_02], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_02_03], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_02_04], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_02_05], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_02_06], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_02_07], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_02_08], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_02_09], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_02_10], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_03_01], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_03_02], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_03_03], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_03_04], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_03_05], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_03_06], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_03_07], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_03_08], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_03_09], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_03_10], iterations])

Task.start(NopEx.Semaphore_PON, :mtRUN,
```

```
                [instances[:Semaphore_04_01], iterations])
        Task.start(NopEx.Semaphore_PON, :mtRUN,
                [instances[:Semaphore_04_02], iterations])
        Task.start(NopEx.Semaphore_PON, :mtRUN,
                [instances[:Semaphore_04_03], iterations])
        Task.start(NopEx.Semaphore_PON, :mtRUN,
                [instances[:Semaphore_04_04], iterations])
        Task.start(NopEx.Semaphore_PON, :mtRUN,
                [instances[:Semaphore_04_05], iterations])
        Task.start(NopEx.Semaphore_PON, :mtRUN,
                [instances[:Semaphore_04_06], iterations])
        Task.start(NopEx.Semaphore_PON, :mtRUN,
                [instances[:Semaphore_04_07], iterations])
        Task.start(NopEx.Semaphore_PON, :mtRUN,
                [instances[:Semaphore_04_08], iterations])
        Task.start(NopEx.Semaphore_PON, :mtRUN,
                [instances[:Semaphore_04_09], iterations])
        Task.start(NopEx.Semaphore_PON, :mtRUN,
                [instances[:Semaphore_04_10], iterations])
        Task.start(NopEx.Semaphore_PON, :mtRUN,
                [instances[:Semaphore_05_01], iterations])
        Task.start(NopEx.Semaphore_PON, :mtRUN,
                [instances[:Semaphore_05_02], iterations])
        Task.start(NopEx.Semaphore_PON, :mtRUN,
                [instances[:Semaphore_05_03], iterations])
        Task.start(NopEx.Semaphore_PON, :mtRUN,
                [instances[:Semaphore_05_04], iterations])
        Task.start(NopEx.Semaphore_PON, :mtRUN,
                [instances[:Semaphore_05_05], iterations])
        Task.start(NopEx.Semaphore_PON, :mtRUN,
                [instances[:Semaphore_05_06], iterations])
        Task.start(NopEx.Semaphore_PON, :mtRUN,
                [instances[:Semaphore_05_07], iterations])
        Task.start(NopEx.Semaphore_PON, :mtRUN,
                [instances[:Semaphore_05_08], iterations])
        Task.start(NopEx.Semaphore_PON, :mtRUN,
                [instances[:Semaphore_05_09], iterations])
        Task.start(NopEx.Semaphore_PON, :mtRUN,
                [instances[:Semaphore_05_10], iterations])
        Task.start(NopEx.Semaphore_PON, :mtRUN,
                [instances[:Semaphore_06_01], iterations])
        Task.start(NopEx.Semaphore_PON, :mtRUN,
                [instances[:Semaphore_06_02], iterations])
        Task.start(NopEx.Semaphore_PON, :mtRUN,
                [instances[:Semaphore_06_03], iterations])
        Task.start(NopEx.Semaphore_PON, :mtRUN,
                [instances[:Semaphore_06_04], iterations])
        Task.start(NopEx.Semaphore_PON, :mtRUN,
                [instances[:Semaphore_06_05], iterations])
        Task.start(NopEx.Semaphore_PON, :mtRUN,
                [instances[:Semaphore_06_06], iterations])
        Task.start(NopEx.Semaphore_PON, :mtRUN,
                [instances[:Semaphore_06_07], iterations])
        Task.start(NopEx.Semaphore_PON, :mtRUN,
                [instances[:Semaphore_06_08], iterations])
        Task.start(NopEx.Semaphore_PON, :mtRUN,
                [instances[:Semaphore_06_09], iterations])
        Task.start(NopEx.Semaphore_PON, :mtRUN,
                [instances[:Semaphore_06_10], iterations])
        Task.start(NopEx.Semaphore_PON, :mtRUN,
                [instances[:Semaphore_07_01], iterations])
        Task.start(NopEx.Semaphore_PON, :mtRUN,
                [instances[:Semaphore_07_02], iterations])
```

```
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_07_03], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_07_04], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_07_05], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_07_06], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_07_07], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_07_08], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_07_09], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_07_10], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_08_01], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_08_02], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_08_03], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_08_04], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_08_05], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_08_06], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_08_07], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_08_08], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_08_09], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_08_10], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_09_01], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_09_02], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_09_03], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_09_04], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_09_05], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_09_06], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_09_07], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_09_08], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_09_09], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_09_10], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_10_01], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_10_02], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
       [instances[:Semaphore_10_03], iterations])
Task.start(NopEx.Semaphore_PON, :mtRUN,
```

```
                        [instances[:Semaphore_10_04], iterations])
                Task.start(NopEx.Semaphore_PON, :mtRUN,
                        [instances[:Semaphore_10_05], iterations])
                Task.start(NopEx.Semaphore_PON, :mtRUN,
                        [instances[:Semaphore_10_06], iterations])
                Task.start(NopEx.Semaphore_PON, :mtRUN,
                        [instances[:Semaphore_10_07], iterations])
                Task.start(NopEx.Semaphore_PON, :mtRUN,
                        [instances[:Semaphore_10_08], iterations])
                Task.start(NopEx.Semaphore_PON, :mtRUN,
                        [instances[:Semaphore_10_09], iterations])
                Task.start(NopEx.Semaphore_PON, :mtRUN,
                        [instances[:Semaphore_10_10], iterations])

            end_code
        end_method

end_fbe
```

APÊNDICE F - Códigos fonte em linguagem NOPL do caso de estudo II

Este apêndice apresenta os códigos fonte em linguagem PON utilizados no caso de estudo apresentado na seção 5.4.

Arquivo cta.nop

```
fbe Semaphore_PON

      public integer atSeconds = 0

      public integer atSemaphoreState = 5

      public integer atHVSS = 0

      public integer atVVSS = 0

      private method mtINC
            params
                  Integer count
            end_params
            code ELIXIR
                  PON.Service.FBE.set_attribute_expr(this,
                    :atSeconds, "@atSeconds + " <> to_string(count))
            end_code
      end_method

      private method mtRUN
            params
                  Integer iterations
            end_params
            code ELIXIR
                  Enum.map(1..iterations,
                        fn(_x) ->
                              mtINC(this, 1)
                        end)
            end_code
      end_method

      private method mtRT
            attribution
                  this.atSeconds = 0
            end_attribution
      end_method

      private method mtHTLG
            attribution
                  this.atSemaphoreState = 0
            end_attribution
      end_method

      private method mtHTLY
            attribution
                  this.atSemaphoreState = 1
            end_attribution
      end_method

      private method mtHTLR
```

```
            attribution
                  this.atSemaphoreState = 2
            end_attribution
      end_method

      private method mtVTLG
            attribution
                  this.atSemaphoreState = 3
            end_attribution
      end_method

      private method mtVTLY
            attribution
                  this.atSemaphoreState = 4
            end_attribution
      end_method

      private method mtVTLR
            attribution
                  this.atSemaphoreState = 5
            end_attribution
      end_method

      private method mtHTLGCBCL
            attribution
                  this.atSemaphoreState = 6
            end_attribution
      end_method

      private method mtHTLYCBCL
            attribution
                  this.atSemaphoreState = 7
            end_attribution
      end_method

      private method mtVTLGCBCL
            attribution
                  this.atSemaphoreState = 8
            end_attribution
      end_method

      private method mtVTLYCBCL
            attribution
                  this.atSemaphoreState = 9
            end_attribution
      end_method

      rule rlCBCL1
            condition
                  subcondition sbCBCL1
                        premise prSeconds
                              this.atSeconds == 2 end_premise
                        and
                        premise prSemaphoreState
                              this.atSemaphoreState == 5 end_premise
                  end_subcondition
            end_condition
            action actCBCL1
                  instigation inCBCL1
                        call this.mtHTLG()
                        call this.mtRT()
                  end_instigation
            end_action
```

```
	end_rule

	rule rlCBCL2
		condition
			subcondition sbCBCL2
				premise prSeconds2
					this.atSeconds == 38 end_premise
				and
				premise prSemaphoreState2
					this.atSemaphoreState == 0 end_premise
			end_subcondition
		end_condition
		action actCBCL2
			instigation inCBCL2
				call this.mtHTLY()
				call this.mtRT()
			end_instigation
		end_action
	end_rule

	rule rlCBCL3
		condition
			subcondition sbCBCL3
				premise prSecondsCBCL2
					this.atSeconds == 30 end_premise
				and
				premise prSemaphoreStateCBCL2
					this.atSemaphoreState == 6 end_premise
			end_subcondition
		end_condition
		action actCBCL3
			instigation inCBCL3
				call this.mtHTLY()
				call this.mtRT()
			end_instigation
		end_action
	end_rule

	rule rlCBCL4
		condition
			subcondition sbCBCL4
				premise pratSeconds3
					this.atSeconds == 5 end_premise
				and
				premise prSemaphoreState3
					this.atSemaphoreState == 1 end_premise
			end_subcondition
		end_condition
		action actCBCL4
			instigation inCBCL4
				call this.mtHTLR()
				call this.mtRT()
			end_instigation
		end_action
	end_rule

	rule rlCBCL5
		condition
			subcondition sbCBCL5
				premise prSecondsCBCL3
					this.atSeconds == 6 end_premise
				and
				premise prSemaphoreStateCBCL3
```

```
                        this.atSemaphoreState == 7 end_premise
            end_subcondition
      end_condition
      action actCBCL5
            instigation inCBCL5
                  call this.mtHTLR()
                  call this.mtRT()
            end_instigation
      end_action
end_rule

rule rlCBCL6
      condition
            subcondition sbCBCL6
                  premise pratSeconds4
                        this.atSeconds == 2 end_premise
                  and
                  premise prSemaphoreState4
                        this.atSemaphoreState == 2 end_premise
            end_subcondition
      end_condition
      action actCBCL6
            instigation inCBCL6
                  call this.mtVTLG()
                  call this.mtRT()
            end_instigation
      end_action
end_rule

rule rlCBCL7
      condition
            subcondition sbCBCL7
                  premise pratSeconds5
                        this.atSeconds == 38 end_premise
                  and
                  premise prSemaphoreState5
                        this.atSemaphoreState == 3 end_premise
            end_subcondition
      end_condition
      action actCBCL7
            instigation inCBCL7
                  call this.mtVTLY()
                  call this.mtRT()
            end_instigation
      end_action
end_rule

rule rlCBCL8
      condition
            subcondition sbCBCL8
                  premise prSecondsCBCL5
                        this.atSeconds == 30 end_premise
                  and
                  premise prSemaphoreStateCBCL5
                        this.atSemaphoreState == 8 end_premise
            end_subcondition
      end_condition
      action actCBCL8
            instigation inCBCL8
                  call this.mtVTLY()
                  call this.mtRT()
            end_instigation
      end_action
```

```
	end_rule

	rule rlCBCL9
		condition
			subcondition sbCBCL9
				premise pratSeconds6
					this.atSeconds == 5 end_premise
				and
				premise prSemaphoreState6
					this.atSemaphoreState == 4 end_premise
			end_subcondition
		end_condition
		action actCBCL9
			instigation inCBCL9
				call this.mtVTLR()
				call this.mtRT()
			end_instigation
		end_action
	end_rule

	rule rlCBCL10
		condition
			subcondition sbCBCL10
				premise prSecondsCBCL6
					this.atSeconds == 6 end_premise
				and
				premise prSemaphoreStateCBCL6
					this.atSemaphoreState == 9 end_premise
			end_subcondition
		end_condition
		action actCBCL10
			instigation inCBCL10
				call this.mtVTLR()
				call this.mtRT()
			end_instigation
		end_action
	end_rule

	rule rlCBCL11
		condition
			subcondition sbCBCL11
				premise prSeconds7
					this.atSeconds <= 17 end_premise
				and
				premise prSemaphoreState7
					this.atSemaphoreState == 0 end_premise
				and
				premise prVehicleSensorState7
					this.atHVSS == 1 end_premise
			end_subcondition
		end_condition
		action actCBCL11
			instigation inCBCL11
				call this.mtHTLGCBCL()
			end_instigation
		end_action
	end_rule

	rule rlCBCL12
		condition
			subcondition sbCBCL12
				premise prSeconds7Full
					this.atSeconds <= 17 end_premise
```

```
                        and
                        premise prSemaphoreState7Full
                              his.atSemaphoreState == 0 end_premise
                        and
                        premise prVehicleSensorState7Full
                              this.atHVSS == 2 end_premise
                  end_subcondition
            end_condition
            action actCBCL12
                  instigation inCBCL12
                        call this.mtHTLGCBCL()
                  end_instigation
            end_action
      end_rule

      rule rlCBCL13
            condition
                  subcondition sbCBCL13
                        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_subcondition
            end_condition
            action actCBCL13
                  instigation inCBCL13
                        call this.mtHTLYCBCL()
                        call this.mtRT()
                  end_instigation
            end_action
      end_rule

      rule rlCBCL14
            condition
                  subcondition sbCBCL14
                        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_subcondition
            end_condition
            action actCBCL14
                  instigation inCBCL14
                        call this.mtHTLYCBCL()
                        call this.mtRT()
                  end_instigation
            end_action
      end_rule
```

```
rule rlCBCL15
      condition
            subcondition sbCBCL15
                  premise prSeconds9
                        this.atSeconds <= 17 end_premise
                  and
                  premise prSemaphoreState9
                        this.atSemaphoreState == 3 end_premise
                  and
                  premise prVehicleSensorState9
                        this.atVVSS == 1 end_premise
            end_subcondition
      end_condition
      action actCBCL15
            instigation inCBCL15
                  call this.mtVTLGCBCL()
            end_instigation
      end_action
end_rule

rule rlCBCL16
      condition
            subcondition sbCBCL16
                  premise prSeconds9Full
                        this.atSeconds <= 17 end_premise
                  and
                  premise prSemaphoreState9Full
                        this.atSemaphoreState == 3 end_premise
                  and
                  premise prVehicleSensorState9Full
                        this.atVVSS == 2 end_premise
            end_subcondition
      end_condition
      action actCBCL16
            instigation inCBCL16
                  call this.mtVTLGCBCL()
            end_instigation
      end_action
end_rule

rule rlCBCL17
      condition
            subcondition sbCBCL17
                  premise prSeconds10
                        this.atSeconds >= 18 end_premise
                  and
                  premise prSeconds210
                        this.atSeconds < 32 end_premise
                  and
                  premise prSemaphoreState10
                        this.atSemaphoreState == 3 end_premise
                  and
                  premise prVehicleSensorState10
                        this.atHVSS == 1 end_premise
            end_subcondition
      end_condition
      action actCBCL17
            instigation inCBCL17
                  call this.mtVTLYCBCL()
                  call this.mtRT()
            end_instigation
      end_action
end_rule
```

```
      rule rlCBCL18
            condition
                  subcondition sbCBCL18
                        premise prSeconds10Full
                              this.atSeconds >= 18 end_premise
                        and
                        premise prSeconds210Full
                              this.atSeconds < 32 end_premise
                        and
                        premise prSemaphoreState10Full
                              this.atSemaphoreState == 3 end_premise
                        and
                        premise prVehicleSensorState10Full
                              this.atHVSS == 2 end_premise
                  end_subcondition
            end_condition
            action actCBCL18
                  instigation inCBCL18
                        call this.mtVTLYCBCL()
                        call this.mtRT()
                  end_instigation
            end_action
      end_rule

end_fbe
```

Arquivo Main.nop

```
fbe Main

      public Semaphore_PON Semaphore_01_01
      public Semaphore_PON Semaphore_01_02
      public Semaphore_PON Semaphore_01_03
      public Semaphore_PON Semaphore_01_04
      public Semaphore_PON Semaphore_01_05
      public Semaphore_PON Semaphore_01_06
      public Semaphore_PON Semaphore_01_07
      public Semaphore_PON Semaphore_01_08
      public Semaphore_PON Semaphore_01_09
      public Semaphore_PON Semaphore_01_10

      public Semaphore_PON Semaphore_02_01
      public Semaphore_PON Semaphore_02_02
      public Semaphore_PON Semaphore_02_03
      public Semaphore_PON Semaphore_02_04
      public Semaphore_PON Semaphore_02_05
      public Semaphore_PON Semaphore_02_06
      public Semaphore_PON Semaphore_02_07
      public Semaphore_PON Semaphore_02_08
      public Semaphore_PON Semaphore_02_09
      public Semaphore_PON Semaphore_02_10

      public Semaphore_PON Semaphore_03_01
      public Semaphore_PON Semaphore_03_02
      public Semaphore_PON Semaphore_03_03
      public Semaphore_PON Semaphore_03_04
      public Semaphore_PON Semaphore_03_05
      public Semaphore_PON Semaphore_03_06
      public Semaphore_PON Semaphore_03_07
      public Semaphore_PON Semaphore_03_08
      public Semaphore_PON Semaphore_03_09
```

```
    public Semaphore_PON Semaphore_03_10

    public Semaphore_PON Semaphore_04_01
    public Semaphore_PON Semaphore_04_02
    public Semaphore_PON Semaphore_04_03
    public Semaphore_PON Semaphore_04_04
    public Semaphore_PON Semaphore_04_05
    public Semaphore_PON Semaphore_04_06
    public Semaphore_PON Semaphore_04_07
    public Semaphore_PON Semaphore_04_08
    public Semaphore_PON Semaphore_04_09
    public Semaphore_PON Semaphore_04_10

    public Semaphore_PON Semaphore_05_01
    public Semaphore_PON Semaphore_05_02
    public Semaphore_PON Semaphore_05_03
    public Semaphore_PON Semaphore_05_04
    public Semaphore_PON Semaphore_05_05
    public Semaphore_PON Semaphore_05_06
    public Semaphore_PON Semaphore_05_07
    public Semaphore_PON Semaphore_05_08
    public Semaphore_PON Semaphore_05_09
    public Semaphore_PON Semaphore_05_10

    public Semaphore_PON Semaphore_06_01
    public Semaphore_PON Semaphore_06_02
    public Semaphore_PON Semaphore_06_03
    public Semaphore_PON Semaphore_06_04
    public Semaphore_PON Semaphore_06_05
    public Semaphore_PON Semaphore_06_06
    public Semaphore_PON Semaphore_06_07
    public Semaphore_PON Semaphore_06_08
    public Semaphore_PON Semaphore_06_09
    public Semaphore_PON Semaphore_06_10

    public Semaphore_PON Semaphore_07_01
    public Semaphore_PON Semaphore_07_02
    public Semaphore_PON Semaphore_07_03
    public Semaphore_PON Semaphore_07_04
    public Semaphore_PON Semaphore_07_05
    public Semaphore_PON Semaphore_07_06
    public Semaphore_PON Semaphore_07_07
    public Semaphore_PON Semaphore_07_08
    public Semaphore_PON Semaphore_07_09
    public Semaphore_PON Semaphore_07_10

    public Semaphore_PON Semaphore_08_01
    public Semaphore_PON Semaphore_08_02
    public Semaphore_PON Semaphore_08_03
    public Semaphore_PON Semaphore_08_04
    public Semaphore_PON Semaphore_08_05
    public Semaphore_PON Semaphore_08_06
    public Semaphore_PON Semaphore_08_07
    public Semaphore_PON Semaphore_08_08
    public Semaphore_PON Semaphore_08_09
    public Semaphore_PON Semaphore_08_10

    public Semaphore_PON Semaphore_09_01
    public Semaphore_PON Semaphore_09_02
    public Semaphore_PON Semaphore_09_03
    public Semaphore_PON Semaphore_09_04
    public Semaphore_PON Semaphore_09_05
    public Semaphore_PON Semaphore_09_06
```

```
      public Semaphore_PON Semaphore_09_07
      public Semaphore_PON Semaphore_09_08
      public Semaphore_PON Semaphore_09_09
      public Semaphore_PON Semaphore_09_10

      public Semaphore_PON Semaphore_10_01
      public Semaphore_PON Semaphore_10_02
      public Semaphore_PON Semaphore_10_03
      public Semaphore_PON Semaphore_10_04
      public Semaphore_PON Semaphore_10_05
      public Semaphore_PON Semaphore_10_06
      public Semaphore_PON Semaphore_10_07
      public Semaphore_PON Semaphore_10_08
      public Semaphore_PON Semaphore_10_09
      public Semaphore_PON Semaphore_10_10

      public method mtSIMULATE
            params
                  Integer iterations
            end_params
            code ELIXIR
                  instances = initialize()

                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_01_01], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_01_02], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_01_03], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_01_04], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_01_05], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_01_06], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_01_07], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_01_08], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_01_09], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_01_10], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_02_01], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_02_02], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_02_03], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_02_04], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_02_05], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_02_06], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_02_07], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_02_08], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_02_09], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_02_10], iterations])
```

```
                Task.start(NopEx.Semaphore_PON,
                  :mtRUN, [instances[:Semaphore_03_01], iterations])
                Task.start(NopEx.Semaphore_PON,
                  :mtRUN, [instances[:Semaphore_03_02], iterations])
                Task.start(NopEx.Semaphore_PON,
                  :mtRUN, [instances[:Semaphore_03_03], iterations])
                Task.start(NopEx.Semaphore_PON,
                  :mtRUN, [instances[:Semaphore_03_04], iterations])
                Task.start(NopEx.Semaphore_PON,
                  :mtRUN, [instances[:Semaphore_03_05], iterations])
                Task.start(NopEx.Semaphore_PON,
                  :mtRUN, [instances[:Semaphore_03_06], iterations])
                Task.start(NopEx.Semaphore_PON,
                  :mtRUN, [instances[:Semaphore_03_07], iterations])
                Task.start(NopEx.Semaphore_PON,
                  :mtRUN, [instances[:Semaphore_03_08], iterations])
                Task.start(NopEx.Semaphore_PON,
                  :mtRUN, [instances[:Semaphore_03_09], iterations])
                Task.start(NopEx.Semaphore_PON,
                  :mtRUN, [instances[:Semaphore_03_10], iterations])

                Task.start(NopEx.Semaphore_PON,
                  :mtRUN, [instances[:Semaphore_04_01], iterations])
                Task.start(NopEx.Semaphore_PON,
                  :mtRUN, [instances[:Semaphore_04_02], iterations])
                Task.start(NopEx.Semaphore_PON,
                  :mtRUN, [instances[:Semaphore_04_03], iterations])
                Task.start(NopEx.Semaphore_PON,
                  :mtRUN, [instances[:Semaphore_04_04], iterations])
                Task.start(NopEx.Semaphore_PON,
                  :mtRUN, [instances[:Semaphore_04_05], iterations])
                Task.start(NopEx.Semaphore_PON,
                  :mtRUN, [instances[:Semaphore_04_06], iterations])
                Task.start(NopEx.Semaphore_PON,
                  :mtRUN, [instances[:Semaphore_04_07], iterations])
                Task.start(NopEx.Semaphore_PON,
                  :mtRUN, [instances[:Semaphore_04_08], iterations])
                Task.start(NopEx.Semaphore_PON,
                  :mtRUN, [instances[:Semaphore_04_09], iterations])
                Task.start(NopEx.Semaphore_PON,
                  :mtRUN, [instances[:Semaphore_04_10], iterations])

                Task.start(NopEx.Semaphore_PON,
                  :mtRUN, [instances[:Semaphore_05_01], iterations])
                Task.start(NopEx.Semaphore_PON,
                  :mtRUN, [instances[:Semaphore_05_02], iterations])
                Task.start(NopEx.Semaphore_PON,
                  :mtRUN, [instances[:Semaphore_05_03], iterations])
                Task.start(NopEx.Semaphore_PON,
                  :mtRUN, [instances[:Semaphore_05_04], iterations])
                Task.start(NopEx.Semaphore_PON,
                  :mtRUN, [instances[:Semaphore_05_05], iterations])
                Task.start(NopEx.Semaphore_PON,
                  :mtRUN, [instances[:Semaphore_05_06], iterations])
                Task.start(NopEx.Semaphore_PON,
                  :mtRUN, [instances[:Semaphore_05_07], iterations])
                Task.start(NopEx.Semaphore_PON,
                  :mtRUN, [instances[:Semaphore_05_08], iterations])
                Task.start(NopEx.Semaphore_PON,
                  :mtRUN, [instances[:Semaphore_05_09], iterations])
                Task.start(NopEx.Semaphore_PON,
                  :mtRUN, [instances[:Semaphore_05_10], iterations])
                Task.start(NopEx.Semaphore_PON,
```

```
                    :mtRUN, [instances[:Semaphore_06_01], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_06_02], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_06_03], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_06_04], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_06_05], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_06_06], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_06_07], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_06_08], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_06_09], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_06_10], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_07_01], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_07_02], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_07_03], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_07_04], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_07_05], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_07_06], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_07_07], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_07_08], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_07_09], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_07_10], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_08_01], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_08_02], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_08_03], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_08_04], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_08_05], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_08_06], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_08_07], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_08_08], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_08_09], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_08_10], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_09_01], iterations])
                  Task.start(NopEx.Semaphore_PON,
                    :mtRUN, [instances[:Semaphore_09_02], iterations])
```

```
                    Task.start(NopEx.Semaphore_PON,
                     :mtRUN, [instances[:Semaphore_09_03], iterations])
                    Task.start(NopEx.Semaphore_PON,
                     :mtRUN, [instances[:Semaphore_09_04], iterations])
                    Task.start(NopEx.Semaphore_PON,
                     :mtRUN, [instances[:Semaphore_09_05], iterations])
                    Task.start(NopEx.Semaphore_PON,
                     :mtRUN, [instances[:Semaphore_09_06], iterations])
                    Task.start(NopEx.Semaphore_PON,
                     :mtRUN, [instances[:Semaphore_09_07], iterations])
                    Task.start(NopEx.Semaphore_PON,
                     :mtRUN, [instances[:Semaphore_09_08], iterations])
                    Task.start(NopEx.Semaphore_PON,
                     :mtRUN, [instances[:Semaphore_09_09], iterations])
                    Task.start(NopEx.Semaphore_PON,
                     :mtRUN, [instances[:Semaphore_09_10], iterations])
                    Task.start(NopEx.Semaphore_PON,
                     :mtRUN, [instances[:Semaphore_10_01], iterations])
                    Task.start(NopEx.Semaphore_PON,
                     :mtRUN, [instances[:Semaphore_10_02], iterations])
                    Task.start(NopEx.Semaphore_PON,
                     :mtRUN, [instances[:Semaphore_10_03], iterations])
                    Task.start(NopEx.Semaphore_PON,
                     :mtRUN, [instances[:Semaphore_10_04], iterations])
                    Task.start(NopEx.Semaphore_PON,
                     :mtRUN, [instances[:Semaphore_10_05], iterations])
                    Task.start(NopEx.Semaphore_PON,
                     :mtRUN, [instances[:Semaphore_10_06], iterations])
                    Task.start(NopEx.Semaphore_PON,
                     :mtRUN, [instances[:Semaphore_10_07], iterations])
                    Task.start(NopEx.Semaphore_PON,
                     :mtRUN, [instances[:Semaphore_10_08], iterations])
                    Task.start(NopEx.Semaphore_PON,
                     :mtRUN, [instances[:Semaphore_10_09], iterations])
                    Task.start(NopEx.Semaphore_PON,
                     :mtRUN, [instances[:Semaphore_10_10], iterations])

            end_code
      end_method

end_fbe
```

APÊNDICE G - Manual de utilização do *Framework* NOP Elixir

O apêndice a seguir é um documento cujo objetivo é fazer uma apresentação das funcionalidades do *framework* NOP Elixir. O documento inicia com requisitos de instalação para desenvolvimentos com o *framework* e, em seguida, parte para um projeto passo-a-passo a fim de demonstrar, de maneira prática, todas as funcionalidades disponíveis no *framework* NOP Elixir. O documento também traz algumas dicas de depuração e utilização das ferramentas próprias da linguagem Elixir.

Objetivo

O objetivo deste documento é apresentar o módulo hex nop_elixir, um simulador de ambiente NOP (Notified Oriented Programing) para linguagem Elixir aproveitando a arquitetura distribuída do Erlang/OTP disponível sobre a máquina virtual BEAM. Neste documento encontra-se as informações necessárias para instalar e utilizar o simulador, mesmo não tendo nenhum contato com a linguagem Elixir ou Erlang. Portanto, se estiver disposto a estudar as referências deste documento, conseguirá fazer suas simulações PON utilizando o módulo nop_elixir.

Antes de começar

O simulador está desenvolvido em Elixir, uma linguagem funcional desenvolvida sobre a plataforma Erlang/OTP pelo brasileiro José Valim[1], portanto para uma melhor simulação será de grande valia um aprofundamento nesta linguagem. Para os alunos com e-mail no domínio *@alunos.utfpr.edu.br* é possível baixar **gratuitamente** o e-book "Programing Elixir > 1.6" no site Pragmatic BookShelf. Abaixo segue o link para a compra:

https://pragprog.com/book/elixir16/programming-elixir-1-6

Instalando o ambiente de desenvolvimento Elixir

Há disponíveis várias interfaces de desenvolvimento para trabalhar com Elixir. Porém, neste trabalho será apresentado apenas o ambiente do Visual Source Code. Tal IDE foi escolhida por estar disponível tanto para sistemas operacionais Windows, Linux quanto Mac OS. Isto posto, a primeira etapa é a instalação da linguagem e suas dependências. Para tal, basta seguir os guias disponibilizados para sua plataforma pelos mantenedores do elixir que se encontram neste link:

https://elixir-lang.org/install.html

Posteriormente, instalar a interface de desenvolvimento Visual Source Code, cujo link está disponível abaixo:

https://code.visualstudio.com/

[1] Sim, você leu direito! Elixir, junto com Lua compõe as duas linguagens brasileiras já em uso comercial pelo mundo a fora.

Uma vez instalados os componentes acima, abra o VS Code e entre na aba extensões identificada pelo botão  para a instalar as extensões que irá precisar para trabalhar com o elixir, são elas:

- **vscode-elixir**: Instala todo o ambiente de desenvolvimento do elixir
- **vscode-elixir-formatter**: uma ajuda a mais na hora de formatar aquele módulo que ficou grande demais
- **ElixirLS**: Um depurador para aquele momento que você não sabe mais o que fazer (falarei detalhadamente sobre depuração logo mais)

O módulo nop_elixir

O módulo nop_elixir está disponível no bitbucket como um repositório git através do seguinte link:

https://bitbucket.org/fnegrini/nop_elixir/

Entretanto, verá que não é necessário tê-lo baixado para poder desenvolver suas simulações, pois ele está distribuído como um módulo hex de forma que o elixir faz controle automático de dependência e versão para você. Contudo sugiro fortemente uma visita às fontes, pois existem vários exemplos que poderão servir de inspiração, além de ajudar a esclarecer algumas dúvidas que porventura terão.

Primeiro projeto passo-a-passo

O objetivo deste capítulo é guiá-lo em uma caminhada passo-a-passo desde a criação até os testes finais de um módulo de simulação dentro do nop_elixir. Para tanto, segue-se aqui um problema simples que será resolvido no decorrer deste capítulo com o intuito de explorar e explanar os elementos do *framework*. Vale lembrar também que a resolução deste exercício está disponível junto com outros nas fontes git que apresentadas no link do capítulo anterior.

Segue-se então o problema a ser resolvido:

> Carl Allen é um homem que perdeu muitas oportunidades por causa da palavra não. Ele decide ir para um seminário de autoajuda para aprender a dizer sim. Respondendo positivamente a todos os convites e oportunidades, Carl passa por experiências incríveis e logo descobre que as mudanças não precisariam ser tão drásticas (Sinopse do filme Sim Senhor, 2009 com Jim Carrey).
>
> A resolução de nosso problema, de forma mais explícita, é a seguinte: Carl possui o atributo resposta que pode ser Sim ou Não. A resolução de nosso exercício se dá criando uma regra que, sempre que Carl mudar sua resposta, ou seja, mudar seu atributo resposta para não, vamos criar uma regra que volte seu atributo resposta para sim.

Criando um projeto

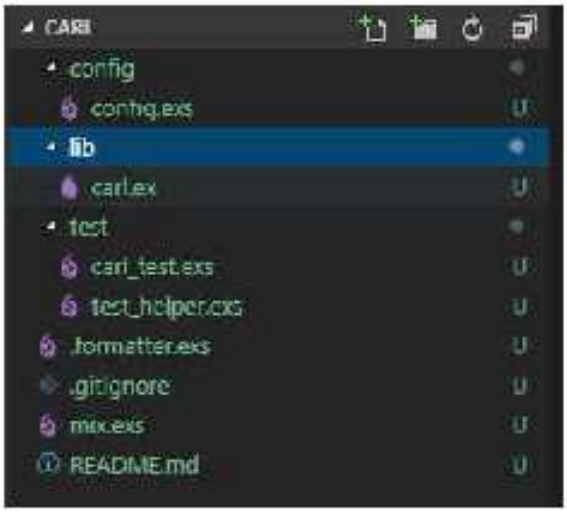
Uma das vantagens de se usar elixir é a criação automática do projeto com uma estrutura de arquivos coesa e organizada. Esta é a única etapa que será feita fora do VS Code através de uma linha de comando (*prompt* para Windows, *shell* para os demais). Abra então seu aplicativo, posicione-se na pasta onde deseja criar o seu projeto e digite o comando:

```
mix new <nome_projeto>
```

Para este documento, o projeto será chamado de *carl*, em homenagem ao nosso protagonista. Portanto o comando ficou assim:

- mix new carl

Em seguida, abra o seu VS Code e informe como pasta principal a pasta carl que acabou de criar. Você deverá ter uma estrutura de pastas como a figura abaixo:



Importante, então, entender o que é cada uma destas pastas:

- **config**: Algumas configurações iniciais. Você pode criar diretivas aqui por exemplo que poderão ser lidas no seu código. Como exemplo, aqui você pode definir o nível de logging que sua aplicação irá trabalhar.
- **lib**: É aqui que você irá codificar. Todos os arquivos que serão criados neste exercício serão colocados dentro desta pasta
- **test**: O mix já cria o projeto com um ambiente TDD configurado. Usarei o ambiente neste documento e recomendo fortemente seu uso em seus projetos.

Utilizando o módulo nop_elixir

O Elixir dispõe de uma plataforma de distribuição muito prática para distribuição de módulos em forma de bibliotecas hex. O módulo nop_elixir já está sendo distribuído desta forma e, como falado anteriormente, não é necessário baixar ou configurar nada para usá-lo em seu

projeto. Para usá-lo, então, é necessário incluí-lo como dependência para seu projeto, conforme descrito a seguir.

Declarando nop_elixir como dependência do projeto

Dentro da raiz do projeto você vai encontrar o arquivo **mix.exs**. No final do arquivo existe a declaração de dependência declarada por **deps**. Inclua a seguinte informação: **{:nop_elixir, "~> 0.0.24"}**

```
defmodule Carl.MixProject do
  use Mix.Project

  def project do
    [
      app: :carl,
      version: "0.1.0",
      elixir: "~> 1.7",
      start_permanent: Mix.env() == :prod,
      deps: deps()
    ]
  end

  # Run "mix help compile.app" to learn about applications.
  def application do
    [
      extra_applications: [:logger, :nop_elixir]
    ]
  end

  # Run "mix help deps" to learn about dependencies.
  defp deps do
    [
      {:nop_elixir, "~> 0.0.14"},
      # {:dep_from_hexpm, "~> 0.3.0"},
      # {:dep_from_git, git: "https://github.com/elixir-lang/my_dep.git", tag: "0.1.0"},
    ]
  end
end
```

Isto fará com que seu projeto dependa do módulo **nop_elixir** com versão mínima 0.0.24. Adicionalmente você pode entrar na página do módulo e verificar qual é a última versão disponível e ajustar a versão mínima do seu projeto. Desta forma o elixir irá mantê-lo atualizado. O link para o módulo hex segue abaixo:

https://hex.pm/packages/nop_elixir

As versões disponíveis estão no canto inferior esquerdo da página do módulo. Recomendo que faça isso e atualize para a última versão antes de prosseguir.

Ao final, sua declaração de dependência deverá ficar assim:

```
  # Run "mix help deps" to learn about dependencies.
  defp deps do
    [
      {:nop_elixir, "~> 0.0.18"},
    ]
```

```
    end
```

Declarando nop_elixir como aplicação

Uma segunda coisa que precisa ser feita antes de iniciar seu desenvolvimento é declarar o nop_elixir como uma aplicação. Desta forma alguns processos do nop_elixir são iniciados junto com sua aplicação garantindo o correto funcionamento do ambiente de simulação. Para isto, no mesmo **mix.exs** localize a definição **application** e inclua o módulo nop_elixir que deverá ficar assim:

```
    def application do
      [
        extra_applications: [:logger, :nop_elixir]
      ]
    end
```

O logger também é importante para apresentar mensagens de debug que foram desenvolvidas no *framework* nop_elixir para avaliação dos ambientes de simulação.

Ativando o Logging no console

Você verá que o Logging no elixir está muito bem resolvido e sem muito esforço poderá definir um dispositivo próprio de saída ou usar um que já tenha sido criado e disponibilizado no hex. Por hora, segue-se a versão básica de configuração e deixar log aparecendo no próprio terminal. Para isso, localize o arquivo **config.exs** na pasta **config** e inclua as linhas abaixo:

```
    config :logger, :console,
    level: :debug, # :error :info :debug
    format: "\n$time $metadata[$level] $levelpad$message\n",
    metadata: [:user_id]
```

A primeira linha (*config)* define que o log será apresentado no console. Já a segunda linha (*level*) define definindo o nível para depuração. Se quiser mudar, basta trocar para os demais níveis que já estão comentados logo ao lado e recompilar tudo. A terceira linha define a formatação do texto de saída. Por último, define que somente logs gerados pelo seu usuário serão apresentados.

Compilando e baixando dependências

Agora o projeto está pronto para ser iniciado. As dependências já foram definidas, basta agora compilar. Para isso, um terminal através do menu **Terminal** do **VS Code** e digite os seguintes comandos:

Comando	O que ele faz
mix deps.get	Faz download e compila as dependências

mix compile	Compila e gera os arquivos para a execução no BEAM
mix test	Executa os testes definidos em carl_test/test.exs

Agora o projeto foi criado e compilado, hora de conhecer melhor o simulador. Para isto, execute os seguintes comando através do **iex**[2]. No terminal, dentro da pasta **carl** inicie o iex carregando o projeto através do seguinte comando:

```
iex -S mix
```

Desta forma seu projeto já foi carregado e, com ele, o framework nop_elixir também. A partir de agora, sempre que me referir ao ambiente interativo **iex**, subentenda que ele deverá ser ativado através deste comando.

Antes de falar dos elementos, um teste rápido? Entre no **iex** com a opção "-S mix" e rode os comandos abaixo:

```
fbe = NOP.Service.FBE.create_fbe(NOP.Element.FBE_dummy, "NOP.element.MyFBE")

NOP.Service.Rule.create_rule(NOP.Element.Rule_dummy3, "NOP.element.Rule",
[fbe])

NOP.Service.FBE.set_attribute(fbe, :value, 5)

:timer.sleep(100)

NOP.Service.FBE.get_attribute(fbe, :value) == 3
```

Se tudo correu bem até aqui, você deve ter visto vários logs aparecendo no seu console. Caso contrário, observe a mensagem de erro e verifique se todas as etapas foram executadas em sua integralidade.

Imergindo nos elementos no framework

O PON é declarativo, portanto um ambiente de simulação também deve ser igualmente declarativo. Claro que há a limitação da linguagem Elixir, mas é possível verificar que os mesmos elementos encontrados na NOPL são igualmente apresentados aqui, tanto que, sempre que possível, será colocado lado-a-lado afim de facilitar a tradução do *framework* com a NOPL. Lembre-se que o objetivo deste framework é permitir a simulação de seu código PON em um ambiente multicore.

Em seguida são apresentados detalhes sobre cada elemento PON convertido para o *framework*.

[2] Iex é a sessão interativa da linguagem Elixir. Para mais detalhes, verifique e-book indicado no início do documento e dê uma lida. Já nas primeiras páginas ele discorre rapidamente sobre a sessão interativa do elixir.

FBE

Antes de falar na FBE, importante falar de algumas funções que estão disponíveis no módulo **NOP.Service.FBE** do nosso framework para enriquecer nosso ambiente de simulação, são elas:

Função	Tipo	Parâmetros	O que faz
create_fbe	Síncrono	• **fbe_type**: Módulo Elixir da FBE • **fbe_name**: Nome da FBE (deve ser único no ambiente) • retorno: PID	Cria um ator FBE com o módulo definido em **fbe_type**
set_attribute	Assíncrono	• **fbe**: ator fbe criado por um **create_fbe** • **atrr_name**: Átomo com o nome do atributo que receberá o novo valor • **attr_value**: Novo valor a ser atribuído em **attr_name.**	Atribui novo valor **attr_value** ao atributo **atrr_name** no ator **fbe.** Esta função dispara o processo de notificações nas **premises** relacionadas ao atributo de nome **attr_name.**
set_attribute_expr	Assíncrono	• **fbe**: ator fbe criado por um **create_fbe** • **atrr_name**: Átomo com o nome do atributo que receberá o novo valor • **expression:** Expressão aritmética cujo resultado será atribuído ao atributo **attr_name**	Função complementar a **set_attribute,** porém permite atribuir um valor calculado a partir de valores de atributos da **fbe**. Para se referenciar a um atributo, utilize o prefixo **@** seguido do nome do valor. Por exemplo se um **fbe** possui um atributo chamado *value* e deseja incrementá-lo em 1 do seu valor atual, utilize: **expression = "@value + 1"**

get_attribute	Síncrono	• **fbe**: ator fbe criado por um **create_fbe** • **atrr_name**: Átomo com o nome do atributo cujo valor será recuperado	Recupera o valor do atributo **attr_name** no ator **fbe**.
get_name	Síncrono	• **fbe**: ator fbe criado por um **create_fbe**	Recupera o nome do ator **fbe**
get_statistics	Síncrono	• **fbe**: ator fbe criado por um **create_fbe**	Recupera o número de notificações recebidas para modificação de atributos
reset_statistics	Assíncrono	• **fbe**: ator fbe criado por um **create_fbe**	Inicializa o contador de notificações recebidas

Voltando então à FBE, para atender a este elemento foi criado o módulo **NOP.Element.FBE,** portanto para se ter uma FBE em seu projeto basta criar um módulo Elixir e incluir o módulo acima através do comando **use.** Dentro do seu módulo será necessário declarar apenas uma função de nome **init_attributes** que será responsável por inicializar sua lista de atributos com valores iniciais em forma de **Map.** Para facilitar o entendimento, segue a solução do problema do Carl, que é exemplificando como seria em NOPL e como ficou no *framework* nop_elixir.

FBE Carl

Apenas relembrando, Carl possui um atributo resposta (answer) que poderá ser sim (yes) ou não (no). Já me adiante no exercício e declarei o método **change_to_yes** que será usado logo à frente no exercício. Portanto, traduzindo isto em FBE em NOPL:

NOPL

```
fbe Carl
  attributes
        integer answer 0
  end_attributes

  methods
        method change_to_yes(answer = 1)
  end_methods
end_fbe
```

A mesma tradução agora feita em nop_elixir ficaria assim:

Simulador nop_elixir

```
defmodule NOP.Carl.FBE do
```

```
  use NOP.Element.FBE

  defp int_attributes() do
    %{:answer => :yes}
  end

  def change_to_yes(carl) do
    NOP.Service.FBE.set_attribute(carl, :answer, :yes)
  end
end
```

Perceba que há uma relação muito próxima entre o NOPL e o ambiente de simulação de forma que é possível, sem grandes dificuldades, fazer uma tradução afim de conseguir simular seu código NOPL dentro do framework. Veja que algumas variações são possíveis, como é o caso de usar os átomos **:yes** e **:no** nos lugares dos valores inteiros **1** e **0** respectivamente. É uma vantagem que deixa o código mais legível e que aconselho o uso sempre que possível.

Hora de criar um arquivo **fbe.ex** dentro da pasta **lib** de nosso projeto e criar nossa FBE Carl (se quiser, pode copiar o código acima). Vamos aproveitar e aproveitar e criar um caso de testes para garantir tudo está funcionando. No arquivo **carl_test.exs** dentro da pasta **test** inclua o seguinte caso de testes.

```
  test "Creating FBE carl" do

    carl =  NOP.Service.FBE.create_fbe(NOP.Carl.FBE, "FBE.Carl")

    assert NOP.Service.FBE.get_name(carl) == "FBE.Carl"

  end
```

Em seguida, abra o terminal e execute um os testes para ver se tudo correu bem.

Premise

As premises não precisam ser definidas com módulos específicos. Aqui há a facilidade de apenas cria-la e usá-la através das funções criadas dentro do módulo **NOP.Service.Premise** do framework. Vamos conhece-las:

Função	Tipo	Parâmetros	O que faz
create_premise	Síncrono	• **premise_name**: Nome da Premise (deve ser único no ambiente) • **fbe**: Ator FBE • **fbe_attr**: Nome do Atributo • **operator**: Operador, valores possíveis(:EQ, :NE, :GE, :LE; GT e :LT)	Cria um ator Premise e ajusta sua premissa como sendo o atributo **fbe_attr** no ator **fbe** na comparação lógica **operator**

		• **operand**: valor constante • retorno: PID	com a constante **operand.**
create_premise	Síncrono	• **premise_name**: Nome da Premise (deve ser único no ambiente) • **fbe_left**: Ator FBE • **fbe_left_attr**: Nome do Atributo • **operator**: Operador, valores possíveis(:EQ, :NE, :GE, :LE; GT e :LT) • **fbe_right:** Ator FBE • **fbe_right_attr**: Nome do Atributo • retorno: PID	Cria um ator Premise e ajusta sua premissa como sendo o atributo **fbe_left_attr** no ator **fbe_left** na comparação lógica **operator** com o atributo **fbe_rigth_atr** no ator **fbe_right**
evaluate_premise	Síncrono	• **premise**: ator premise criado por um **create_premise**	Recupera o valor lógico atual da **premise**
get_name	Síncrono	• **premise**: ator premise criado por um **create_premise**	Recupera o nome do ator **premise**
get_statistics	Síncrono	• **premise**: ator premise criado por um **create_premise**	Recupera o número de notificações recebidas
reset_statistics	Assíncrono	• **premise**: ator premise criado por um **create_premise**	Inicializa o contador notificações

Como nada precisa ser criado, vamos apenas criar um caso de testes para garantir que nossa FBE consegue se comunicar com uma premise corretamente. Agora que você já sabe onde, inclua o seguinte caso de testes abaixo e refaça seus testes para ver se tudo está correto até aqui.

```
      test "FBE carl linking to premise" do

        carl =  NOP.Service.FBE.create_fbe(NOP.Carl.FBE, "FBE.Carl")

        premise  =  NOP.Service.Premise.create_premise("Premise.Is_no",
carl, :answer, :EQ, :no)

        NOP.Service.FBE.set_attribute(carl, :answer, :no)

        :timer.sleep(100)

        assert NOP.Service.Premise.evaluate_premise(premise) == true

      end
```

Condition e Subcondition

No Desenvolvimento do framework o conceito NOPL de **Condition** e **Subcondition** foi incorporado no serviço **FBE.Service.Condition** de forma que sempre que for traduzir qualquer um dos elementos de sua NOPL irá usar o mesmo elemento e já explicarei em detalhes como isto funciona, antes porém vamos às funções que você tem disponível referido módulo:

Função	Tipo	Parâmetros	O que faz
create_condition	Síncrono	• **condition_name**: Nome da Condition (deve ser único no ambiente) • **premise_list**: Lista de atores lógicos (Premise, Condition ou Rule) • **expression**: Expressão a ser avaliada • retorno: PID	Cria um ator Condition e ajusta sua expressão como sendo a expressão lógica **expression** contendo o nome dos atores.
evaluate_condition	Síncrono	• **condition**: ator Condition criado por um **create_condition**	Recupera o valor lógico atual da **condition**
get_name	Síncrono	• **condition**: ator Condtition criado por um **create_condition**	Recupera o nome do ator **condition**
get_statistics	Síncrono	• **condition**: ator Condtition criado por um **create_condition**	Recupera o número de notificações recebidas
reset_statistics	Assíncrono	• **condition**: ator Condtition criado por um **create_condition**	Inicializa o contador de notificações

Como pode perceber facilmente, a **Condition** aceita como entrada qualquer elemento lógico, ou seja, aceita uma **Premise,** outra **Condition** ou até mesmo uma **Rule** (eu sei! Isto não está no NOPL, mas eu não conto para o professor se você não contar). Para um melhor entendimento, veja o caso de testes abaixo. Aproveite e já o coloque no seu projeto para garantirmos que tudo está funcionando bem até aqui.

```
    test "FBE carl linking to premise and condition" do

      carl =  NOP.Service.FBE.create_fbe(NOP.Carl.FBE, "FBE.Carl")

      premise = NOP.Service.Premise.create_premise(
```

```
            "Premise.Is_no", carl, :answer, :EQ, :no)

          condition1 = NOP.Service.Condition.create_condition(
            "Condition.Always_true", [], "true")

          condition2 = NOP.Service.Condition.create_condition(
            "Condition.Expression",
            [premise, condition1],
            "( Premise.Is_no and Condition.Always_true )")

          NOP.Service.FBE.set_attribute(carl, :answer, :no)

          :timer.sleep(100)

          assert NOP.Service.Condition.evaluate_condition(condition2) ==
true

        end
```

Rule

Tal como o FBE, a Rule também é um elemento que deverá ser modularizado em seu projeto afim de que seja declarativo. Antes de falar de como se criar uma *Rule* utilizando o *framework*, seguem os serviços disponíveis em **PON.Service.Rule**:

Função	Tipo	Parâmetros	O que faz
create_rule	Síncrono	• **rule_name**: Nome da Rule (deve ser único no ambiente) • **fbe_list**: Lista de atores fbe que serão usados na criação das Conditions e Instigations • retorno: PID	Cria um ator Rule. O parâmetro **fbe_list** será passado diretamente para as funções **create_element_list** e **create_instigation_list**
evaluate_rule	Síncrono	• **rule**: ator Rule criado por um **create_rule**	Recupera o valor lógico atual da **rule**
get_name	Síncrono	• **rule**: ator Rule criado por um **create_rule**	Recupera o nome do ator **rule**
run_instigations	Assíncrono	• **rule**: ator Rule criado por um **create_rule**	Dispara manualmente as Instigations da Rule independente da condição lógica

get_statistics	Síncrono	• **rule**: ator Rule criado por um **create_rule**	Recupera o número de notificações recebidas
reset_statistics	Assíncrono	• **rule**: ator Rule criado por um **create_rule**	Inicializa o contador de notificações

Infelizmente a linguagem Elixir não permite deixar tão claro a declaração d *Rule* quanto é no NOPL, contudo não é tão difícil. São necessárias as seguintes etapas:

- Criar um módulo e incluir o módulo **NOP.Element.Rule** através do comando **use**
- Declarar uma função **create_element_list** para construir sua expressão lógica. Esta função deverá retornar uma lista de elementos lógicos que serão avaliados de maneira simples: todos deverão ser verdadeiros para que a Rule seja ativada.
- Declarar uma função **create_instigation_list** para construir sua lista de instigações. Esta função deverá retornar uma lista de funções que serão disparadas em paralelo através da função **Task.start,** portanto todos serão executados de forma distribuída e não há garantias de ordem de execução.

Rule Carl

Transcorrendo, então, o problema do Carl. Este se resume em mudar a resposta do Carl para sim (yes) toda vez que ele pensar em mudar sua resposta para não (no). Portanto, antes de partir para a *Rule* no framework, segue-se como ela ficaria em NOPL:

NOPL

```
rule rlAnswerIsNo
  condition
        premise imp prIsNo Carl.answer == 0
  end_condition
  action
        instigation inToYes Carl.change_to_yes();
  end_action
end_rule
```

De forma análoga, tem-se no framework a *Rule* definida da seguinte forma:

```
defmodule NOP.Carl.RuleNo do
  use NOP.Element.Rule

  defp create_element_list([carl]) do

    premise_no = NOP.Service.Premise.create_premise(
      "NOP.Carl.premise.No", carl, :answer, :EQ, :no)

    condition = NOP.Service.Condition.create_condition(
      "NOP.Carl.condition.No", [premise_no], "NOP.Carl.premise.No")

    [condition]
```

```
    end

    defp create_instigation_list([carl]) do

      [{NOP.Carl.FBE, :change_to_yes, [carl]}]

    end

  end
```

Importante, porém, uma observação importante. Como *Rule*, *Condition* e *Premise* são considerados elementos lógicos, o elemento *Condition* passou a ser supérfluo na definição acima. Portanto a *Rule* pode ser "otimizada" para uma definição mais simplificada tal como essa:

```
  defmodule NOP.Carl.RuleNo do
    use NOP.Element.Rule

    defp create_element_list([carl]) do

      premise_no = NOP.Service.Premise.create_premise(
        "NOP.Carl.premise.No", carl, :answer, :EQ, :no)

      [premise_no]

    end

    defp create_instigation_list([carl]) do

      [{NOP.Carl.FBE, :change_to_yes, [carl]}]

    end

  end
```

Agora que está tudo criado de forma declarativa, importante criar mais um caso de testes para garantir que tudo ficou correto até aqui:

```
    test "Create FBE rule and testing Rule" do

      carl  = NOP.Service.FBE.create_fbe(NOP.Carl.FBE, "FBE.Carl")

      NOP.Service.Rule.create_rule(NOP.Carl.RuleNo,
        "Rule.Carl.No",  [carl])

      NOP.Service.FBE.set_attribute(carl, :answer, :no)

      :timer.sleep(100)
```

```
    assert NOP.Service.FBE.get_attribute(carl, :answer) == :yes

  end
```

Veja que o caso de testes altera o atributo **:answer** para **:no** e a **rule** faz com que a mesma seja alterada para **:yes**. Importante lembrar que este teste já está ocorrendo de forma distribuída.

Definindo a função *init*

Mas toda vez que precisar iniciar o ambiente vou precisar declarar Rule e FBE? Não necessariamente. É possível encapsular esta inicialização para facilitar a criação dos ambientes de simulação. A sugestão aqui é a seguinte: no arquivo criado com o mesmo nome do projeto dentro da pasta **lib** (no caso do exemplo, **carl.ex**), criar uma função **init** que cria todo ambiente de maneira completa e retorna a FBE. Este retorno é importante para o caso de fazer alguma operação com a *FBE*. Abaixo, a definição completa da função *init*:

```
  def init(name) do
    carl  = NOP.Service.FBE.create_fbe(NOP.Carl.FBE, "FBE." <> name)

    NOP.Service.Rule.create_rule(NOP.Carl.RuleNo,
      "Rule." <> name,  [carl])

    carl
  end
```

Agora é possível criar o ambiente facilmente através de um único comando. Crie agora um último caso de testes para garantir que isto está funcionando também:

```
  test "Create total envirment" do

    carl = Carl.init("Carl")

    NOP.Service.FBE.set_attribute(carl, :answer, :no)

    :timer.sleep(100)

    assert NOP.Service.FBE.get_attribute(carl, :answer) == :yes

  end
```

Depurando e observando processos

O Elixir possui ferramentas muito boas para depurar e observar seu projeto. Não é objetivo deste documento entrar em maiores detalhes, mas apenas apresentar o mínimo e essencial para utilização em suas simulações.

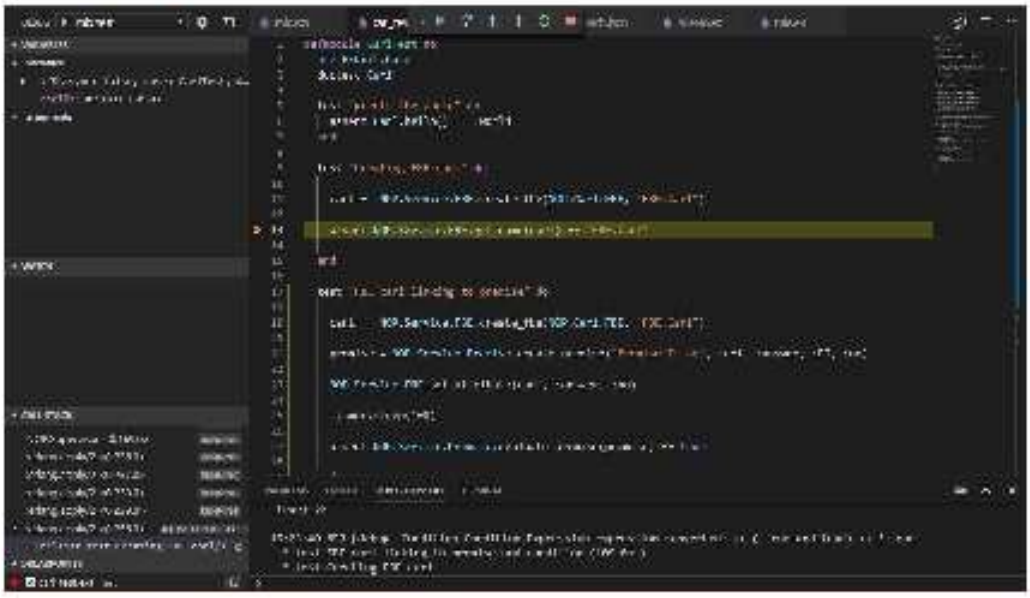
Depuração

A depuração depende no pacote ElixirLS que foi comentado no primeiro capítulo deste documento. Se ainda não deixou configurado, sugiro que volte ao capítulo para então continuar os próximos passos para configurar seu ambiente de depuração.

O primeiro passo é criar um arquivo de configuração para depuração que é feito quase que automaticamente. Entre no ambiente de depuração através do botão  , lá deve haver uma barra de opções informando que não há configurações. Clique nela e escolha a opção **Add configuration.** Esta operação irá criar um arquivo **launch.json** que já deixa configurada a depuração para seus casos de teste. Com isso, então é possível depurar seus casos de teste. Salve o arquivo e faça um teste:

Abra o arquivo **carl_test.exs** na pasta **test**, coloque um break-point (use a tecla F9) e execute em modo depuração com a opção **mix test** para ver o que acontece

Você deverá ver algo assim:



Maiores detalhes podem ser encontrados no site do desenvolvedor aqui:

https://code.visualstudio.com/api/extension-guides/debugger-extension

Importante! Talvez você receba uma notificação de que precisa atualizar sua versão do OTP para que o depurador funcione corretamente. Se isto acontecer, baixe e instale a última versão disponível aqui:

https://www.erlang.org/downloads

Se você usa Windows, talvez seja necessário atualizar o caminho para apontar para a nova biblioteca para que a extensão consiga acessar corretamente a nova versão da OTP.

Observando processos

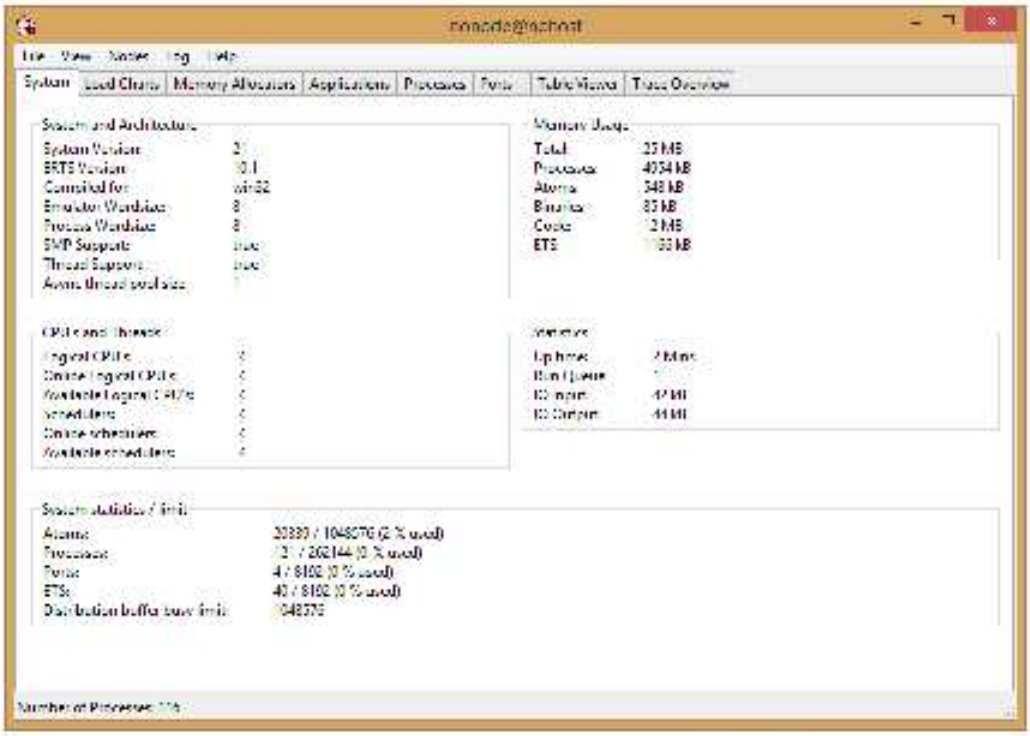
Observar processos é uma ferramenta poderosa para analisar o comportamento de sua simulação. Conforme apresentado, cada elemento da NOPL se torna um ator, que se converte em um processo dentro da máquina virtual BEAM. É possível observar vários atributos, mas para isto é necessário iniciar no modo interativo (**iex**). Inicie o iex carregando o seu projeto (opção **-S** mix) e crie alguns clones de Carl (segue abaixo um script):

```
Carl1 = Carl.init("Carl1")
Carl2 = Carl.init("Carl2")
Carl3 = Carl.init("Carl3")
Carl4 = Carl.init("Carl4")
Carl5 = Carl.init("Carl5")
```

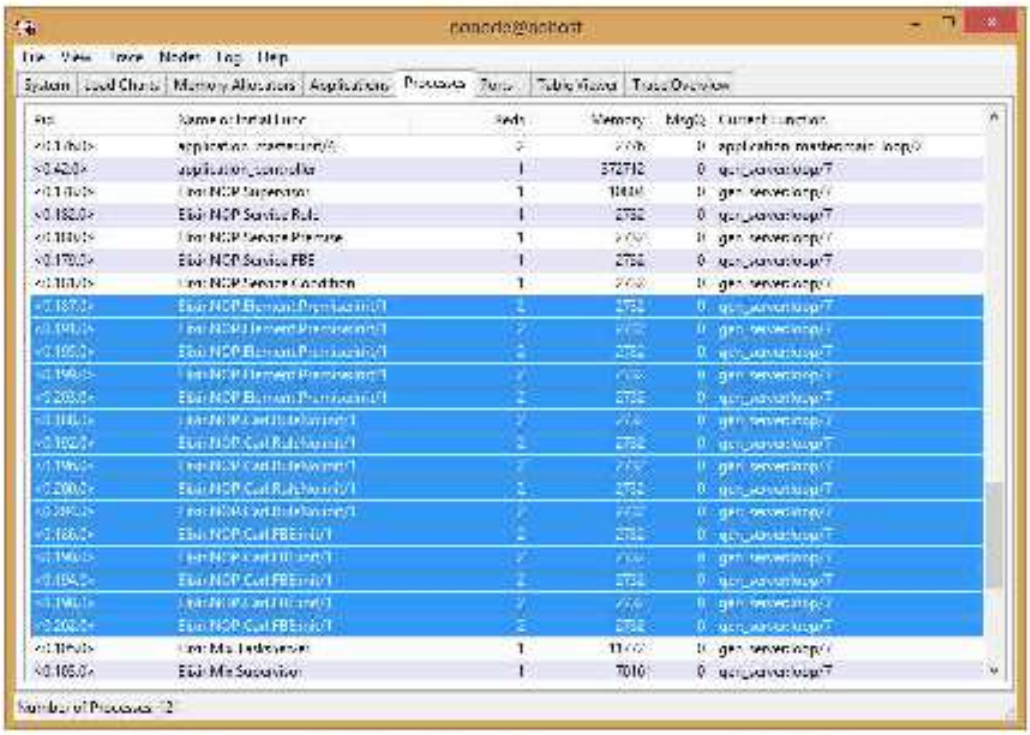
Em seguida abra o **Observer** através do seguinte comando:

```
:observer.start
```

Você verá uma janela como esta



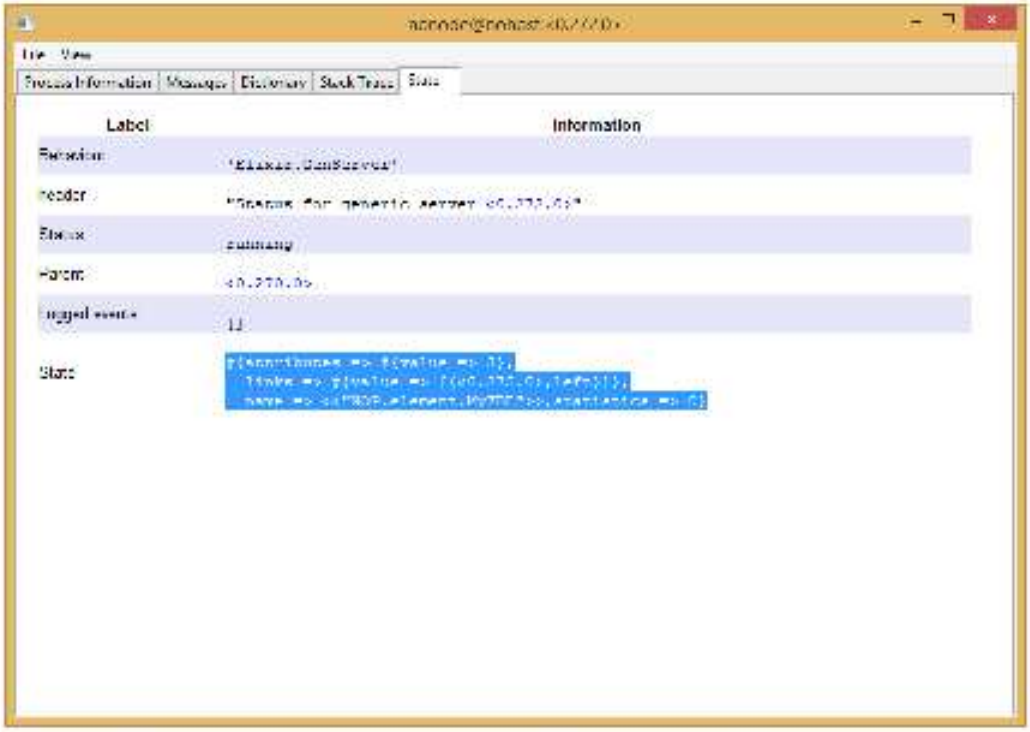
Vá até a aba processos e localize os processos. Eles iniciam com Elixir.NOP.*



Duplo click e você poderá ver detalhes de cada um deles. Chamo a atenção para a aba State, que contém as informações de cada ator. Vou repassar rapidamente cada informação para facilitar o entendimento destas informações

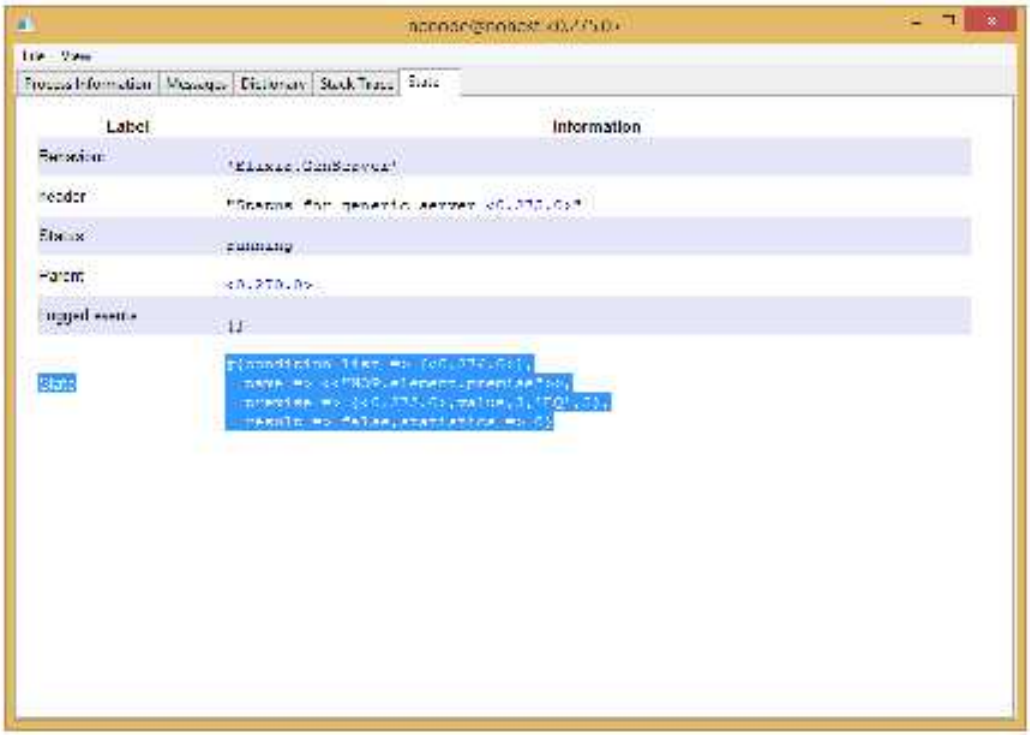
Entendendo o Estado de cada processo

FBE



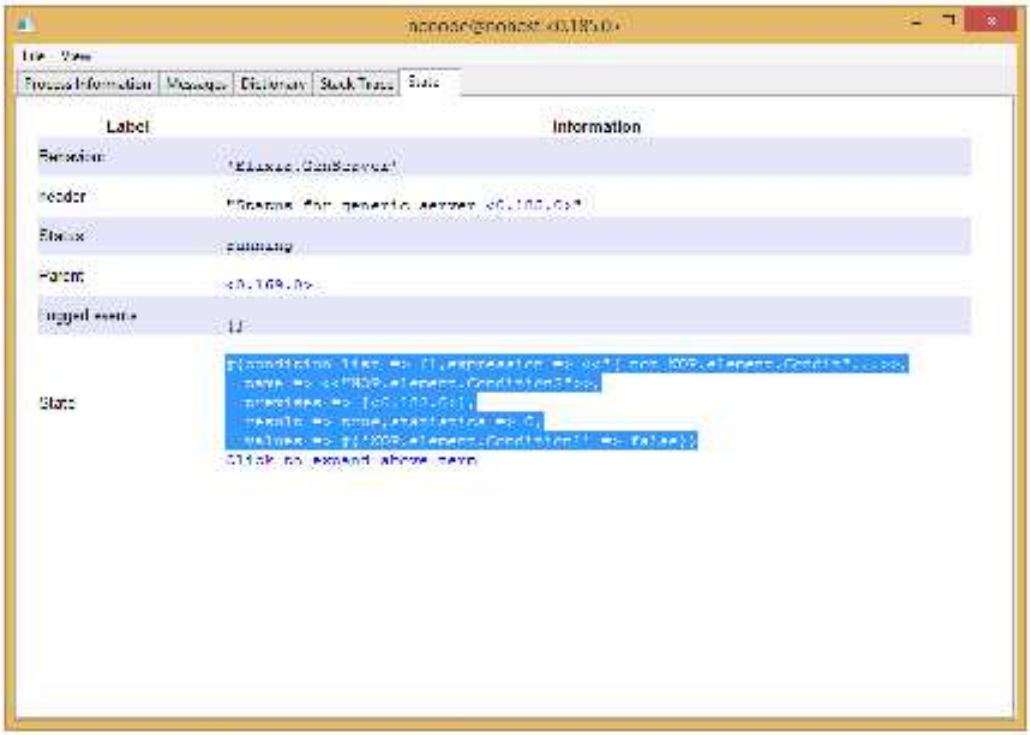
- **atributes**: São os atributos do FBE criados pela função **init_attributes()**
- **links**: São a relação de premises que serão notificadas na mudança do atributo correspondente.
- **name**: O nome da FBE
- **statistics**: O contador acumulado de notificações

Premise



- **condition_list**: Lista de Conditions que serão modificadas quando a Premise alterar seu estado lógico
- **name**: o nome da Premise
- **premise**: a declaração da premise que pode ser:
    - fbe, atributo, valor do atributo, operador operando
    - fbe esq, atributo esq, valor esq, operador, fbe dir, atributo dir, valor dir
- **result**: Estado lógico atual da Premise
- **statistics**: O contador acumulado de notificações

Condition/Subcondition



- **condition_list**: lista de elementos lógicos que serão notificados quando a condição mudar seu valor lógico
- **expression**: a expressão a ser avaliada
- **name**: nome da Condition
- **premises**: lista de elementos lógicos que fazem parte da sua expressão lógica
- **values**: conjunto de valores lógicos atuais de cada premise composto por um Map Nome => valor
- **result**: o valor lógico atual da Condition
- **statistics**: O contador acumulado de notificações

Rule

- **condition_list**: lista de elementos lógicos que serão notificados quando a condição mudar seu valor lógico
- **element_list**: lista de elementos lógicos que são avalidados na rule
- **instigation_list**: lista de funções que serão disparadas em paralelo caso a rule tenha estado lógico verdadeiro
- **values**: conjunto de valores lógicos atuais de cada premise composto por um Map Nome => valor
- **statistics**: O contador acumulado de notificações

Colhendo estatísticas

O framework pon_elixir foi criado pensando em facilitar a coleta notificações, por isto algumas funções estão disponíveis para recuperação em todos os elementos.

Função	Escopo	Descrição
NOP.Service.FBE.get_statistics	FBE	Recupera contador de notificações de um processo específico do tipo FBE.
NOP.Service.Premise.get_statistics	Premise	Recupera contador de notificações de um processo específico do tipo Premise.

NOP.Service.Condition.get_statistics	Condition	Recupera contador de notificações de um processo específico do tipo Condition.
NOP.Service.Rule.get_statistics	Rule	Recupera contador de notificações de um processo específico do tipo Rule
NOP.Service.FBE.reset_statistics	FBE	Inicializa estatísticas contagem de notificações de um processo específico do tipo FBE.
NOP.Service.Premise.reset_statistics	Premise	Inicializa estatísticas contagem de notificações de um processo específico do tipo Premise.
NOP.Service.Condition.reset_statistics	Condition	Inicializa estatísticas contagem de notificações de um processo específico do tipo Condition.
NOP.Service.Rule.reset_statistics	Rule	Inicializa estatísticas contagem de notificações de um processo específico do tipo Rule
NOP.Service.FBE.get_statistics_all	FBE	Recupera soma das notificações de todos os processos do tipo FBE.
NOP.Service.Premise.get_statistics_all	Premise	Recupera soma das notificações de todos os processos do tipo Premise.
NOP.Service.Condition.get_statistics_all	Condition	Recupera soma das notificações de todos os processos do tipo Condition.
NOP.Service.Rule.get_statistics_all	Rule	Recupera soma das

		notificações de todos os processos do tipo Rule
NOP.Service.FBE.reset_statistics_all	FBE (Todas)	Inicializa estatísticas contagem de notificações de todos os processos do tipo FBE.
NOP.Service.Premise.reset_statistics_all	Premise (Todas)	Inicializa estatísticas contagem de notificações de todos os processos do tipo Premise.
NOP.Service.Condition.reset_statistics_all	Condition (Todas)	Inicializa estatísticas contagem de notificações de todos os processos do tipo Condition.
NOP.Service.Rule.reset_statistics_all	Rule (Todas)	Inicializa estatísticas contagem de notificações de todos os processos do tipo Rule
NOP.Application.get_statistics_from_total	Global	Recupera soma das notificações de todos os processos do framework
NOP.Application.reset_statistics_total	Global	Inicializa estatísticas contagem de notificações de todos os processos do framework
NOP.Application.wait_up_to_end_all_process	Global	Aguarda o encerramento de todos os processos. Use para contar tempo de execução de uma simulação.

Dica final para contabilizar tempos

A melhor forma de calcular o tempo real que uma aplicação gasta para sua execução é através da função **NOP.Application.wait_up_to_end_all_process**, pois esta fica inspecionando se há mensagens na fila de execução de todos os processos relacionados a elementos PON e só finaliza sua execução quando todos os processos estão inativos e sem nenhuma mensagem

pendente. É um pequeno *overhead* de processamento, mas tem se demonstrado bem assertivo no cálculo de tempo real de execução. Esta função aliada à função Erlang **:timer.tc** irão lhe trazer os números que precisa para saber como seu algoritmo está se comportando. O pequeno trecho abaixo é um exemplo de como se utilizam estas funções:

```
    NOP.Application.reset_element_list_total()
    {time_in_microseconds, _ret_val} =  :timer.tc(
      fn ->
        NopEx.mtSIMULATE(nil, 2000)
        NOP.Application.wait_up_to_end_all_process()
      end)

    IO.puts("The liquid time is #{div(time_in_microseconds,1000)}")
    IO.puts("The notification count is
#{NOP.Application.get_statistics_from_total()}")
```

Para acelerar seus testes, outra dica é salvar este trecho de código em um arquivo com extensão **.exs**. Com isso você consegue executá-lo com mais facilidade chamando de dentro do **iex** através do comando **"c nome_do_arquivo.exs":**

Agora é sua vez

Perceba que o projeto que criamos juntos no decorrer deste documento foi relativamente simples e em um ambiente altamente distribuído quase não faz sentido. Vamos então tentar fazer um bom proveito deste poderoso ambiente, eis o desafio:

> Crie um ambiente que gere 1000 instâncias de Carl e que, para cada instância de Carl, altere o atributo ***answer*** para ***no*** outras 1000. Cada instância deverá ser executada em um processo paralelo separado.

A solução para este problema está disponível no repositório, na pasta **samples**, mas antes de ir lá ver como se resolve sugiro gastar alguns minutos tentando resolver este problema. Depois, compare sua solução com a solução do repositório e tire suas conclusões.

Ah! Mais uma dica: ao executar seu teste de stress, deslige o *logging*, pois o mesmo é um processo individual que acaba criando um gargalo no seu ambiente de simulação. Para isso, entre no arquivo **config.exs** dentro da pasta **config** e altere o nível para **:error.** Em seguida execute o comando:

- mix deps.compile

Agora é com você! Boas simulações!

Se ainda sim restarem dúvidas, envie-me um e-mail no: negrini@alunos.utfpr.edu.br que eu respondo na medida do possível

APÊNDICE H - Comparativo NOPL Erlang-Elixir com NOPL-Namespaces

O target NOPL-*Namespaces* foi apresentado durante a disciplina de Tópicos Avançados em Engenharia de Software por Athayde et ali (2016). Esta materialização consiste na geração de código alvo compilável em C++ no qual os elementos PON são traduzidos em *Namespaces* desta linguagem. Como consequência, o produto final é bastante estático em termos de ocupação de memória, porém permite integração com outros códigos C++ clássicos. O Código 23 apresenta um exemplo de uma tradução de um *Method* PON em neste compilador.

Código 23. Exemplo de tradução de *Method* PON para target NOPL-*Namespaces*

C++ Namespaces

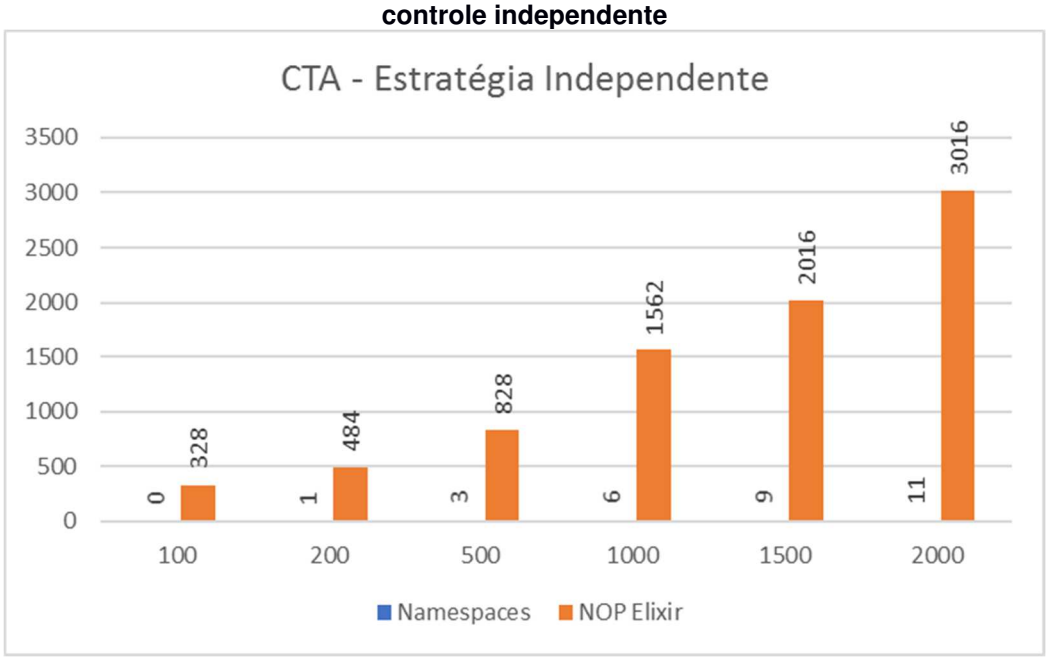
```
namespace method {
    namespace main {
        namespace Semaphore_01_01 {
            namespace mtHorGreen {
                void mtHorGreen() {
                    #ifdef _DEBUG_
                    std::cout << "mtHorGreen" << std::endl;
                    #endif
                    instance::main::Semaphore_01_01::at::atSemaphoreState::setValue(0);
                }
            }
```

Fonte: Autoria própria

Uma vez concluído o desenvolvimento do compilador NOPL Erlang-Elixir, um primeiro comparativo com a compilação NOPL para C++ *Namespaces* tornou-se uma escolha óbvia. Isto porque ambas estavam disponíveis e a compilação do código fonte em NOPL para a estratégia independente, apresentada na no primeiro caso de uso descrito na seção 5.3[37], estava disponível para ambos os compiladores. O resultado do comparativo pode ser visto na Figura 96.

[37] O código fonte para a estratégia de congestionamento facilitado (CBCF), descrito na seção 5.4, apresentou erros de compilação para target NOPL C++ *Namespaces*, impedindo comparativo adicional neste caso de uso.

Figura 96. Comparativo entre NOPL Erlang-Elixir e NOPL-Namespaces para estratégia de controle independente



Fonte: Autoria própria

O gráfico demonstra uma diferença considerável entre os *targets*, sendo que o *target* NOPL-*Namespaces* foi extremamente mais rápido em suas execuções. Esta considerável diferença de desempenho se dá pelo fato de esta solução tratar apenas com lógica em memória estática, com o mínimo de acesso a dispositivos de entrada e saída e sem nenhuma sobrecarga para controle de concorrência como ocorre com o *target* NOPL Erlang-Elixir.

APÊNDICE I - Comparativo *Framework* NOP Elixir com solução *multicore* especialista Elixir

Com objetivo de fornecer valores quantitativos comparativos para os resultados alcançados nos dois experimentos apresentados neste trabalho, foi proposto o desenvolvimento das estratégias de controle independente de congestionamento facilitado em linguagem Elixir. Desta forma, ter-se-ia um valor de referência de um programa especialista.

Estratégia Independente (IND)

Iniciou-se, então, o desenvolvimento da estratégia de tráfego independente, a saber, a mesma estratégia apresentada na seção 5.3 deste trabalho. Todo o tempo gasto no desenvolvimento foi tabulado de forma a quantificar-se o seu esforço. O tempo total pode ser visto na Tabela 20 juntamente com a tabulação do tempo gasto para desenvolvimento do código equivalente PON, disponível no APÊNDICE E.

Tabela 20. Comparativo de tempo de desenvolvimento entre Elixir e NOPL para estratégia de tráfego independente

Elixir		
Data Ini	**Data Fim**	**Tempo**
05/06/2019 11:00	05/06/2019 17:00	6:00:00
05/06/2019 15:00	05/06/2019 19:00	4:00:00
06/06/2019 15:00	06/06/2019 18:00	3:00:00
Total		13:00:00

PON		
Data Ini	**Data Fim**	**Tempo**
07/09/2018 11:00	07/09/2018 12:00	1:00:00
08/09/2018 11:00	08/09/2018 13:00	2:00:00
		0:00:00
Total		3:00:00

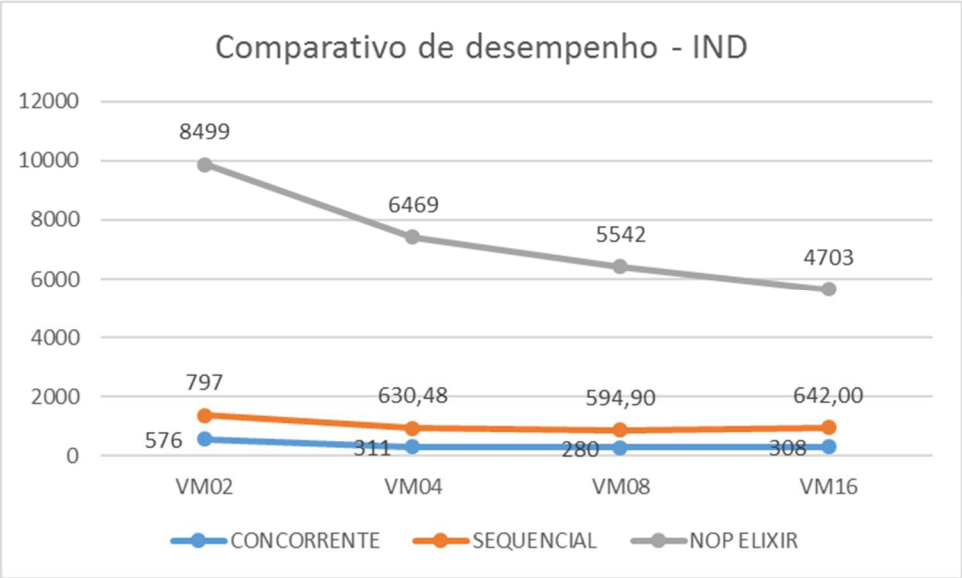
Fonte: Autoria própria

Como é possível observar, o desenvolvimento em linguagem Elixir com objetivo de execução *multicore* ainda foi mais trabalhosa mesmo com todo o ferramental

disponibilizado para este fim. Importante ressaltar que neste tempo não foram considerados tempos de estudo ou testes com bibliotecas e ferramentas.

Finalizado o desenvolvimento, o programa Elixir foi executado nos mesmos ambientes virtuais apresentados na Tabela 13, descritos na seção 5.2. Contudo, o programa foi executado de duas formas distintas: a primeira, chamada de sequencial, no qual o processo principal Elixir se responsabiliza por atualizar os controles de tempo de todos os semáforos de maneira sequencial; e a segunda, chamada de concorrente, no qual é criado um processo para cada semáforo para atualizar os controles de tempo. Cada forma de execução foi repetida cinco vezes e sua média foi tabulada. O resultado pode ser visto na Figura 97.

Figura 97. Comparativo de desempenho entre *Framework* NOP Elixir e programação Elixir especialista para estratégia de tráfego independente



Fonte: Autoria própria

O resultado especialista se apresentou melhor que o *Framework* NOP Elixir. A hipótese para este resultado se dá pelo fato de no *Framework* NOP Elixir ocorrer muito mais mensagens, gerando uma sobrecarga no gerenciamento dos processos Erlang. Como informativo, o Framework gerou 2.861.271 mensagens contra 400.000 na versão especialista.

Estratégia de congestionamento facilitado (CBCF)

Finalizados os comparativos da primeira estratégia, partiu-se para o desenvolvimento em Elixir da estratégia de congestionamento facilitado (CBCF). Como recurso inicial, para ambos os códigos fonte Elixir e NOPL, foi usado o código fonte da primeira estratégia como base. Desta forma, não houve um desenvolvimento do zero, mas sim, um complemento da primeira estratégia.

Portanto, da mesma forma como foi feita na primeira estratégia, também foi computado o tempo de desenvolvimento. O resultado pode ser visto na Tabela 21 juntamente com a tabulação do tempo gasto para desenvolvimento do código equivalente PON, disponível no APÊNDICE F.

Tabela 21. Comparativo de tempo de desenvolvimento entre Elixir e NOPL para estratégia de congestionamento facilitado

Elixir		
Data Ini	**Data Fim**	**Tempo**
15/06/2019 15:30	15/06/2019 17:30	2:00:00
		0:00:00
		0:00:00
Total		2:00:00

PON		
Data Ini	**Data Fim**	**Tempo**
15/09/2018 17:00	15/09/2018 18:40	1:40:00
		0:00:00
		0:00:00
Total		1:40:00

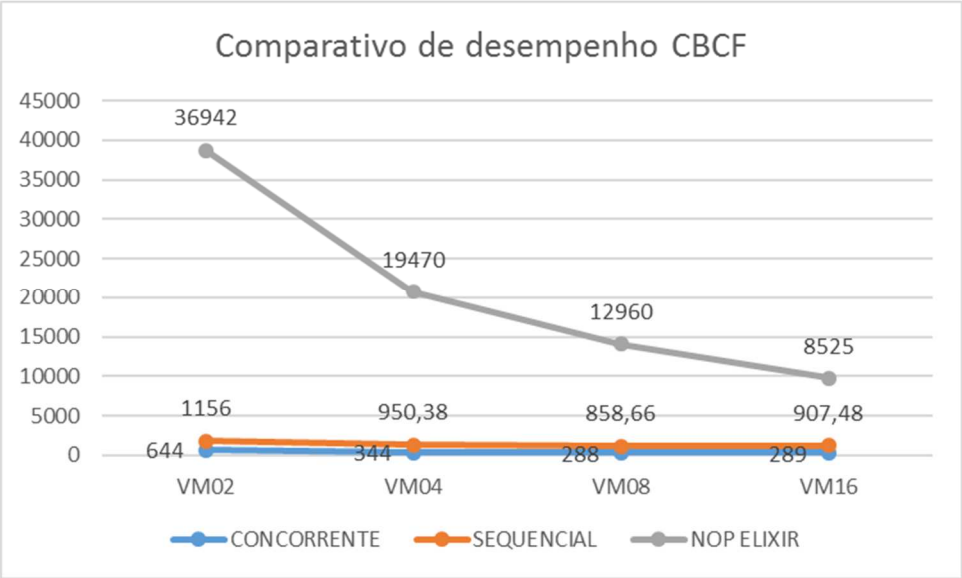
Fonte: Autoria própria

Ao analisar esta tabela, ao contrário do acontecido no desenvolvimento da primeira estratégia, a primeira coisa que se percebe é que os tempos de desenvolvimento são muito próximos. Este fato ocorreu porque, tanto em Elixir, quanto em NOPL, foi aproveitado o código da primeira estratégia como forma de aceleração de desenvolvimento. Portanto, para o desenvolvimento em Elixir, toda a preocupação com paralelismos e concorrência foi já estava resolvida, necessitando apenas o complemento lógico com a inclusão dos sensores. Desta forma, a complexidade dos

desenvolvimentos se equiparou, fato este comprovado na equivalência de tempo de desenvolvimento.

Finalizado o desenvolvimento, tal como na primeira estratégia, o programa Elixir foi executado nos mesmos ambientes virtuais apresentados na Tabela 13 descritos na seção 5.2. Tal como ocorreu na primeira estratégia também, o programa também foi executado de duas formas distintas: a primeira, chamada de sequencial, no qual o processo principal Elixir se responsabiliza por atualizar os controles de tempo de todos os semáforos de maneira sequencial; e a segunda, chamada de concorrente, no qual é criado um processo para cada semáforo para atualizar os controles de tempo. Cada forma de execução foi repetida cinco vezes e sua média foi tabulada. O resultado pode ser visto na Figura 98.

Figura 98. Comparativo de desempenho entre *Framework* NOP Elixir e programação Elixir especialista para estratégia de congestionamento facilitado



Fonte: Autoria própria

Tal como ocorreu na primeira estratégia, o resultado especialista se apresentou melhor que o *Framework* NOP Elixir. A hipótese para este resultado também é a mesma da sua antecessora, ou seja, se dá pelo fato de no *Framework* NOP Elixir ocorrer muito mais mensagens, gerando uma sobrecarga no gerenciamento dos processos Erlang. Como informativo, nesta estratégia o Framework gerou 9.217.687 mensagens contra 400.000 na versão especialista.

APÊNDICE J - Comparativo *Framework* NOP Elixir com (NEGRINI, PORDEUS e SIMÃO, 2019)

As pesquisas nos laboratórios da UTFPR no tocante à comparação de desempenho do paradigma PON em software com demais paradigmas existentes, resultou em uma publicação em folha resumo no Congresso SODEBRAS 2019 (NEGRINI, PORDEUS e SIMÃO, 2019). Posteriormente, o mesmo foi submetido em forma de artigo completo, porém, até a data da publicação deste trabalho, não foi obtido retorno de sua aceitação ao respectivo congresso. Contudo, é certo que a folha resumo terá publicação.

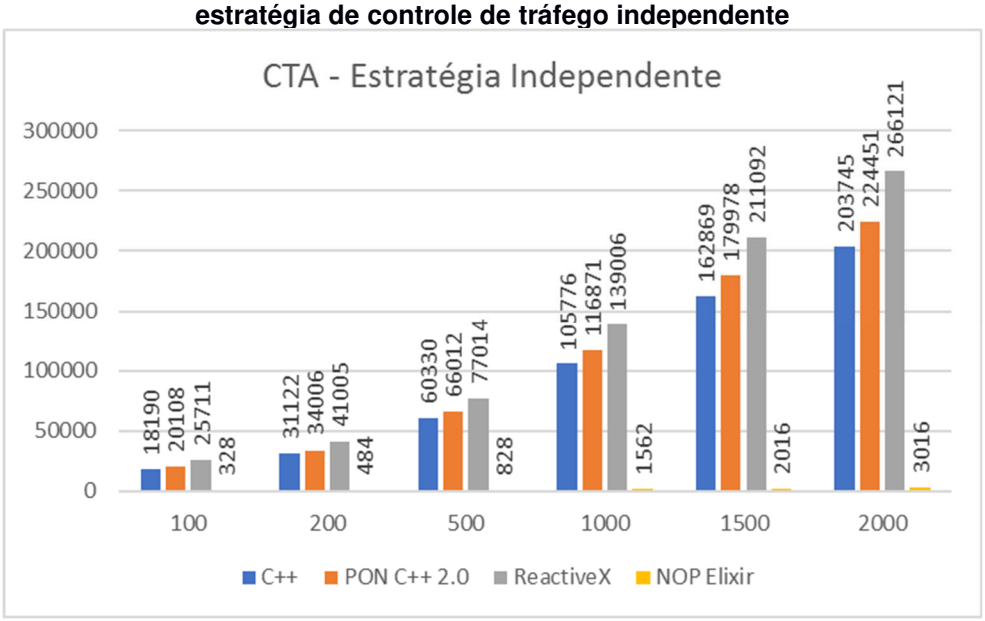
Este trabalho apresenta, em suma, um comparativo de desempenho para avaliação das estratégias de controle de tráfego independente, congestionamento facilitado (CBCF) e trânsito facilitado (CBTF) tal como descritas em (RENAUX, R., *et al.*, 2015). Não por coincidência, as duas primeiras estratégias foram as mesmas utilizadas nos dois casos de estudo apresentados neste trabalho nas seções 5.3 e 5.4. Outrossim, todos os valores apresentados foram computados no mesmo computador do autor, permitindo uma equiparação em termos de hardware para todos os experimentos.

Detalhando ainda mais o trabalho apresentado no resumo, ele compara as estratégias de controle de semáforos à luz de três paradigmas de programação, a saber: C++ clássico, *Framework* PON C++ 2.0 e por meio da biblioteca ReactiveX (REACTIVEX, 2019). Como até o momento não há comparativos do *Framework* NOP Elixir para a terceira estratégia de controle (CBTF), este apêndice faz um comparativo apenas das duas primeiras estratégias, a saber, a estratégia independente e a estratégia de congestionamento facilitado (CBCF).

O comparativo consiste em controlar os semáforos dos cruzamentos de 10 (dez) ruas verticais com 10 (dez) ruas horizontais em vários ciclos de tempo, representando cada segundo um ciclo. Foram dispostos seis conjuntos de repetições, a saber conjuntos de 100, 200, 500, 1000, 1500 e 2000 ciclos. Cada conjunto de repetições foi executado cinco vezes para cada paradigma e coletado o seu menor tempo. Todas as execuções foram feitas em um notebook modelo DELL com processador Core i5 5200U *quadcore* de 2.2 GHZ, 8GB de memória RAM e 512 GM de memória SSD.

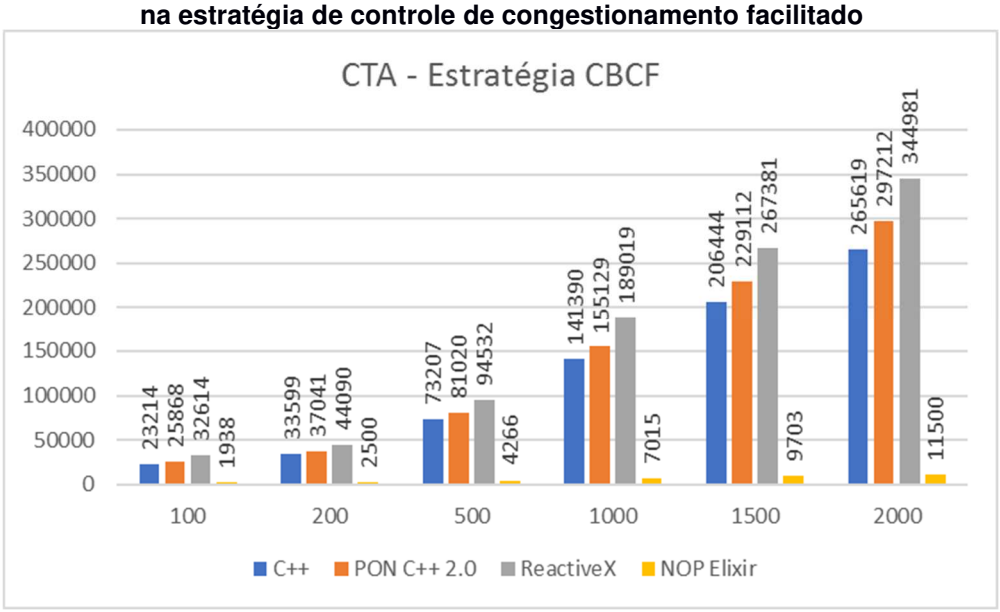
Os resultados podem ser vistos na Figura 99 e Figura 100 respectivamente para as estratégias de controle independente e controle de congestionamento facilitado.

Figura 99. Comparativo do *Framework* NOP Elixir com (NEGRINI, PORDEUS e SIMÃO, 2019) na estratégia de controle de tráfego independente



Fonte: Autoria própria

Figura 100. Comparativo do *Framework* NOP Elixir com (NEGRINI, PORDEUS e SIMÃO, 2019) na estratégia de controle de congestionamento facilitado



Fonte: Autoria própria

Como é possível observar, há uma diferença considerável de desempenho nos resultados do *Framework* NOP Elixir para os demais paradigmas. No tocante à comparação com os paradigmas C++ e *Framework* PON C++ 2.0 a diferença se dá

basicamente pelo fato de as duas primeiras serem serializadas, desta forma não se beneficiaram do ambiente *multicore*[38] como ocorre no *Framework* NOP Elixir. No tocante à biblioteca ReactiveX, uma hipótese é a sobrecarga de tarefas causada por esta biblioteca.

[38] Máquina usada para execução dos testes possuía quatro núcleos.

APÊNDICE K – Código fonte da classe *CodeGenerationElixir*

Arquivo CodeGenerationElixir.h

```
#ifndef _CODE_GENERATION_ELIXIR_H_
#define _CODE_GENERATION_ELIXIR_H_

#define MAIN_FOLDER "nop_ex"
#define MAIN_MODULE "NopEx"

#include "Compiler.h"

class Instance;
class Premise;
class Attribute;
class Condition;
class Rule;
class Method;
class Fbe;
class Subcondition;
class Instigation;

#include <string>
#include <iostream>
#include <fstream>
#include <sstream>
#include <list>
#include <map>

using namespace std;

class CodeGenerationElixir : public Compiler {

    public:

        CodeGenerationElixir();
        virtual ~CodeGenerationElixir();

    private:

        std::map<std::string, Fbe*> fbeList;

        void clearFbeList();

        void addFbeToList(std::string name,  Fbe *fbe);

        std::map<std::string, Fbe*>* getFbeList();

        void generateCodeInstance(Instance *instance, int level);
```

```
    void iterateOverInstances(Instance *instance, int level);

    void createFileStructure();

    void setConfigurationFiles();

    void checkFBE(Fbe *fbe);

    void createFBE(Fbe *fbe, std::string path);

    bool directoryExists(std::string path);

    void checkAgregatedAttributes(std::fstream *filestream, Fbe *fbe);

    void iterateOverAttributes(std::fstream *filestream, Fbe *fbe);

    void generateCodeAttribute(std::fstream *filestream, Attribute *attribute);

    std::string elixirNameofFBE(Fbe *fbe);

    void iterateOverMethods(std::fstream *filestream, Fbe *fbe);

    void generateCodeMethod(std::fstream *filestream, Fbe *fbe, Method *method);

    void iterateOverRules(Fbe *fbe, std::string path);

    void generateCodeRule(Fbe *fbe, std::string path, Rule *rule);

    std::string elixirNameofRule(Fbe *fbe, Rule *rule);

    void iterateOverCondition(std::fstream *filestream, Fbe *fbe, Rule *rule,
      Condition *condition);

    void generateCodeSubcondition(std::fstream *filestream, Fbe *fbe, Rule *rule,
      Subcondition *subcondition);

    void generateCodePremise(std::fstream *filestream, Fbe *fbe, Rule *rule,
      Premise *premise);

    std::string elixirNameofPremise(Fbe *fbe, Rule *rule, Premise *premise);

    std::string elixirNameofSubcondition(Fbe *fbe, Rule *rule,
      Subcondition *subcondition);

    void iterateOverInstigations(std::fstream *filestream, Fbe *fbe, Rule *rule);

    void generateCodeInstigation(std::fstream *filestream, Fbe *fbe, Rule *rule,
      Instigation *instigation);
```

```
        void generateMainRules(std::fstream *filestream, Fbe *fbeMain);

        void iterateOverMainRules(Fbe *fbe, std::string path,
          std::fstream *filestreamMain);

        void generateCodeMainRule(Fbe *fbe, std::string path, Rule *rule,
          std::fstream *filestreamMain);

        void determineMainRuleFbeList(Fbe *fbe, Rule *rule);

        std::string capitalize(std::string word);

    public:

        void generateCode();

    public:


    private:

    std::fstream fileElixir;

};

#endif
```

Arquivo CodeGenerationElixir.cpp

```
#include "generation/elixir/CodeGenerationElixir.h"

#include "NOPGraph.h"

#include "elements/Fbe.h"
#include "elements/Instance.h"
#include "elements/Attribute.h"
#include "elements/Attribution.h"
#include "elements/Method.h"
#include "elements/Premise.h"
#include "elements/Subcondition.h"
#include "elements/Conjunction.h"
#include "elements/Condition.h"
#include "elements/Rule.h"
#include "elements/Action.h"
#include "elements/Instigation.h"
#include "elements/Call.h"
#include "elements/Expression.h"
#include "elements/Factor.h"
#include "elements/ElementFactor.h"
#include "elements/Symbol.h"
#include "elements/Target.h"
#include "elements/Type.h"
#include "elements/Visibility.h"
#include "elements/Param.h"
#include "elements/CodeBlock.h"

#include <map>
#include <iostream>
#include <sys/types.h>
#include <sys/stat.h>

CodeGenerationElixir::CodeGenerationElixir() : Compiler() {

    //

}

CodeGenerationElixir::~CodeGenerationElixir(){

    //

}

void CodeGenerationElixir::createFileStructure(){

    std::cout << "\nGenerating file structure..." << std::endl;

    std::stringstream commandStream;
```

```
    commandStream << "rmdir /Q /S " << MAIN_FOLDER;
    // Eliminate old structure
    system(commandStream.str().c_str());

    // Create project system
    std::stringstream commandStream2;
    commandStream2 << "mix new  " << MAIN_FOLDER;
    system(commandStream2.str().c_str());

}

void CodeGenerationElixir::setConfigurationFiles(){

    std::cout << "\nGenerating configuration file \"mix.exs\"..." << std::endl;

    std::stringstream filenameStream;
    filenameStream << MAIN_FOLDER << "/mix.exs";
    std::fstream filestream;

    filestream.open(filenameStream.str(), std::fstream::out | std::fstream::trunc);

    // Set dependencies and initialize elixir app on mix.exs file
    filestream << "defmodule "<< MAIN_MODULE<< ".MixProject do\n";
    filestream << "  use Mix.Project\n";
    filestream << "\n";
    filestream << "  def project do\n";
    filestream << "    [\n";
    filestream << "      app: :" << MAIN_FOLDER << ",\n";
    filestream << "      version: \"0.1.0\",\n";
    filestream << "      elixir: \"~> 1.7\",\n";
    filestream << "      start_permanent: Mix.env() == :prod,\n";
    filestream << "      deps: deps()\n";
    filestream << "    ]\n";
    filestream << "  end\n";
    filestream << "\n";
    filestream << "  # Run \"mix help compile.app\" to learn about applications.\n";
    filestream << "  def application do\n";
    filestream << "    [\n";
    filestream << "      extra_applications: [:logger, :nop_elixir]\n";
    filestream << "    ]\n";
    filestream << "  end\n";
    filestream << "\n";
    filestream << "  # Run \"mix help deps\" to learn about dependencies.\n";
    filestream << "  defp deps do\n";
    filestream << "    [\n";
    filestream << "      {:nop_elixir, \"~> 0.0.22\"}\n",
    filestream << "    ]\n";
    filestream << "  end\n";
    filestream << "end\n";
```

```
    filestream.flush();
    filestream.close();

    // Get dependencies
    std::stringstream commandStream;
    commandStream << "cd " MAIN_FOLDER << " && mix deps.get";
    system(commandStream.str().c_str());

}


void CodeGenerationElixir::generateCode() {

    std::cout << "\nGenerating elixir project..." << std::endl;

    Instance *mainInstance = graph->getFbeMainInstance();
    createFileStructure();

    setConfigurationFiles();
    // Create main FBe
    Fbe *fbeMain = graph->getFbe("Main");


    std::stringstream filenameStream;
    filenameStream << MAIN_FOLDER << "/lib/" << MAIN_FOLDER << ".ex";
    fileElixir.open(filenameStream.str(), std::fstream::out | std::fstream::trunc);

    fileElixir << "defmodule " << MAIN_MODULE << " do\n";
    fileElixir << "  @moduledoc \"\"\"\n";
    fileElixir << "  Documentation for " << MAIN_MODULE <<".\n";
    fileElixir << "  \"\"\"\n";
    fileElixir << "\n";
    fileElixir << "  @doc \"\"\"\n";
    fileElixir << "  Hello world.\n";
    fileElixir << "\n";
    fileElixir << "  ## Examples\n";
    fileElixir << "\n";
    fileElixir << "      iex> "<< MAIN_MODULE << ".hello()\n";
    fileElixir << "      :world\n";
    fileElixir << "\n";
    fileElixir << "  \"\"\"\n";
    fileElixir << "  def hello do\n";
    fileElixir << "    :world\n";
    fileElixir << "  end\n";
    fileElixir << "\n";
    iterateOverMethods(&fileElixir, fbeMain);
    fileElixir << "\n";
    fileElixir << "  def initialize() do\n";
    fileElixir << "    instances = %{}\n";
    generateCodeInstance(mainInstance, 0);
```

```
    fileElixir << "\n";
    fileElixir << "    create_main_rules(instances)\n";
    fileElixir << "\n";
    fileElixir << "    instances\n";
    fileElixir << "  end\n";
    fileElixir << "  def create_main_rules(instances) do\n";
    fileElixir << "\n";
    generateMainRules(&fileElixir, fbeMain);
    fileElixir << "\n";
    fileElixir << "  end\n";
    fileElixir << "end\n";
    fileElixir << "\n";

    fileElixir.flush();
    fileElixir.close();

    // Compile project
    std::stringstream commandStream;
    commandStream << "cd " MAIN_FOLDER << " && mix compile";
    system(commandStream.str().c_str());

    // Run self test
    std::stringstream commandStream2;
    commandStream2 << "cd " MAIN_FOLDER << " && mix test";
    system(commandStream2.str().c_str());

    std::cout << "\n Run command " << MAIN_MODULE <<
      ".initialize() to start NOP environment..." << std::endl;
}

void CodeGenerationElixir::generateMainRules(std::fstream *filestream, Fbe *fbeMain){

    std::stringstream ss;
    ss << MAIN_FOLDER << "/lib/" /*<< fbe->getName()*/;
    std::string path = ss.str();

    iterateOverMainRules(fbeMain, path, filestream);

}

void CodeGenerationElixir::iterateOverMainRules(Fbe *fbe, std::string path,
  std::fstream *filestreamMain){

    std::map<std::string, Rule*> *rules = fbe->getRules();

    for (std::map<std::string, Rule*>::iterator it = rules->begin();
      it != rules->end(); ++it) {

        Rule *rule = it->second;
```

```
        generateCodeMainRule(fbe, path, rule, filestreamMain);

    }

}

void CodeGenerationElixir::generateCodeMainRule(Fbe *fbe, std::string path, Rule *rule,
  std::fstream *filestreamMain){

    std::cout << "\nInterpreting Rule " << rule->getName() << " of FBE " <<
      fbe->getName() << " as an actor" << std::endl;
    // Create Rule File
    std::stringstream filenameStream;
    filenameStream << path << "/" << rule->getName() << ".ex";
    std:string filename = filenameStream.str();
    std::fstream filestream;
    std::string rule_param;
    std::string rule_param_call;

    clearFbeList();
    determineMainRuleFbeList(fbe, rule);
    rule_param = "[";
    rule_param_call = "[";

    for (std::map<std::string, Fbe*>::iterator it = fbeList.begin();
      it != fbeList.end(); ++it) {

        rule_param += it->first + ", ";

        rule_param_call += "instances[:" + it->first + "], ";

    }

    rule_param += "]";
    rule_param_call += "]";

    // Initialization call
    *filestreamMain << "    NOP.Service.Rule.create_rule(" <<
       elixirNameofRule(fbe, rule);
    *filestreamMain << ", \"" << elixirNameofRule(fbe, rule) << "\", [" <<
       rule_param_call << "])\n\n";

    filestream.open(filename, std::fstream::out | std::fstream::trunc);
    filestream << "defmodule "<< elixirNameofRule(fbe, rule) << " do\n";
    filestream << "  use NOP.Element.Rule\n";
    filestream << "\n";
    filestream << "  defp create_element_list([" << rule_param << "]) do\n";
    filestream << "\n";

    Condition *condition = rule->getCondition();
```

```
    iterateOverCondition(&filestream, fbe, rule, condition);

    filestream << "  end\n";
    filestream << "\n";
    filestream << "  defp create_instigation_list([" << rule_param << "]) do\n";
    filestream << "\n";
    filestream << "    [\n";

    iterateOverInstigations(&filestream, fbe, rule);

    filestream << "    ]\n";
    filestream << "\n";
    filestream << "  end\n";
    filestream << "\n";
    filestream << "end\n";
    filestream.flush();
    filestream.close();

}

bool CodeGenerationElixir::directoryExists(std::string path){
    struct stat info;

    if( stat( path.c_str(), &info ) != 0 )
        return false;
    else if( info.st_mode & S_IFDIR )  // S_ISDIR() doesn't exist on my windows
        return true;
    else
        return false;
}

void CodeGenerationElixir::checkAgregatedAttributes(std::fstream *filestream, Fbe *fbe) {

    std::map<std::string, Attribute*> *attributes = fbe->getAttributes();

    for (std::map<std::string, Attribute*>::iterator it =
      attributes->begin(); it != attributes->end(); ++it) {

        Attribute *attribute = it->second;

        //TODO: Generate a creation process like:
        // <attribute_name> = NOP.Service.FBE.create_fbe(Elixir.<FBE_TYPE>.FBE, "atribute
_name")

    }

}

void CodeGenerationElixir::iterateOverAttributes(std::fstream *filestream, Fbe *fbe) {
```

```
    std::map<std::string, Attribute*> *attributes = fbe->getAttributes();

    for (std::map<std::string, Attribute*>::iterator it = attributes->begin();
      it != attributes->end(); ++it) {

        Attribute *attribute = it->second;

        generateCodeAttribute(filestream, attribute);

    }

}

void CodeGenerationElixir::generateCodeAttribute(std::fstream *filestream,
  Attribute *attribute) {

    int typeID = attribute->getType()->getTypeId();

    if((typeID==Type::STRING_TYPE)||(typeID==Type::CHAR_TYPE)){
        *filestream << "        :" << attribute->getName() << " => \"" <<
          attribute->getFactor()->getStringValue() << "\",\n";
    }else{
        *filestream << "        :" << attribute->getName() << " => " <<
          attribute->getFactor()->getStringValue() << ",\n";
    }


}

std::string CodeGenerationElixir::capitalize(std::string word){
    std::string capitalized = word;
    capitalized[0] = toupper(capitalized[0]);
    return capitalized;
}

std::string CodeGenerationElixir::elixirNameofFBE(Fbe *fbe){
    std::stringstream s;
    s << MAIN_MODULE << "." << capitalize(fbe->getName());
    return s.str();
}

void CodeGenerationElixir::iterateOverMethods(std::fstream *filestream, Fbe *fbe){

    std::map<std::string, Method*> *methods = fbe->getMethods();

    for (std::map<std::string, Method*>::iterator it = methods->begin();
      it != methods->end(); ++it) {

        Method *method = it->second;
```

```
        generateCodeMethod(filestream, fbe, method);

    }

}

void CodeGenerationElixir::generateCodeMethod(std::fstream *filestream, Fbe *fbe,
  Method *method) {

    std::map<std::string, Attribution*> *attributions = method->getAttributions();

    std::string paramsCode = "this";

    // Params
    std::map<std::string, Param*> *params = method->getParams();
    for (std::map<std::string, Param*>::iterator it = params->begin();
      it != params->end(); ++it) {

        Param *param = it->second;
        paramsCode = paramsCode +  ", " + param->getName();

    }

    *filestream << "  def " << method->getName() << "(" << paramsCode << ") do \n";

    // Attribution
    for (std::map<std::string, Attribution*>::iterator it = attributions->begin();
      it != attributions->end(); ++it) {

        Attribution *attribution = it->second;

        *filestream << "    NOP.Service.FBE.set_attribute(" <<
           attribution->getElement()->getInstanceName() <<", :" <<
           attribution->getElement()->getAttributeName() << ", " <<
           attribution->getFactor()->getStringValue() << ")\n";

    }

    // Code
    std::map<std::string, CodeBlock*> *codeBlocks = method->getCodeBlocks();
    for (std::map<std::string, CodeBlock*>::iterator it = codeBlocks->begin();
      it != codeBlocks->end(); ++it) {

        CodeBlock *codeBlock = it->second;

        if(codeBlock->getTarget()->getTargetId() == Target::ELIXIR_TARGET){
            *filestream << std::endl;
            *filestream << codeBlock->getCode() << std::endl;
            *filestream << std::endl;
```

```
        }
    }

    *filestream << "  end\n";

}

void CodeGenerationElixir::createFBE(Fbe *fbe, std::string path){

    std::cout << "\nInterpreting FBE " << fbe->getName() << " as an actor" << std::endl;

    // Create folder
    std::stringstream comandStream;
    comandStream << "cd "<< MAIN_FOLDER << "/lib && mkdir " << fbe->getName();
    std::string command = comandStream.str();
    system(command.c_str());

    // Create FBE Definition
    std::stringstream filenameStream;
    filenameStream << path << "/fbe.ex";
    std:string filename = filenameStream.str();
    std::fstream filestream;

    filestream.open(filename, std::fstream::out | std::fstream::trunc);

    // Set dependencies and initialize elixir app on mix.exs file
    filestream << "defmodule " << elixirNameofFBE(fbe) << " do\n";
    filestream << "  use NOP.Element.FBE\n";
    filestream << "\n";
    filestream << "  defp int_attributes() do\n";

    checkAgregatedAttributes(&filestream, fbe);

    filestream << "    %{\n";

    iterateOverAttributes(&filestream, fbe);

    filestream << "    }\n";
    filestream << "  end\n";
    filestream << "\n";

    iterateOverMethods(&filestream, fbe);

    filestream << "\n";
    filestream << "  def initialize(name) do\n";
    filestream << "\n";
    filestream << "    this = NOP.Service.FBE.create_fbe(" <<
      elixirNameofFBE(fbe) << ", name)\n";
    filestream << "\n";
```

```
    std::map<std::string, Rule*> *rules = fbe->getRules();
    for (std::map<std::string, Rule*>::iterator it = rules->begin();
      it != rules->end(); ++it) {
        Rule *rule = it->second;
        filestream << "    NOP.Service.Rule.create_rule(" << elixirNameofRule(fbe, rule);
        filestream << ", \"" << elixirNameofRule(fbe, rule) << "\", [this])\n\n";
    }

    filestream << "    this\n";
    filestream << "  end\n";
    filestream << "\n";
    filestream << "end\n";
    filestream.flush();
    filestream.close();

    // Crete Rule Definition - one file per Rule
    iterateOverRules(fbe, path);
}

void CodeGenerationElixir::checkFBE(Fbe *fbe){

    std::stringstream ss;
    ss << MAIN_FOLDER << "/lib/" << fbe->getName();
    std::string path = ss.str();

    //Skip Main FBE Creation
    if(!strcmp(fbe->getName().c_str(),"Main"))
        return;

    //Check if FBE implemented: check if respective folder exists
    if(!directoryExists(path)){
        createFBE(fbe, path);
    }

}

void CodeGenerationElixir::generateCodeInstance(Instance *instance, int level) {

    Fbe *fbe = instance->getFbe();

    checkFBE(fbe);

    //Skip Main FBE Creation
    if(strcmp(instance->getName().c_str(),"this")){
        fileElixir << "    instances = Map.put(instances, :" <<
          instance->getName() << ", " << elixirNameofFBE(fbe) <<
          ".initialize(\"" << instance->getName() << "\"))\n\n";
    }

    level++;
```

```
    iterateOverInstances(instance, level);

}

void CodeGenerationElixir::iterateOverInstances(Instance *instance, int level) {

    std::map<std::string, Instance*> *instances = instance->getInstances();

    for (std::map<std::string, Instance*>::iterator it = instances->begin();
      it != instances->end(); ++it) {

        Instance *instance = it->second;

        // TODO - GAMBI
        if (instance->getName() != "this") {

            generateCodeInstance(instance, level);

        }

    }

}

std::string CodeGenerationElixir::elixirNameofRule(Fbe *fbe, Rule *rule){
    std::stringstream s;
    s << MAIN_MODULE << "." <<  capitalize(fbe->getName()) << "." <<
      capitalize(rule->getName()) ;
    return s.str();
}

std::string CodeGenerationElixir::elixirNameofPremise(Fbe *fbe, Rule *rule,
  Premise *premise){
    std::stringstream s;
    s << MAIN_MODULE << "." <<  capitalize(fbe->getName()) << "." <<
      capitalize(rule->getName()) << "." << capitalize(premise->getName()) ;
    return s.str();
}

void CodeGenerationElixir::generateCodePremise(std::fstream *filestream, Fbe *fbe,
  Rule *rule, Premise *premise){

    std::cout << "\nInterpreting Premise " << premise->getName() << " of rule " <<
      fbe->getName() << "." << rule->getName() << " as an actor" << std::endl;

    *filestream << "    " << premise->getName() <<
      " = NOP.Service.Premise.create_premise(";
    *filestream << "\"" << elixirNameofPremise(fbe, rule, premise) << "\"";
```

```
    Expression *expression = premise->getExpression();
    Factor *leftFactor = expression->getLeftFactor();

    if (leftFactor->getFactorId() == Factor::ELEMENT_FACTOR) {

        ElementFactor *elementLeft = (ElementFactor*)leftFactor;

        *filestream << ", " << elementLeft->getInstanceName() << ", :" <<
          elementLeft->getAttributeName() << ", ";

    }

    Symbol *symbol = expression->getSymbol();

    switch (symbol->getSymbolId())
    {
        case Symbol::EQUAL_SYMBOL:
            *filestream << ":EQ, ";
            break;

        case Symbol::NOT_EQUAL_SYMBOL:
            *filestream << ":NE, ";
            break;

        case Symbol::LESSER_THAN_SYMBOL:
            *filestream << ":LT, ";
            break;

        case Symbol::GREATER_THAN_SYMBOL:
            *filestream << ":GT, ";
            break;

        case Symbol::LESS_OR_EQUAL_SYMBOL:
            *filestream << ":LE, ";
            break;

        case Symbol::GREATER_OR_EQUAL_SYMBOL:
            *filestream << ":GE, ";
            break;

        default:
            *filestream << ":EQ, ";
            break;
    }


    Factor *rightFactor = expression->getRightFactor();

    if (rightFactor->getFactorId() == Factor::ELEMENT_FACTOR) {
```

```
        ElementFactor *elementRight = (ElementFactor*)leftFactor;

        *filestream << elementRight->getInstance()->getName() << ", :" <<
          elementRight->getAttribute()->getName() << ")\n";

    }else{
        *filestream << rightFactor->getStringValue() << ")\n";
    }

    //New line
    *filestream << "\n";

}

std::string CodeGenerationElixir::elixirNameofSubcondition(Fbe *fbe, Rule *rule,
  Subcondition *subcondition){
    std::stringstream s;
    s << MAIN_MODULE << "." <<  capitalize(fbe->getName()) << "." <<
      capitalize(rule->getName()) << "." << capitalize(subcondition->getName()) ;
    return s.str();
}


void CodeGenerationElixir::generateCodeSubcondition(std::fstream *filestream, Fbe *fbe,
  Rule *rule, Subcondition *subcondition) {

    std::cout << "\nInterpreting Subcondition " << subcondition->getName() <<
      " of rule " << fbe->getName() << "." << rule->getName() <<
      " as an actor" << std::endl;

    std::map<std::string, Premise*> *premises = subcondition->getPremises();

    std::stringstream conditionStream;
    std::stringstream expressionStream;
    std::string conjuctionStr;
    bool first = true;

    Conjunction *conjunction = subcondition->getConjunction();

    switch (conjunction->getConjunctionId())
    {
        case Conjunction::NO_CONJUNCTION:
            conjuctionStr = " and not ";
            break;

        case Conjunction::AND_CONJUNCTION:
            conjuctionStr = " and ";
            break;

        case Conjunction::OR_CONJUNCTION:
```

```
            conjuctionStr = " or ";
            break;

        default:
            break;
    }

    for (std::map<std::string, Premise*>::iterator it = premises->begin();
      it != premises->end(); ++it) {

        Premise *premise = it->second;

        generateCodePremise(filestream, fbe, rule, premise);

        conditionStream << premise->getName() << ", ";

        if (first){
            expressionStream << elixirNameofPremise(fbe, rule, premise);
            first = false;
        }else{
            expressionStream << conjuctionStr << elixirNameofPremise(fbe, rule, premise);
        }

    }

    *filestream << "    " << subcondition->getName() <<
      " = NOP.Service.Condition.create_condition(";
    *filestream << "\"" << elixirNameofSubcondition(fbe, rule, subcondition) << "\"";
    *filestream << ",  [" << conditionStream.str() << "],";
    *filestream << "\"( " << expressionStream.str() << " )\"";
    *filestream << ")\n";
    *filestream << "\n";

}

void CodeGenerationElixir::iterateOverCondition(std::fstream *filestream, Fbe *fbe,
  Rule *rule, Condition *condition){

    std::stringstream ruleStream;
    std::map<std::string, Subcondition*> *subconditions = condition->getSubconditions();

    for (std::map<std::string, Subcondition*>::iterator it = subconditions->begin();
      it != subconditions->end(); ++it) {

        Subcondition *subcondition = it->second;

        generateCodeSubcondition(filestream, fbe, rule, subcondition);

        ruleStream << subcondition->getName() << ",";
    }
```

```
    std::map<std::string, Premise*> *premises = condition->getPremises();

    for (std::map<std::string, Premise*>::iterator it = premises->begin();
      it != premises->end(); ++it) {

        Premise *premise = it->second;

        generateCodePremise(filestream, fbe, rule, premise);

        ruleStream << premise->getName() << ",";
    }

    *filestream << "    [" << ruleStream.str() << "]\n";
}

void CodeGenerationElixir::generateCodeInstigation(std::fstream *filestream, Fbe *fbe,
  Rule *rule, Instigation *instigation){

    std::list<Call*> *calls = instigation->getCalls();

    for (std::list<Call*>::iterator it = calls->begin(); it != calls->end(); ++it) {

        Call *call = ((Call*)*it);

        // raising exception
        *filestream << "      {" << elixirNameofFBE(call->getInstance()->getFbe()) <<
          ", :" << call->getMethodName() << ", [" << call->getInstanceName() << "]},\n";

        //*filestream << "      {" << elixirNameofFBE(fbe) << ", :" << call-
>getMethodName() << ", [" << "this" << "]},\n";

    }
}

void CodeGenerationElixir::iterateOverInstigations(std::fstream *filestream, Fbe *fbe, Ru
le *rule){

    Action *action = rule->getAction();

    std::map<std::string, Instigation*> *instigations = action->getInstigations();

    for (std::map<std::string, Instigation*>::iterator it = instigations->begin();
      it != instigations->end(); ++it) {

        Instigation *instigation = it->second;

        generateCodeInstigation(filestream, fbe, rule, instigation);

    }
```

```
}

void CodeGenerationElixir::generateCodeRule(Fbe *fbe, std::string path, Rule *rule){

    std::cout << "\nInterpreting Rule " << rule->getName() << " of FBE " <<
      fbe->getName() << " as an actor" << std::endl;
    // Create Rule File
    std::stringstream filenameStream;
    filenameStream << path << "/" << rule->getName() << ".ex";
    std:string filename = filenameStream.str();
    std::fstream filestream;

    filestream.open(filename, std::fstream::out | std::fstream::trunc);
    filestream << "defmodule "<< elixirNameofRule(fbe, rule) << " do\n";
    filestream << "  use NOP.Element.Rule\n";
    filestream << "\n";
    filestream << "  defp create_element_list([this]) do\n";
    filestream << "\n";

    Condition *condition = rule->getCondition();

    iterateOverCondition(&filestream, fbe, rule, condition);

    filestream << "  end\n";
    filestream << "\n";
    filestream << "  defp create_instigation_list([this]) do\n";
    filestream << "\n";
    filestream << "    [\n";

    iterateOverInstigations(&filestream, fbe, rule);

    filestream << "    ]\n";
    filestream << "\n";
    filestream << "  end\n";
    filestream << "\n";
    filestream << "end\n";
    filestream.flush();
    filestream.close();

}
void CodeGenerationElixir::iterateOverRules(Fbe *fbe, std::string path) {

    std::map<std::string, Rule*> *rules = fbe->getRules();

    for (std::map<std::string, Rule*>::iterator it = rules->begin();
      it != rules->end(); ++it) {

        Rule *rule = it->second;

        generateCodeRule(fbe, path, rule);
```

```
    }

}

void CodeGenerationElixir::clearFbeList(){

    fbeList.clear();
}

void CodeGenerationElixir::addFbeToList(std::string name,  Fbe *fbe){


    if (fbeList.count(name) == 0) {
        fbeList.insert(std::pair<std::string, Fbe*>(name, fbe));
    }

}

std::map<std::string, Fbe*>* CodeGenerationElixir::getFbeList(){

    return &fbeList;
}

void CodeGenerationElixir::determineMainRuleFbeList(Fbe *fbe, Rule *rule){

    Condition *condition = rule->getCondition();

    std::map<std::string, Subcondition*> *subconditions = condition->getSubconditions();

    for (std::map<std::string, Subcondition*>::iterator it = subconditions->begin();
      it != subconditions->end(); ++it) {

        Subcondition *subcondition = it->second;

        Conjunction *conjunction = subcondition->getConjunction();

        std::map<std::string, Premise*> *premises = subcondition->getPremises();

        for (std::map<std::string, Premise*>::iterator it = premises->begin();
          it != premises->end(); ++it) {

            Premise *premise = it->second;

            Expression *expression = premise->getExpression();

            Factor *leftFactor = expression->getLeftFactor();

            if (leftFactor->getFactorId() == Factor::ELEMENT_FACTOR) {
```

```
            ElementFactor *elementLeft = (ElementFactor*)leftFactor;
            addFbeToList(elementLeft->getInstanceName(), nullptr);
        }
        Factor *rightFactor = expression->getRightFactor();
        if (rightFactor->getFactorId() == Factor::ELEMENT_FACTOR) {
            ElementFactor *elementRight = (ElementFactor*)rightFactor;
            addFbeToList(elementRight->getInstanceName(), nullptr);
        }

    }

}
std::map<std::string, Premise*> *premisesCond = condition->getPremises();
for (std::map<std::string, Premise*>::iterator it = premisesCond->begin();
  it != premisesCond->end(); ++it) {
    Premise *premise = it->second;
    Expression *expression = premise->getExpression();
    Factor *leftFactor = expression->getLeftFactor();
    if (leftFactor->getFactorId() == Factor::ELEMENT_FACTOR) {
        ElementFactor *elementLeft = (ElementFactor*)leftFactor;
        addFbeToList(elementLeft->getInstanceName(), nullptr);
    }
    Factor *rightFactor = expression->getRightFactor();
    if (rightFactor->getFactorId() == Factor::ELEMENT_FACTOR) {
        ElementFactor *elementRight = (ElementFactor*)rightFactor;
        addFbeToList(elementRight->getInstanceName(), nullptr);
    }
```

```
    }

    Action *action = rule->getAction();

    std::map<std::string, Instigation*> *instigations = action->getInstigations();

    for (std::map<std::string, Instigation*>::iterator it = instigations->begin();
      it != instigations->end(); ++it) {

        Instigation *instigation = it->second;

        std::list<Call*> *calls = instigation->getCalls();

        for (std::list<Call*>::iterator it = calls->begin(); it != calls->end(); ++it) {

            Call *call = ((Call*)*it);

            addFbeToList(call->getInstanceName(), nullptr);

        }

    }

}
```