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

FELIPE DOS SANTOS NEVES

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

DISSERTAÇÃO

CURITIBA
2021

FELIPE DOS SANTOS NEVES

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

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

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

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

CURITIBA

2021

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Campus Curitiba

FELIPE DOS SANTOS NEVES

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

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

Data de aprovação: 03 de Setembro de 2021

Prof Adriano Francisco Ronszcka, Doutorado - Sem Vinculo Oficial

Prof Fernando Schutz, - Universidade Tecnológica Federal do Paraná

Prof Herve Panetto, Doutorado - Universite de Lorraine

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

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

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

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

AGRADECIMENTOS

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

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

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

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

“Each of us is carving a stone,
erecting a column, or cutting
a piece of stained glass
in the construction of something
much bigger than ourselves.”
(Adrienne Clarkson)
“Cada um de nós está a esculpir
uma pedra, a erguer uma coluna,
ou a cortar um pedaço de vidro
manchado na construção de algo
muito maior do que nós próprios.”
(Adrienne Clarkson)

RESUMO

NEVES, Felipe dos Santos. ***Framework* PON C++ 4.0: Contribuição para a concepção de aplicações no Paradigma Orientado a Notificações por meio de Programação Genérica**. 2021. 139 f. Dissertação (Mestrado em Computação Aplicada) – Universidade Tecnológica Federal do Paraná. Curitiba, 2021.

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

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

ABSTRACT

NEVES, Felipe dos Santos. **Framework NOP 4.0: Contribution to the development of applications in the Notification Oriented Paradigm through Generic Programming**. 2021. 139 p. Dissertation (Master's Degree in Applied Computing) – Universidade Tecnológica Federal do Paraná. Curitiba, 2021.

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

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

LISTA DE CÓDIGOS

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS, SIGLAS E ACRÔNIMOS

Sigla	Original	Tradução
UTFPR	Universidade Tecnológica Federal do Paraná	
PPGCA	Programa de Pós-Graduação em Computação Aplicada	
CPGEI	Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial	
PON	Paradigma Orientado a Notificações	
TDD	*Test Driven Development*	Desenvolvimento Orientado a Testes
STL	*Standard Template Library*	Biblioteca Template Padrão
LingPON	Linguagem do PON	
PON-HD	PON em *Hardware* Digital	
NOCA	*Notification Oriented Computer Architecture*	Arquitetura de Computação Orientada a Notificações
POO	Paradigma Orientado a Objetos	
PI	Paradigma Imperativo	
PD	Paradigma Declarativo	
PF	Paradigma Funcional	
PP	Paradigma Procedimental	
PL	Paradigma Lógico	
POE	Paradigma Orientado a Eventos	
POAt	Paradigma Orientado a Atores	
POAg	Paradigma Orientado a Agentes	
POC	Paradigma Orientado a Componentes	
SBR	Sistema Baseado em Regras	
CON	Controle Orientado a Notificações	
API	*Application Programming Interface*	Interface de Programação de Aplicativos
SBR	Sistema Baseado em Regras	
RNA	Rede Neural Artificial	
BP	*Backpropagation*	Retropropagação
MLP	*Multi Layer Perception*	Percepção Multicamadas
CBCF	Controle Baseado em Congestionamento Facilitado	

SUMÁRIO

1 EXPERIMENTOS COM O *FRAMEWORK* PON C++ 4.0 E SEUS RESULTADOS

O desenvolvimento do *Framework* PON C++ 4.0 foi proposto com o objetivo de introduzir recursos que facilitem o desenvolvimento de aplicações em PON, assim como utilizar recursos modernos da linguagem C++ que permitiriam ganhos de desempenho quando comparado ao *Framework* PON C++ 2.0, particularmente por ser até então o de melhor desempenho. Desta forma se torna necessária a realização de testes que validem o *Framework* PON C++ 4.0 do ponto de vista funcional, assim como do ponto de vista de desempenho.

Os resultados de todos os testes realizados em linguagem de programação C++[1] foram obtidos por meio do *framework* Google Benchmark, de modo a permitir a reprodutibilidade dos testes e resultados de forma simples. Além disso, o Google Benchmark garante a confiabilidade dos tempos de execução apresentados, pois ele gerencia o número de iterações necessárias de forma dinâmica para obter um resultado estatisticamente estável (GOOGLE, 2020).

Além das aplicações desenvolvidas para os testes de desempenho, na Seção 1.2 é apresentada uma nova da implementação do jogo *NOPUnreal*. Esse jogo já foi introduzido na Seção **??**. Entretanto, nesta seção ele é reimplementado com o *Framework* PON C++ 4.0, o que permite a comparação em termos de facilidade de desenvolvimento e verbosidade com a implementação anterior desenvolvida com o *Framework* PON C++ 2.0.

Após apresentadas estas aplicações, a Seção 1.3 apresenta a pesquisa de opinião relativa ao uso do *Framework* PON C++ 4.0 realizada com desenvolvedores do grupo de pesquisa do PON apenas com o intuito de se obter algum *feedback* ou alimentação externa. Por fim, na Seção 1.4, é realizada uma reflexão sobre os resultados apresentados neste capítulo.

1.1 TESTES DE DESEMPENHO

Nas seções seguintes são apresentadas as aplicações desenvolvidas com o propósito de avaliar o desempenho do *Framework* PON C++ 4.0. A Seção 1.1.1 apresenta a aplicação de sensores, a Seção 1.1.2 apresenta o algoritmo *Bitonic Sort*, a Seção 1.1.3 apresenta o algoritmo *Random Forest* e, por fim, a Seção 1.1.4 apresenta a aplicação de controle de semáforos.

Cada um destes *benchmarks* apresenta seu próprio propósito. O objetivo da aplicação

[1] Para permitir os testes com o Google Benchmark, aplicações na linguagem de programação C também são compiladas em C++

de sensores tem como objetivo permitir a comparação do desempenho do novo *Framework* PON C++ 4.0 com o estável *Framework* PON C++ 2.0, assim como comparar o desempenho de ambos os *frameworks* com POO. A aplicação de controle de tráfego automatizado tem o propósito de permitir avaliar o comportamento do paralelismo instroduzidas pelo *Framework* PON C++ 4.0, de modo que esta aplicação apresenta comparações com o *Framework* PON C++ Elixir/Erlang, pelo fato deste ser um *framework* que faz amplo uso das capacidades de paralelismo da linguagem de programação Elixir. Por fim, ambos os *benchmarks* universalizados por meio dos algoritmos *Bitonic Sort* e *Random Forest* tem como principal objetivo permitir a comparação do desempenho do *Framework* PON C++ 4.0 com o PP, assim como avaliar os benefícios da utilização de paralelismo para a execução destes algoritmos.

1.1.1 Aplicação de sensores

A chamada aplicação de sensores materializa uma solução para uma rede de sensores simulados, na qual cada sensor possui um estado (ativado ou desativado) que pode ser observado. Uma *Rule* determina o comportamento do sensor, quando o sensor é ativado esta *Rule* é aprovada, reiniciando os estados de ativação e leitura do sensor.

A representação desta aplicação sob a forma de um *FBE* e uma *Rule* foi previamente apresentada na Figura **??** da Seção **??**. O Código 1 em LingPON, poderia ser utilizado para gerar código nos *frameworks*, conforme feito em (SKORA, 2020). Porém, neste presente trabalho, por questões de possibilitar uma otimização mais fina do código, as implementações com o *Framework* PON C++ 2.0 e 4.0 foram escritas manualmente, servindo o código em LingPON de guia tão somente. O Código 2 apresenta a estrutura utilizada para implementar o sensor com o *Framework* PON C++ 4.0, utilizando um total de apenas 13 linhas.

Código 1 – *FBE Sensor* em LingPON

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

Fonte: Fonte: Autoria própria

Código 2 – Código da estrutura do sensor com o *Framework* PON C++ 4.0

```
struct NOPSensor{
   NOP::SharedAttribute<bool> atIsRead{ NOP::BuildAttribute(false) };
   NOP::SharedAttribute<bool> atIsActivated{ NOP::BuildAttribute(false) };
   NOP::SharedPremise prIsActivated{ NOP::BuildPremise<bool>(atIsActivated, true,
       NOP::Equal()) };
   NOP::SharedPremise prIsNotRead{ NOP::BuildPremise<bool>(atIsRead, false, NOP::Equal()) };
   NOP::SharedRule rlSensor{ NOP::BuildRule(
      NOP::BuildCondition<NOP::Conjunction>(prIsActivated, prIsNotRead),
      NOP::BuildAction(NOP::BuildInstigation([&]() { this->Read(); this->Deactivate(); })))
   };
   void Read() const { atIsRead->SetValue(true); }
   void Activate() const { atIsActivated->SetValue(true); atIsRead->SetValue(false); }
   void Deactivate() const { atIsActivated->SetValue(false); }
};
```

Fonte: Autoria própria

O Código 3 apresenta a estrutura utilizada para implementar o sensor com o *Framework* PON C++ 2.0. Comparando o Código 2 com o Código 3 é interessante observar que a implementação com o *Framework* PON C++ 4.0 é mais simples que com a implementação com o *Framework* PON C++ 2.0, utilizando muito menos linhas de código. Por sua vez, o Código 4 apresenta a estrutura utilizada para implementar o sensor com o POO em C++, utilizando um total de apenas 8 linhas.

Código 3 – Código da estrutura do sensor com o *Framework* PON C++ 2.0

```
struct NOP2Sensor : public FBE
{
```

```
  Boolean* atIsActivated, atIsRead;
  Premise* prIsActivated, prIsNotRead;
  Instigation* inSensor;
  RuleObject* rlSensor;
  Method* mtSensor;
  NOP2Sensor() {
    BOOLEAN(this, atIsActivated, false);
    BOOLEAN(this, atIsRead, false);
    mtSensor = new MethodPointer<NOP2Sensor>(this, &NOP2Sensor::ProcessSensor);
  }
  void ProcessSensor() { atIsRead->setValue(true); atIsActivated->setValue(false); }
};

class SensorApp : public NOPApplication {
public:
  SensorApp(int count) : NOPApplication() {
    SingletonFactory::changeStructure(SingletonFactory::NOPVECTOR);
    SingletonScheduler::changeScheduler(SchedulerStrategy::NO_ONE);
    for (int i = 0; i < count; i++) {
      sensors.push_back(std::make_shared<NOP2Sensor>());
    }
    for (auto& sensor : sensors) {
      PREMISE(sensor->prIsActivated, sensor->atIsActivated, true,
            Premise::EQUAL, Premise::STANDARD, false);
      PREMISE(sensor->prIsNotRead, sensor->atIsRead, false,
            Premise::EQUAL, Premise::STANDARD, false);
      INSTIGATION_NAMED(sensor->inSensor, sensor->mtSensor);
      RULE(sensor->rlSensor, SingletonScheduler::getInstance(), Condition::CONJUNCTION);
      sensor->rlSensor->addPremise(sensor->prIsActivated);
      sensor->rlSensor->addPremise(sensor->prIsNotRead);
      sensor->rlSensor->addInstigation(sensor->inSensor);
      sensor->rlSensor->end();
    }
  }
  std::vector<std::shared_ptr<NOP2Sensor>> sensors;
};
```

Fonte: Autoria própria

Para testar esta aplicação são instanciados um total de 100.000 sensores com uma taxa de aprovação que varia para cada iteração de forma que a cada iteração apenas uma fração dos sensores é ativada[2]. O gráfico da Figura 1 mostra o resultado dos testes, apresentando o tempo de execução com relação à taxa de aprovação.

Código 4 – Código da estrutura do sensor em POO

```
struct OOPSensor {
   inline static std::atomic<int> counter{ 0 };
   bool isRead{ false };
   bool isActivated{ false };
   void Read() { isRead = true; }
   void Activate() { isActivated = true; isRead = false; }
   void Deactivate() { isActivated = false; }
};
```

Fonte: Autoria própria

No gráfico da Figura 1 é possível observar como ambos os *frameworks* apresentam um comportamento similar, com desempenho superior ao POO para taxas de aprovação mais baixa e com desempenho inferior para taxas de aprovação mais alta, sendo que o tempo de execução cresce de maneira linear com a taxa de aprovação das regras, conforme esperado pela

[2] Teste executado em um computador com processor Ryzen 5 3600 a 3.6GHz e 16 GB de RAM DDR4 a 3000MHz (dual channel) e sistema operacional Windows 10 64 bits.

complexidade linear do PON. As aplicações utilizando o PON ainda apresentam tempos de execução superiores ao POO em determinados casos devido ao custo computacional em tempo de execução adicionado pela utilização das estruturas de dados utilizadas para a materialização do mecanismo de notificações. Isto fica mais evidente quanto mais sensores são ativados, amenizando assim o efeito das redundâncias estruturais e temporais do POO/PI. A Figura 2 detaca os tempos de execução com o *Framework* PON C++ 4.0 relativos aos do *Framework* PON C++ 2.0, reforçando como o mesmo apresenta desempenho superior em todos os cenários.

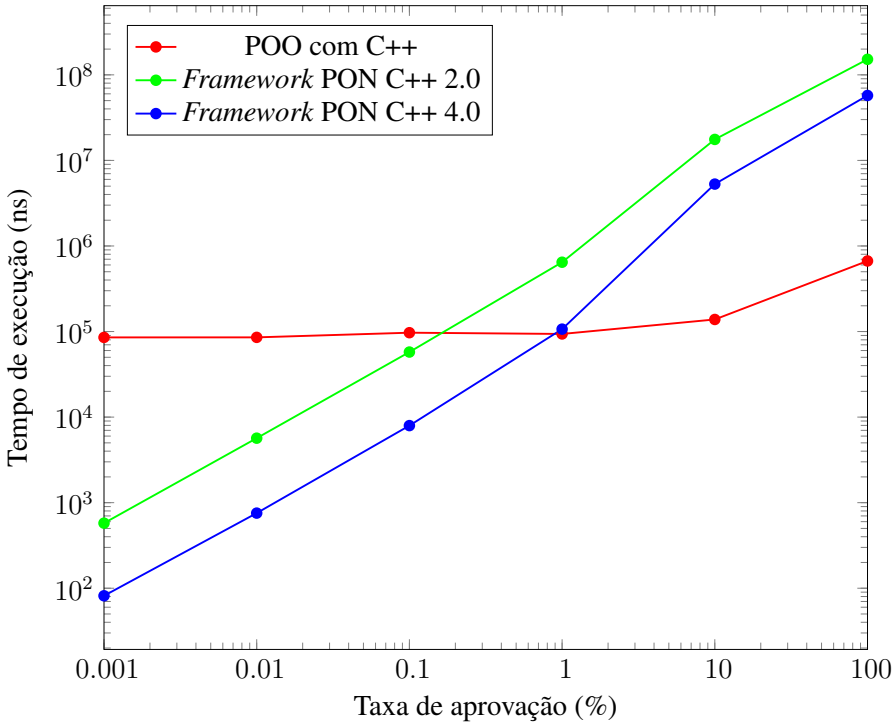


Figura 1 – Testes de desempenho da aplicação do sensor
Fonte: Autoria própria

Outra análise realizada neste mesmo cenário da aplicação de sensores foi o do consumo de memória das aplicações. Para este teste a aplicação instancia uma quantidade de 10.000 sensores, sem realizar a aprovação de nenhuma *Rule*. Essa instanciação dos sensores se repete ao longo de pelo menos 10 segundos, permitindo traçar o perfil de consumo de memória da aplicação, utilizando as ferramentas de análise do Visual Studio 2019.

O consumo de memória da aplicação em POO com C++ é mostrado na Figura 3. Naturalmente, devido à natureza da aplicação ser bastante simples, como não há uso de nenhum *framework*, atinge um máximo de 2 MB durante o teste. A baixa taxa de amostragem combinada ao rápido tempo de execução desta aplicação não permite observar as curvas no consumo de memória sendo alocada e desalocada neste cenário. É interessante saber o consumo de memória da aplicação em POO, pois esta serve como referência para comparar o consumo dos *frameworks*.

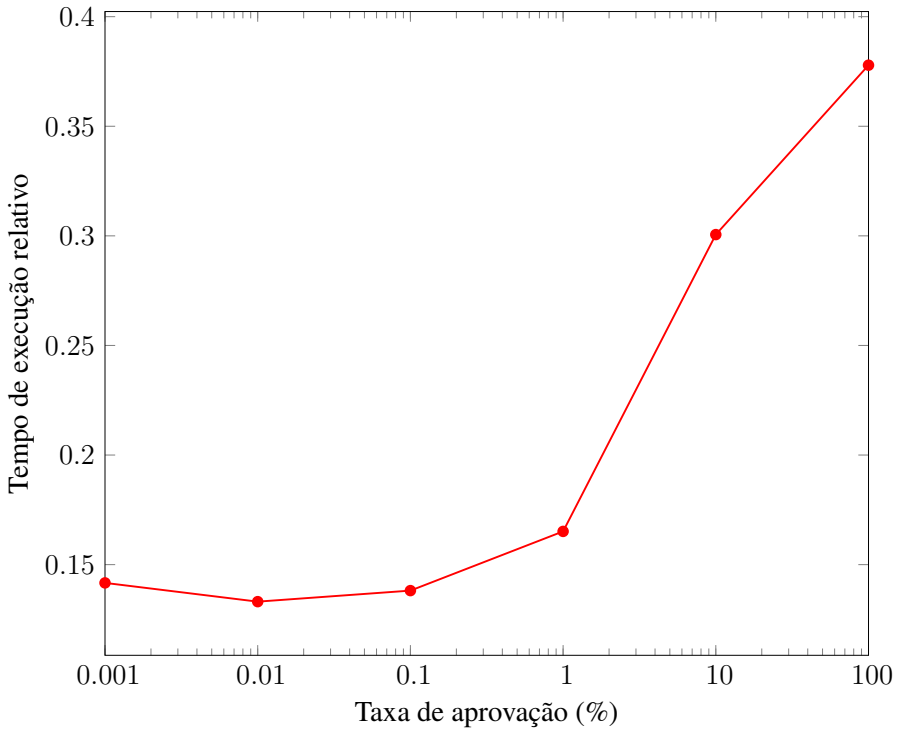
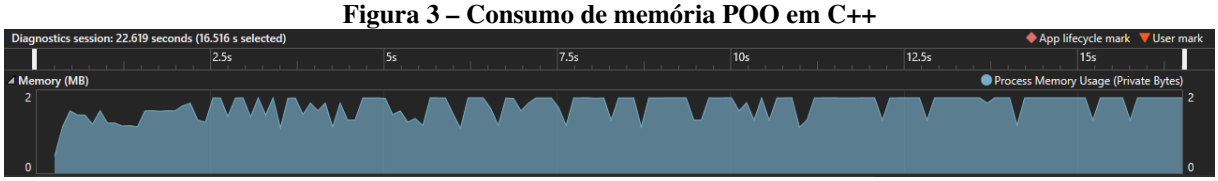


Figura 2 – Tempos de execução da aplicação do sensor com o *Framework* PON C++ 4.0 relativo ao *Framework* PON C++ 2.0
Fonte: Autoria própria

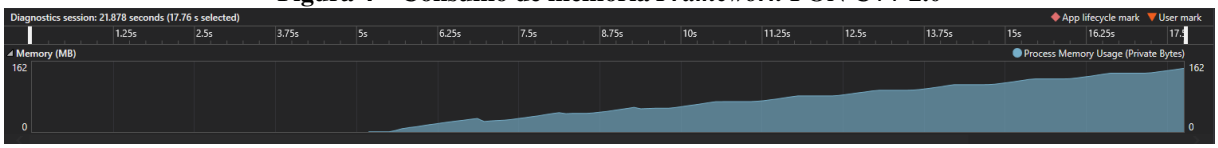
Figura 3 – Consumo de memória POO em C++



Fonte: Autoria própria

O consumo de memória da aplicação com o *Framework* PON C++ 2.0 é mostrado na Figura 4, é interessante observar a curva crescente que demonstra que a memória não está sendo corretamente desalocada, fazendo com que o consumo de memória apenas aumente ao longo do tempo, até o ponto em que a memória do sistema se esgote e a aplicação falhe. Um teste isolado realizando a instanciação das entidades uma única vez apresentou um consumo de 69 MB de RAM. Xavier (2014) também já havia encontrado problemas no mecanismo de alocação de memória do *Framework* PON C++ 2.0, evidenciados neste experimento.

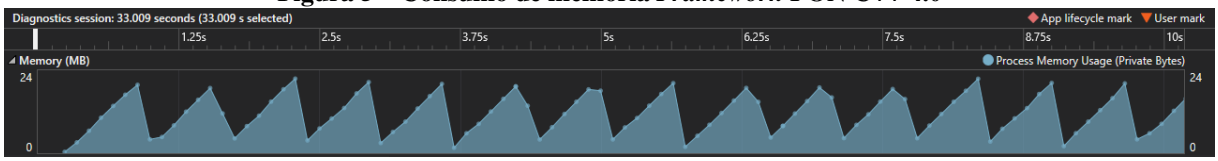
Figura 4 – Consumo de memória *Framework* PON C++ 2.0



Fonte: Autoria própria

O consumo de memória da aplicação com o *Framework* PON C++ 4.0 é mostrado na Figura 5, na qual pode ser observado um comportamento esperado da memória sendo alocada para as entidades até atingir cerca de 24 MB, e então sendo apropriadamente desalocada antes da nova iteração, demonstrando a estabilidade do gerenciamento de memória, sendo realizado com a utilização dos *smart pointers*.

Figura 5 – Consumo de memória *Framework* PON C++ 4.0



Fonte: Autoria própria

Desta forma, fica explícito que o *Framework* PON C++ 4.0 apresenta ganhos de desempenho significativos com relação ao *Framework* PON C++ 2.0. No cenário da aplicação de sensores, quando comparado ao *Framework* PON C++ 2.0, o *Framework* PON C++ 4.0 reduziu em 62% o tempo de execução no pior caso, e em 85% no melhor caso. Além da redução nos tempos de execução, também reduziu em quase dois terços o consumo de memória e não apresentou o problema de vazamento de memória apresentado no gráfico da Figura 4.

Uma aplicação de cunho similar, porém com contexto ainda mais restrito ao grupo de pesquisa, a aplicação mira ao alvo. Esta aplicação apresenta nível de complexidade e lógica similares à aplicação de sensores e é apresentada como curiosidade no Apêndice F.

1.1.2 Aplicação *Bitonic Sort*

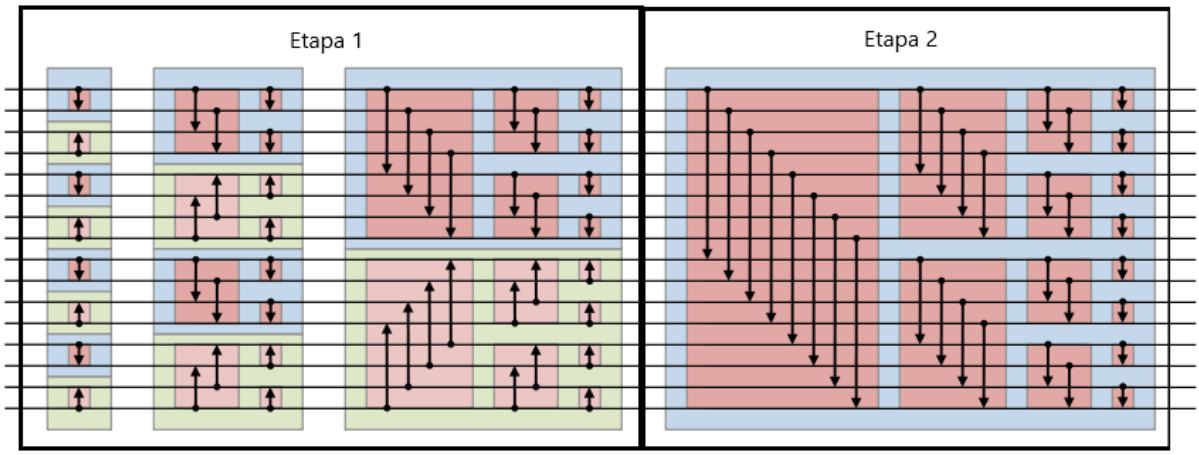
O *Bitonic Sort* é um algoritmo de ordenação originalmente proposto por Batcher (1968). Esta seção discorre sobre os detalhes do algoritmo, assim como sua implementação em PON, apresentando comparações com a implementação tradicional do algoritmo em linguagem de programação C.

Uma sequência é dita como *bitonic* caso sua primeira parte seja crescente e a segunda parte decrescente, de forma que para uma sequência $[0...n-1]$ a mesma é *bitonic* caso exista um índice *i* nos limites $0 <= i <= n-1$ de modo que $x_0 <= x_1 <= ... <= x_i$ e $x_i >= x_{i+1} >= ... >= x_{n-1}$. Como exemplo, a sequência $[5, 6, 7, 8, 4, 3, 2, 1]$ é *bitonic*, pois pode ser dividida em duas sequências $[5, 6, 7, 8]$ e $[1, 2, 3, 4]$, que são, respectivamente, crescentes e decrescentes.

Desta forma, uma sequência *bitonic* pode ser ordenada por um conjunto de comparado-

res que operam em pares de valores da sequência, no contexto da chamada ordenação *bitonic*. O algoritmo de ordenação *Bitonic Sort* pode ser dividido em duas etapas, sendo que primeiramente a sequência é transformada em uma sequência *bitonic* (etapa 1) e subsequentemente essa sequência *bitonic* é ordenada de forma crescente (etapa 2). Estas duas etapas são ilustradas na Figura 6. Os comparadores dessas etapas podem ser executados de forma paralela permitindo melhor desempenho do algoritmo em ambientes multiprocessados.

Figura 6 – Processo de ordenação com *Bitonic Sort*



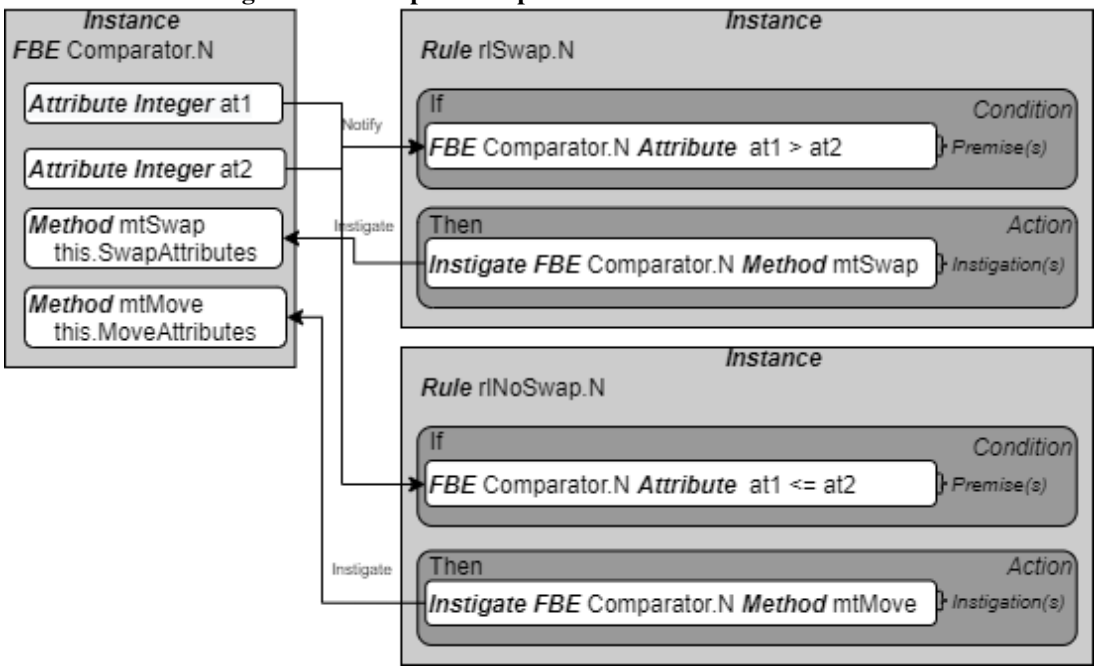
Fonte: Adaptado de Mullapudi (2014)

O algoritmo do *Bitonic Sort* pode ser dividido em vários estágios, dado o fato que as comparações são realizadas entre dois elementos e os mesmos são comparados apenas uma vez dentro de cada estágio, de tal modo que as comparações dentro de cada estágio podem ser realizadas de forma paralela. A existência desses estágios paralelizáveis favorece uma implementação em PON devido ao seu paralelismo intrínseco. Deste modo, os comparadores do *Bitonic Sort* podem ser representados por meio de *Rules* (PORDEUS, 2020; PORDEUS *et al.*, 2021).

Em uma interpretação para a implementação em PON os diferentes estágios podem ser representados por *Attributes* independentes, de modo que a ordenação ocorre movendo os valores do estágio anterior para o próximo. Assim, cada comparador é materializado como um *FBE* com duas *Rules*, conforme ilustrado na Figura 7, sendo que uma *Rule* opera no caso da comparação ser verdadeira e a outra no caso da comparação ser falsa. Isto visto que pela divisão em etapas, é necessário mover os valores da etapa anterior para a próxima, invertendo os valores apenas quando a comparação é verdadeira. Os métodos *mtMove* e *mtSwap* são responsáveis por

mover os valores de *at1* e *at2* do *FBE* do comparador atual para os *Attributes* dos comparadores das etapas seguintes.

Figura 7 – *Rules* para comparador do *Bitonic Sort* em PON



Fonte: Autoria própria

Na estrutura do ordenador implementado com o *Framework* PON C++ 4.0, todas as entidades do PON são criadas de forma dinâmica na construtora da *struct*, baseando-se no número de elementos, sendo passado como parâmetro. A função *Sort*, por sua vez, atribuí os valores aos *Attributes* pertinentes ao primeiro estágio de comparação, conforme mostrado no Código 5, e a ordenação é realizada pela interação das entidades do PON já declaradas na construção do objeto, ao final da operação retornando o vetor *out* que armazena os resultados da ordenação. Devido à extensão dos códigos, essa seção traz apenas trechos das implementações, com o código completo disponível para referência no Apêndice G.

Código 5 – Trecho de código da estrutura NOPBitonicSorter

```
std::vector<T> NOPBitonicSorter::Sort(const std::vector<T>& input){
   for (auto i = 0; i < input.size(); i++) {
      elements[1][i]->SetValue(input[i]);
   }
   return out;
}
```

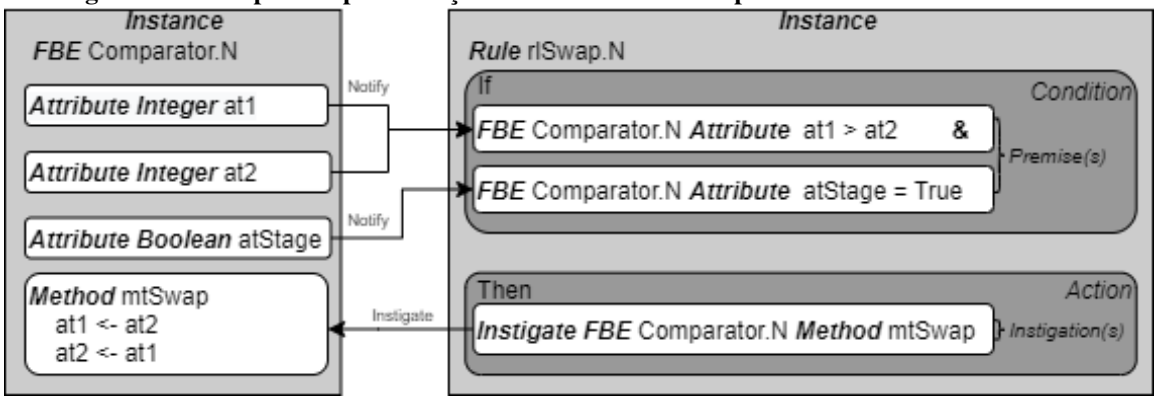
Fonte: Autoria própria

Essa implementação apresenta elementos completamente independentes e paralelizáveis, de modo que qualquer alteração no estado de um *Attribute* do estágio de entrada do

ordenador é refletida na saída do mesmo. Entretanto, esta implementação em PON apresenta desempenho ruim, devido ao fato de ser necessário mover os dados entre os diferentes estágios. A operação de mover os dados necessita da aprovação de uma *Rule* para cada comparação para sua execução. Além disso, também pode ocorrer execução de *Rules* desnecessárias decorrentes da ordem de avaliação dos comparadores, porque os valores nos estágios intermediários são substituídos pelos estágios anteriores à medida que as *Rules* são aprovadas, causando a aprovação de regras desnecessárias durante o processo. Dados estes problemas, se torna necessária uma nova interpretação deste problema em PON por meio da proposta de uma solução mais eficiente.

Esta solução mais eficiente proposta se aproveita da característica da divisão em estágios do algoritmo, desta vez não utilizando *Attributes* para os estágios intermediários, mas sim uma *Premise* adicional aos comparadores que controla sua execução apenas durante o seu estágio. Desta forma também é possível reduzir o número de *Rules* pela metade, utilizando apenas uma *Rule* por comparador. Esta interpretação é representada na Figura 8.

Figura 8 – *Rules* para implementação mais eficiente do comparador do *Bitonic Sort* em PON



Fonte: Autoria própria

O Código 6 mostra em detalhe o corpo da função de ordenação desta implementação, onde os valores dos *Attributes* são atribuídos de acordo com o vetor de entrada, porém a ordenação é realizada por meio da mudança do estado dos *Attributes* de controle de cada estágio para *true*.

Código 6 – Trecho de código da estrutura NOPBitonicSorterStages

```
template<typename T>
std::vector<T> NOPBitonicSorterStages::Sort(std::vector<T>& input)
{
   for (size_t i = 0; i < input.size(); i++)
   {
      elements[i]->SetValue(input[i], NOP::NoNotify);
   }
   for (const auto& stage : stages)
   {
      stage->SetValue(true);
      stage->SetValue(false);
   }
   for (auto i = 0; i < elements.size(); i++)
   {
      out[i] = elements[i]->GetValue();
   }
   return out;
}
```

Fonte: Autoria própria

A Figura 9 mostra como a implementação com a divisão em estágios apresenta tempos de execução significativamente menores que a implementação original em PON, sendo que para o caso da ordenação de 64 elementos a implementação em estágios chega a ser 20.000 vezes mais rápida.

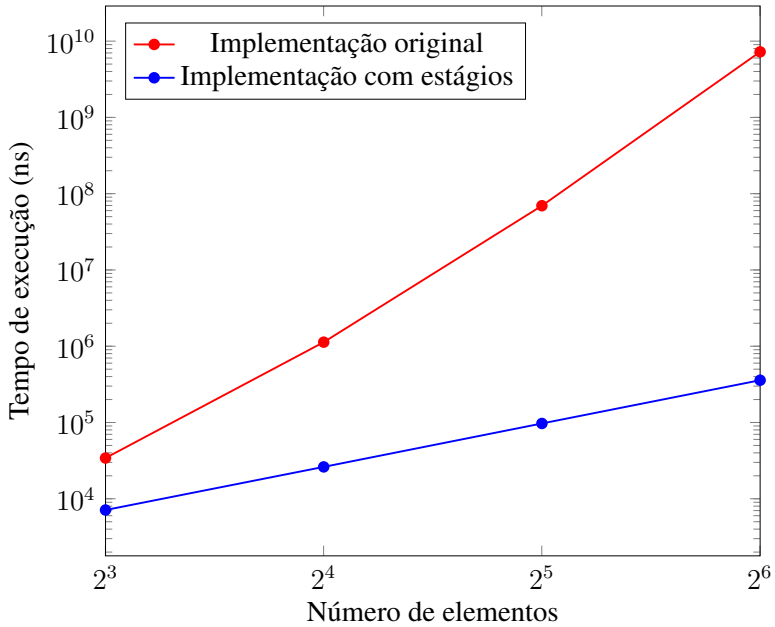
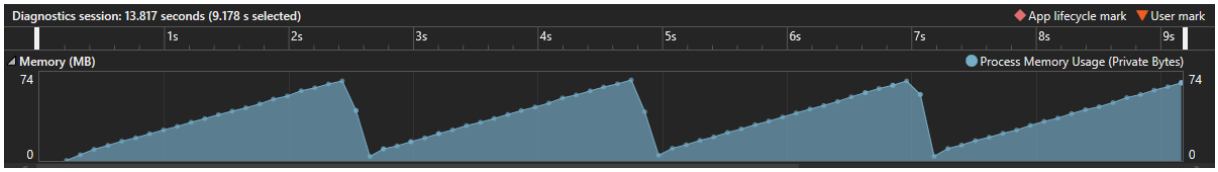


Figura 9 – Testes de desempenho da aplicação do *Bitonic Sort* com diferentes implementações em PON
Fonte: Autoria própria

Além disso, o número reduzido de *Rules* utilizadas faz com que a implementação em estágio tenha o consumo de memória cerca de 50% menor. Utilizando as ferramentas de análise do Visual Studio 2019 foi possível traçar o perfil do consumo de memória das aplicações. Neste teste, para o caso do ordenador de 1024 elementos, o total de memória alocado pela aplicação chega a 74 MB na implementação original e 35 MB na implementação em estágios.

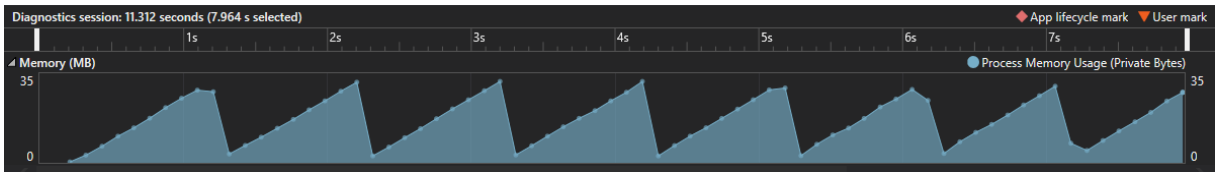
Esse alto consumo de memória é decorrente do grande número de elementos do PON utilizados na construção do programa, sendo que para o ordenador de 1024 elementos são necessários 28.160 comparadores. Os gráficos do consumo de memória para a implementação original e a implementação em estágios são apresentados nas Figuras 10 e 11 respectivamente.

Figura 10 – Consumo de memória para a aplicação do algoritmo *Bitonic Sort* com o *Framework* PON C++ 4.0 na implementação original



Fonte: Autoria própria

Figura 11 – Consumo de memória para a aplicação do algoritmo *Bitonic Sort* com o *Framework* PON C++ 4.0 na implementação em estágios



Fonte: Autoria própria

Além das implementações em PON, foi escolhida uma implementação em C do algoritmo, disponível na íntegra no Apêndice H(PITSIANIS, 2008), de modo a servir como base de comparação para o desempenho da implementação em PON. Nos testes foi avaliado o tempo de execução para realizar a ordenação de 32, 64, 128, 256, 512, 1024, 2048 e 4096 elementos[3]. Na avaliação do tempo de execução da implementação em PON não é considerado o tempo gasto na inicialização da estrutura em si, mas sim da execução da função *Sort*. Os resultados são exibidos no gráfico da Figura 12. Para esta comparação foi utilizada a implementação em estágios com o PON.

O tempo de execução do processo de ordenação com o PON foi superior ao da implementação em C utilizando o PP e, quanto maior o número de elementos, maior a diferença entre os tempos de execução, chegando a uma diferença de 2000 vezes no caso de ordenação de 4096 elementos. O problema da implementação em PON se dá devido ao elevado número de comparadores, que por sua vez representam um número elevado de entidades do PON instanciadas para a construção do comparador, cada uma com um significativo custo de memória associado, além do elevado número de notificações necessárias para a realização do processo de ordenação.

[3] Teste executado em um computador com processor Ryzen 5 3600 a 3.6GHz e 16 GB de RAM DDR4 a 3000MHz (dual channel) e sistema operacional Windows 10 64 bits.

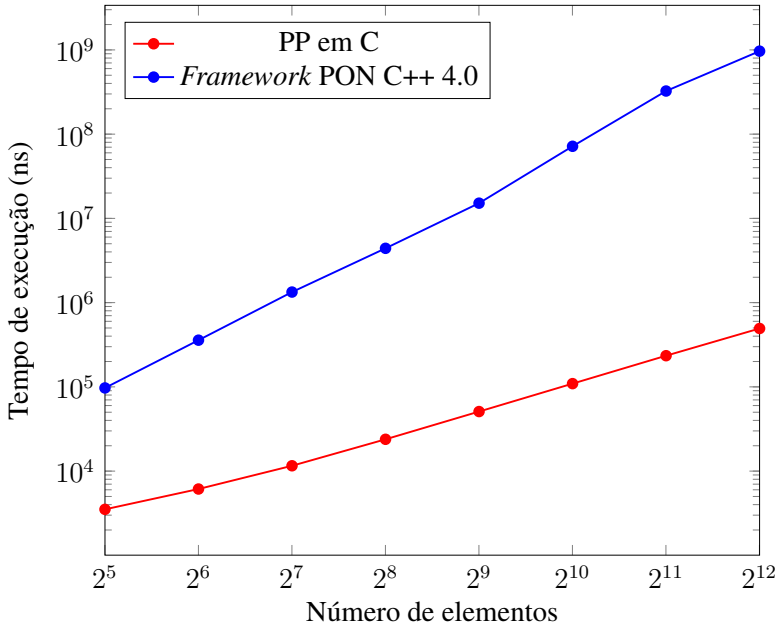


Figura 12 – Testes de desempenho da aplicação do algoritmo *Bitonic Sort*
Fonte: Autoria própria

De forma a aproveitar a natureza paralelizável do algoritmo, também é possível paralelizar a implementação em PON com o *Framework* PON C++ 4.0, fazendo usos do mecanismo de notificações paralelas, de forma que cada notificação pode ser processada em uma *thread* separada, potencialmente reduzindo os tempos de execução para aplicações com elevado número de entidades a serem notificadas.

Esta implementação é bastante simples com o *Framework* PON C++ 4.0, pois basta utilizar a função *SetValue* com um parâmetro de template <*NOP::Parallel*> para que as notificações sejam realizadas de forma paralela. O Código 7 mostra a função de ordenação de forma paralela, similar ao já apresentado no Código 6.

No gráfico da Figura 13 é possível observar que a paralelização oferece ganhos de desempenho para a ordenação de números elevados de elementos, enquanto para números menores de elementos o tempo de execução aumentou. Por sua vez, na Figura 14, são comparados os tempos de execução da aplicação com paralelização relativos aos da execução sequencial. Para a ordenação de 4096 elementos, a execução de forma paralelizada teve seu tempo de execução reduzido em cerca de 50%.

Além de observar os tempos de execução, a fim de verificar os efeitos do paralelismo nessa aplicação, é possível observar a utilização de CPU durante a execução, também utilizando as ferramentas de análise do Visual Studio 2019, da mesma forma que o consumo de memória foi observado nas outras aplicações.

Ao avaliar o desempenho da aplicação para ordenar 4096 elementos durante a execução

Código 7 – *Bitonic Sort* paralelizado com o *Framework* PON C++ 4.0

```
template<typename T>
std::vector<T> NOPBitonicSorterStages::Sort(std::vector<T>& input)
{
   for (size_t i = 0; i < input.size(); i++)
   {
      elements[i]->SetValue(input[i], NOP::NoNotify);
   }

   for (const auto& stage : stages)
   {
      stage->SetValue<NOP::Parallel>(true);
      stage->SetValue<NOP::Parallel>(false);
   }

   for (auto i = 0; i < elements.size(); i++)
   {
      out[i] = elements[i]->GetValue();
   }

   return out;
}
```

Fonte: Autoria própria

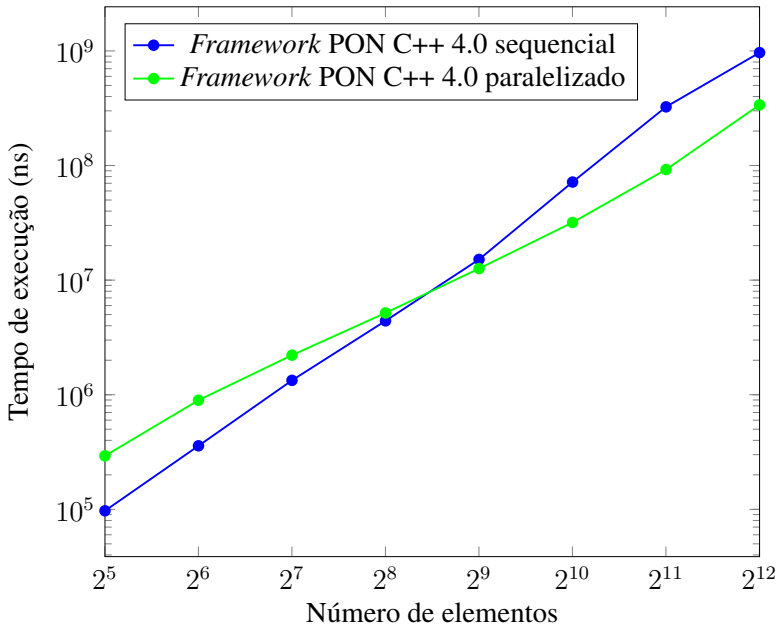


Figura 13 – Comparação da paralelização na aplicação do algoritmo *Bitonic Sort*
Fonte: Autoria própria

sequencial, com o gráfico da utilização de CPU na Figura 15, a utilização de CPU não passa de 8%. Isso se dá pelo fato do ambiente de testes possuir 12 núcleos, de modo que uma aplicação sem paralelismo executando de forma sequencial somente consegue utilizar um dos núcleos, equivalente a 8.33% do processamento disponível.

Já a implementação paralelizada consegue fazer uma utilização muito melhor da CPU. Como ilustrado na 16, a utilização de CPU varia, atingindo quase 100% em alguns momentos. Ou seja, essa implementação utiliza de melhor forma os recursos disponíveis, com isso atingindo um desempenho melhor que a implementação sequencial.

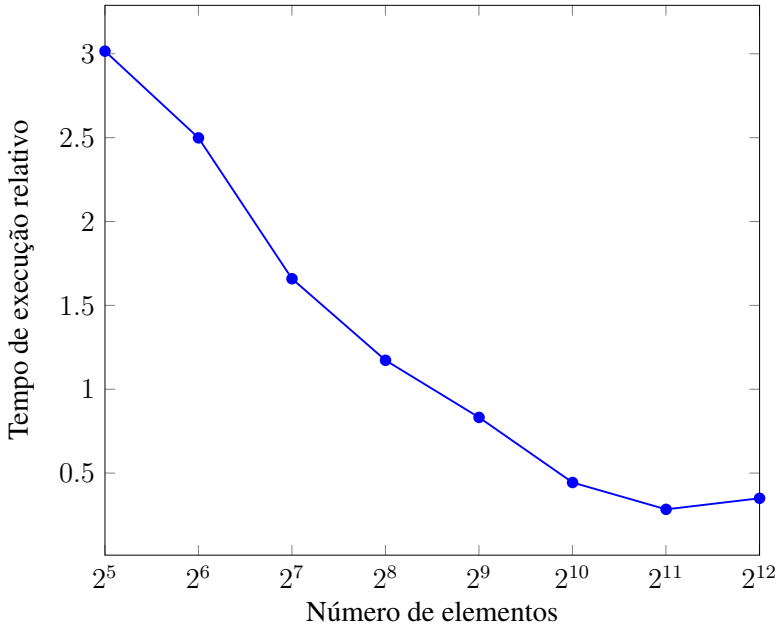
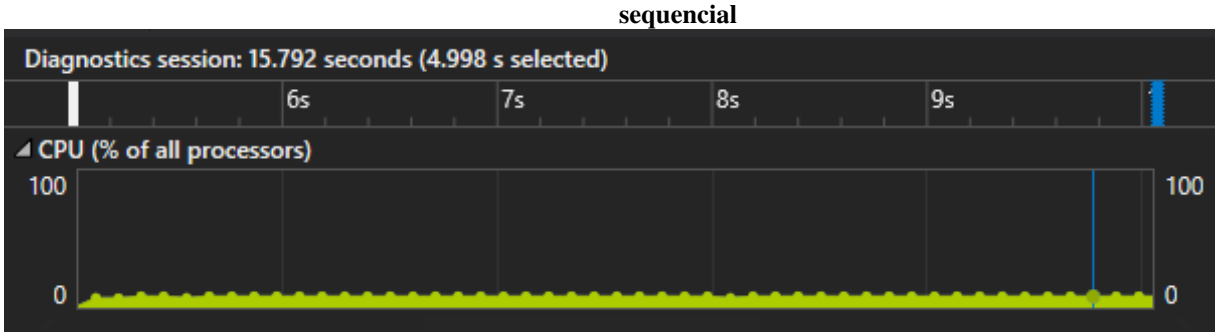


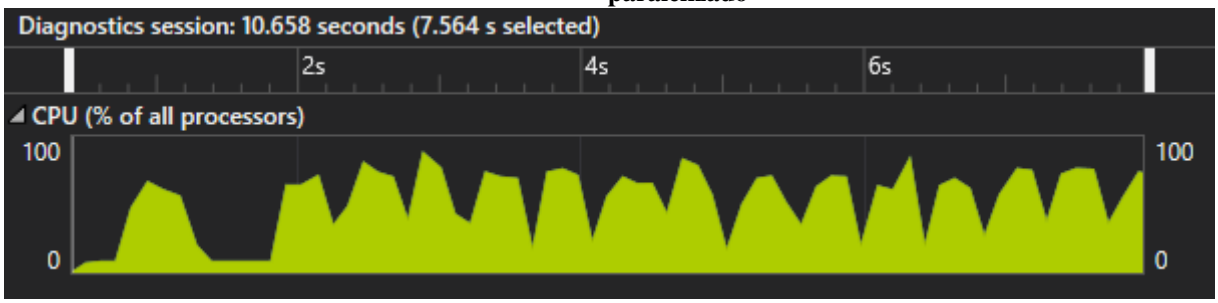
Figura 14 – Tempos de execução do algoritmo *Bitonic Sort* com o *Framework* PON C++ 4.0 paralelizado relativo ao sequencial
Fonte: Autoria própria

Figura 15 – Utilização de CPU durante execução do algoritmo *Bitonic Sort* com o *Framework* PON C++ 4.0 sequencial



Fonte: Autoria própria

Figura 16 – Utilização de CPU durante execução do algoritmo *Bitonic Sort* com o *Framework* PON C++ 4.0 paralelizado



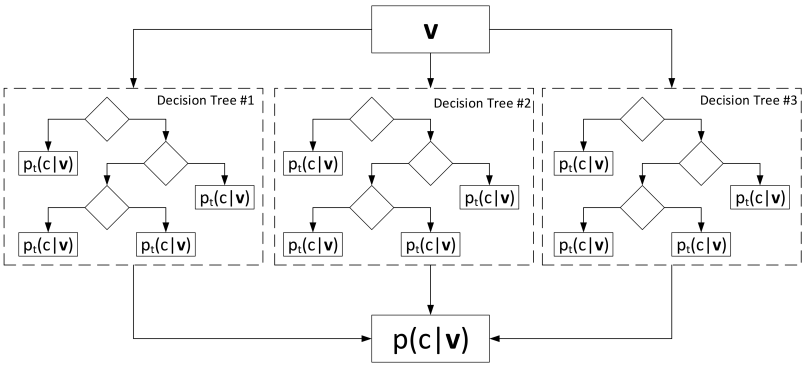
Fonte: Autoria própria

1.1.3 Aplicação *Random Forest*

O *Random Forest* é um algoritmo popular de aprendizado de máquina, utilizado em muitas aplicações para classificação e regressão. O algoritmo consiste em um conjunto de árvores de decisão, treinadas individualmente e combinadas para obter um resultado com menor erro de classificação/regressão. Para esta aplicação é comparado o desempenho da implementação em PON com implementações em linguagem de programação *Python* e C. Em tempo, Python é linguisticamente mais alto-nível que a linguagem C justamente.

Isto dito, no *Random Forest*, cada árvore é avaliada separadamente, viabilizando a execução de maneira paralela. Ao final da execução de todas as árvores os resultados são combinados para todo o conjunto (CRIMINISI *et al.*, 2011). A Figura 17 ilustra a estrutura das árvores de decisão (*decision trees*) independentes do *Random Forest*. No contexto do PON, a implementação do *Random Forest* é interessante devido ao fato deste algoritmo ser construído essencialmente por meio de expressões lógicas *if-else*. Deste modo, o PON pode permitir eliminar as redundâncias temporais e estruturais, além de também permitir a paralelização da execução das árvores (PORDEUS, 2020).

Figura 17 – Árvores de decisão do algoritmo *Random Forest*



Fonte: Pordeus (2020)

Uma implementação realizada originalmente por Pordeus (2020) [4] possibilita a geração de código em PON com base nas árvores de decisão geradas com o auxílio da biblioteca scikit-learn[5] na linguagem de programação Python. Esta implementação era capaz de gerar código específico para o PON em LingPON e PON-HD, assim como para o PP em linguagem de programação C. Esta implementação foi modificada para gerar código específico para o PON utilizando o *Framework* PON C++ 4.0, com base na implementação para geração de código

[4] Implementação disponível em https://github.com/leonardopordeus/RP
[5] Maiores detalhes sobre a biblioteca podem ser encontrados em https://scikit-learn.org/stable/

em LingPON já existente. A implementação em proposta neste trabalho contempla apenas a utilização do algoritmo para a classificação de dados, sendo que o treinamento e geração das árvores de classificação são realizados por meio do uso da biblioteca scikit-learn.

No Código 8 é apresentado como exemplo uma das *Rules* geradas por esta implementação. Esta *Rule* é composta por diversas *Premises* que são capazes de avaliar o valor de *Attributes*, que representam os dados de entrada da classificação, com relação aos valores atribuídos durante o processo de treinamento das árvores. Ainda, devido a extensão dos códigos gerados aqui são disponibilizados apenas trechos, com o código-fonte completo para o caso de uma árvore disponível no Apêndice I.

Código 8 – *Rule* do algoritmo *Random Forest* para o *Framework* PON C++ 4.0

```
/*
rule rlTree_0_5
    premise prTree_0_5_1
      this.attr3 > 75
    end_premise
    and
    premise prTree_0_5_2
      this.attr2 <= 495
    end_premise
    and
    premise prTree_0_5_3
      this.attr3 <= 165
    end_premise
    and
    premise prTriggerTree0
      this.trigger_tree_0 == true
    end_premise
  end_condition
  action sequential
    instigation sequential
      call this.mt_count_versicolor()
      call this.mtTrigger1()
      call this.mtRstTrigger0()
    end_instigation
  end_action
end_rule
*/
NOP::SharedRule rlTree_0_5 = NOP::BuildRule(
      NOP::BuildCondition<NOP::Conjunction>(
         NOP::BuildPremise(attr3, 75, NOP::Greater()),
         NOP::BuildPremise(attr2, 495, NOP::LessEqual()),
         NOP::BuildPremise(attr3, 165, NOP::LessEqual()),
         NOP::BuildPremise(trigger_tree_0, true, NOP::Equal())
      ),
      NOP::BuildAction(
         NOP::BuildInstigation(mt_count_versicolor,mtTrigger1,mtRstTrigger0)
      )
   );
```

Fonte: Autoria própria

O algoritmo *Random Forest* possibilita a utilização de um número variável de árvores de decisão. Dessa forma, com a implementação supracitada, também é possível gerar código para diferentes números de árvores. Nos testes foram utilizados os valores de 1, 10, 20, 50 e 100 árvores. Na Tabela **??** são relacionados os números de elementos necessários para a construção de cada versão do classificador com diferentes números de árvores.

Número de árvores	1	10	20	50	100
Attributes	7	34	64	154	304
Premises	8	39	56	80	93
Conditions	9	90	173	437	839
Rules	9	90	173	437	839
Actions	6	60	120	300	600
Instigations	6	60	120	300	600
Methods	6	60	120	300	600

Tabela 1 – Número de elementos em relação ao número de árvores
Fonte: Autoria própria

Esse elevado número de entidades e, por sua vez, de linhas de código faz com que o tempo de compilação dessa aplicação seja bastante elevado. No computador utilizado nos testes, com processor Ryzen 5 3600 e 16 GB de RAM DDR4, o tempo de compilação da aplicação de testes chegou a ser superior a 3 minutos, pois somente o arquivo contendo o código-fonte definindo a estrutura das árvores contém mais de 32.000 linhas de código.

A Figura 18 exibe o resultado dos testes de tempo de execução das diferentes implementações do algoritmo *Random Forest*. As implementações em PP (C) e PON (C++) utilizam o código gerado, enquanto a aplicação em linguagem de programação Python utiliza a função disponível na própria biblioteca, *model.predict(data)*. O desempenho deste algoritmo em PON, com o uso do *Framework* PON C++ 4.0, foi muito inferior ao desempenho da implementação em linguagem de programação C. Ainda assim, os tempos de execução foram menores que os da aplicação em linguagem de programação Python.

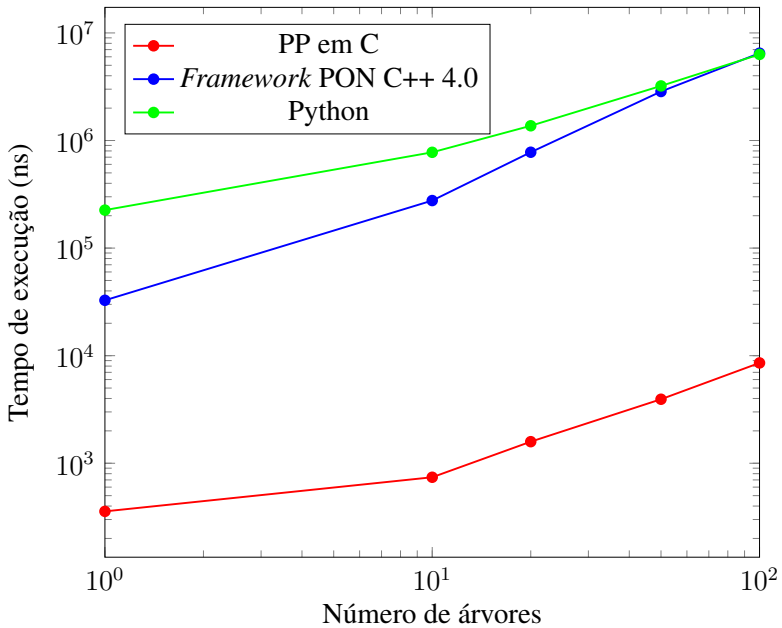


Figura 18 – Testes de desempenho da aplicação do algoritmo *Random Forest*
Fonte: Autoria própria

Além disso, ao contrário da aplicação *Bitonic Sort*, a aplicação de paralelismo, usando a

função *SetValue<NOP::Parallel>*, não resultou em redução nos tempos de execução. Conforme pode ser observado na Figura 19 a implementação paralelizada aumentou os tempos de execução quando comparada com a implementação sequencial. Isso pode ser justificado pelo fato da implementação utilizar diversos *Attributes* e *Premises* de gatilho, que, na prática, forçam a execução de certas etapas a ocorrer de forma sequencial, limitando os benefícios da paralelização. Ainda, conforme o gráfico da Figura 20 que ilustra o tempo de execução da implementação paralelizada relativo ao da implementação sequencial, a implementação paralelizada apresenta tempos de execução pelo menos cinco vezes mais lentos.

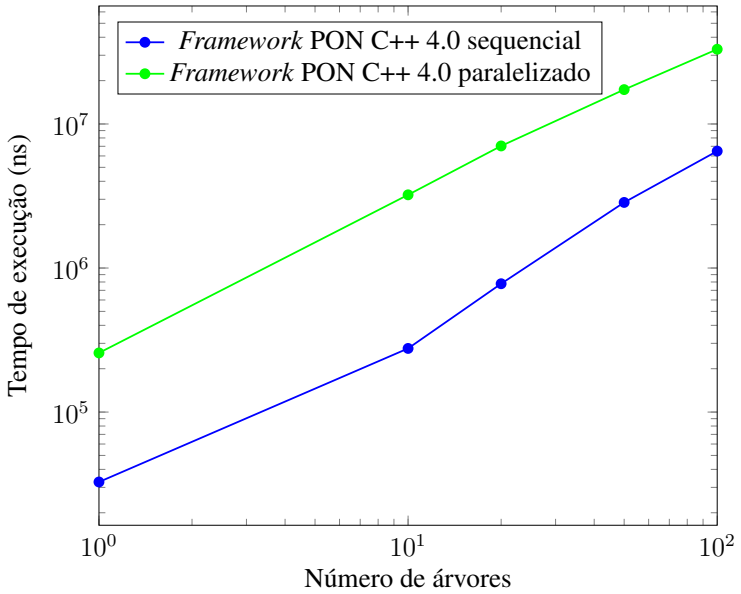


Figura 19 – Comparação da paralelização na aplicação do algoritmo *Random Forest*
Fonte: Autoria própria

Para avaliar a utilização de CPU foi utilizada a implementação do algoritmo utilizando 100 árvores. Do mesmo modo que a aplicação *Bitonic Sort*, a utilização de CPU para a implementação sequencial do *Random Forest* também fica limitada a 8%. Diferente do observado na aplicação *Bitonic Sort*, a utilização de CPU para a implementação sequencial do *Random Forest* não utiliza de forma eficiente todos os núcleos da CPU. De acordo com a Figura 22, o consumo de CPU fica próximo de apenas 50%. Isto é refletido no resultado dos tempos de execução que não apresentam redução na execução paralelizada.

Em suma, foi concluído que nesta implementação do algoritmo *Random Forest* em *Framework* PON C++ 4.0, a utilização de *Attributes* e *Premises* de gatilho limitam a paralelização. Essa limitação na paralelização faz com que não seja possível utilizar de forma equilibrada todos os núcleos da CPU, de modo que o desempenho não apresenta benefícios com a execução paralelizada. De fato, ocorre justamente o contrário, pois o custo de execução dos mecanismos

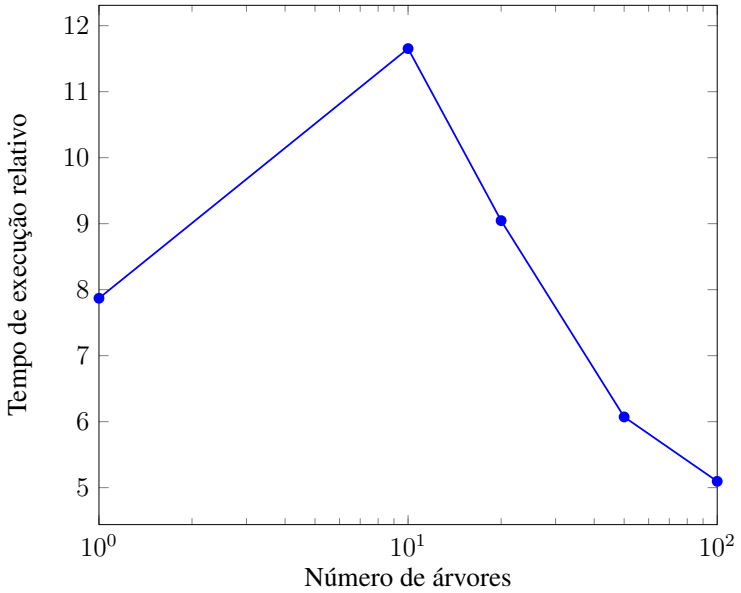
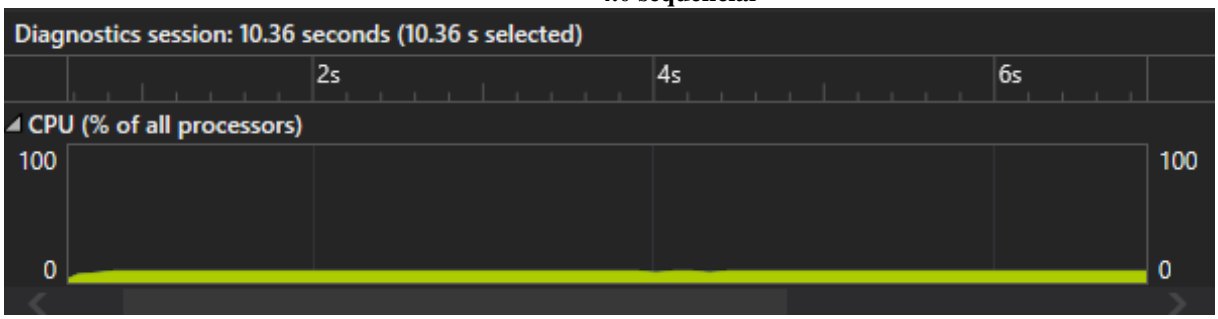


Figura 20 – Tempos de execução do algoritmo *Random Forest* com o *Framework* PON C++ 4.0 paralelizado relativo ao sequencial
Fonte: Autoria própria

de paralelização sobrepassa os benefícios de desempenho da mesma, aumentando o tempo de execução total da aplicação.

Figura 21 – Utilização de CPU durante execução do algoritmo *Random Forest* com o *Framework* PON C++ 4.0 sequencial



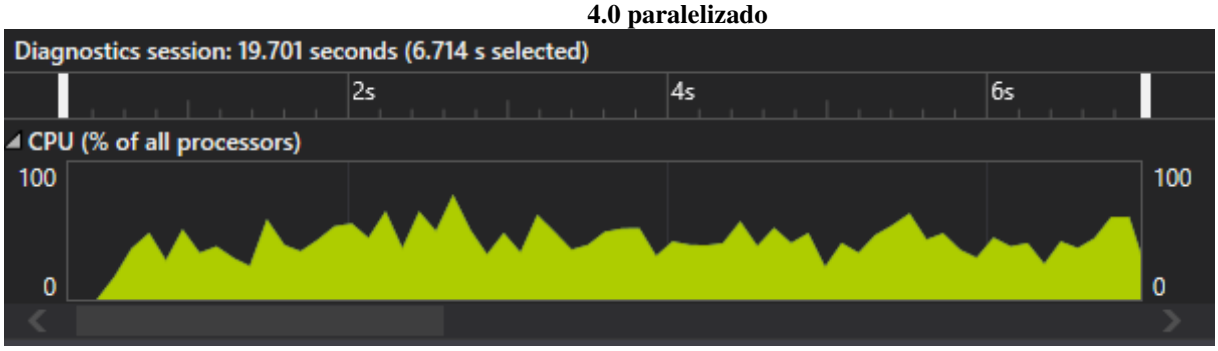
Fonte: Autoria própria

1.1.4 Aplicação de Controle de Tráfego Automatizado (CTA)

A aplicação de Controle de Tráfego Automatizado (CTA) é mais um caso de estudo frequentemente utilizado para avaliar o desempenho do PON. Esta aplicação foi escolhida por permitir avaliar o *Framework* PON C++ 4.0 do ponto de vista de implementação do paralelismo, por meio da comparação com outro *framework* que materializa esta propriedade, o *Framework* PON Elixir/Erlang (NEGRINI, 2019).

O ambiente de simulação desta aplicação proposto por RENAUX *et al.* (2015) é com-

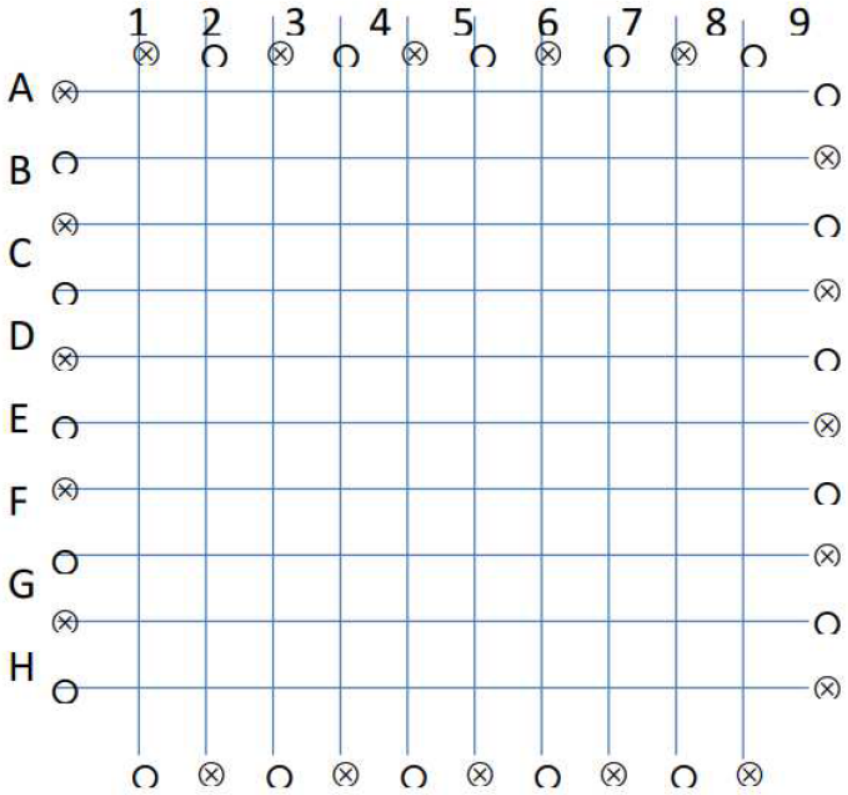
Figura 22 – Utilização de CPU durante execução do algoritmo *Random Forest* com o *Framework* PON C++ 4.0 paralelizado



Fonte: Autoria própria

posto por uma matriz 10x10, com um total de 100 interseções. As linhas e colunas da matriz representam ruas enquanto as interseções representam os cruzamentos com semáforos. Uma representação gráfica deste ambiente é apresentada na Figura 23.

Figura 23 – Ambiente de simulação

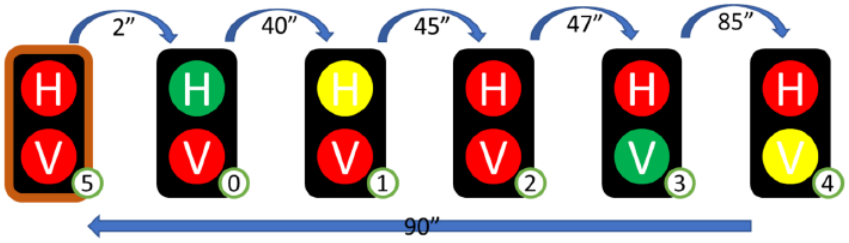


Fonte: RENAUX *et al.* (2015)

Neste experimento são consideradas duas estratégias que regem o comportamento dos semáforos, a estratégia de controle independente e estratégia de controle baseado em congestionamento facilitado (CBCF). Na estratégia de controle independente cada semáforo possui tempos fixos para cada estado (NEGRINI, 2019). O diagrama de estados com a temporização

para esta estratégia é apresentado na Figura 24. Neste diagrama as cores representam o estado em si (verde, amarelo e vermelho), enquanto os círculos H representam o estado do semáforo da via horizontal do cruzamento, e o círculo V representa o estado do semáforo da via vertical, de acordo com a disposição apresentada na Figura 23.

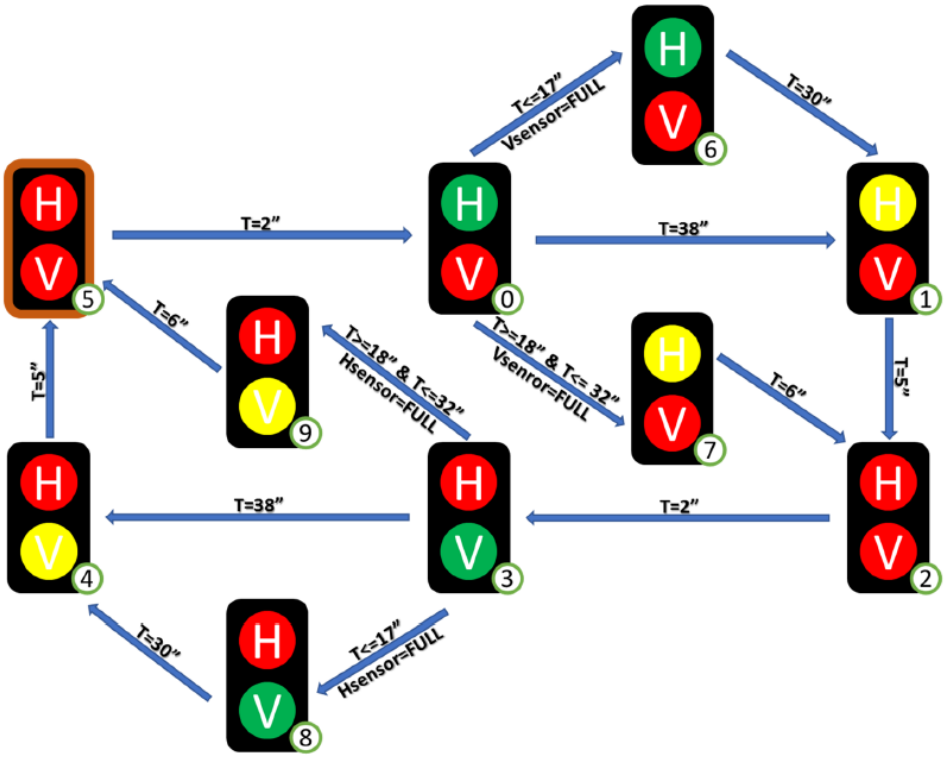
Figura 24 – Estados do CTA com estratégia independente



Fonte: Negrini (2019)

Na estratégia de controle CBCF, diferente da estratégia independente, um sensor é utilizado para detectar a porcentagem de veículos parados, sendo que se o sensor detecta que a porcentagem de veículos parados está acima de 60% e o tempo total do semáforo vermelho é menor que 30 segundos, ele tem seu tempo ajustado para 30 segundos. O diagrama de estados para esta estratégia é ilustrado na Figura 25.

Figura 25 – Estados do CTA com estratégia CBCF



Fonte: Negrini (2019)

Para a execução deste experimento, foi utilizado código em LingPON tanto para a aplicação o *Framework* PON C++ 4.0 como para o *Framework* PON Elixir/Erlang. Devido às limitações atuais no compilador LingPON para *Framework* PON C++ 4.0, o código foi modificado apenas de forma a utilizar o mecanismo de notificações paralelas (com *SetValue<NOP::Parallel>*). O Código 9 apresenta um trecho deste código em LingPON, enquanto o Código 10 apresenta um trecho do código gerado em *Framework* PON C++ 4.0, contendo um *Method* e uma *Rule*, para o *FBE* do semáforo na estratégia independente. Os códigos-fonte completos são disponibilizados no Apêndice J.

Código 9 – Trecho do FBE para o CTA na estratégia independente em LingPON

```
fbe Semaphore_CTA
   private integer atSemaphoreState = 5
   public integer atSeconds = 0
   private method mtResetTimer
      assignment
         this.atSeconds = 0
      end_assignment
   end_method
   private method mtHorizontalTrafficLightGREEN
      assignment
         this.atSemaphoreState = 0
      end_assignment
   end_method
   ...
   rule rlHorizontalTrafficLightGreen
      condition
            premise prSeconds
               this.atSeconds == 2
            end_premise
            and
            premise prSemaphoreState
               this.atSemaphoreState == 5
            end_premise
      end_condition
      action sequential
         instigation parallel
            call this.mtHorizontalTrafficLightGREEN()
         end_instigation
      end_action
   end_rule
   ...
end_fbe
```

Fonte: Autoria própria

Código 10 – Trecho do FBE para o CTA na estratégia independente em *Framework* PON C++ 4.0

```
SemaphoreCTA::SemaphoreCTA()
   : atSeconds{NOP::BuildAttribute<int>(0)},
     atSemaphoreState{NOP::BuildAttribute<int>(5)},
     prSeconds{NOP::BuildPremise<>(atSeconds, 2, NOP::Equal())},
     prSemaphoreState{
        NOP::BuildPremise<int>(atSemaphoreState, 5, NOP::Equal())},
     ...
   rlHorizontalTrafficLightGreen = NOP::BuildRule(
      NOP::BuildCondition(CONDITION(*prSeconds && *prSemaphoreState),
                     prSeconds, prSemaphoreState),
      NOP::BuildAction(NOP::BuildInstigation<NOP::Parallel>(
         METHOD(mtHorizontalTrafficLightGREEN();)))

   );
   ...
}

void SemaphoreCTA::mtHorizontalTrafficLightGREEN()
{
   atSemaphoreState->SetValue<NOP::Parallel>(0);
}
```

Fonte: Autoria própria

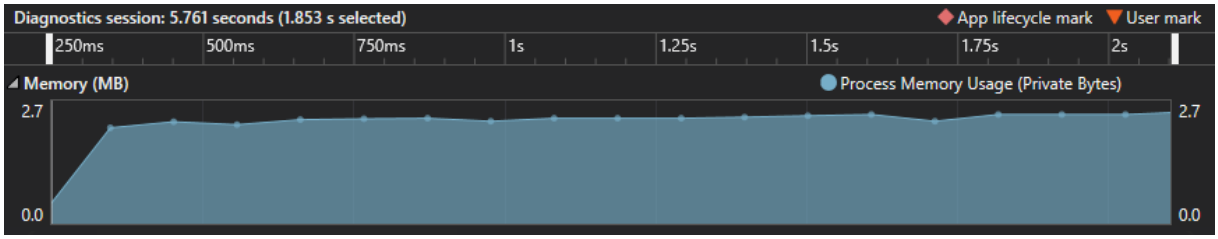
O experimento utilizado para a avaliação de desempenho consiste em iterar quatro mil vezes em uma matriz 10x10 de semáforos, sendo que cada iteração incrementa o tempo do *Attribute atSeconds* de cada semáforo. Desta forma, o experimento permite avaliar o tempo de execução da lógica, sem considerar o tempo de espera real entre os estados do semáforo, que existiriam em uma situação prática, mas que não são relevantes do ponto de vista de desempenho.

Para fins de comparação com o *Framework* PON Elixir/Erlang são considerados os resultados apresentados por Negrini (2019) utilizando o ambiente VM16, que utiliza uma instância de ambiente virtualizado na nuvem da Amazon com processadores AMD EPYC série 7000 com 16 núcleos a uma frequência de *clock* de 2,5 GHz em todos os núcleos (NEGRINI, 2019), enquanto os experimentos com o *Framework* PON C++ 4.0 *Framework* utilizaram um processador Ryzen 5 3600 com 6 núcleos e 12 *threads* a 3,6 GHz.

O consumo de memória da aplicação durante a execução com a estratégia de controle independente atingiu um máximo de 2,7 MB, enquanto para a estratégia de controle CBCF o consumo de memória checou a 4,8 MB, devido ao maior número de *Rules* utilizadas neste método de controle. O consumo de memória durante a execução das duas estratégias é mostrado nas Figuras 26 e 27. A aplicação com o *Framework* PON Elixir/Erlang consumiu, respectivamente, cerca de 800 MB e 700 MB nas estratégias independente e CBCF.

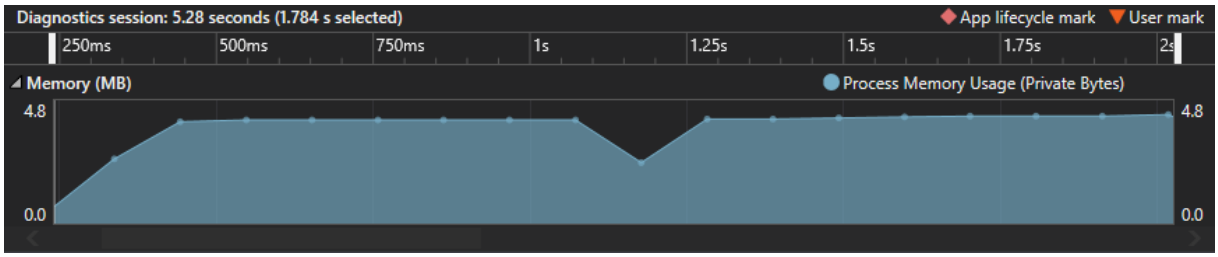
De modo a observar a paralelização o uso de CPU também foi observado o uso de CPU durante a execução das duas estratégias. Como pode ser observado nos gráficos das Figuras 28 e 29, o uso de CPU oscila entre cerca de 40% e 70% em ambos os cenários.

Figura 26 – Consumo de memória para a aplicação de semáforo com estratégia independente com o *Framework* PON C++ 4.0



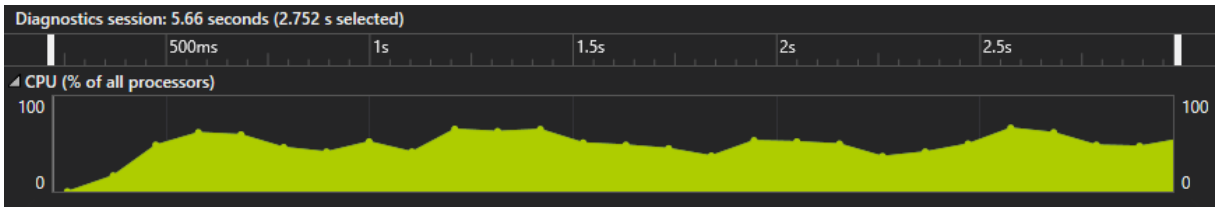
Fonte: Autoria própria

Figura 27 – Consumo de memória para a aplicação de semáforo com estratégia CBCF com o *Framework* PON C++ 4.0



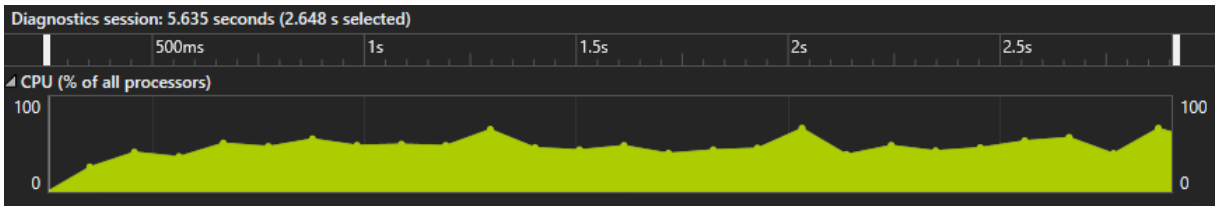
Fonte: Autoria própria

Figura 28 – Uso de CPU durante execução do Semáforo com estratégia independente com o *Framework* PON C++ 4.0



Fonte: Autoria própria

Figura 29 – Uso de CPU durante execução de semáforo com estratégia CBCF com o *Framework* PON C++ 4.0



Fonte: Autoria própria

O tempo de execução das estratégias de controle independente e CTBF foram de 336 ms e 434 ms respectivamente, utilizando o *Framework* PON C++ 4.0. No caso dos testes com o *Framework* PON Elixir/Erlang os tempos de execução foram de 4.703 ms e 8.525 ms. Esses resultados são comparados no gráfico da Figura 30. A aplicação com o *Framework* PON C++ 4.0 apresenta tempos de execução uma ordem de grandeza inferiores aos tempos de execução da mesma aplicação com o *Framework* PON Elixir/Erlang, mesmo considerando a execução da aplicação em Elixir/Erlang em um ambiente com poder de processamento muito superior, além

de apresentar consumo de memória mais de cem vezes menor.

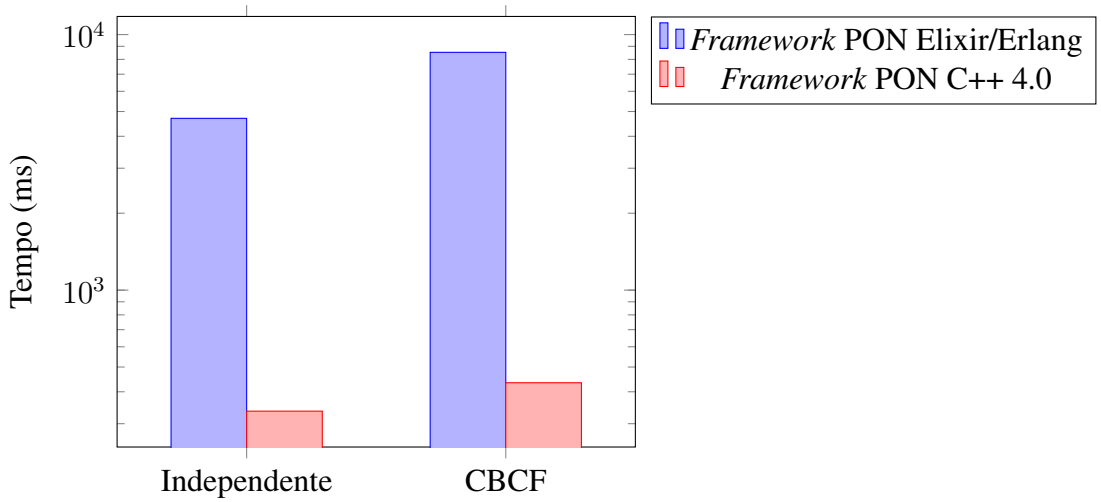


Figura 30 – Tempo de execução da aplicação de semáforo com o *Framework* PON C++ 4.0 *Framework* PON Elixir/Erlang
Fonte: Autoria própria

O desempenho superior do *Framework* PON C++ 4.0 era esperado, visto que Elixir é uma linguagem de programação de mais alto nível, utilizando o PF, com maior grau de abstração, que por sua vez é associada a custos computacionais elevados quando comparado a uma linguagem como C++. O objetivo desta comparação não é apenas observar como aplicações em C++ possuem desempenho superior a aplicações com Elixir/Erlang, pois ambas as linguagens de programação têm propostas diferentes, mas sim demonstrar no contexto do PON como o *Framework* PON C++ 4.0 permite o desenvolvimento de aplicações que usam paralelismo ao mesmo tempo que consegue entregar um alto desempenho com baixos tempos de execução e baixo consumo de memória, o que não é possível com outros *frameworks* do PON.

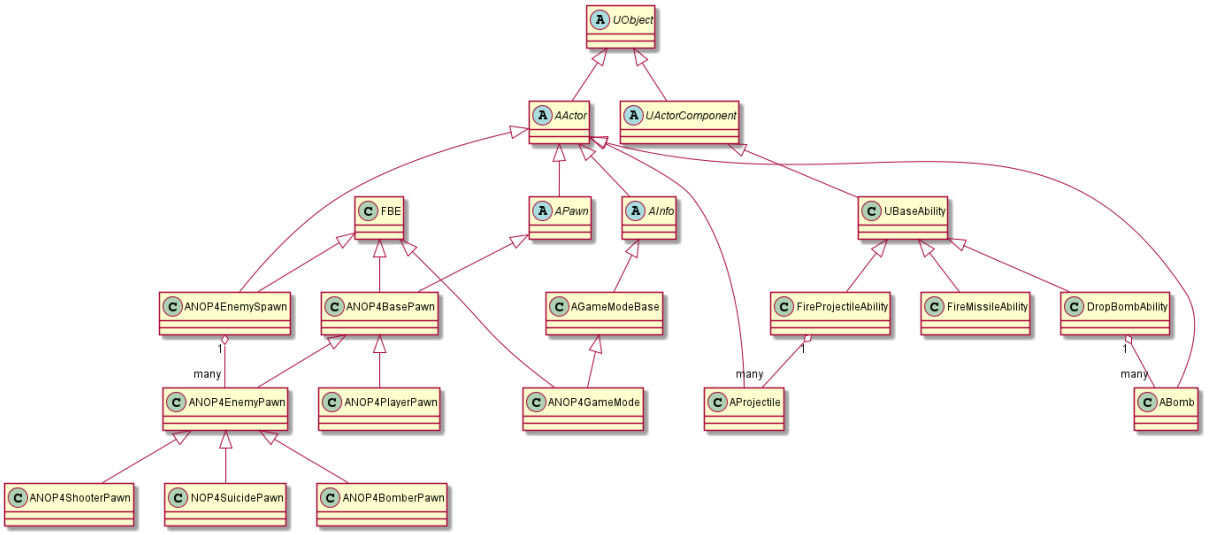
1.2 JOGO NOPUNREAL COM *FRAMEWORK* PON C++ 4.0

O jogo apresentado na Seção **??**, desenvolvido com o *Framework* PON C++ 2.0, foi reimplementado com o *Framework* PON C++ 4.0 de forma a permitir comparar a verbosidade do código nas duas implementações. A implementação é feita em parte diretamente no POO, utilizando as bibliotecas da Unreal Engine principalmente para os cálculos matemáticos e implementações de nível gráfico da *engine* em si, enquanto a parte do jogo relativa ao PON é dada pela lógica de controle automatizado dos inimigos. No total, o jogo contém 17 *Rules* em sua composição.

A implementação com o *Framework* PON C++ 4.0 utiliza como base a implementação

original com o *Framework* PON C++ 2.0, utilizando a mesma estrutura de classes, apenas modificando o código referente às *Rules*. Desta forma o diagrama de classes, apresentado na Figura 31, é equivalente ao da Figura **??**. Como ambos os *frameworks* apresentam funcionalidades equivalentes o processo de implementação com o *Framework* PON C++ 4.0 foi bastante simples, bastando converter o formato de código das estruturas do PON para o modelo do novo *framework*. Esse processo levou apenas uma tarde de trabalho. Como resultado, as funcionalidades do jogo foram mantidas.

Figura 31 – Diagrama de classes do jogo desenvolvido com o *Framework* PON C++ 4.0



Fonte: Autoria própria

Foi comparada a diferença na quantidade de código necessária para implementar o jogo utilizando o POO em C++, o *Framework* PON C++ 2.0 e o *Framework* PON C++ 4.0. Utilizando a ferramenta *cloc*[6], foi avaliado o número de linhas contidas nos códigos que compõem cada uma destas implementações. Conforme apresentado na Tabela 2, a implementação com o *Framework* PON C++ 4.0 conseguiu reduzir em 14.75% o número de linhas de código quando comparado com a implementação com o *Framework* PON C++ 2.0, e possui apenas 7.6% mais linhas de código que a implementação com o POO em C++. A implementação com o *Framework* PON C++ 4.0 permitiu converter diretamente o código com o *Framework* PON C++ 2.0 ao mesmo tempo que simplificou o código reduzindo o número de linhas de código totais.

[6] Ferramenta disponível em https://github.com/AlDanial/cloc

Tabela 2 – Linhas de código para a composição do jogo NOPUnreal

Versão	Linhas de código
POO em C++	1407
Framework PON C++ 2.0	1776
Framework PON C++ 4.0	1514

Fonte: Autoria própria

1.3 PESQUISA DE OPINIÃO DE DESENVOLVEDORES

De forma a permitir avaliar os resultados referentes à facilidade de uso de *framework*, que é algo que não há como avaliar de forma objetiva, apenas subjetiva, foi montado um questionário com diversas questões referentes ao uso dos *frameworks* do PON. O questionário completo está disponível no Apêndice K. O resultado desta pesquisa é apresentado meramente a fim de curiosidade, por contar com uma amostra muito pequena sem significância estatística devido ao pequeno número de desenvolvedores que já utilizaram o *Framework* PON C++ 4.0. O questionário contou com apenas três respostas[7], sendo que todas estas três pessoas possuem experiência utilizando tanto o *Framework* PON C++ 4.0 como o *Framework* PON C++ 2.0.

Foi perguntado aos desenvolvedores, numa escala de 0 a 5, o quanto consideravam que o *Framework* PON C++ 4.0 apresenta melhorias sobre o *Framework* PON C++ 2.0. Como pode ser observado na Figura 32, é consenso entre os desenvolvedores que o *Framework* PON C++ 4.0 apresenta melhorias significativas sobre o *Framework* PON C++ 2.0.

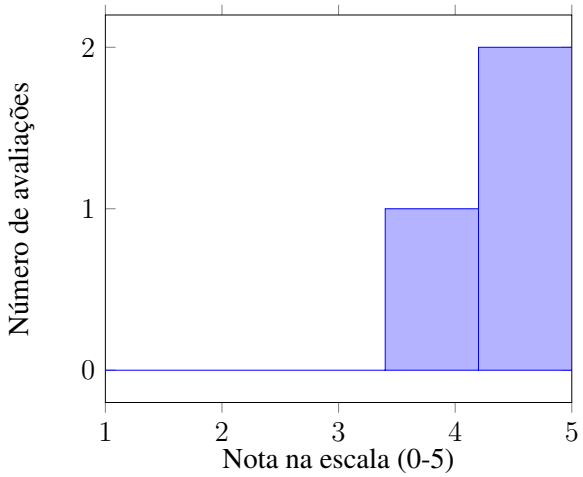


Figura 32 – Resultado da pesquisa de avaliação da melhoria do *Framework* PON C++ 4.0 sobre o *Framework* PON C++ 2.0
Fonte: Autoria própria

Em maiores detalhes, foi requisitado aos desenvolvedores avaliar ambos os *frameworks* de acordo com diversos critérios, como facilidade de aprendizado, facilidade de uso, verbosidade,

[7] Respostas dos alunos do grupo de pesquisa do PON Lucas Skora, Gustavo Chierici e Lucas Mamann.

versatilidade, desempenho e aderência aos fundamentos do PON. Para cada critério o usuário pode responder com fraco, moderado, satisfatório, muito bom ou excelente. Para facilitar a visualização dos resultados as respostas foram convertidas para uma escala de 1 a 5, onde 1 representa fraco, e 5 excelente, e o resultado considera a média aritmética das respostas. Conforme apresentado no gráfico da Figura 33, o *Framework* PON C++ 4.0 é considerado superior em todos os critérios avaliados.

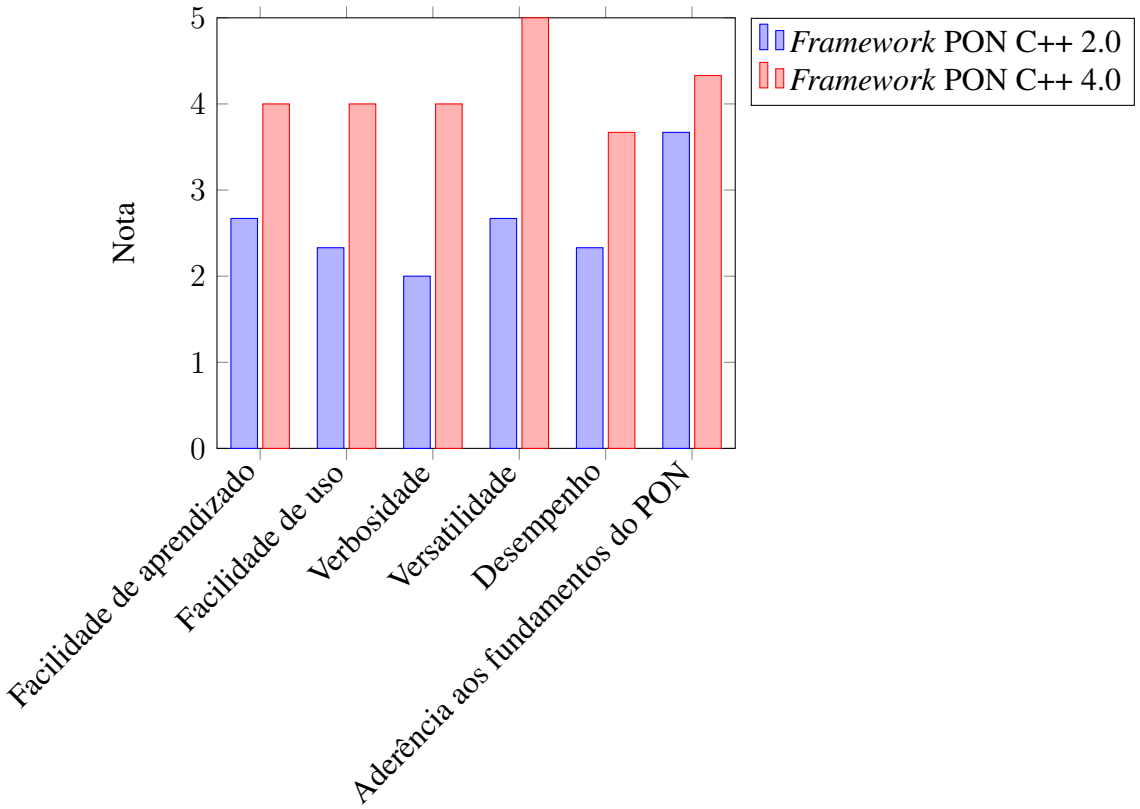


Figura 33 – Resultado da pesquisa de avaliação da melhoria do *Framework* PON C++ 4.0 sobre o *Framework* PON C++ 2.0
Fonte: Autoria própria

1.4 REFLEXÕES SOBRE OS RESULTADOS OBTIDOS

Por meio dos testes unitários realizados, é possível afirmar que o *Framework* PON C++ 4.0 consegue atender de forma satisfatória as funcionalidades necessárias para a construção de aplicações em PON. Considerando as implementações atuais, é possível comparar a diferença na quantidade de código necessária para implementar o *Framework* PON C++ 2.0 e o *Framework* PON C++ 4.0. Utilizando a ferramenta *cloc*, foi avaliado o número de linhas contidas nos códigos que compõem o *Framework* PON C++ 2.0 e o *Framework* PON C++ 4.0. Nessa comparação são consideradas apenas as linhas de código, desconsiderando linhas em branco e comentários.

Tabela 3 – Linhas de código para a composição do *framework*

Versão	Linhas de código
Framework PON C++ 2.0	6587
Framework PON C++ 4.0	970

Fonte: Autoria própria

A grande diferença no número de linhas de código se dá principalmente pela estrutura genérica do *Framework* PON C++ 4.0 que permite eliminar a repetição de código que era presente no *Framework* PON C++ 2.0. Essa redução no número de linhas de código é importante, pois isso facilita a manutenção do código por outros desenvolvedores, pois reduz o tamanho da base de código com a qual um novo desenvolvedor precisa se habituar ao trabalhar com o código, assim como torna mais simples eventuais alterações e melhorias na estrutura do *framework*.

Dentre os objetivos da implementação do *Framework* PON C++ 4.0 estão aumentar facilidade de uso e reduzir da curva de aprendizado, entretanto esses critérios são bastante subjetivos. Ademais, com o desenvolvimento do jogo NOPUnreal foi possível constatar de forma objetiva a simplificação do código por meio da redução do número de linhas de código utilizadas na implementação do mesmo com o *Framework* PON C++ 4.0 quando comparado com a implementação com o *Framework* PON C++ 2.0.

A adição de novos recursos da linguagem de programação C++ ao *Framework* PON C++ 4.0, como o uso de *smart pointers*, e o uso de expressões *lambda*, além de facilitar o uso e desenvolvimento, não trouxe prejuízos no desempenho uma vez comparado com o *Framework* PON C++ 2.0 quando aplicado para o desenvolvimento da aplicação de sensores, sendo que o *Framework* PON C++ 4.0 apresentou tempos de execução menores que o *Framework* PON C++ 2.0 neste caso. Entretanto, essa melhoria de desempenho ainda não foi suficiente a ponto de superar o desempenho da aplicação no POO em C++ para este cenário.

No entanto, a utilização desses recursos mais avançados do C++ no *Framework* PON C++ 4.0 causa um aumento na complexidade do código, principalmente quando comparado com o *Framework* PON C++ 2.0, devido à utilização de construções mais novas e com sintaxe rebuscada (como expressões *lambdas*, *fold expressions*). Ainda assim são apresentadas soluções que permitem abstrair essas construções mais complexas de modo suficiente para facilitar a aplicação por desenvolvedores com conhecimento menos avançado dessas sintaxes.

O desenvolvimento da aplicação *Bitonic Sort* demonstra o potencial do PON para o desenvolvimento de algoritmos paralelizáveis, apesar dos elevados tempos de execução e consumo de memória, visto que parte destes custos computacionais podem ser atribuídos à

implementação do *Framework* PON C++ 4.0.

Além disso, ao se avaliar o desempenho da versão paralelizada da aplicação *Bitonic Sort* pode se observar que a aplicação da paralelização no mecanismo de notificações pode resultar em ganhos de desempenho referentes ao tempo de execução em determinados cenários. Porém, a paralelização também pode causar degradação no desempenho, como no caso da aplicação *Random Forest*, dependendo da estrutura da aplicação, devido ao natural custo computacional atrelado ao processo de execução de *threads* em C++.

Nem toda aplicação que pode ser paralelizada consegue obter ganhos de desempenho como, por exemplo, a implementação do algoritmo *reverse* da STL paralelizado, que no compilador para C++ da Microsoft (MSVC) teve desempenho 1.6 vezes mais lento que a implementação sequencial. Isso não significa que estes algoritmos não devam ser paralelizados, mas sim que o *hardware* atual para o qual o código é compilado não traz ganho de desempenho (O'NEAL, 2018).

Por fim, principalmente durante o desenvolvimento das aplicações para os algoritmos *Bitonic Sort* e *Random Forest*, foi possível constatar a maneira como o *Framework* PON C++ 4.0 facilita o desenvolvimento de aplicações em PON, pois o desenvolvimento destas mesmas aplicações com o *Framework* PON C++ 2.0 não foi realizado justamente por ser extremamente complicada. Por exemplo, na implementação do algoritmo para o *Bitonic Sort*, as *Rules* são criadas de forma dinâmica durante a inicialização, o que só foi possível realizar de forma simples devido aos facilitadores de desenvolvimento, por meio dos *Builder* e gerenciamento de memória com *smart pointers*, introduzidos pelo *Framework* PON C++ 4.0.

O *Framework* PON C++ 4.0 avança sobre o que já havia sido apresentado no *Framework* PON C++ 2.0 no que diz respeito à programação em alto nível ao disponibilizar interfaces de programação mais fáceis de utilizar, dessa forma contribuindo para a programação em alto nível. O *Framework* PON C++ 4.0 também é o primeiro *framework* na linguagem C++ a contemplar de forma satisfatória a propriedade de paralelismo do PON, ao usar recursos modernos da linguagem C++ que permitem o desenvolvimento de aplicações *multithread* de forma bastante simples. Os benefícios da aplicação do paralelismo no *framework* são particularmente destacados nos resultados da aplicação *Bitonic Sort* apresentados na Seção 1.1.2, enquanto na Seção 1.1.3, com a aplicação do algoritmo *Random Forest*, demonstra que existem casos em que a paralelização pode não trazer melhorias ao desempenho. Em suma, os benefícios da paralelização dependem das características da aplicação, sendo que o *Framework* PON C++ 4.0 disponibiliza as ferramentas

necessárias que permitem ao desenvolvedor facilmente utilizar a paralelização.

Além disso, no tocante à propriedade do paralelismo, o desenvolvimento da aplicação de controle de semáforos permitiu comparar o *Framework* PON C++ 4.0 com o *Framework* PON Elixir/Erlang. Com este experimento, fica demonstrada a capacidade de paralelismo do *Framework* PON C++ 4.0, cujos níveis de utilização de CPU atingiram níveis altos, comparáveis aos do *Framework* PON Elixir/Erlang, ao mesmo tempo que mantém tempos de execução mais baixos, característicos das implementações de *framework* do PON em C++. O *Framework* PON C++ 4.0 consegue materializar propriedades de desempenho e paralelismo do PON simultaneamente, algo que não foi atingido por nenhum dos outros *frameworks* existentes.

No que diz respeito ao desempenho, como era esperado, o *Framework* PON C++ 4.0 não consegue apresentar os ganhos esperados quando comparado a aplicações no PI devido ao alto custo computacional das estruturas utilizadas na composição do *framework* em si. Entretanto, quando comparado com as outras materializações em C++, é possível constatar que o *Framework* PON C++ 4.0 apresenta uma melhora no desempenho das aplicações, reduzindo tanto os tempos de execução como consumo de memória.

REFERÊNCIAS

BANASZEWSKI, R. F. **Paradigma Orientado a Notificações: Avanços e Comparações**. 2009. Dissertação (Mestrado) — UTFPR, 2009.

BATCHER, Kenneth. Sorting networks and their applications. *In*: . *[S.l.: s.n.]*, 1968. v. 32, p. 307–314.

CRIMINISI, Antonio; SHOTTON, Jamie; KONUKOGLU, Ender. Decision forests: A unified framework for classification, regression, density estimation, manifold learning and semi-supervised learning. **Foundations and Trends in Computer Graphics and Vision**, v. 7, p. 81–227, 01 2011.

GOOGLE. **google/benchmark**. 2020. Disponível em: https://github.com/google/benchmark.

MULLAPUDI, Amrutha. **Bitonic Sort**. *[S.l.]*: University at Buffalo, 2014. Lecture for CSE633: Parallel Algorithms (Spring 2014).

NEGRINI, F. **Tecnologia NOPL Erlang-Elixir – paradigma orientado a notificações via uma abordagem orientada a microatores assíncronos**. 2019. Dissertação (Mestrado) — UTFPR, CPGEI, 2019.

O'NEAL, Billy. **Using C++17 Parallel Algorithms for Better Performance**. Microsoft, 2018. Disponível em: https://devblogs.microsoft.com/cppblog/using-c17-parallel-algorithms-for-better-performance/.

PITSIANIS, Nikos. **CMake**. 2008. Disponível em: https://courses.cs.duke.edu//fall08/cps196.1/Pthreads/bitonic.c.

PORDEUS, L.F.; LINHARES, R.R.; STADZISZ, P.C.; SIMãO, J.M. Nop-dh – evaluation over bitonic sort algorithm. **Microprocessors and Microsystems**, p. 104314, 2021. ISSN 0141-9331. Disponível em: https://www.sciencedirect.com/science/article/pii/S0141933121004750.

PORDEUS, L. F. **ArqTotalPON - Contribuição para Arquitetura de Computação própria e efetiva ao Paradigma Orientado a Notificações**. 2020. Dissertação (Mestrado) — UTFPR, CPGEI, 2020.

RENAUX, Douglas P. B.; LINHARES, Robson Ribeiro; SIMAO, Jean Marcelo; STADZISZ, Paulo Cézar. **CTA SImulator - Concepts of Operations**. 2015. Disponível em: http://www.dainf.ct.utfpr.edu.br/~douglas/CTA_CONOPS.pdf.

SKORA, Lucas Eduardo Bonancio. **Criação de alvos de compilação para a NOPL e ferramentas de serialização-desserialização para o Grafo PON**. *[S.l.]*: CPGEI/UTFPR, 2020. Trabalho realizado na disciplina Paradigmas de Programação.

XAVIER, R. D. **Paradigmas de desenvolvimento de software: comparação entre abordagens orientada a eventos e orientada a notificações**. 2014. Dissertação (Mestrado) — UTFPR, PPGCA, 2014.

APÊNDICES

APÊNDICE A

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DIRETORIA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO APLICADA (PPGCA)
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA E
INFORMÁTICA INDUSTRIAL (CPGEI)

AVANÇOS NA APLICAÇÃO DE FUTEBOL DE ROBÔS EM *FRAMEWORK* PON C++ 2.0 DO PARADIGMA ORIENTADO A NOTIFICAÇÕES – PON

Disciplinas:
Tópicos Especiais em EC: Paradigma Orientado a Notificações - TEC0301 (CPGEI)
Tópicos Avançados em Sistemas Embarcados 2 - CASE102 (PPGCA)

Alunos:
1. Anderson Eduardo de Lima
2. Felipe dos Santos Neves
3. Lucas Tachini Garcia
4. Luis Henrique Sant'Ana
5. Omero Francisco Bertol

CURITIBA
2020

LISTA DE FIGURAS

LISTA DE QUADROS

SUMÁRIO

1 INTRODUÇÃO

Nestas considerações iniciais serão apresentadas a contextualização do problema objeto deste trabalho acadêmico, os objetivos e a organização estrutural das seções que constituem este relatório técnico.

1.1 CONTEXTUALIZAÇÃO

Este trabalho tem por objetivo relatar o que foi colocado em prática no tocante aos tópicos abordados nas disciplinas referentes ao Paradigma Orientado a Notificações (PON) registradas como “Tópicos Especiais em EC: Paradigma Orientado a Notificações – TEC0301” no Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI) e como “Tópicos Avançados em Sistemas Embarcados 2 – CASE102” no Programa de Pós-Graduação em Computação Aplicada (PPGCA).

Para experimentar as técnicas explanadas nas disciplinas em questão em uma aplicação real, foi desenvolvido um projeto orientado a regras e baseado nos conceitos e propriedades do PON para um simulador de Futebol de Robôs (RoboCup) executado em máquina virtual.

O desenvolvimento deste estudo deve contemplar o regulamento de funcionamento de uma competição de Futebol de Robôs (RoboCup), um estudo teórico e prático dos elementos fundamentais da tecnologia PON, a configuração de um ambiente de simulação do Futebol de Robôs em máquina virtual (Apêndice A), a análise de *Regras* (*Rules*) implementadas em trabalhos existentes (Apêndices B e C) e o desenvolvimento no *Framework* PON C++ 2.0 de um conjunto de *Rules* para um software de controle de uma partida de Futebol de Robôs (Apêndices D e E).

1.2 OBJETIVOS

Com o objetivo de colocar em prática os conceitos vistos em classe no contexto das disciplinas supracitadas, foi proposto o desenvolvimento de um aplicativo de controle de robôs em tecnologia PON, em contexto de Futebol de Robôs, para execução em ambiente de simulação.

1.3 ESTRUTURA DO TRABALHO

Este relatório técnico está dividido em 5 seções e 5 apêndices. Nesta presente e primeira seção, nomeadamente “Introdução”, foi explicado o assunto tema do trabalho e também foram abordados a motivação, os objetivos e a estrutura geral do trabalho.

Subsequentemente, na segunda seção nomeadamente “Paradigma Orientado a Notificações – PON” será mostrada a estrutura do PON, seus elementos essenciais na forma de entidades *Atributos* (*Attributes*), *Métodos* (*Methods*), *Condição* (*Condition*), *Premissas* (*Premises*), *Ação* (*Action*), *Instigações* (*Instigations*) e *Regras* (*Rules*), o fluxo de execução no PON e a apresentação das linguagens de programação que materializam a tecnologia PON.

Por sua vez, na terceira seção, nomeadamente “Informações sobre o Funcionamento do Futebol de Robôs”, são apresentadas informações sobre as *Rules* de como funciona uma partida de Futebol de Robôs (RoboCup).

Notadamente, na quarta seção, nomeadamente “Desenvolvimento da Aplicação para Controle do Futebol de Robôs”, será apresentado o desenvolvimento do software de controle dos robôs visando a simulação de uma partida de Futebol de Robôs em ambiente de simulação, o qual foi disponibilizado em uma máquina virtual.

Por fim, na quinta e última seção, nomeadamente de “Conclusão”, serão apresentadas as considerações finais apontando as conclusões alcançadas com o desenvolvimento deste trabalho acadêmico.

Outrossim, no “Apêndice A: Utilização do Ambiente de Futebol de Robôs em Máquina Virtual”, será apresentado um tutorial com os passos para configuração da máquina virtual e a execução do ambiente de simulação do Futebol de Robôs.

Ainda, no “Apêndice B: Projeto RoboCup2012 em Ambiente de Simulação” será detalhado o projeto do aplicativo de controle do Futebol de Robôs “RoboCup2012” em tecnologia PON, para execução no ambiente de simulação e que foi disponibilizado para este estudo.

Por sua vez, o “Apêndice C: Projeto RoboCup2012 Desenvolvido em Santos (2017)”, será discutido o trabalho de Santos (2017) que realizou a proposição de uma nova versão da linguagem de programação específica para o Paradigma Orientado a Notificações (PON), nomeada LingPON. A versão 1.2 implementada da LingPON foi empregada no desenvolvimento de um software que controla partidas de Futebol de Robôs (RoboCup) e que foi então comparado com outros dois softwares equivalentes, a saber, desenvolvidos utilizando o *Framework* PON C++ 2.0 e a LingPON versão 1.0. Os softwares desenvolvidos em Santos (2017) nas diferentes linguagens de programação que materializam a tecnologia PON foram também disponibilizados para a realização deste estudo.

Em tempo, as *Rules* implementadas pelos discentes no Projeto “RoboCup2012” em ambiente de simulação serão apresentadas no Apêndice D as *Rules* implementadas pelos alunos Anderson Eduardo de Lima, Felipe dos Santos Neves, Lucas Tachini Garcia e Omero Francisco Bertol; e no Apêndice E as *Rules* implementas pelo aluno Luis Henrique Sant’Ana.

2 PARADIGMA ORIENTADO A NOTIFICAÇÕES – PON

Nesta segunda seção serão feitas as considerações inicias sobre o Paradigma Orientado a Notificações (PON), seus elementos essenciais na forma de *Atributo*s (*Attributes*), *Métodos* (*Methods*), *Condição* (*Condition*), *Premissas* (*Premises*), *Ação* (*Action*), *Instigações* (*Instigations*) e *Regras* (*Rules*), o fluxo de execução no PON e a apresentação das linguagens de programação que materializam a tecnologia PON.

2.1 ELEMENTOS FUNDAMENTAIS DO PON

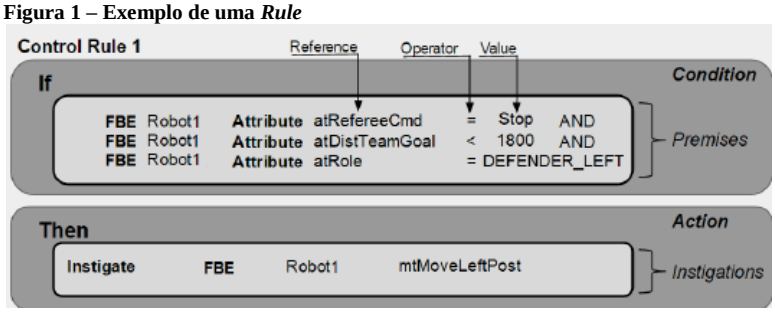
O Paradigma Orientado a Objeto (POO), classificado como subparadigma do Paradigma Imperativo, ou os Sistemas Baseados em Regras, englobados pelo Paradigma Declarativo, sofrem limitações intrínsecas de seus paradigmas (RONSZCKA, 2012, p. 27).

Esses paradigmas levam ao forte acoplamento em expressões lógico causais deste modo levando ao processamento desnecessário com redundâncias causais e/ou utilização custosa de estruturas de dados. Essas limitações com frequência comprometem o desempenho das aplicações. Nesse contexto, existem motivações para buscas de alternativas aos Paradigma Imperativo e Paradigma Declarativo, com o objetivo de diminuir ou eliminar as desvantagens desses paradigmas (RONSZCKA, 2012, p. 28).

Nesse contexto, o Paradigma Orientado a Notificações (PON) foi apresentado como uma alternativa. O PON pretende eliminar algumas deficiências dos paradigmas usuais em relação à avaliação de expressões causais desnecessárias e acopladas. Para tal, o PON faz uso de um mecanismo baseado no relacionamento de entidades computacionais que proporcionam um fluxo de execução de maneira reativa através de notificações precisas e pontuais (RONSZCKA, 2012, p. 28).

Na estrutura do Paradigma Orientado a Notificações (PON), as entidades computacionais que possuem *Atributo*s (*Attributes*) e *Métodos* (*Methods*) são genericamente chamadas de *Fact Base Elements* (FBEs). Por meio de seus *Attributes* e *Methods*, as entidades FBEs são passíveis de correlação lógico-casual, normalmente proveniente de regras “se-então” (ou “If-Then”), por meio de entidades *Rules*, esquematizada na Figura 1, o que constituem elementos fundamentais do PON (SANTOS, 2017, p. 19; RONSZCKA, 2019, p. 28).

Figura 1 – Exemplo de uma *Rule*



Fonte: Santos (2017, p. 19).

Uma *Rule*, como mostra novamente a Figura 1, é decomposta em entidades *Condition* (*Condição*, trata da decisão da *Rule*) e *Action* (*Ação*, trata da execução das ações associadas a *Rule*). Por sua vez, a *Condition* é decomposta em uma ou mais entidades *Premises* (*Premissas*, realiza as verificações que definem a tomada de decisão) e uma *Action* é decomposta em uma ou mais entidades *Instigations* (*Instigações*, "instiga" *Methods* responsáveis por realizar serviços ou habilidades) (BANASZEWSKI, 2009, p. 10; SANTOS, 2017, p. 19; VALENÇA, 2012, p. 8-9; RONSZCKA, 2019, p. 28-29).

2.2 ATTRIBUTES E METHODS

Todo e qualquer recurso de uma aplicação expressa os estados ou valores de situações por meio de entidades chamadas de *Atributos* (*Attributes*), bem como disponibiliza seus serviços ou funcionalidades por meio de entidades chamadas de *Métodos* (*Methods*) (BANASZEWSKI, 2009, p. 9).

O conceito de *Atributos* (*Attributes*) e *Métodos* (*Methods*) representam uma evolução dos conceitos de *Atributos*, que definem a porção de dados; e *Métodos*, que implementam o comportamento, de classe do Paradigma Orientado a Objetos. A diferença, segundo Ronszcka (2012), está no desacoplamento implícito da classe proprietária e a colaboração pontual para com as entidades *Premises* e *Instigations*.

No contexto do desenvolvimento do aplicativo em tecnologia PON para controle de uma partida de Futebol de Robôs, tem-se como exemplo de *Attribute* PON o elemento "atClosestToGoal" (Quadro 1, Linha 4), implementado para definir se o jogador está próximo (*true*) ao gol ou não (*false*). Já no caso do *Attribute* "atPoxX" (Quadro 1, Linha 5), ele será utilizado para representar o valor da posição do robô em relação ao eixo "X" do gráfico cartesiano (corresponde a distância da linha de gol defendido, até a linha de gol atacado).

Quadro 1 – Exemplo da declaração de *Attributes e MethodPointers*

```
// Declaração dos Attributes "atClosetToGoal" e "atPosX" na classe "RobotPON"
// no arquivo de cabeçalhos "RobotPON.h"
1.  class RobotPON : public Robot, public FBE {
2.  public:
3.      …
4.      Boolean* atClosestToGoal;
5.      Double* atPosX;
6.  };
```

Fonte: Autoria própria.

Em tempo, também no contexto do aplicativo de controle de uma partida de Futebol de Robôs, tem-se como exemplos de *Methods* PON os elementos "*mtClosestToGoal*" e "*mtNotClosestToGoal*" (Quadro 2, Linhas 4-5). Estes elementos do tipo *MethodPointers* permitem que métodos implementadas em C++ sejam utilizadas diretamente como *Methods*. Eles são criados atrelando o método que modifica o estado de um jogador para indicar se ele está próximo ao gol (*Attribute* "atClosestToGoal" é setado com o valor *true*), em "closestToGoal" (Quadro 2, Linhas 7 e 15-17); ou não (*Attribute* "atClosestToGoal" é setado com o valor *false*), com o método "notClosestToGoal" (Quadro 2, Linhas 8 e 18-20).

Quadro 2 – Exemplo da implementação de *MethodsPointers*

```
// Implementação das interfaces dos Methods na classe "StrategyPON"
// no arquivo de cabeçalhos "StrategyPON.h"
1.  class StrategyPON : public Strategy, public FBE {
2.  protected:
3.      …
4.      MethodPointer<StrategyPON>* mtClosestToGoal;
5.      MethodPointer<StrategyPON>* mtNotClosestToGoal;
6.      …
7.      void closestToGoal();
8.      void notClosestToGoal();
9.  };

// Criação das Instigations "mtClosestToGoal" e "mtNotClosestToGoal"
// no arquivo de códigos-fontes StrategyPON.cpp Method que instância as Instigations
10. void StrategyPON::initMethodPointers() {
11.     …
12.     mtClosestToGoal = new MethodPointer<StrategyPON>(this, &StrategyPON::closestToGoal);
13.     mtNotClosestToGoal = new MethodPointer<StrategyPON>(this,
                &StrategyPON::notClosestToGoal);
14. }

// Implementação do comportamento dos Methods no arquivo de códigos-fontes StrategyPON.cpp
15. void StrategyPON::closestToGoal(){
16.     this->robot->atClosestToGoal->setValue(true);
17. }

18. void StrategyPON::notClosestToGoal(){
19.     this->robot->atClosestToGoal->setValue(false);
20. }
```

Fonte: Autoria própria.

2.3 CONDITION E PREMISES

As *Premissas* (*Premises*) que compõem a entidade *Condição* (*Condition*), como apresentadas no exemplo de *Rules* ainda na Figura 1, são responsáveis pela realização das verificações que definem a tomada de decisão de uma *Rule*.

A implementação da tecnologia PON, no aplicativo de controle de uma partida de Futebol de Robôs, no que diz respeito as *Premises* está exemplificada no Quadro 3. No primeiro momento do exemplo, declara-se a *Premise* "prNotClosestToGoalX" como um ponteiro (*) para objetos da classe "Premise" (Quadro 3, Linha 4). Na segunda etapa do processo no método "initPremises", a *Premise* "prNotClosestToGoalX" é instanciada a partir da classe "Premise" no método "PREMISE" com o seguinte pseudocódigo da expressão lógica: *atPosX <= 7985* (Quadro 3, Linha 8). Neste contexto, a *Premise* "prNotClosestToGoal" é responsável por verificar se um robô "não" está próximo a linha do gol atacado verificando se o estado/valor do *Attribute* "atPosX" (Quadro 1, Linha 5) é menor ou igual (SMALLERTHAN) ao valor *double* 7985 (*new Double(this, 7980)*).

Quadro 3 – Exemplo da implementação de *Premises*

```
// Declaração do Attribute "prClosetToGoal", como um ponteiro para objetos da classe "Premise",
// no arquivo de cabeçalhos "StrategyPON.h"
1.  class StrategyPON : public Strategy, public FBE {
2.  protected:
3.    …
4.    Premise *prNotClosestToGoalX;
5.  };

// Method que instância as Premises: criação da Premise "prClosestToGoal"
6.  void StrategyPON::initPremises() {
7.    ...
8.    PREMISE(prNotClosestToGoalX, this->robot->atPosX, new Double(this, 7980),
        Premise::SMALLERTHAN, Premise::STANDARD, false);
9.  }
```

Fonte: Autoria própria.

2.4 ACTION E INSTIGATIONS

As *Instigações* (*Instigations*), que compõem a entidade *Ação* (*Action*), como demonstradas no exemplo de *Rules* novamente na Figura 1, são responsáveis através de seus *Methods* pela realização dos serviços ou habilidades associadas a uma *Rule*.

A demonstração da forma de implementação das *Instigations*, novamente no aplicativo de controle da partida de Futebol de Robôs, pode ser observada no Quadro 2. A *Instigation* é criada e atrelada diretamente ao objeto *Rule* "rlNotClosestToGoalX" (Quadro 4, Linha 4) por meio do método INSTIGATION (Quadro 4, Linha 10). Esta

Instigation instigará o *Method* “mtNotClosestToGoal” (Quadro 2, Linha 5).

2.5 RULES

Uma *Rule* é decomposta em entidades *Condition* (condição, conjunto de *Premises*) e *Action* (ação, conjunto de *Instigations*), como pode ser observado ainda na Figura 1. A *Condition* é formada pelas *Premises* associadas à *Rule* de modo que a ativação delas define a execução ou não da *Action*. Cada *Rule* desempenha o papel de conjunção ou disjunção unicamente para todas as *Conditions* em que está atrelada.

Uma das *Rules* implementada no aplicativo de controle da partida de Futebol de Robôs (Apêndice D), pode ser observada no Quadro 4. Primeiramente, declaram-se a *Rule* “rlNotClosestToGoalX” como um ponteiro (*) para objetos da classe “RuleObject” (Quadro 4, Linha 4). A seguir, na próxima etapa do processo no *Method* “initRules”, cria-se a *Rule* com 3 (três) linhas de implementações: a) *Rule* “rlNotClosestToGoalX” é instânciada a partir da classe “RuleObject” no *Method* “RULE” (Quadro 4, Linha 8); b) adiciona-se a *Premise* “prNotClosestToGoalX” (Quadro 3, Linha 4) na *Rule* “rlNotClosestToGoalX” (Quadro 4, Linha 9); e c) adiciona-se a *Instigation* criada a partir do *MethodPointer* “mtNotClosestToGoal” com o método INSTIGATION na *Rule* “rlNotClosestToGoalX” (Quadro 4, Linha 10).

Quadro 4 – Exemplo da implementação de *Rules*

```
// Declaração da Rule "rlNotClosetToGoalX", como um ponteiro para objetos da classe "RuleObject"
// no arquivo de cabeçalhos StrategyPON.h
1.  class StrategyPON : public Strategy, public FBE {
2.  protected:
3.     ...
4.     RuleObject* rlNotClosestToGoalX;
5.  };

// Method que instância as Rules: criação da Rule "rlNotClosestToGoalX"
6.  void StrategyPON::initRules(){
7.     …
8.     RULE(rlNotClosestToGoalX, scheduler, Condition::SINGLE);
9.     rlNotClosestToGoalX->addPremise(prNotClosestToGoalX);
10.    rlNotClosestToGoalX->addInstigation(INSTIGATION(this->mtNotClosestToGoal));
11. }
```

Fonte: Autoria própria.

A *Rule* “rlNotClosestToGoalX”, implementada novamente no Quadro 4, se comporta de forma a determinar se o robô “não” está próximo do gol em relação a coordenada “x”, valor do *Attribute* “atPosX” (Quadro 1, Linha 5), expressão lógica implementada na *Premise* “prNotClosestToGoalX” (Quadro 3, Linha 4). Se não está próximo ao gol, então define o estado do *Attribute* “atClosestToGoal” (Quadro 1, Linha

4) como *false*, comportamento implementação no *Method* “mtNotClosestToGoal” (Quadro 2, Linha 11).

2.6 FLUXO DE EXECUÇÃO NO PON

No Paradigma Orientado a Notificações (PON) o fluxo de execução de uma aplicação inicia-se a partir da mudança de estado de um *Attribute* que “notifica” todas as *Premises* pertinentes, a fim de que estas reavaliem seus estados lógicos. Caso o valor lógico de uma *Premise* se altere, a *Premise* colabora com a avaliação lógica de uma ou de um conjunto de *Conditions* conectadas, o que ocorre por meio da “notificação” sobre a mudança relacionada ao seu estado lógico (VALENÇA, 2012, p. 29; RONSZCKA, 2019, p. 29; BANASZEWSKI, 2009, p. 11).

Consequentemente, cada *Condition* notificada avalia o seu valor lógico de acordo com as notificações recebidas das *Premises* e com um dado operador lógico, sendo este em geral um operador de conjunção (*and*) ou disjunção (*or*). Assim, no caso de uma conjunção, por exemplo, quando todas as *Premises* que integram uma *Condition* são satisfeitas, a *Condition* também é satisfeita. Isto resulta na aprovação de sua respectiva *Rule* que pode “então” ser executada (*Action*) (VALENÇA, 2012, p. 29).

Ainda segundo Valença (2012, p. 29), com a *Rule* aprovada a sua respectiva *Action* é ativada. Uma *Action*, por sua vez, é conectada a um ou vários *Instigations*. As *Instigations* colaboram com as atividades das *Actions*, acionando a execução de algum serviço por meio dos seus *Methods*. Usualmente, os *Methods* alteram os estados dos *Attributes*, recomeçando assim o ciclo de notificações.

2.7 LINGUAGENS DE PROGRAMAÇÃO NO PON

Santos (2017, p. 21) relata que os conceitos do PON foram primeiramente materializados no Paradigma Orientado a Objetos, através de um *framework* desenvolvido na Linguagem de Programação C++ em 2007 pelo Prof. Jean Marcelo Simão e que teve posteriormente a versão 1.0 desenvolvida no trabalho de Banaszewski (2009). Em 2012, a nova versão nomeada de *Framework* PON C++ 2.0 foi implementada nos trabalhos de Valença (2012) e Ronszcka (2012).

No caso da categoria de linguagem de programação específica para o PON, nomeada de LingPON, a versão 1.0 com o respectivo compilador foi desenvolvida e apresentada no projeto de mestrado de Ferreira (2015). Posteriormente, o trabalho de Santos (2017) propôs a LingPON versão 1.2.

3 INFORMAÇÕES SOBRE O FUNCIONAMENTO DO FUTEBOL DE ROBÔS

Nesta segunda seção serão apresentadas informações de como uma partida de Futebol de Robôs é jogada. Tais informações são necessárias para que haja o entendimento de *Rules* implementadas.

3.1 REQUISITOS FUNCIONAIS

Aqui, estão apresentados requisitos funcionais baseados na regulamentação atual da competição de Futebol de Robôs (RoboCup), documento “Rules of the RoboCup Small Size League” (RULES, 2020), a fim de simplesmente garantir uma partida dentro desta regulamentação. São elas:

a) Quando em “fora de jogo”, a bola não pode ser manipulada, ou seja, um jogador não pode realizar *dribbling* ou *shooting*. *Dribbling* é o ato manipular a bola de maneira geral, podendo mover-se com ela.
b) Não é permitido executar *dribbling* por mais de 1m, ou seja, manter contato com a bola (segurá-la) por mais de um metro.
c) Após cobranças, o jogador que cobrará não pode encostar na bola após ela ter se movido 50mm (o que torna a bola em jogo) sem que outro jogador a tenha tocado após esse movimento. Essa infração é chamada de toque duplo e possui uma tolerância equivalente a movimentação para a bola estar em jogo, há uma tolerância de 50mm para o toque duplo. Ou seja, o jogador deve executar passe ou chute quando em cobranças.
d) Quando em *stop*, os robôs devem se movimentar abaixo de 1,5m/s, devendo manter-se a 500mm de distância da bola, não podendo manipulá-la, e devem se posicionar a 200mm da área de defesa do oponente. Há uma tolerância de 2 segundos.
e) Quando em *halt*, nenhum robô pode mover-se ou manipular a bola. Há uma tolerância de 2 segundos.
f) Em *direct kicks* pode ser realizado gol com o chute. Os jogadores defensores devem se posicionar a pelo menos 500mm da bola e os jogadores atacantes podem se posicionar próximos à bola.
g) O *indirect kick* possui regras semelhantes ao *direct kick*, mas não pode ser realizado gol com o chute (pelo menos um jogador atacante que não o *kicker*, após a cobrança, deve tocar na bola antes da validade de um gol).

h) Após o comando *force start*, ambos os times podem manipular a bola.
i) Após o comando de pênalti, o goleiro defensor deve tocar a linha do gol e um jogador atacante pode se aproximar da bola e será o cobrador. Todos os outros robôs devem se posicionar a uma distância mínima demarcada por uma linha a 400mm da marca de pênalti, paralela à linha do gol.
j) Após o comando *normal start*, deve ser avaliado também se o estado anterior era pênalti, para execução de chute do cobrador.
k) Após o comando *normal start*, deve ser avaliado também se o estado anterior era *kick-off*, para execução de chute ou passe do cobrador.
l) De mesmo modo, observa-se que apenas o goleiro pode entrar na sua área de defesa.
m) Um robô não pode executar "chute sem mira".
n) Um robô não pode chutar a bola de forma que esta ultrapasse a velocidade de 6,5m/s.
o) Um robô não deve ir de encontro direto a outro robô para evitar "colisão" e "empurrão".
p) Quando em chute a gol, a bola não pode ser elevada mais do que 150mm, ou o gol é declarado como inválido.
q) Quando a bola ultrapassa as linhas de campo, há cobranças como tiro de meta ou laterais.

3.2 REGRAS

A seguir estão apresentadas *Regras* (*Rules*), *Premissas* (*Premises*) e *Métodos* (*Methods*) necessários para que o funcionamento dos robôs siga de acordo com o estipulado pelos requisitos apresentados na seção "2.1 Requisitos Funcionais". São elas:

a) Premissa que avalia se o comando atual é *normal start*, se o anterior era pênalti, se o jogador é cobrador e método que realiza chute.
b) Limitar a velocidade de chute ou passe nos respectivos métodos.
c) Impedir avanço sobre robô adversário, especialmente se este está se deslocando na direção do robô em questão. Limitação em premissa ou método de movimentação.
d) Limitação no método de chute.
e) Não há razão para um robô posicionar-se além das linhas de campo.

3.3 NOTAS IMPORTANTES

Não é necessário haver um goleiro segundo o documento “Rules of the RoboCup Small Size League”, nas seções “3.1 Number Of Robots” e “4.3.5 Choosing Keeper Id” (RULES, 2020).

Pode-se segurar a bola, desde que não haja restrição a todos os seus graus de liberdade, pelo menos 80% da área da bola (RULES, 2020) deve ser visível quando em vista superior, não eleve a bola do chão e seja possível que outro robô adquira a posse de bola. Uma forma de implementação, portanto, pode-se ser exercer força limitada na bola em direção ao robô com posse.

4 DESENVOLVIMENTO DA APLICAÇÃO PARA CONTROLE DO FUTEBOL DE ROBÔS

Nesta quarta seção, será feita a apresentação da proposta de trabalho de conclusão de disciplina que envolve o desenvolvimento de *Rules* no software de controle dos robôs disponibilizado para estudo (Apêndice B) visando a simulação de uma partida de Futebol de Robôs em ambiente de simulação.

4.1 ATIVIDADE PROPOSTA

A atividade proposta durante a disciplina era de realizar a implementação de um novo conjunto de *Rules* com base no trabalho desenvolvido na dissertação do então mestrando Leonardo Araujo Santos (SANTOS, 2017), afim de melhorar o comportamento dos robôs.

4.2 DIFICULDADES ENCONTRADAS

Durante o desenvolvimento do trabalho foram encontradas diversas dificuldades, principalmente no que se refere ao ambiente proposto.

No uso da máquina virtual disponibilizada (Apêndice A), em computadores com processador AMD ocorriam algumas falhas de execução, e foi necessário o *download* do código fonte e a recompilação da aplicação “grSim”. Esse problema foi causado pelo fato das bibliotecas já virem pré-compiladas, e ao utilizar um processador diferente as mesmas eventualmente vem a apresentar falhas.

Além desse problema inicial também foi constatado que a versão de código disponibilizado neste ambiente não correspondia à versão final da dissertação, mas sim uma versão mais antiga.

Por conta disso inicialmente foi feita uma tentativa de atualizar o código (Apêndice B) com as novas *Rules*, utilizando um outro código disponibilizado da versão final de Santos (2017) (Apêndice C), porém isso se mostrou bastante complicado pois havia uma diferença muito grande entre a estrutura base da interface dos robôs sob a qual essas versões estavam implementadas, conforme a regulamentação de competição de Futebol de Robôs (RoboCup) descrito no capítulo 3, portanto ao tentar compilar e executar o código resultava em muitos erros de compilação, que quando resolvidos apenas geravam erros de execução (*core dump*). Esses erros podem ter sido causados devido a diferenças nas bibliotecas sob as quais os mesmos foram implementados.

Não obtendo sucesso foi observado que não seria muito produtivo prosseguir nessa tentativa de recompilar o código. Desta forma o que se seguiu foi uma tentativa de se adaptar as *Rules* propostas por Santos (2017) (Apêndice C), no código funcional, porém mais uma vez a tarefa se mostrou complicada, devido estrututa geral do código e falta de algumas partes, tornou-se impossivel a mesclagem dos códigos, funcional e de Santos (2017) (Apêndice C).

4.3 TRABALHO DESENVOLVIDO

Considerando as dificuldades apresentadas foi decidido finalmente que cada um dos alunos ficaria responsável pela implementação de 5 *Rules* próprias (Apêndices D e E), de modo a exercitar melhor os conhecimentos individuais sobre o PON e também obter como resultado um código funcional no qual seria possível observar a aplicação das *Rules* implementadas durante uma partida do Futebol de Robôs.

O Quadro 5 apresenta as *Rules* implementadas pelos integrantes do trabalho. Neste quadro é descrito o nome da variável adicionada ao código para novas *Rules*, uma descrição de funcionamento dela e seu responsável. As *Rules* têm papel de modificar o comportamento dos jogadores durante a partida.

Quadro 5 – Conjunto de *Rules* implementadas na aplicação para controle do Futebol de Robôs
(continua)

Rule	**Objetivo**	**Responsável**
rlPositionToShoot	Posicionar jogador para chutar ao gol quando estiver próximo. Verifica *Attributes* de estado do jogo e executa o *Method* que seta *flag* "atReadyToShoot" para *true*.	Felipe dos Santos Neves (Apêndice D)
rlShootToGoal	Chutar ao gol após estar posicionado. Verifica *Attributes* de controle de chute a gol e executa o *Method* que produz a ação de chute "mtShootToGoal".	Felipe dos Santos Neves (Apêndice D)
rlResetShootFlag	Resetar *flags* de controle após realizar chute. Verifica os mesmos *Attributes* de controle de chute e também posse de bola. O chute é considerado feito quando "atBallIsMine" é *false*. Então as *flags* de controle de chute são atribuídas como *false*.	Felipe dos Santos Neves (Apêndice D)
rlMaxDrible	Passar a bola após exceder determinado número de passes. O *Attribute* "atDribleCount" é incrementado a cada toque e "atMaxDribleCount" é randomizado entre 1 e 6 a cada execução da *Rule*. O *Method* força um passe e o *reset* de "atDribleCount".	Felipe dos Santos Neves (Apêndice D)

Quadro 5 – Conjunto de *Rules* implementadas na aplicação para controle do Futebol de Robôs
(continua)

Rules	**Objetivo**	**Responsável**
rlPassGkCloseGoal	Forçar goleiro a passar a bola ao ficar com a posse dela muito próximo ao gol. Verifica se a distância ao gol "atBallDistTeamGoal" é menor que 1800 e outros *Attributes* de controle de posse de bola. Força o passe a outro robô. Só é aplicada em robô do tipo defesa.	Felipe dos Santos Neves (Apêndice D)
rlPenalty	Posicionar o jogador atacante para a posição de pênalti do campo. Tratamento para movimentar o jogador atacante para a posição de pênalti. Posição calculada com base nas dimensões do campo. O estimulo desta *Rule* é realizado pelo painel *Referee Box* no botão *Penalty*	Anderson Eduardo de Lima (Apêndice D)
rlYellowCardBlue	Captura do evento de cartão amarelo do time azul. Realiza o tratamento do estimulo do painel *Referee Box* no botão *Yellow Card* para os jogadores do time azul.	Anderson Eduardo de Lima (Apêndice D)
rlRedCardBlue	Captura do evento de cartão vermelho do time azul. Realiza o tratamento do estimulo do painel *Referee Box* no botão *Red Card* para os jogadores do time azul.	Anderson Eduardo de Lima (Apêndice D)
rlLimiteHorizontal	Evitar a saída de robôs do limite horizontal do campo (regra da categoria *Small Size League* - SSL). Verifica se o robô encontra-se fora do limite horizontal, pré-estabelecido, do campo.	Anderson Eduardo de Lima (Apêndice D)
rlLimiteVertical	Evitar a saída de robôs do limite vertical do campo (regra do SSL). Verifica se o robô encontra-se fora do limite vertical, pré-estabelecido, do campo.	Anderson Eduardo de Lima (Apêndice D)
rlClosestToGoal	Determinar se o robô está próximo do gol. Se está próximo ao gol (coordenadas "x" e "y") então define o estado "atClosestToGoal" como *true*.	Omero Francisco Bertol (Apêndice D)
rlNotClosestToGoalX	Determinar se o robô "não" está próximo do gol em relação a coordenada "x". Se não está próximo ao gol na coordenada "x" então define o estado "atClosestToGoal" como *false*.	Omero Francisco Bertol (Apêndice D)
rlBallFar	Reposicionar o robô. Redefinir chute e drible quando a bola estiver longe. Baseada na "Rule 5" de Santos (2017, p. 179).	Omero Francisco Bertol (Apêndice D)
rlNotClosestToGoalY	Determinar se o robô "não" está próximo do gol em relação a coordenada "y". Se não está próximo ao gol em uma das coordenadas "y" então define o estado "atClosestToGoal" como *false*.	Omero Francisco Bertol (Apêndice D)

Quadro 5 – Conjunto de *Rules* implementadas na aplicação para controle do Futebol de Robôs (conclusão)

Rule	**Objetivo**	**Responsável**
rlClosestToGoalKick	Determinar se o robô deverá realizar o chute a gol. Se o jogador está pronto, está com a bola e está próximo ao gol então chuta para o gol.	Omero Francisco Bertol (Apêndice D)
rlRobotWantsToMoveX	Avalia necessidade de movimento em X.	Lucas Tachini Garcia (Apêndice D)
rlRobotWantsToMoveY	Avalia necessidade de movimento em Y.	Lucas Tachini Garcia (Apêndice D)
rlRobotWantsToMove	Confirma a necessidade de movimento em X ou Y.	Lucas Tachini Garcia (Apêndice D)
rlRobotNotRetrained	Esta *Rule* deve agrupar condições negadas que possam vir a restringir o movimento do jogador.	Lucas Tachini Garcia (Apêndice D)
rlRobotMoveGoalKeeper	Permite ou não que o movimento do goleiro seja executado.	Lucas Tachini Garcia (Apêndice D)
rlNextMoveToRestrictedArea	Avalia movimentação para áreas não permitidas.	Lucas Tachini Garcia (Apêndice D)
rlFixRobotNotGkmove	Modifica o movimento do jogador que não seja o goleiro.	Lucas Tachini Garcia (Apêndice D)
rlFixRobotGkmove	Modifica o movimento do goleiro.	Lucas Tachini Garcia (Apêndice D)
rlRedCardYellow	Aplica cartão vermelho para um jogador do time amarelo.	Lucas Tachini Garcia (Apêndice D)
rlBallOutsideEnemyTeam	Se a bola sai da área do jogo no campo adversário, força o jogador a voltar para sua posição inicial.	Lucas Tachini Garcia (Apêndice D)
rlPassWhenOff	Determinar se o robô deve passar a bola e retornar ao centro do campo. Jogador com a bola, avançado no campo adversário e fora de centro, passa a bola e retorna ao centro.	Luis Henrique Sant'Ana (Apêndice D)

Fonte: Autoria própria.

A finalidade das *Rules* criadas foi presenciar a modificação no comportamento dos jogadores durante simulação causada pela adição das instruções PON. Como muitas das funcionalidades descritas no projeto "RoboCup2012" desenvolvido em Santos (2017) estavam faltando algumas partes do código, desta forma impossibilitando a compilação, algumas destas novas funcionalidades não estão adequadas às regras vigentes para uma partida de Futebol de Robôs Oficial (RoboCup).

5 CONCLUSÃO

Com o objetivo de aplicar na prática os temas abordados nas disciplinas com enfoque no Paradigma Orientado a Notificações (PON) foi proposto a aplicação da tecnologia PON, configuração do ambiente de simulação (Apêndice A), análise de *Rules* em trabalhos existentes (Apêndices B e C) e o desenvolvimento de novas *Rules* do software para o simulador visando controle do Futebol de Robôs (Apêndices D e E).

Notadamente, foram levantadas informações sobre o regulamento da competição do Futebol de Robôs (RoboCup), como é realizada a condução de uma partida, *Rules*, *Premises* e *Methods* necessários para que o funcionamento de um jogador robô siga em acordo com os requisitos funcionais estabelecidos no regulamento.

Subsequentemente, realizou-se a apresentação do Paradigma Orientado a Notificações (PON) como uma alternativa para outros paradigmas de programação, entre eles, o Paradigma Imperativo e Paradigma Declarativo. O estudo da tecnologia PON mostrou seus elementos fundamentais materializados na forma de seus elementos essenciais na forma de *Atributo*s (*Attributes*), *Métodos* (*Methods*), *Condição* (*Condition*), *Premissas* (*Premises*), *Ação* (*Action*), *Instigações* (*Instigations*) e *Regras* (*Rules*). O fluxo de execução de aplicações no PON foi destacado mostrando o desacoplamento em expressões lógico-causais, inexistente em outros paradigmas, o que evita o processamento desnecessário com redundâncias causais e/ou utilização custosa de estruturas de dados. Importância também foi dada as linguagens de programação que materializam a tecnologia PON, são elas: a) *Framework* PON C++ 2.0, desenvolvido na Linguagem de Programação C++; e b) LingPON, linguagem específica para o PON.

Por fim, foi apresentado o desenvolvimento de novas *Rules*, com base no trabalho de Santos (2017), afim de melhorar o comportamento dos jogadores no aplicativo para controle de Futebol de Robôs, executado em ambiente de simulação configurado em máquina virtual. Tarefa essa que apresentou diversas dificuldades, principalmente, no que se refere ao ambiente proposto como por exemplo, na questão das bibliotecas já estarem pré-compiladas e se mostraram incompatíveis com processadores diferentes provocando falhas na sua utilização. A nível de códifos-fontes a versão disponibilizada para estudo não correspondia à versão final de Santos (2017), mas sim uma versão anterior. Considerando as dificuldades, foram então apresentadas as *Rules* que cada um dos alunos implementou aplicando os conhecimentos individuais adiquiridos sobre a tecnologia PON.

REFERÊNCIAS

BANASZEWSKI, Roni Fabio. **Paradigma Orientado a Notificações: Avanços e comparações**. 2009. 285 f. Dissertação de Mestrado – Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial (CPGEI), Universidade Tecnológica Federal do Paraná (UTFPR). Curitiba, 2009. Disponível em: <http://livros01.livrosgratis.com.br/cp087456.pdf>. Acesso em: 16 fev. 2020.

FERREIRA, Cleverson Avelino**. Linguagem e compilador para o paradigma orientado a notificações (PON)**: avanços e comparações. 2015. 227 f. Dissertação (Mestrado em Computação Aplicada) – Universidade Tecnológica Federal do Paraná, Curitiba, 2015. Disponível em: < http://repositorio.utfpr.edu.br/jspui/handle/1/1414>. Acesso em: 22 mai. 2020.

RONSZCKA, Adriano Francisco. **Contribuição para a concepção de aplicações no paradigma orientado a notificações (PON) sob o viés de padrões**. 2012. 224 f. Dissertação de Mestrado – Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI), Universidade Tecnológica Federal do Paraná (UTFPR). Disponível em: <http://repositorio.utfpr.edu.br/jspui/bitstream/1/327/1/CT_CPGEI_M_%20Ronszcka, %20Adriano%20Francisco_2012.pdf>. Acesso em: 23 mar. 2020.

RONSZCKA, Adriano Francisco. **Método para a criação de linguagens de programação e compiladores para o Paradigma Orientado a Notificações em plataformas distintas**. 2019. 375 f. Tese de Doutorado – Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial (CPGEI), Universidade Tecnológica Federal do Paraná (UTFPR). Curitiba, 2019. Disponível em: <http://repositorio.utfpr.edu.br/jspui/handle/1/4234>. Acesso em: 25 fev. 2020.

RULES of the RoboCup Small Size League 2019. **RoboCup Federation**, 19 mar. 2020. Disponível em: <https://robocup-ssl.github.io/ssl-rules/sslrules.html>. Acesso em: 08 fev. 2020.

SANTOS, Leonardo Araujo. **Linguagem e compilador para o paradigma orientado a notificações: Avanços para facilitar a codificação e sua validação em uma aplicação de controle de Futebol de Robôs**. 2017. 293 f. Dissertação de Mestrado – Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial (CPGEI), Universidade Tecnológica Federal do Paraná (UTFPR). Curitiba, 2017. Disponível em: <http://repositorio.utfpr.edu.br/jspui/handle/1/2778>. Acesso em: 16 fev. 2020.

VALENÇA, Glauber Zárate. **Contribuição para a materialização do Paradigma Orientado a Notificações (PON) via framework e wizard**. 2012. 205 f. Dissertação (Mestrado em Computação Aplicada) – Programa de Pós-graduação em Computação Aplicada (PPGCA), Universidade Tecnológica Federal do Paraná (UTFPR). Curitiba, 2012. Disponível em: <http://repositorio.utfpr.edu.br:8080/jspui/handle/1/393>. Acesso em: 16 fev. 2020.

APÊNDICE B

Comparisons on Game Development with Unreal Engine 4 using Object-Oriented Paradigm (OOP in C++) versus Notification Oriented Paradigm

Felipe dos Santos Neves
Graduate student at *PPGCA*[1] - *DAINF*[2] - *UTFPR*[3]
Report in article form to the course Programming Paradigms
(1st trimester of 2020) at *CPGEI*[4] - *UTFPR*[3] – Prof. Jean Marcelo Simão.
Curitiba, Paraná, Brazil
fneves@alunos.utfpr.edu.br

***Abstract*—This paper draws comparisons on the usage of the novel Notification Oriented Paradigm for game development with the traditional approach using Object-Oriented C++. For this purpose, a simple game is developed using the Unreal Engine 4, implementing the same game logic with both the Object-Oriented Paradigm and Notification Oriented Paradigm. A formal software development cycle, from requirements to design and implementation is used to ensure that both implementations have the same final result, enabling the comparisons of the time spent in development and number of lines of code necessary for each implementation. The software developed using the Notification Oriented Paradigm generated more lines of code, however less time was spent during development, showing that in this case using the Notification Oriented Paradigm was beneficial to reduce the software development time.**

***Index Terms*—Game Development Methodologies, Unreal Engine 4, NOP Versus OOP Experimentation**

I. Introduction

The development of electronic video games dates back to the early 1950s, where computers first began to be available to civilians, and the first graphical video game appeared in 1958, with the vastly know game called Pong, which was entirely hardware based [1]. Since then game Development has grown to a multibillion industry, reaching $120.1 billion revenue in 2019 [2].

Game Development is a widely multidisciplinary field, involving field such as arts, design, business and software development. And as digital games grow in size and complexity each year the industry pushes forward the fields of both hardware and software development.

In the early days of computer assisted game development most games had software build from the ground up specifically for that purpose, but in the core of any modern game is a Game Engine, which is the core software responsible for handling the doing the rendering, calculating physics, lighting, playing sounds and much more. These engines provide the tools for the developers to bring their design to work without with much more ease than having to rebuild everything for each game. Of course, some cutting edge development still requires the development of a new engine from time to time to better fit the requirements of the game that is being developed [3].

Given the nature of such applications, as computer games can be roughly described as a group of objects that interact with each other and that have to be rendered on a screen on a fixed interval, both the games and the engines themselves are almost always implemented in the Object-Oriented paradigm. The engine used in this study is the Unreal Engine version 4, which has its API available in C++. This is one of the most used tools in professional game development, and shows how important the Object-Oriented paradigm is in the game industry.

Given this context this paper's goal is to show how the novel Notification Oriented paradigm can be used as a great asset to extend toolset developers have, where the Object-Oriented paradigm shows its limitations. In its current implementation there is a version of the Notification Oriented Paradigm available as a C++ library [4], which makes it suitable and easy to interface with the Unreal Engine API, providing a convenient way to study the viability of the application of the Notification Oriented Paradigm for game development.

Given the objective of this paper to compare the two approaches to the development of a game we establish a set of requirements that should be present in both implementations. Also, after developing for the initial requirements there will be a second set of modified requirements, where some of the requirements are modified and new ones are added, for the purpose of evaluating how easy it would be to modify the software after it is already done for the initial requirements.

II. Unreal Engine

The Unreal Engine was chosen for this project due to it being one of the most used engines in commercial game Development, as illustrated in Figure 1, from data gathered from Steam [5], the major games storefront for PC. Besides

[1]Programa de Pós-Graduação em Computação Aplicada
[2]Departamento Acadêmico de Informática
[3]Universidade Tecnológica Federal do Paraná
[4]Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial

that, it also has its API in C++, making it easy to interface with the NOP C++ Framework 2.0 [4], which could be compiled as a library and loaded in the Unreal Engine project.

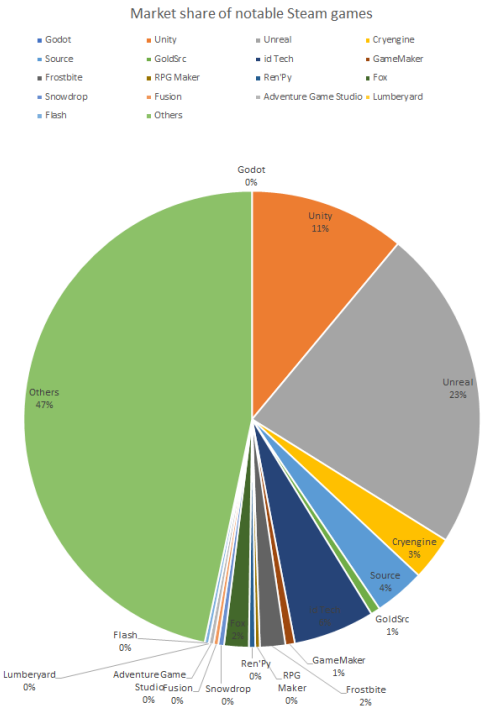


Fig. 1: Game engines market share from Steam [5]

While most of the game's implementations can be done in C++, the Unreal Engine also has its own visual programming language called *Blueprints*, that allows the users to use the API through visual blocks and connections. This is an excellent way to speed up the development, as it is very easy and intuitive, and also enables users without programming language to use the engine. There is extensive documentation provided for both the C++ and *Blueprints* and the Unreal Engine website [6], in this paper only the core concepts required in order to provide a basic understanding of the implementations of this project.

While being easier to use the *Blueprints* has a very robust interface to call methods from the native C++ API. The only downside is that it may be slower, and using the native API can result in a faster implementation, depending on how the code is written, and also allows for a more advanced use. For the most part for simpler games they can be made entirely using *Blueprints*, or used in complement with core implementations in C++, which the engine provides the tools to export to be used with *Blueprints*. This means that you can expand write your own classes in C++ and use the available tools from the engine, through some clever use of macros, and convert it to a pure *Blueprints* object.

A. Core Concepts

One of the core concepts of the Unreal Engine is that every object is derived from the base class UObject, which contains the base structure for every object that you want to be handled by the engine. This is because the engine expects to handle the two core loops for every object BeginPlay and Tick. Also, every object that is created is inserted into a list of all objects in the game world, so that the engine can handle all of them. The engine handles both graphical and non-graphical objects as well.

The BeginPlay function relates to a usual C++ class constructor. The BeginPlay function is called every time a instance of the class is invoked into the game world. It is very similar to what a constructor function would do in a normal C++ program, but the difference in this case is that the engine uses the constructor to pre-load assets, such as graphics and sounds, into memory. This is done to provide better memory efficiency and speed, as you cannot have heavy objects being loaded into memory during gameplay, so the engine pre-loads all the classes before the level starts, that would be what happens during a loading phase. And then there is the BeginPlay function that is only called when the object is effectively spawned into the world.

Another very important function is the Tick function. This function is actually optional, as it can be very resource intensive, as it is called for every tick of the engine, or for every frame that is drawn on the render. Objects that do not need to run every tick can disable this feature, but it is usually enabled for most objects that have any sort of interaction.

As it becomes evident from these concepts, for graphical objects in general, the code must run loops that update on every frame, what could contradict the principles of the Notification Oriented Paradigm, that would avoid those loops altogether. But as the game engine abstracts most of these graphical elements and the Notification Oriented Paradigm can be used, not to substitute the traditional Object Oriented approach, but to complement and improve the design of the other non-graphical related aspects of the game code, such as the enemy behavior and game systems and rules.

III. Notification Oriented Paradigm

A. Objectives

The main objectives of the Notification Oriented Paradigm are to solve some of the many problems present in the Imperative and Declarative Paradigms [7] [8]. The first problem that can be mentioned is the redundancy present on the Imperative Paradigm, since it is strongly dependent on loops (such as if, while and for loops) for evaluating variable states during the program execution, but most of these variables will never change during each interaction, resulting in a code that will run multiple times without achieving any different results [7].

Besides that one of the other main problems is distribution, since the same variables can be evaluated in multiple different

sections of the code, and the passivity of its elements bring a close dependency to the data and commands within a program, making it harder to split into multiple execution units. In scenarios where you cannot control the execution order of the parts of your code it can cause many problems [8].

As such is the case of Unreal Engine objects, where the engine is responsible for choosing which elements to evaluate first, the developer has no input on that matter, and that can bring problems where there is a close dependency on variables that may change their states before a dependent state has its chance to be evaluated.

B. NOP Entities

This section presents in greater detail the structures that compose the Notification Oriented Paradigm implementation.

1) Fact Base Element: The Figure 2 describes the mains entity of the Notification Oriented Paradigm, the Fact Base Element (FBE). It is a class used to describe the states and services of our objects. An FBE contains attributes and methods, which are collaborative objects. The difference between an FBE and a traditional Object Oriented class is that the attributes and methods are not simple passive entities [8].

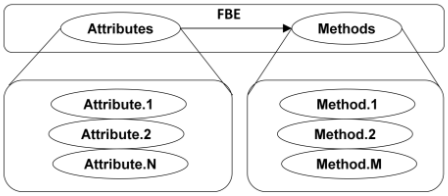


Fig. 2: FBE Structure
[8]

2) Attribute: The attributes store the values that represents the current states of the properties of given FBE. It is different from a traditional member variable from the point that it is able to notify its relevant premises when its state changes. The ability to notify the premises, that can be members of the same object or be part of a different entity is what gives the advantages in respect to parallel execution and decoupling.

3) Premises: The premises represent the evaluation of the state of a given attribute. When the attribute notifies the premises, it is then evaluated against a given value or against another attribute. The premises can be evaluated as true or false, and when its value changes it notifies the relevant conditions.

4) Condition: The condition is composed from premises, that are evaluated with either a conjunction or disjunction operation. It notifies the relevant rules when it is approved.

5) Rule: A rule, as illustrated in Figure 3, is composed by a condition and an action, in a causal manner, when the rule's condition becomes true it then the rule becomes approved for execution, executing its action. Indirectly, the rule is composed by premises, and instigations.

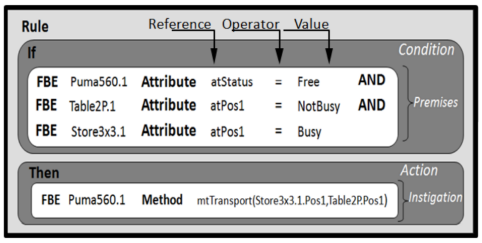


Fig. 3: Rule Structure
[8]

6) Action: The action may contain multiple instigations. Its activated when the rule is approved. It is responsible for calling the connected instigations.

7) Instigation: The responsibility of the instigation is to call the methods of the objects, passing their appropriate parameters, when required. These methods can either be NOP attribute changes or also regular C++ function calls. By allowing the call of the regular code functions it gives the flexibility to integrate with other frameworks such as the Unreal Engine API.

The chain of events from the attribute changes to the method execution as described in the Section III-B is also illustrated in the Figure 4.

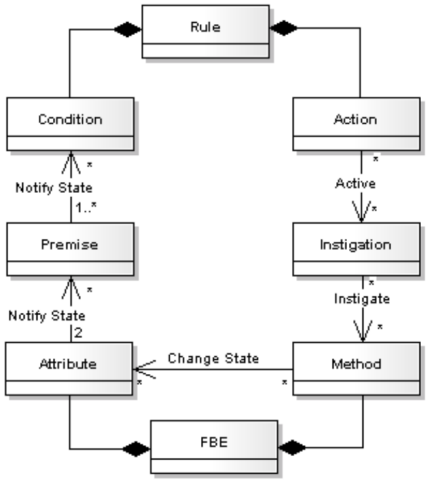


Fig. 4: NOP Notification Mechanism Diagram
[8]

IV. Programming Concepts

As defined in by Peter Van Roy [9], programming paradigms are composed of smaller conceptual units, called programming

concepts. He defines 4 concepts that he considers the most important.

1) Record: A data structure, as a group of references that can be accessed through indexes to each item. Is ins present in the PON C++ framework as inherited from C++. It can be seen in the FBE structure, which is composed of other structures such as attributes and methods.

2) Lexically Scoped Closures: In the PON C++ framework, as all the entities described in Section III-B are implemented as classes from C++ we inherit this concept.

3) Independence (Concurrency): A program can be constructed as independent parts. When two parts do not interact, they are concurrent, as opposed to when the order of execution is defined and they are sequential. All the PON entities are by definition independent and support parallelism, as their behavior is determined by the notification mechanism, which supports concurrency. In this project it is very clear to observe this behavior as all the objects constructed in the Unreal Engine act as separate and independent threads therefore are executed concurrently.

4) Named State: Is the capacity of storing information in a computational element with a name, so that it can be accessed and modified through the code. It is very important in the PON framework so that attributes, premises, conditions and rules can be defined, and the instances of these elements can be shared through references to their names.

These concepts where later analyzed and classified in relation to the Notification Oriented Paradigm [10]. And these concepts are then validated and discussed bellow derived from the observations applying the Notification Oriented Paradigm through the C++ Framework in this project in Table I.

TABLE I: Programming Concepts in the NOP

Concept	Present
Record	Yes
Lexically Scoped Closures	Yes
Independence (Concurrency)	Yes
Named State	Yes

V. Implementation

Regarding the design of the game itself the was to reduce the complexity related to the graphics so that it would be possible to focus more on the software development itself. For that purpose the game was designed as a top-down shooter, which is composed of a player and enemies that are able to move in a two-dimensional environment. This is a appropriate approach for this study as it gives two degrees of freedom for the movement of the entities while still remaining fairly easy to implement when compared to full three dimensional environment. The graphics themselves are also implement just as necessary to provide a minimum viable demo, and to give suitable player feedback so that the game can be enjoyed.

As for the NOP implementation specifics, it is still constrained to the Object-Oriented approach that comes from the engine for the basic objects structure, but these objects are now also Fact-Based Entities. To comply with the NOP concepts, the basic iteration loops of the Unreal Engine are still used, but variable evaluation and function calls during these loops are avoided, begin only used for the attribution of updated values to the FBE's attributes.

In Figure 5 is shown a screenshot of the implemented game, with the player in the center, surrounded by enemies with behavior following the game logic implemented using the NOP.

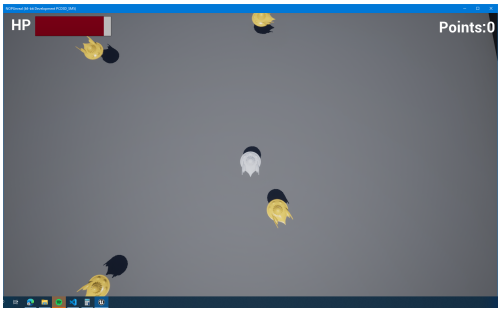


Fig. 5: In-Game screenshot

A. Requirements Specification

Given the fact that the two implementations have the same objective it was clear that writing a good set of requirements to guide both of them. Having a strict set of Functional and Non-Functional Requirements allows us to ensure that both implementations would have the same final functionality. These full set of requirements is described in Table II and III.

The initial set of requirements in Table III was designed to achieve a minimum viable product, a simple but functional game. We do not enter in much detail about the graphical aspects of the project, such as models, textures, sounds and effects as they are outside of the scope of the study.

As one of our objectives was to evaluate how both implementations could be modified a second set of modified requirements was defined with the objective to improve upon the original game design and is presented in Table IV. With these modifications it is possible to evaluate how easy is to modify an already finished code with the different approaches.

One of the benefits of developing with NOP based on functional requirements is how easy it is to relate the requirements with the code, due to how the program is structured, it more clearly reproduces the rules of a requirement into the code, making it easier to understand and to develop. This relationship between rules and requirements is extremely beneficial to the system, as it facilitates and encourages better traceability of the requirements during the whole product life cycle [11].

B. Class Diagram

The classes were designed to respect a similar hierarchy in both implementation, as well as reutilizing elements that

TABLE II: Non-Functional Requirements

Requirement	Description
NFR-1	The game shall be implemented using C++, the NOP C+++ Framework 2.0 and blueprints with Unreal Engine 4
NFR-2	All elements shall be implemented using both only C++ and C++ with the NOP Framework while keeping the same functionality
NFR-3	The game shall be be composed of a single level that can be played with both implementations

did not change in both implementations, such the classes for the abilities, derived from UAbility. The Object-Oriented only design is presented in Figure 6 and the NOP design in Figure 7. The only difference in both class diagrams is that in the NOP design the Player and Enemy classes derive from both AActor class and the FBE class and the names are different to separate both implementations, as they run simultaneously in the same application. The similar structure also allows to reutilize some of the code used to implement basic functionality, such as controlling the player, moving the objects and using weapons, only requiring minor modifications between each implementation and thus keeping both codes as similar as possible, and this is important to be able to evaluate the differences in the amount of lines of code as well.

C. Rules Diagram

For the NOP implementation there are a total of 8 classes that are derived from the FBE class, meaning that they have a set of rules in their implementation, and it is present below the textual description of all the rules and their conditions. These rules were designed in order to achieve the requirements presented in the section V-A. The rules are described in tables V to XXI.

One of the advantages of a software developed using NOP is that it can become very easy to trace the requirements into the code. As, ideally, all of the logic must be contained within the rules are linked directly to a requirement and the code where it is implemented, via the rules, premises, attributes and methods involved. In Table XXV it is shown the link between the requirements and the rules used for their implementation. Some requirements may not link to any rule as they were not implemented using rules, an therefore these requirements become harder to trace in the code. It can be observed that the requirements that describe the enemy and generic game behavior are the ones that have a rule, because these are the features that were more appropriate to be implemented with rules.

TABLE III: Initial Functional Requirements

Requirement	Description
FR-1	The player shall be able to choose from the C++ and NOP versions in the menu
FR-2	The level shall be composed of a player versus enemies inside a closed arena
FR-3	All the objects shall be able to move in two dimensions
FR-4	The player shall win when all the enemies are dead
FR-5	The player shall lose when it dies
FR-6	The player shall be able to move with WASD keys
FR-7	The player shall be able to shoot with directional keys
FR-8	The enemy type 1 shall follow the player while its HP is below half its maximum value
FR-9	The enemy type 1 shall flee from the player while its HP is at or bellow half its maximum value
FR-10	The enemy type 1 shall fire a projectile always when it has a clear line of shot towards the player and it is not on cooldown
FR-11	The enemy type 1 shall fire a missile always when it has a clear line of shot towards the player and it is not on cooldown
FR-12	The enemy type 2 shall flee from the player while its HP is at or bellow half its maximum value
FR-13	The enemy type 2 shall follow the player while its over 500 units away from the player
FR-14	The enemy type 2 shall follow a straight line its at or under 500 units away from the player
FR-15	The enemy type 2 shall place a bomb when the player has a clear path towards its current position and it is not on cooldown
FR-16	The enemy type 3 shall always follow the player
FR-17	The enemy type 3 shall explode when it collides
FR-18	The enemy type 3 shall cause 2 damage in a 200 units radius when exploding
FR-19	An entity shall not cause damage to itself
FR-20	The projectile ability shall have a cooldown of 2 seconds to the enemies
FR-21	The projectile ability shall have a cooldown of 1 second to the player
FR-22	The missile ability shall have a cooldown of 5 seconds to the enemies
FR-23	The bomb ability shall have a cooldown of 5 seconds to the enemies
FR-24	A projectile shall cause 1 damage to the object it collides
FR-25	A missile shall cause 2 damage to the object it collides
FR-26	A bomb shall detonate 3 seconds after being created
FR-27	A enemy type 3 shall cause 2 damage in a 200 units radius when exploding
FR-28	The player shall move at a speed of 500 units per second
FR-29	The enemy type 1 shall move at a speed of 300 units per second
FR-30	The enemy type 2 shall move at a speed of 250 units per second
FR-31	The enemy type 3 shall move at a speed of 400 units per second
FR-32	The objects shall not be able to move over each other
FR-33	The projectile shall follow a straight line with a speed of 1000 units per second
FR-34	The missile shall follow the player with a speed of 1000 units per second
FR-35	The level shall spawn a enemy of each type in 3 corners of the arena
FR-36	The player shall be able to pause the game
FR-37	The player shall be able to continue the game from the pause menu
FR-38	The enemy type 1 shall have 2 HP
FR-39	The enemy type 2 shall have 5 HP
FR-40	The enemy type 3 shall have 1 HP
FR-41	The player shall have 20 HP

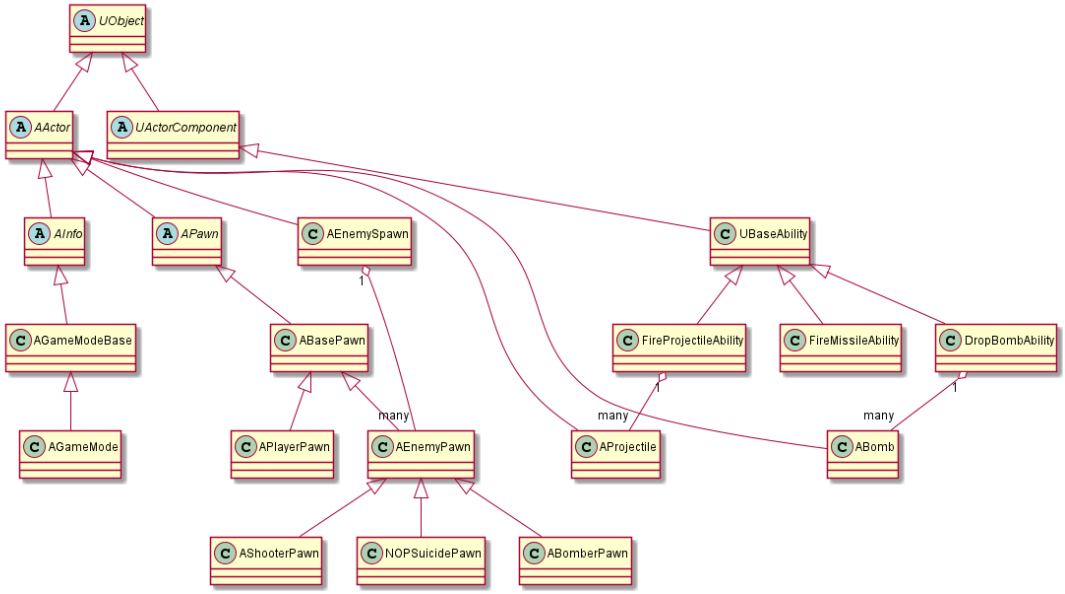


Fig. 6: OO Class Diagram

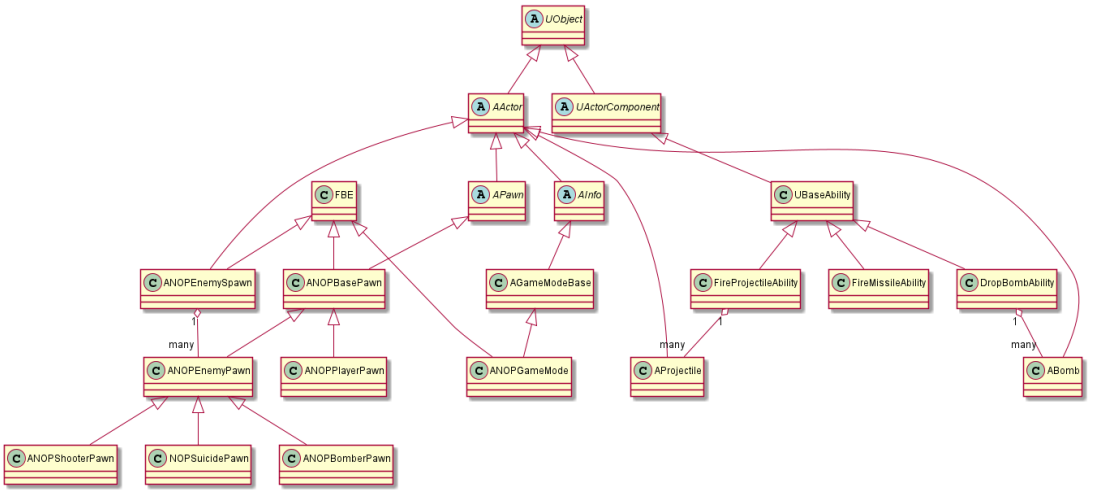


Fig. 7: NOP Class Diagram

TABLE IV: Modified Functional Requirements

Requirement	Description
FR-8	The enemy type 1 shall follow the player while it is over half of its maximum HP and it is the closest type 1 enemy to the player or it is over a distance of 800 units
FR-35	The level shall spawn enemies dynamically
FR-42	The player shall win when reaching 20 points
FR-43	The player shall be awarded 2 points when destroying a enemy type 1
FR-44	The player shall be awarded 3 points when destroying a enemy type 2
FR-45	The player shall be awarded 1 point when destroying a enemy type 3
FR-46	The player shall not be awarded any points when enemies destroy each other
FR-47	The level shall only spawn new enemies on the corners of the arena
FR-48	The level shall spawn a new enemy type 1 every 10 seconds until a maximum of 5 simultaneous enemies type 1 exist
FR-49	The level shall spawn a enemy type 3 for each two enemies of other types destroyed
FR-50	The level shall spawn a enemy type 2 every 5 seconds, while there are less than 5 enemies of any type in the arena
FR-51	The level shall initially spawn a enemy type 1 in each of the corners of the arena
FR-52	The enemy type 1 shall circle around the player with a 800 units radius while its HP is over half of its maximum HP and it is not the closest type 1 enemy to the player

TABLE V: Rule 1

RL-1	
Class	NOPBasePawn
Type	Conjunction
Description	Handle pawn' death
Premises	
Pawn's HP is at or below 0	

TABLE VI: Rule 2

RL-2	
Class	NOPBasePawn
Type	Conjunction
Description	Gives rewards on pawn's death
Premises	
Pawn's HP is equal ow less than 0	

TABLE VII: Rule 3

RL-3	
Class	NOPBomberPawn
Type	Conjunction
Description	Use the drop bomb ability
Premises	
Ability cooldown is ready	
Player has free path to current position	

TABLE VIII: Rule 4

RL-4	
Class	NOPBomberPawn
Type	Conjunction
Description	Set strategy to flee
Premises	
Pawn's HP is equal ow less than the maximum	

TABLE IX: Rule 5

RL-5	
Class	NOPBomberPawn
Type	Conjunction
Description	Set strategy to follow
Premises	
Pawn's HP is greater than half the maximum	
Distance to player is greater than 500 units	

TABLE X: Rule 6

RL-6	
Class	NOPBomberPawn
Type	Conjunction
Description	Set strategy to roam
Premises	
Pawn's HP is less or equal half the maximum	
Distance to player is less or equal 500 units	

TABLE XI: Rule 7

RL-6	
Class	NOPEnemySpawn
Type	Conjunction
Description	Spawn an enemy type 1
Premises	
Spawn cooldown is ready	
Number of simultaneous enemies of type 1 is less than 5	

TABLE XII: Rule 8

RL-7	
Class	NOPEnemySpawn
Type	Conjunction
Description	Spawn an enemy type 2
Premises	
Spawn cooldown is ready	
Number of simultaneous total enemies is less than 5	

TABLE XIII: Rule 9

RL-8	
Class	NOPEnemySpawn
Type	Conjunction
Description	Spawn an enemy type 3
Premises	
Spawn cooldown is ready	
Number of simultaneous total enemies is less than 5	

TABLE XIV: Rule 10

RL-9	
Class	NOPShooterPawn
Type	Conjunction
Description	Fire a projectile
Premises	
Has clear line of shot to player	
Projectile ability cooldown is ready	

TABLE XV: Rule 11

RL-10	
Class	NOPShooterPawn
Type	Conjunction
Description	Fire a missile
Premises	
Has clear line of shot to player	
Missile ability cooldown is ready	

TABLE XVI: Rule 12

RL-11	
Class	NOPShooterPawn
Type	Conjunction
Description	Set strategy to flee
Premises	
Pawn's HP is less or equal half the maximum	

TABLE XVII: Rule 13

RL-12	
Class	NOPShooterPawn
Type	Disjunction
Description	Set strategy to follow
Premises	
Sub condition 1	Type: Conjunction
Pawn's HP is greater than half the maximum	
Pawn is the closest of its type to the player	
Sub condition 2	Type: Conjunction
Pawn's HP is greater than half the maximum	
Pawn over 800 units away from the player	

TABLE XVIII: Rule 14

RL-13	
Class	NOPShooterPawn
Type	Conjunction
Description	Set strategy to roam
Premises	
Pawn is not the closest of its type to the player	
Pawn's HP is greater than half the maximum	
Pawn under 800 units away from the player	

TABLE XIX: Rule 15

RL-14	
Class	NOPUnrealGameMode
Type	Conjunction
Description	Player wins the game
Premises	
Game has started	
Player has 20 or more points	
Player is alive	

TABLE XX: Rule 16

RL-15	
Class	NOPUnrealGameMode
Type	Conjunction
Description	Player loses the game
Premises	
Game has started	
Player is dead	

TABLE XXI: Rule 17

RL-16	
Class	NOPUnrealGameMode
Type	Conjunction
Description	Set game as started
Premises	
Game has not started	
Game startup is finished	

VI. Evaluations and Comparisons

For an objective perspective in comparing the amount of effort required to develop the software it was chosen to compare both implementations by evaluating the difference in lines of code and time required to develop the software using both approaches, as they can be useful as indicators of how much effort was required during the development. Given that the developer was not experienced with neither the Unreal Engine API nor the NOP C++ Framework it was chosen to only evaluate their time used in the development of the second set of requirements, since most of the time spent during the development of the first version was actually spent learning how to use the tools as well as writing basic functions used for both versions and therefore would not reflect the real difference in the amount of effort required.

For the time evaluation each requirement, either new or modified, is linked to the amount of time spent in the Object-Oriented and Notification Oriented approaches, detailed in Table XXII. For this purpose it is only considered the time effectively spent writing code.

TABLE XXII: Time spent for development

Requirement	State	NOP	OOP
FR-35	Modified	0 min	0 min
FR-42	New	5 min	20 min
FR-43	New	15 min	5 min
FR-44	New	5 min	5 min
FR-45	New	5 min	5 min
FR-46	New	0 min	0 min
FR-47	New	5 min	5 min
FR-48	New	10 min	30 min
FR-49	New	10 min	30 min
FR-50	New	10 min	30 min
FR-51	New	10 min	10 min
FR-8	Modified	30 min	90 min
FR-9	Modified	30 min	30 min
Total		135 min	220 min

In the same manner the total lines of code are compared using the *cloc* tool [12]. This tool can count the total lines of code while eliminating comments and blank lines. Only the source code used for each implementation is considered for this purpose, and code that is common for both implementations is accounted for in each one of them. The results are shown in Table XXIII. The number of files is the same for both implementations.

TABLE XXIII: Lines of code written

Requirement Version	OOP	NOP
Initial	1272 lines	1407 lines
Final	1583 lines	1776 lines

Another analysis was to count the number of words in the same manner, this time using the command *wc* from *Linux*. The results are shown in Table XXIV.

TABLE XXIV: Words written

Requirement Version	OOP	NOP
Initial	54800 words	59315 words
Final	67682 words	75329 words

VII. Conclusion and Future Works

The results presented clearly show that working with NOP can bring great improvements to the process of game development, and other types of software as well, whether by reducing the development time due to a more comprehensive code structure, specially when writing code for behavior that can be translated well into rules, which is the case in many aspects of game development, since a lot of the effort is dedicated to writing the game logic and non-playable characters behavior.

As shown in the results from Table XXII, the time spent modifying the NOP code was 38.62% when compared with the OOP only code. And another major improvement is that all the NOP rules can be directly traced into specific requirements, meaning that any changes in the project can be more easily translated into code, as it reduces the risk of having the logic spread among to many places in the code, which makes it harder to do any changes without breaking other behaviors or leaving deprecated code in some places, which may easily lead to bugs and inconsistency in the software behavior.

There are however some clear drawbacks, that come from working with a novel technology. The framework itself can be very verbose, as it is shown in the increased amount of code written necessary for the same logic, as show in Table XXIII and Table XXIV, as the NOP code has 12.16% more lines and 11.13% more words, however this is small problem, since this comes from having to copy the initialization and declaration from the NOP structure, and doing so still took less time than the OOP approach.

Further development of the NOP technology and the C++ framework itself can help yield even better results, reducing the total time and amount of code required for software development. The development of more projects with the NOP framework can also be useful to ground the statement that the use of the technology can help reduce the effort required in software development.

References

[1] J. Gold, *Object-oriented game development*. Pearson Education Limited, 2004.

[2] D. Takahashi, "Superdata: Games hit $120.1 billion in 2019, with fortnite topping 1.8$ billion," Dec 2019. [Online]. Available: https://venturebeat.com/2020/01/02/superdata-games-hit-120-1-billion-in-2019-with-fortnite-topping-1-8-billion/

[3] J. Gregory, *Game engine architecture*. CRC Press, 2019.

[4] A. F. Ronszcka, "Contribuição para a concepção de aplicações no paradigma orientado a notificações (pon) sob o viés de padrões," Master's thesis, UTFPR, 2012.

[5] L. D. Wen, "Steam-engines," https://github.com/limdingwen/Steam-Engines, 2018.

[6] E. Games. (2020) Unreal engine 4 documentation. [Online]. Available: https://docs.unrealengine.com/en-US/index.html

TABLE XXV: Correlation Matrix

Requirement	RL-1	RL-2	RL-3	RL-4	RL-5	RL-6	RL-7	RL-8	RL-9	RL-10	RL-11	RL-12	RL-13	RL-14	RL-15	RL-16	RL-17
FR-1																	
FR-2																	
FR-3																	
FR-4																	
FR-5	x															x	
FR-6																	
FR-7																	
FR-8													x				
FR-9												x					
FR-10										x							
FR-11											x						
FR-12				x													
FR-13					x												
FR-14						x											
FR-15			x														
FR-16																	
FR-17																	
FR-18																	
FR-19																	
FR-20										x							
FR-21																	
FR-22											x						
FR-23			x														
FR-24																	
FR-25																	
FR-26																	
FR-27																	
FR-28																	
FR-29																	
FR-30																	
FR-31																	
FR-32																	
FR-33																	
FR-34																	
FR-35																	
FR-36																	
FR-37																	
FR-38																	
FR-39																	
FR-40																	
FR-41																	
FR-42															x		x
FR-43		x															
FR-44		x															
FR-45		x															
FR-46		x															
FR-47																	
FR-48							x										
FR-49									x								
FR-50								x									
FR-51																	x
FR-52														x			

APÊNDICE D

Conjunto de testes do *Framework* PON C++ 4.0

```
#include <atomic>
#include <queue>
#include <random>
#include <thread>

#include "gtest/gtest.h"
#include "libnop/framework.h"

TEST(Attribute, Int)
{
   const int val{123456};
   NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(val);
   EXPECT_EQ(at1->GetValue(), val);
}

TEST(Attribute, Bool)
{
   const bool val{true};
   NOP::SharedAttribute<bool> at1 = NOP::BuildAttribute(val);
   EXPECT_EQ(at1->GetValue(), val);
}

TEST(Attribute, Class)
{
   class Rectangle
   {
      int width, height;

     public:
      Rectangle(int w, int h) : width{w}, height{h} {};
      inline bool operator==(Rectangle i) const
      {
         return (width == i.width) && (height == i.height);
      }
      inline bool operator!=(Rectangle i) const
      {
         return (width != i.width) || (height != i.height);
      }
   };

   const Rectangle val1{1, 2};
   NOP::SharedAttribute<Rectangle> at1 = NOP::BuildAttribute(val1);
   EXPECT_EQ(at1->GetValue(), val1);

   const Rectangle val2{3, 4};
   NOP::SharedAttribute<Rectangle> at2 = NOP::BuildAttribute(val2);
   EXPECT_NE(at1->GetValue(), at2->GetValue());

   at2->SetValue(val1);
   EXPECT_EQ(at1->GetValue(), at2->GetValue());
}

TEST(Attribute, String)
{
   const std::string val{"test"};
   NOP::SharedAttribute<std::string> at1 = NOP::BuildAttribute(val);
   EXPECT_EQ(at1->GetValue(), val);
}

TEST(Premise, Simple)
{
   NOP::SharedAttribute<int> at1 = NOP::BuildAttribute<int>(-1);
   NOP::SharedAttribute<int> at2 = NOP::BuildAttribute<int>(-2);

   NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());

   // On initialization should not be approved
   EXPECT_FALSE(pr1->Approved());
   at1->SetValue(1);
   EXPECT_FALSE(pr1->Approved());
```

```
   at2->SetValue(1);
   EXPECT_TRUE(pr1->Approved());
}

TEST(Premise, Custom)
{
   NOP::SharedAttribute<int> at1 = NOP::BuildAttribute<int>(1);
   NOP::SharedAttribute<int> at2 = NOP::BuildAttribute<int>(1);
   NOP::SharedPremise prDouble = NOP::BuildPremise(
      at1, at2, [](auto a1, auto a2) { return a1 == 2 * a2; });

   // On initialization should not be approved
   EXPECT_FALSE(prDouble->Approved());
   at1->SetValue(2 * at2->GetValue());
   EXPECT_TRUE(prDouble->Approved());
   at1->SetValue(0);
   EXPECT_FALSE(prDouble->Approved());
}

TEST(Condition, Conjunction)
{
   NOP::SharedAttribute<int> at1 = NOP::BuildAttribute<int>(-1);
   NOP::SharedAttribute<int> at2 = NOP::BuildAttribute<int>(-2);
   NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());

   NOP::SharedAttribute<int> at3 = NOP::BuildAttribute<int>(-3);
   NOP::SharedAttribute<int> at4 = NOP::BuildAttribute<int>(-4);
   NOP::SharedPremise pr2 = NOP::BuildPremise(at3, at4, NOP::Equal());

   NOP::SharedCondition cn1 =
      NOP::BuildCondition(CONDITION(*pr1 && *pr2), pr1, pr2);

   // On initialization should not be approved
   EXPECT_FALSE(cn1->Approved());
   at1->SetValue(1);
   at2->SetValue(1);
   EXPECT_FALSE(cn1->Approved());
   at3->SetValue(1);
   at4->SetValue(1);
   EXPECT_TRUE(cn1->Approved());
}

TEST(Condition, Disjunction)
{
   NOP::SharedAttribute<int> at1 = NOP::BuildAttribute<int>(-1);
   NOP::SharedAttribute<int> at2 = NOP::BuildAttribute<int>(-2);
   NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());

   NOP::SharedAttribute<int> at3 = NOP::BuildAttribute<int>(-3);
   NOP::SharedAttribute<int> at4 = NOP::BuildAttribute<int>(-4);
   NOP::SharedPremise pr2 = NOP::BuildPremise(at3, at4, NOP::Equal());

   NOP::SharedCondition cn1 =
      NOP::BuildCondition(CONDITION(*pr1 || *pr2), pr1, pr2);

   // On initialization should not be approved
   EXPECT_FALSE(cn1->Approved());
   at1->SetValue(1);
   at2->SetValue(1);
   EXPECT_TRUE(cn1->Approved());
   at3->SetValue(1);
   at4->SetValue(1);
   EXPECT_TRUE(cn1->Approved());
   at1->SetValue(-1);
   EXPECT_TRUE(cn1->Approved());
   at3->SetValue(-1);
   EXPECT_FALSE(cn1->Approved());
}

TEST(Condition, SubCondition)
{
   NOP::SharedAttribute<int> at1 = NOP::BuildAttribute<int>(-1);
   NOP::SharedAttribute<int> at2 = NOP::BuildAttribute<int>(-2);
   NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());

   NOP::SharedAttribute<int> at3 = NOP::BuildAttribute<int>(-3);
   NOP::SharedAttribute<int> at4 = NOP::BuildAttribute<int>(-4);
   NOP::SharedPremise pr2 = NOP::BuildPremise(at3, at4, NOP::Equal());

   NOP::SharedCondition cn1 = NOP::BuildCondition(CONDITION(*pr1), pr1);
   NOP::SharedCondition cn2 =
```

```
      NOP::BuildCondition(CONDITION(*pr2 && *cn1), pr2, cn1);

   // On initialization should not be approved
   EXPECT_FALSE(cn2->Approved());
   at1->SetValue(1);
   at2->SetValue(1);
   EXPECT_TRUE(cn1->Approved());
   EXPECT_FALSE(cn2->Approved());
   at3->SetValue(1);
   at4->SetValue(1);
   EXPECT_TRUE(cn2->Approved());
}

TEST(Condition, MasterRule)
{
   NOP::SharedAttribute<int> at1 = NOP::BuildAttribute<int>(-1);
   NOP::SharedAttribute<int> at2 = NOP::BuildAttribute<int>(-2);
   NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());

   NOP::SharedAttribute<int> at3 = NOP::BuildAttribute<int>(-3);
   NOP::SharedAttribute<int> at4 = NOP::BuildAttribute<int>(-4);
   NOP::SharedPremise pr2 = NOP::BuildPremise(at3, at4, NOP::Equal());

   NOP::SharedCondition cn1 = NOP::BuildCondition(CONDITION(*pr1), pr1);
   NOP::SharedRule rl1 =
      NOP::BuildRule(cn1, NOP::BuildAction(NOP::BuildInstigation()));

   NOP::SharedCondition cn2 =
      NOP::BuildCondition(CONDITION(*pr2 && *rl1), pr2, rl1);

   EXPECT_FALSE(cn2->Approved());
   at1->SetValue(1);
   at2->SetValue(1);
   NOP::Scheduler::Instance().FinishAll();
   EXPECT_TRUE(rl1->Approved());
   EXPECT_FALSE(cn2->Approved());
   at3->SetValue(1);
   at4->SetValue(1);
   NOP::Scheduler::Instance().FinishAll();
   EXPECT_TRUE(cn2->Approved());
}

TEST(Complete, Basic)
{
   NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(-1);
   NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(-2);
   NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());
   NOP::SharedCondition cn1 = NOP::BuildCondition(CONDITION(*pr1), pr1);

   std::atomic<int> executionCounter{0};
   NOP::Method mt = [&] { executionCounter++; };
   NOP::SharedInstigation in1 = NOP::BuildInstigation(mt);
   NOP::SharedAction ac1 = NOP::BuildAction(in1);

   NOP::SharedRule rl1 = NOP::BuildRule(cn1, ac1);
   NOP::Scheduler::Instance().FinishAll();

   // Nothing should be approved upon initialization
   EXPECT_FALSE(pr1->Approved());
   EXPECT_FALSE(cn1->Approved());
   EXPECT_FALSE(rl1->Approved());

   at1->SetValue(1);
   NOP::Scheduler::Instance().FinishAll();

   // Nothing should be approved yet
   EXPECT_FALSE(pr1->Approved());
   EXPECT_FALSE(cn1->Approved());
   EXPECT_FALSE(rl1->Approved());

   at2->SetValue(1);
   NOP::Scheduler::Instance().FinishAll();

   // Everything should be approved
   EXPECT_TRUE(pr1->Approved());
   EXPECT_TRUE(cn1->Approved());
   EXPECT_TRUE(rl1->Approved());
   EXPECT_EQ(executionCounter, 1);

   at2->SetValue(10);
   NOP::Scheduler::Instance().FinishAll();
```

```
   EXPECT_FALSE(pr1->Approved());
   EXPECT_FALSE(cn1->Approved());
   EXPECT_FALSE(rl1->Approved());

   at1->SetValue(10);
   NOP::Scheduler::Instance().FinishAll();

   // Everything should be approved
   EXPECT_TRUE(pr1->Approved());
   EXPECT_TRUE(cn1->Approved());
   EXPECT_TRUE(rl1->Approved());
   EXPECT_EQ(executionCounter, 2);
}

#ifdef LIBNOP_LOG_ENABLE
TEST(Complete, Logger)
{
   NOP::Logger::Instance().SetLogFileName("test.log");
   NOP::Logger::Instance().SetLogFileName("test2.log");
   NOP::Logger::Instance().SetLogFileName("test.log");

   struct Test : NOP::FBE
   {
      explicit Test(const std::string_view name) : FBE(name) {}
      NOP::SharedAttribute<int> at1 =
         NOP::BuildAttributeNamed("at1", this, -1);
      // Build without parent
      NOP::SharedAttribute<int> at2 =
         NOP::BuildAttributeNamed("at2", nullptr, -2);
      NOP::SharedPremise pr1 =
         NOP::BuildPremiseNamed("pr1", this, at1, at2, NOP::Equal());
      NOP::SharedCondition cn1 =
         NOP::BuildConditionNamed("cn1", this, CONDITION(*pr1), pr1);

      std::atomic<int> executionCounter{0};
      NOP::Method mt = [&] { executionCounter++; };
      NOP::SharedInstigation in1 =
         NOP::BuildInstigationNamed("in1", this, mt);
      NOP::SharedAction ac1 = NOP::BuildActionNamed("ac1", this, in1);

      NOP::SharedRule rl1 = NOP::BuildRuleNamed("rl1", this, cn1, ac1);
   };

   Test test{"TestFBE"};

   NOP::Scheduler::Instance().FinishAll();

   // Nothing should be approved upon initialization
   EXPECT_FALSE(test.pr1->Approved());
   EXPECT_FALSE(test.cn1->Approved());
   EXPECT_FALSE(test.rl1->Approved());

   test.at1->SetValue(1);
   NOP::Scheduler::Instance().FinishAll();

   // Nothing should be approved yet
   EXPECT_FALSE(test.pr1->Approved());
   EXPECT_FALSE(test.cn1->Approved());
   EXPECT_FALSE(test.rl1->Approved());

   test.at2->SetValue(1);
   NOP::Scheduler::Instance().FinishAll();

   // Everything should be approved
   EXPECT_TRUE(test.pr1->Approved());
   EXPECT_TRUE(test.cn1->Approved());
   EXPECT_TRUE(test.rl1->Approved());
   EXPECT_EQ(test.executionCounter, 1);

   test.at2->SetValue(10);
   NOP::Scheduler::Instance().FinishAll();

   EXPECT_FALSE(test.pr1->Approved());
   EXPECT_FALSE(test.cn1->Approved());
   EXPECT_FALSE(test.rl1->Approved());

   test.at1->SetValue(10);
   NOP::Scheduler::Instance().FinishAll();

   // Everything should be approved
```

```
   EXPECT_TRUE(test.pr1->Approved());
   EXPECT_TRUE(test.cn1->Approved());
   EXPECT_TRUE(test.rl1->Approved());
   EXPECT_EQ(test.executionCounter, 2);
}
#endif

TEST(Complete, MultipleCycles)
{
   NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(-1);
   NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(1);
   NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());
   NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Single>(pr1);

   std::atomic<int> executionCounter{0};
   NOP::Method mt = [&] { executionCounter++; };
   NOP::SharedInstigation in1 = NOP::BuildInstigation(mt);
   NOP::SharedAction ac1 = NOP::BuildAction(in1);

   NOP::SharedRule rl1 = NOP::BuildRule(cn1, ac1);

   for (int i = 1; i <= INT16_MAX; i++)
   {
      at1->SetValue(1);
      NOP::Scheduler::Instance().FinishAll();

      at1->SetValue(-1);
      EXPECT_EQ(executionCounter, i);
   }

   EXPECT_EQ(executionCounter, INT16_MAX);
}

TEST(Counter, Default)
{
   NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(1);
   NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(-1);
   NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());
   NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Single>(pr1);

   std::atomic<int> executionCounter{0};
   NOP::Method mt = [&] { executionCounter++; };
   NOP::SharedInstigation in1 = NOP::BuildInstigation(mt);
   NOP::SharedAction ac1 = NOP::BuildAction(in1);

   NOP::SharedRule rl1 = NOP::BuildRule(cn1, ac1);
   NOP::Scheduler::Instance().FinishAll();

   constexpr int executions{INT8_MAX};
   for (int i = 0; i < executions; i++)
   {
      at2->SetValue(-1);
      at2->SetValue(1);
      NOP::Scheduler::Instance().FinishAll();
   }
   EXPECT_EQ(executionCounter, executions);
}

TEST(Complete, InitialStatesDisapproved)
{
   NOP::Scheduler& scheduler = NOP::Scheduler::Instance();

   NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(1);
   NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(2);
   NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());
   NOP::SharedCondition cn1 = NOP::BuildCondition(CONDITION(*pr1), pr1);

   std::atomic<int> executionCounter{0};
   NOP::Method mt = [&] { executionCounter++; };
   NOP::SharedInstigation in1 = NOP::BuildInstigation(mt);
   NOP::SharedAction ac1 = NOP::BuildAction(in1);

   NOP::SharedRule rl1 = NOP::BuildRule(cn1, ac1);
   scheduler.FinishAll();

   EXPECT_FALSE(pr1->Approved());
   EXPECT_FALSE(cn1->Approved());
   EXPECT_FALSE(rl1->Approved());
   NOP::Scheduler::Instance().FinishAll();
   EXPECT_EQ(executionCounter, 0);
}
```

```
TEST(Complete, InitialStatesApproved)
{
   NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(1);
   NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(1);
   NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());
   NOP::SharedCondition cn1 = NOP::BuildCondition(CONDITION(*pr1), pr1);

   std::atomic<int> executionCounter{0};
   NOP::Method mt = [&] { executionCounter++; };
   NOP::SharedInstigation in1 = NOP::BuildInstigation(mt);
   NOP::SharedAction ac1 = NOP::BuildAction(in1);

   NOP::SharedRule rl1 = NOP::BuildRule(cn1, ac1);

   NOP::Scheduler::Instance().FinishAll();

   EXPECT_TRUE(pr1->Approved());
   EXPECT_TRUE(cn1->Approved());
   EXPECT_TRUE(rl1->Approved());

   EXPECT_EQ(executionCounter, 1);
}

TEST(Complete, SharedEntities1)
{
   NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(1);
   NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(1);
   NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());
   NOP::SharedCondition cn1 = NOP::BuildCondition(CONDITION(*pr1), pr1);

   std::atomic<int> executionCounter{0};
   NOP::Method mt = [&] { executionCounter++; };
   // clone methods
   NOP::SharedInstigation in1 = NOP::BuildInstigation(mt);
   NOP::SharedAction ac1 = NOP::BuildAction(in1);
   NOP::SharedRule rl1 = NOP::BuildRule(cn1, ac1);

   NOP::SharedInstigation in2 = NOP::BuildInstigation(mt);
   NOP::SharedAction ac2 = NOP::BuildAction(in2);
   NOP::SharedRule rl2 = NOP::BuildRule(cn1, ac2);

   NOP::Scheduler::Instance().FinishAll();

   EXPECT_EQ(executionCounter, 2);
}

TEST(Complete, SharedEntities2)
{
   NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(1);
   NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(1);
   NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());
   NOP::SharedCondition cn1 = NOP::BuildCondition(CONDITION(*pr1), pr1);
   NOP::SharedCondition cn2 = NOP::BuildCondition(CONDITION(*pr1), pr1);
   std::atomic<int> executionCounter{0};
   NOP::Method mt = [&] { executionCounter++; };
   // clone methods
   NOP::SharedInstigation in1 = NOP::BuildInstigation(mt);
   NOP::SharedAction ac1 = NOP::BuildAction(in1);
   NOP::SharedRule rl1 = NOP::BuildRule(cn1, ac1);
   NOP::SharedRule rl2 = NOP::BuildRule(cn2, ac1);

   NOP::Scheduler::Instance().FinishAll();

   EXPECT_EQ(executionCounter, 2);
}

TEST(Complete, Renotification)
{
   NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(-1);
   NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(-2);
   NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());
   NOP::SharedCondition cn1 = NOP::BuildCondition(CONDITION(*pr1), pr1);
   std::atomic<int> executionCounter{0};
   NOP::SharedInstigation in1 =
      NOP::BuildInstigation([&] { executionCounter++; });
   NOP::SharedAction ac1 = NOP::BuildAction(in1);
   NOP::SharedRule rl1 = NOP::BuildRule(cn1, ac1);

   at1->SetValue(1, false);
   at2->SetValue(1, false);
```

```
   NOP::Scheduler::Instance().FinishAll();

   EXPECT_EQ(executionCounter, 1);

   at2->SetValue(1, true);
   NOP::Scheduler::Instance().FinishAll();
   EXPECT_EQ(executionCounter, 2);

   at2->SetValue(1, true);
   NOP::Scheduler::Instance().FinishAll();
   EXPECT_EQ(executionCounter, 3);
}

TEST(Complete, RecursiveRule)
{
   NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(0);
   NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(0);
   NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, std::less<>());
   NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Single>(pr1);
   int executionCounter{0};
   auto test = [&]()
   {
      executionCounter++;
      at1->SetValue<NOP::ReNotify>(at1->GetValue() + 1);
   };
   NOP::SharedInstigation in1 = NOP::BuildInstigation(test);
   NOP::SharedAction ac1 = NOP::BuildAction(in1);
   NOP::SharedRule rl1 = NOP::BuildRule(cn1, ac1);

   at2->SetValue(10);
   NOP::Scheduler::Instance().FinishAll();

   EXPECT_EQ(executionCounter, 10);
}
TEST(Condition, Composition)
{
   NOP::SharedAttribute<int> at1 = NOP::BuildAttribute<int>(-1);
   NOP::SharedAttribute<int> at2 = NOP::BuildAttribute<int>(-2);
   NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());

   NOP::SharedAttribute<int> at3 = NOP::BuildAttribute<int>(-3);
   NOP::SharedAttribute<int> at4 = NOP::BuildAttribute<int>(-4);
   NOP::SharedPremise pr2 = NOP::BuildPremise(at3, at4, NOP::Equal());

   NOP::SharedAttribute<int> at5 = NOP::BuildAttribute<int>(0);
   NOP::SharedAttribute<int> at6 = NOP::BuildAttribute<int>(0);
   NOP::SharedPremise pr3 = NOP::BuildPremise(at5, at6, NOP::Equal());

   NOP::SharedCondition cn1 =
      NOP::BuildCondition(CONDITION((*pr1 || *pr2) && !*pr3), pr1, pr2, pr3);

   at1->SetValue(1);
   at2->SetValue(1);
   NOP::Scheduler::Instance().FinishAll();
   EXPECT_TRUE(pr1->Approved());
   EXPECT_FALSE(pr2->Approved());
   EXPECT_TRUE(pr3->Approved());
   EXPECT_FALSE(cn1->Approved());

   at5->SetValue(1);
   NOP::Scheduler::Instance().FinishAll();
   EXPECT_FALSE(pr3->Approved());
   EXPECT_TRUE(cn1->Approved());

   at2->SetValue(-1);
   NOP::Scheduler::Instance().FinishAll();
   EXPECT_FALSE(pr1->Approved());
   EXPECT_FALSE(cn1->Approved());

   at3->SetValue(1);
   at4->SetValue(1);
   NOP::Scheduler::Instance().FinishAll();
   EXPECT_TRUE(pr2->Approved());
   EXPECT_TRUE(cn1->Approved());
}
TEST(Flag, Exclusive)
{
   NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(0);
   NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(0);

   NOP::SharedAttribute<bool> atExclusive = NOP::BuildAttribute(false);
```

```
   NOP::SharedPremise pr1 = NOP::BuildPremise(at1, 1, NOP::Equal());
   NOP::SharedPremise pr2 = NOP::BuildPremise(at2, 2, NOP::Equal());

   NOP::SharedPremise prExclusive =
      NOP::BuildPremise(atExclusive, true, NOP::Equal());

   NOP::SharedCondition cn1 =
      NOP::BuildCondition<NOP::Conjunction>(pr1, prExclusive);
   NOP::SharedCondition cn2 =
      NOP::BuildCondition<NOP::Conjunction>(pr2, prExclusive);

   std::atomic<int> executionCounter1{0};
   auto test1 = [&]()
   {
      std::this_thread::sleep_for(std::chrono::milliseconds(1));
      EXPECT_TRUE(atExclusive->GetValue());
      atExclusive->SetValue<NOP::Exclusive>(false);
      std::this_thread::sleep_for(std::chrono::milliseconds(20));
      ++executionCounter1;
      at1->SetValue(0);
      atExclusive->SetValue<NOP::Exclusive>(true);
   };
   NOP::SharedInstigation in1 = NOP::BuildInstigation(test1);
   NOP::SharedAction ac1 = NOP::BuildAction(in1);
   NOP::SharedRule rl1 = NOP::BuildRule(cn1, ac1);

   std::atomic<int> executionCounter2{0};
   auto test2 = [&]()
   {
      std::this_thread::sleep_for(std::chrono::milliseconds(5));
      EXPECT_TRUE(atExclusive->GetValue());
      atExclusive->SetValue<NOP::Exclusive>(false);
      std::this_thread::sleep_for(std::chrono::milliseconds(10));
      ++executionCounter2;
      at2->SetValue(0);
      atExclusive->SetValue<NOP::Exclusive>(true);
   };

   NOP::SharedInstigation in2 = NOP::BuildInstigation(test2);
   NOP::SharedAction ac2 = NOP::BuildAction(in2);
   NOP::SharedRule rl2 = NOP::BuildRule(cn2, ac2);

   atExclusive->SetValue<NOP::Exclusive>(true);
   {
      std::thread t1{[&]() { at1->SetValue<NOP::Exclusive>(1); }};
      std::thread t2{[&]() { at2->SetValue<NOP::Exclusive>(2); }};
      std::this_thread::sleep_for(std::chrono::milliseconds(10));
      t1.join();
      t2.join();
   }

   NOP::Scheduler::Instance().FinishAll();

   EXPECT_EQ(executionCounter1, 1);
   EXPECT_EQ(executionCounter2, 1);
}

TEST(Example, Alarm)
{
   struct Alarm
   {
      int alarmCount{0};

      NOP::SharedAttribute<bool> atState{NOP::BuildAttribute<bool>(false)};
      NOP::SharedPremise prAlarm{
         NOP::BuildPremise(atState, true, NOP::Equal())};
      NOP::SharedCondition cnAlarm{
         NOP::BuildCondition(CONDITION(*prAlarm), prAlarm)};
      NOP::Method mtNotify { METHOD(alarmCount++;) };
      NOP::SharedInstigation inAlarm{NOP::BuildInstigation(mtNotify)};
      NOP::SharedAction acAlarm{NOP::BuildAction(inAlarm)};
      NOP::SharedRule rlAlarm{NOP::BuildRule(cnAlarm, acAlarm)};
   };

   Alarm al1;
   Alarm al2;

   for (int i = 0; i < 10; i++)
   {
      al1.atState->SetValue(!al1.atState->GetValue());
```

```
      al2.atState->SetValue<NOP::ReNotify>(true);
      NOP::Scheduler::Instance().FinishAll();
   }
   EXPECT_EQ(al1.alarmCount, 5);
   EXPECT_EQ(al2.alarmCount, 10);
}

TEST(Example, AlarmSimplified)
{
   struct Alarm
   {
      int alarmCount{0};

      NOP::SharedAttribute<bool> atState{NOP::BuildAttribute<bool>(false)};
      NOP::SharedPremise prAlarm{
         NOP::BuildPremise(atState, true, NOP::Equal())};
      NOP::SharedRule rlAlarm
      {
         NOP::BuildRule(
            NOP::BuildCondition(CONDITION(*prAlarm), prAlarm),
            NOP::BuildAction(NOP::BuildInstigation(METHOD(alarmCount++;))))
      };
   };

   Alarm al1;
   Alarm al2;

   for (int i = 0; i < 10; i++)
   {
      al1.atState->SetValue(!al1.atState->GetValue());
      al2.atState->SetValue<NOP::ReNotify>(true);
      NOP::Scheduler::Instance().FinishAll();
   }
   EXPECT_EQ(al1.alarmCount, 5);
   EXPECT_EQ(al2.alarmCount, 10);
}

#ifdef LIBNOP_SCHEDULER_ENABLE
TEST(Complete, Competing)
{
   NOP::SharedAttribute<bool> atShouldExecute = NOP::BuildAttribute(true);
   NOP::SharedAttribute<bool> atResourceAvailable = NOP::BuildAttribute(false);
   NOP::SharedPremise prShouldExecute =
      NOP::BuildPremise(atShouldExecute, true, NOP::Equal());
   NOP::SharedPremise prResourceAvailable =
      NOP::BuildPremise(atResourceAvailable, true, NOP::Equal());

   NOP::SharedCondition cn1 =
      NOP::BuildCondition(CONDITION(*prShouldExecute && *prResourceAvailable),
                     prShouldExecute, prResourceAvailable);

   std::atomic<int> executionCounter1{0};
   NOP::Method mt1 = [&]()
   {
      atResourceAvailable->SetValue(false);
      ++executionCounter1;
   };
   NOP::Method mt2 = [&]()
   {
      atResourceAvailable->SetValue(false);
      ++executionCounter1;
   };
   NOP::SharedInstigation in1 = NOP::BuildInstigation(mt1);
   NOP::SharedAction ac1 = NOP::BuildAction(in1);
   NOP::SharedInstigation in2 = NOP::BuildInstigation(mt2);
   NOP::SharedAction ac2 = NOP::BuildAction(in2);

   NOP::SharedRule rl1 = NOP::BuildRule(cn1, ac1);
   NOP::SharedRule rl2 = NOP::BuildRule(cn1, ac2);

   atResourceAvailable->SetValue(true);
   NOP::Scheduler::Instance().FinishAll();

   EXPECT_EQ(executionCounter1, 1);

   // Use function just to apply code coverage
   mt1();
   mt2();
}
#endif
```

```
TEST(Condition, EnumConjunction)
{
   NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(-1);
   NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(-1);
   NOP::SharedPremise pr1 = NOP::BuildPremise(at1, 1, NOP::Equal());
   NOP::SharedPremise pr2 = NOP::BuildPremise(at2, 2, NOP::Equal());

   NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Conjunction>(pr1, pr2);

   std::atomic<int> executionCounter{0};
   NOP::Method mt = [&] { ++executionCounter; };
   NOP::SharedInstigation in1 = NOP::BuildInstigation(mt);
   NOP::SharedAction ac1 = NOP::BuildAction(in1);

   NOP::SharedRule rl1 = NOP::BuildRule(cn1, ac1);
   NOP::Scheduler::Instance().FinishAll();
   EXPECT_FALSE(rl1->Approved());
   EXPECT_EQ(executionCounter, 0);

   at1->SetValue(1);
   NOP::Scheduler::Instance().FinishAll();
   EXPECT_FALSE(rl1->Approved());
   EXPECT_EQ(executionCounter, 0);

   at2->SetValue(2);
   NOP::Scheduler::Instance().FinishAll();
   EXPECT_TRUE(rl1->Approved());
   EXPECT_EQ(executionCounter, 1);
}

TEST(Condition, EnumDisjunction)
{
   NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(-1);
   NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(-1);
   NOP::SharedPremise pr1 = NOP::BuildPremise(at1, 1, NOP::Equal());
   NOP::SharedPremise pr2 = NOP::BuildPremise(at2, 2, NOP::Equal());

   NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Disjunction>(pr1, pr2);

   std::atomic<int> executionCounter{0};
   NOP::Method mt = [&] { ++executionCounter; };
   NOP::SharedInstigation in1 = NOP::BuildInstigation(mt);
   NOP::SharedAction ac1 = NOP::BuildAction(in1);

   NOP::SharedRule rl1 = NOP::BuildRule(cn1, ac1);
   NOP::Scheduler::Instance().FinishAll();
   EXPECT_FALSE(rl1->Approved());
   EXPECT_EQ(executionCounter, 0);

   at1->SetValue(1);
   NOP::Scheduler::Instance().FinishAll();
   EXPECT_TRUE(rl1->Approved());
   EXPECT_EQ(executionCounter, 1);

   at2->SetValue(2);
   NOP::Scheduler::Instance().FinishAll();
   EXPECT_TRUE(rl1->Approved());
   EXPECT_EQ(executionCounter, 1);
}

TEST(Flag, NoNotify)
{
   NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(-1);
   NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(-2);
   NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());
   NOP::SharedCondition cn1 = NOP::BuildCondition(CONDITION(*pr1), pr1);

   std::atomic<int> executionCounter{0};
   NOP::Method mt = [&] { executionCounter++; };
   NOP::SharedInstigation in1 = NOP::BuildInstigation(mt);
   NOP::SharedAction ac1 = NOP::BuildAction(in1);

   NOP::SharedRule rl1 = NOP::BuildRule(cn1, ac1);
   at1->SetValue<NOP::Default>(1);
   at2->SetValue<NOP::NoNotify>(1);
   NOP::Scheduler::Instance().FinishAll();

   EXPECT_FALSE(rl1->Approved());
   EXPECT_EQ(executionCounter, 0);
}
```

```
TEST(Flag, ParallelNoNotify)
{
   NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(-1);
   NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(-2);
   NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());
   NOP::SharedCondition cn1 = NOP::BuildCondition(CONDITION(*pr1), pr1);

   std::atomic<int> executionCounter{0};
   NOP::Method mt = [&] { executionCounter++; };
   NOP::SharedInstigation in1 = NOP::BuildInstigation(mt);
   NOP::SharedAction ac1 = NOP::BuildAction(in1);

   NOP::SharedRule rl1 = NOP::BuildRule(cn1, ac1);

   at1->SetValue<NOP::Parallel>(1);
   at2->SetValue<NOP::Parallel | NOP::NoNotify>(1);
   NOP::Scheduler::Instance().FinishAll();

   EXPECT_FALSE(rl1->Approved());
   EXPECT_EQ(executionCounter, 0);
}

TEST(Flag, ReNotify)
{
   NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(-1);
   NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(-2);
   NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());
   NOP::SharedCondition cn1 = NOP::BuildCondition(CONDITION(*pr1), pr1);

   std::atomic<int> executionCounter{0};
   NOP::Method mt = [&] { executionCounter++; };
   NOP::SharedInstigation in1 = NOP::BuildInstigation(mt);
   NOP::SharedAction ac1 = NOP::BuildAction(in1);

   NOP::SharedRule rl1 = NOP::BuildRule(cn1, ac1);

   at1->SetValue<NOP::Default>(1);
   at2->SetValue<NOP::Default>(1);

   NOP::Scheduler::Instance().FinishAll();

   EXPECT_TRUE(rl1->Approved());
   EXPECT_EQ(executionCounter, 1);

   at2->SetValue<NOP::ReNotify>(1);
   NOP::Scheduler::Instance().FinishAll();

   EXPECT_TRUE(rl1->Approved());
   EXPECT_EQ(executionCounter, 2);
}

TEST(Flag, Parallel)
{
   NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(-1);
   NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(-2);
   NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());
   NOP::SharedCondition cn1 = NOP::BuildCondition(CONDITION(*pr1), pr1);

   std::atomic<int> executionCounter{0};
   NOP::Method mt = [&] { executionCounter++; };
   NOP::SharedInstigation in1 = NOP::BuildInstigation(mt);
   NOP::SharedAction ac1 = NOP::BuildAction(in1);

   NOP::SharedRule rl1 = NOP::BuildRule(cn1, ac1);
   NOP::SharedRule rl2 = NOP::BuildRule(cn1, ac1);

   at1->SetValue<NOP::Parallel>(1);
   at2->SetValue<NOP::Parallel>(1);
   NOP::Scheduler::Instance().FinishAll();

   EXPECT_TRUE(rl2->Approved());
   EXPECT_TRUE(rl1->Approved());
   EXPECT_EQ(executionCounter, 2);
}

TEST(Flag, ReNotifyParallel)
{
   NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(-1);
   NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(-2);
   NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());
   NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Single>(pr1);
```

```
   auto ptr = NOP::BuildPremise(at1, 1, NOP::Equal());

   std::atomic<int> executionCounter{0};
   NOP::Method mt = [&] { executionCounter++; };
   NOP::SharedInstigation in1 = NOP::BuildInstigation(mt);
   NOP::SharedAction ac1 = NOP::BuildAction(in1);

   NOP::SharedRule rl1 = NOP::BuildRule(cn1, ac1);
   NOP::SharedRule rl2 = NOP::BuildRule(cn1, ac1);

   at1->SetValue<NOP::Parallel>(1);
   at2->SetValue<NOP::Parallel>(1);
   NOP::Scheduler::Instance().FinishAll();

   EXPECT_TRUE(rl2->Approved());
   EXPECT_TRUE(rl1->Approved());
   EXPECT_EQ(executionCounter, 2);

   at1->SetValue<NOP::Parallel | NOP::ReNotify>(1);
   NOP::Scheduler::Instance().FinishAll();

   // Everything should be approved
   EXPECT_TRUE(rl2->Approved());
   EXPECT_TRUE(rl1->Approved());
   EXPECT_EQ(executionCounter, 4);
}

TEST(Flag, ReNotifyNoNotify)
{
   NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(-1);
   NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(-2);
   NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());
   NOP::SharedCondition cn1 = NOP::BuildCondition(CONDITION(*pr1), pr1);

   std::atomic<int> executionCounter{0};
   NOP::Method mt = [&] { executionCounter++; };
   NOP::SharedInstigation in1 = NOP::BuildInstigation(mt);
   NOP::SharedAction ac1 = NOP::BuildAction(in1);

   NOP::SharedRule rl1 = NOP::BuildRule(cn1, ac1);

   at1->SetValue<NOP::NoNotify | NOP::ReNotify>(1);
   at2->SetValue<NOP::NoNotify | NOP::ReNotify>(1);
   NOP::Scheduler::Instance().FinishAll();

   EXPECT_FALSE(rl1->Approved());
   EXPECT_EQ(executionCounter, 0);
}

#ifdef LIBNOP_SCHEDULER_ENABLE
TEST(Scheduler, FIFO)
{
   NOP::Scheduler::Instance().SetStrategy(NOP::FIFO);
   NOP::SharedAttribute<bool> atDelay = NOP::BuildAttribute(false);
   NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(0);
   NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(0);
   NOP::SharedPremise pr1 = NOP::BuildPremise(at1, 1, NOP::Equal());
   NOP::SharedPremise pr2 = NOP::BuildPremise(at2, 2, NOP::Equal());
   NOP::SharedPremise prDelay = NOP::BuildPremise(atDelay, true, NOP::Equal());

   NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Single>(pr1);
   NOP::SharedCondition cn2 = NOP::BuildCondition<NOP::Single>(pr2);
   NOP::SharedCondition cnDelay = NOP::BuildCondition<NOP::Single>(prDelay);

   std::atomic<int> executionCounter{0};
   NOP::Method mt1 = [&] { EXPECT_EQ(1, ++executionCounter); };
   NOP::SharedInstigation in1 = NOP::BuildInstigation(mt1);
   NOP::SharedAction ac1 = NOP::BuildAction(in1);

   NOP::Method mt2 = [&] { EXPECT_EQ(2, ++executionCounter); };
   NOP::SharedInstigation in2 = NOP::BuildInstigation(mt2);
   NOP::SharedAction ac2 = NOP::BuildAction(in2);

   NOP::SharedRule rl1 = NOP::BuildRule(cn1, ac1);
   NOP::SharedRule rl2 = NOP::BuildRule(cn2, ac2);

   NOP::Method mtDelay = [&]
   { std::this_thread::sleep_for(std::chrono::milliseconds(10)); };
   NOP::SharedInstigation inDelay = NOP::BuildInstigation(mtDelay);
   NOP::SharedAction acDelay = NOP::BuildAction(inDelay);
   NOP::SharedRule rlDelay = NOP::BuildRule(cnDelay, acDelay);
```

```
   atDelay->SetValue(true);
   at1->SetValue(1);
   at2->SetValue(2);
   NOP::Scheduler::Instance().FinishAll();

   EXPECT_EQ(executionCounter, 2);
}

TEST(Scheduler, LIFO)
{
   NOP::Scheduler::Instance().SetStrategy(NOP::LIFO);
   NOP::SharedAttribute<bool> atDelay = NOP::BuildAttribute(false);
   NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(0);
   NOP::SharedAttribute<int> at2 = NOP::BuildAttribute(0);
   NOP::SharedPremise pr1 = NOP::BuildPremise(at1, 1, NOP::Equal());
   NOP::SharedPremise pr2 = NOP::BuildPremise(at2, 2, NOP::Equal());
   NOP::SharedPremise prDelay = NOP::BuildPremise(atDelay, true, NOP::Equal());

   NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Single>(pr1);
   NOP::SharedCondition cn2 = NOP::BuildCondition<NOP::Single>(pr2);
   NOP::SharedCondition cnDelay = NOP::BuildCondition<NOP::Single>(prDelay);

   std::atomic<int> executionCounter{0};
   NOP::Method mt1 = [&] { EXPECT_EQ(2, ++executionCounter); };
   NOP::SharedInstigation in1 = NOP::BuildInstigation(mt1);
   NOP::SharedAction ac1 = NOP::BuildAction(in1);

   NOP::Method mt2 = [&] { EXPECT_EQ(1, ++executionCounter); };
   NOP::SharedInstigation in2 = NOP::BuildInstigation(mt2);
   NOP::SharedAction ac2 = NOP::BuildAction(in2);

   NOP::SharedRule rl1 = NOP::BuildRule(cn1, ac1);
   NOP::SharedRule rl2 = NOP::BuildRule(cn2, ac2);

   NOP::Method mtDelay = [&]
   { std::this_thread::sleep_for(std::chrono::milliseconds(100)); };
   NOP::SharedInstigation inDelay = NOP::BuildInstigation(mtDelay);
   NOP::SharedAction acDelay = NOP::BuildAction(inDelay);
   NOP::SharedRule rlDelay = NOP::BuildRule(cnDelay, acDelay);

   atDelay->SetValue(true);
   std::this_thread::sleep_for(std::chrono::milliseconds(10));
   at1->SetValue(1);
   at2->SetValue(2);
   NOP::Scheduler::Instance().FinishAll();

   EXPECT_EQ(executionCounter, 2);
}

TEST(Scheduler, Priority)
{
   NOP::Scheduler::Instance().SetStrategy(NOP::Priority);
   NOP::SharedAttribute<bool> atDelay = NOP::BuildAttribute(false);
   NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(0);
   NOP::SharedPremise pr1 = NOP::BuildPremise(at1, 1, NOP::Equal());
   NOP::SharedPremise prDelay = NOP::BuildPremise(atDelay, true, NOP::Equal());

   NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Single>(pr1);
   NOP::SharedCondition cnDelay = NOP::BuildCondition<NOP::Single>(prDelay);

   std::atomic<int> executionCounter{0};
   NOP::Method mt1 = [&] { EXPECT_EQ(2, ++executionCounter); };
   NOP::SharedInstigation in1 = NOP::BuildInstigation(mt1);
   NOP::SharedAction ac1 = NOP::BuildAction(in1);

   NOP::Method mt2 = [&] { EXPECT_EQ(1, ++executionCounter); };
   NOP::SharedInstigation in2 = NOP::BuildInstigation(mt2);
   NOP::SharedAction ac2 = NOP::BuildAction(in2);

   NOP::Method mt3 = [&] { EXPECT_EQ(3, ++executionCounter); };
   NOP::SharedInstigation in3 = NOP::BuildInstigation(mt3);
   NOP::SharedAction ac3 = NOP::BuildAction(in3);

   NOP::SharedRule rl1 = NOP::BuildRule(cn1, ac1, 0);
   NOP::SharedRule rl2 = NOP::BuildRule(cn1, ac2, 123);
   NOP::SharedRule rl3 = NOP::BuildRule(cn1, ac3, -1);

   NOP::Method mtDelay = [&]
   { std::this_thread::sleep_for(std::chrono::milliseconds(10)); };
   NOP::SharedInstigation inDelay = NOP::BuildInstigation(mtDelay);
```

```
   NOP::SharedAction acDelay = NOP::BuildAction(inDelay);
   NOP::SharedRule rlDelay = NOP::BuildRule(cnDelay, acDelay);

   atDelay->SetValue(true);
   at1->SetValue(1);
   NOP::Scheduler::Instance().FinishAll();

   EXPECT_EQ(executionCounter, 3);
}
#endif

TEST(Methods, Async)
{
   NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(-1);
   NOP::SharedPremise pr1 = NOP::BuildPremise(at1, 1, NOP::Equal());

   NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Single>(pr1);

   std::atomic<int> executionCounter{0};
   NOP::SharedInstigation in1 =
      NOP::BuildInstigation(ASYNC_METHOD(++executionCounter;));
   NOP::SharedAction ac1 = NOP::BuildAction(in1);

   NOP::SharedRule rl1 = NOP::BuildRule(cn1, ac1);
   at1->SetValue(1);
   NOP::Scheduler::Instance().FinishAll();
   EXPECT_TRUE(rl1->Approved());
   // Should not be executed yet
   EXPECT_EQ(executionCounter, 0);
   std::this_thread::sleep_for(std::chrono::milliseconds(100));
   EXPECT_EQ(executionCounter, 1);
}

TEST(Methods, AsyncMultiple)
{
   NOP::Scheduler::Instance().SetStrategy(NOP::EStrategy::None);
   NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(-1);
   NOP::SharedPremise pr1 = NOP::BuildPremise(at1, 1, NOP::Equal());

   NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Single>(pr1);

   std::atomic<int> executionCounter{0};
   NOP::SharedInstigation in1 =
      NOP::BuildInstigation(ASYNC_METHOD(++executionCounter;));
   NOP::SharedAction ac1 = NOP::BuildAction(in1);

   NOP::SharedRule rl1 = NOP::BuildRule(cn1, ac1);
   at1->SetValue(1);
   at1->SetValue(0);
   at1->SetValue(1);
   at1->SetValue(0);
   at1->SetValue(1);
   NOP::Scheduler::Instance().FinishAll();
   std::this_thread::sleep_for(std::chrono::milliseconds(200));
   EXPECT_EQ(executionCounter, 3);
}

TEST(Methods, AsyncMultipleSlow)
{
   NOP::Scheduler::Instance().SetStrategy(NOP::EStrategy::None);
   NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(-1);
   NOP::SharedPremise pr1 = NOP::BuildPremise(at1, 1, NOP::Equal());

   NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Single>(pr1);

   std::atomic<int> executionCounter{0};
   NOP::SharedInstigation in1 = NOP::BuildInstigation(
      ASYNC_METHOD(std::this_thread::sleep_for(std::chrono::milliseconds(10));
                ++executionCounter;));
   NOP::SharedAction ac1 = NOP::BuildAction(in1);

   NOP::SharedRule rl1 = NOP::BuildRule(cn1, ac1);
   at1->SetValue(1);
   at1->SetValue(0);
   at1->SetValue(1);
   at1->SetValue(0);
   at1->SetValue(1);
   NOP::Scheduler::Instance().FinishAll();
   std::this_thread::sleep_for(std::chrono::milliseconds(20));
   EXPECT_EQ(executionCounter, 3);
}
```

```
TEST(Parallel, Action)
{
   NOP::Scheduler::Instance().SetStrategy(NOP::EStrategy::None);
   NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(-1);
   NOP::SharedPremise pr1 = NOP::BuildPremise(at1, 1, NOP::Equal());

   NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Single>(pr1);

   std::atomic<int> executionCounter{0};
   NOP::SharedInstigation in1 =
      NOP::BuildInstigation(METHOD(++executionCounter;));
   NOP::SharedInstigation in2 =
      NOP::BuildInstigation(METHOD(++executionCounter;));
   NOP::SharedAction ac1 = NOP::BuildAction<NOP::Parallel>(in1, in2);

   NOP::SharedRule rl1 = NOP::BuildRule(cn1, ac1);
   at1->SetValue(1);
   at1->SetValue(0);
   at1->SetValue(1);
   at1->SetValue(0);
   at1->SetValue(1);
   NOP::Scheduler::Instance().FinishAll();
   EXPECT_EQ(executionCounter, 6);
}

TEST(Parallel, Instigation)
{
   NOP::Scheduler::Instance().SetStrategy(NOP::EStrategy::None);
   NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(-1);
   NOP::SharedPremise pr1 = NOP::BuildPremise(at1, 1, NOP::Equal());

   NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Single>(pr1);

   std::atomic<int> executionCounter{0};
   NOP::SharedInstigation in1 = NOP::BuildInstigation<NOP::Parallel>(
      METHOD(++executionCounter;), METHOD(++executionCounter;));

   NOP::SharedAction ac1 = NOP::BuildAction<NOP::Parallel>(in1);

   NOP::SharedRule rl1 = NOP::BuildRule(cn1, ac1);
   at1->SetValue(1);
   at1->SetValue(0);
   at1->SetValue(1);
   at1->SetValue(0);
   at1->SetValue(1);
   NOP::Scheduler::Instance().FinishAll();
   EXPECT_EQ(executionCounter, 6);
}
```

APÊNDICE E

README.md 7/25/2021

Framework NOP C++ 4.0 (libnop)

This framwork was developed to make it easier to build applications using the NOP in C++. It uses CMake to build the library and applications.

Requirements

- Unix or Windows (OSX is untested)
- CMake 3.14
- C++20 compliant compiler (MSVC 14.2 / GCC 10 or GCC 11 is recomended)

Usage instructions

Check the template project for a ready to use template: https://nop.dainf.ct.utfpr.edu.br/nop-applications/framework-application/framework-cpp-4-application/template-application

Check libnop_gtest/test.cpp file for more examples on how to use the framework.

Build options

The following options must be set during cmake generation.

Usage options:

- LIBNOP_SCHEDULER_ENABLE: Enable use of schedluer for executing Rules (reduces performance)
- LIBNOP_LOG_ENABLE: Enable using logs (heavy performance hit)

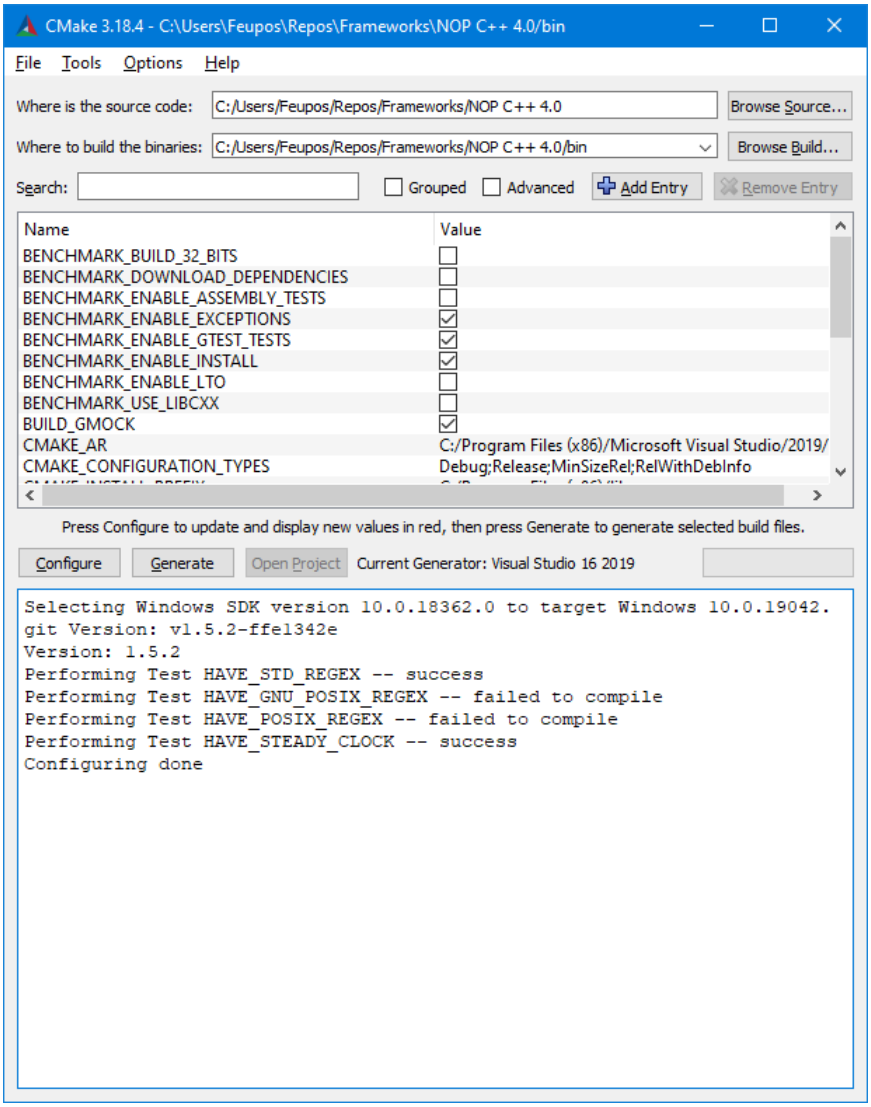
Development options:

- LIBNOP_TEST_ENABLE: Enable building unity tests
- LIBNOP_COVERAGE_ENABLE: Enable code coverage
- LIBNOP_BENCHMARK_ENABLE: Enable build benchmarks
- LIBNOP_BENCHMARK_SENSOR_ENABLE: Enable build sensor benchmark
- LIBNOP_BENCHMARK_BITONIC_ENABLE: Enable build bitonic benchmark
- LIBNOP_BENCHMARK_RANDOMFOREST_ENABLE: Enable build random forest benchmark
- LIBNOP_FW2_TEST_ENABLE: Enable build NOP C++ Framework 2.0 tests

For maximum performance, all options are disabled by default. As an alternative you can add "#define LIBNOP_SCHEDULER_ENABLE" before including "#include "NOPFramework.h"" or "#include "libnop/framework.h"" to enable the scheduler.

Hot to build (Windows)

Download and install the CMake GUI for windows and generate the project for your IDE. https://cmake.org/download/



1. Set source folder - Ex: nop-framework-cpp-4/
2. Set build folder - Ex: nop-framework-cpp-4/build
3. Configure the desired options
4. Generate
5. Open Project

Hot to build (Unix)

On Ubuntu 20.04 you must install gcc 10/11 and libtbb and cmake as follows:

1. `sudo apt-get install -y g++-11`

2. `sudo apt-get install -y libtbb-dev`
3. `sudo apt-get install python3-pip`
4. `sudo pip install cmake`

With the dependencies installed you can compile:

1. `mkdir build && cd build`
2. `cmake .. -DCMAKE_CXX_COMPILER=g++-11`
3. `sudo make install`

Tests and benchmark (Unix)

After building you can run the tests and benchmarks:

1. `libnop_gtest/libnop_gtest`
2. `libnop_gbench/libnop_gbench`

How to use

Include the header "libnop/framework.h" in your files.

To utilize this library in other `CMake` projects you can add the following lines to your CMakeLists.txt:

```
find_package(libnop)
if(NOT libnop_FOUND)
    include(FetchContent)
    FetchContent_Declare(libnop
        GIT_REPOSITORY https://nop.dainf.ct.utfpr.edu.br/nop-
implementations/frameworks/nop-framework-cpp-4.git
        GIT_TAG master
    )

    FetchContent_GetProperties(libnop)
    if(NOT libnop_POPULATED)
        FetchContent_MakeAvailable(libnop)
    endif()
endif()
```

C++20 is required for use:

```
set_target_properties(<target> PROPERTIES
    CXX_STANDARD 20
)
```

Alternatively you can clone the repository and install the library in your system:

1. `git clone https://nop.dainf.ct.utfpr.edu.br/nop-implementations/frameworks/nop-framework-cpp-4.git`
2. `cd nop-framework-cpp-4.git`
3. `mkdir build`
4. `cd build`
5. `cmke .. -DCMAKE_CXX_COMPILER=g++-11`
6. `sudo make install`

Then import with CMake with find_package:

```
find_package(libnop)
```

CI/CD

To make use of code coverage make sure you have a valid runner: https://nop.dainf.ct.utfpr.edu.br/help/ci/runners/README https://docs.gitlab.com/runner/install/

How to use

To use this framework you can just define and initialize the NOP entities as desired.

For more details about the NOP entities you may check the NOP bibligraphy, such as the paper regarding this framework available here.

All the NOP entities are defined as shared pointers, so they can be easily shared in the code.

You must declare the entity and the initialize the pointer using one of the available builders.

The following header must be included: `#include "libnop/framework;h"`.

Attribute

The Attribute has a template parameter that can be any data type. It has a single builder that receives the initial value of the attribute.

```
template <typename T>
using SharedAttribute = std::shared_ptr<Attribute<T>>;
```

Example:

```
NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(1);
```

Premise

The Premise can be built with two Attributes or an Attribute and a value. The third parameter is the Premise operation between it Attributes.

```
template <typename T, typename Func>
auto BuildPremise(std::shared_ptr<Attribute<T>> lhs, T rhs, Func op);

template <typename T, typename Func>
auto BuildPremise(std::shared_ptr<Attribute<T>> lhs,
                  std::shared_ptr<Attribute<T>> rhs, Func op);

/* The operation op can be:
    NOP::Equal(),
    NOP::Different(),
    NOP::Greater(),
    NOP::GreaterEqual(),
    NOP::Less(),
    NOP::LessEqual()
*/
```

Example:

```
NOP::SharedAttribute<int> at1 = NOP::BuildAttribute<int>(-1);
NOP::SharedAttribute<int> at2 = NOP::BuildAttribute<int>(-2);

NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());
NOP::SharedPremise pr2 = NOP::BuildPremise(at1, 0, NOP::Different());
```

Condition

The Condition is built using as arguments one or more Premises, Conditions (as Sub-Bonditions) or Rules (as Master Rules).

The simple condition builder with the condition type as a template parameter:

```
template <EConditionType type, typename... Args>
auto BuildCondition(Args... args);

/* The condition type can be:
    NOP::Single,
    NOP::Conjunction,
    NOP::Disjunction
*/
```

More complex conditions can be composed as a boolean expression, and passing the pointers as arguments. The operation can be passed as a lambda expression, where the CONDITION macro can be used to facilitate the definition.

```
template <typename... Args>
auto BuildCondition(std::function<bool(void)> operation, Args... args);

#define CONDITION(expression) [=, this]() { return bool(expression); }
```

Example:

```
NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Conjunction>(pr1,
pr2);
NOP::SharedCondition cn2 =
        NOP::BuildCondition(CONDITION((*pr1 || *pr2) && !*pr3), pr1, pr2,
pr3);
```

Method

The Method is stored as a std::function with no parameters. It can be initialized with a lambda expression. The macro METHOD can be used to facilitate the definition.

```
using Method = std::function<void(void)>;
#define METHOD(expression) [&]() { expression }
```

Example:

```
int counter{0};
NOP::Method mt1{ METHOD(counter++;) };
```

Instigation

The Instigation is built with one or more Methods as parameters. A template parameter can be used to define a parallel Instigation (all Methods are executed in parallel).

```
template <NOPFlag Flag = Default, typename... Args>
auto BuildInstigation(Args... methods)
```

Example:

```
NOP::SharedInstigation in1 = NOP::BuildInstigation(mt1, mt2);
NOP::SharedInstigation in2 = NOP::BuildInstigation<NOP::Parallel>(mt3,
mt4);
```

Action

The Action is built with one or more Instigations as parameters. A template parameter can be used to define a parallel Action (all Instigations are instigated in parallel).

```
template <NOPFlag Flag = Default, typename... Args>
auto BuildAction(Args... instigation);
```

Example:

```
NOP::SharedAction ac1 = NOP::BuildAction(in1, in2);
NOP::SharedAction ac2 = NOP::BuildAction<NOP::Parallel>(in3, in4);
```

Rule

The Rule is build using an Action and a Condition. When using the Scheduler you can also define a priority value (bigger values has higher priority).

```
std::shared_ptr<Rule> BuildRule(std::shared_ptr<Condition> condition,
                                std::shared_ptr<Action> action,
                                const int priority = 0)
```

Example:

```
NOP::SharedRule rl1 = NOP::BuildRule(cn1, ac1);
NOP::SharedRule rl2 = NOP::BuildRule(cn1, ac1, 10);
```

FBE

The FBE class can be inherited by any classes containing Attributes or Methods. It is not required, but facilitates organizing the NOP entities.

```
class Example: public NOP::FBE
{
    NOP::SharedAttribute<int> at1;
};
```

Scheduler

The Scheduler can be used to ensure determinism when executing the Rules. It executes Rules one by one, while rechecking the condition for each one. Beware, performance is heavily

impacted when using the Scheduler.

To use the Scheduler you must use the compile definition -DLIBNOP_SCHEDULER_ENABLE-=ON or define `#define LIBNOP_SCHEDULER_ENABLE` before including `#include "libnop/framework;h"`.

The Scheduler execution strategy can be set at any point. The ignore_conflict flag may be used if conflict resolution is not desired.

```
void SetStrategy(EStrategy strategy, bool ignore_conflict = false);

/* Valid strategy types are:
NOP::None,
NOP::FIFO,
NOP::LIFO,
NOP::Priority
*/
```

Example:

```
NOP::Scheduler::Instance().SetStrategy(NOP::FIFO);
```

Logger

The framework provides debuggin facilities by means of a Logger.

The Logger can be used to log the NOP execution flow. Beware, performance is heavily impacted when using the Logger.

To use the Scheduler you must use the compile definition -DLIBNOP_LOGGER_ENABLE-=ON or define `#define LIBNOP_LOGGER_ENABLE` before including `#include "libnop/framework;h"`.

The log file location can be set with:

```
void Logger::SetLogFileName(const std::string& file_name);
```

All the NOP entities proved auxiliary builder that can name the entities for logging purposes. Those builders are prefixed Named, and have an additional name parameter. The FBE constructor also accepts a name parameter. When those entities are members of an FBE they may also pass the FBE pointer as a parameter.

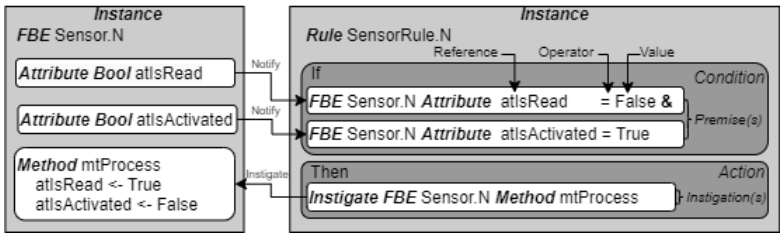
Example:

```
NOP::Logger::Instance().SetLogFileName("test.log");
```

```
struct Test : NOP::FBE
{
    explicit Test(const std::string_view name) : FBE(name) {}
    NOP::SharedAttribute<int> at1 =
        NOP::BuildAttributeNamed("at1", this, -1);
    NOP::SharedAttribute<int> at2 =
        NOP::BuildAttributeNamed("at2", this, -2);
    NOP::SharedPremise pr1 =
        NOP::BuildPremiseNamed("pr1", this, at1, at2, NOP::Equal());
    NOP::SharedCondition cn1 =
        NOP::BuildConditionNamed<NOP::Single:("cn1", this, pr1);

    std::atomic<int> executionCounter{0};
    NOP::Method mt = [&] { executionCounter++; };
    NOP::SharedInstigation in1 =
        NOP::BuildInstigationNamed("in1", this, mt);
    NOP::SharedAction ac1 = NOP::BuildActionNamed("ac1", this, in1);

    NOP::SharedRule rl1 = NOP::BuildRuleNamed("rl1", this, cn1, ac1);
};

Test test{"TestFBE"};
```

FBE example

The following FBE and Rule can be implemented as an example.



As described using NOPL:

```
fbe Sensor
  public boolean atIsRead = false
  public boolean atIsActivated = false
  private method mtProcess
    attribution
      this.atIsRead = true
      this.atIsActivated = false
    end_attribution
  end_method
  rule rlSensor
    condition
        premise prIsActivated
          this.atIsActivated == true
```

```
        end_premise
        and
        premise prIsNotRead
          this.atIsRead == false
        end_premise
    end_condition
    action sequential
        instigation sequential
          call this.mtProcess()
        end_instigation
    end_action
  end_rule
end_fbe
```

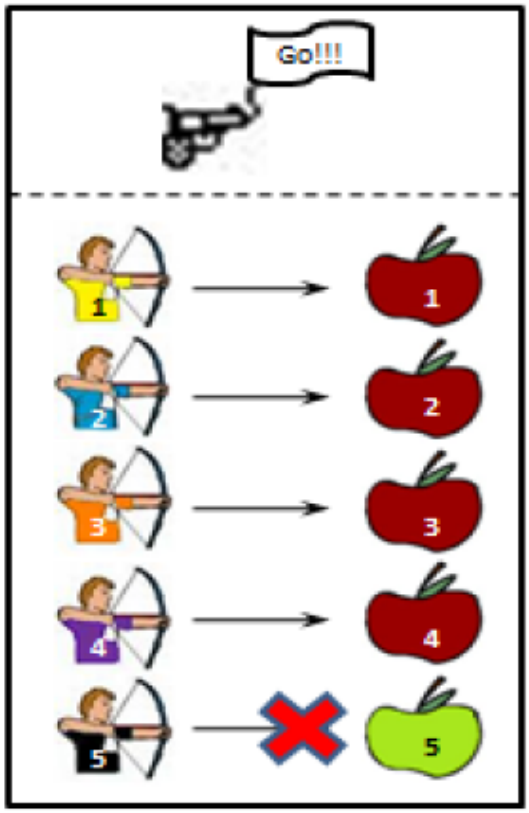
And defined using the framework:

```
struct NOPSensor: NOP::FBE
{
    NOP::SharedAttribute<bool> atIsRead{NOP::BuildAttribute(false)};
    NOP::SharedAttribute<bool> atIsActivated{NOP::BuildAttribute(false)};
    NOP::Method mtProcess{METHOD(atIsRead->SetValue(true); atIsActivated-
>SetValue(false);)};

    NOP::SharedPremise prIsActivated{
        NOP::BuildPremise(atIsActivated, true, NOP::Equal())};
    NOP::SharedPremise prIsNotRead{
        NOP::BuildPremise(atIsRead, false, NOP::Equal())};

    NOP::SharedRule rlSensor{NOP::BuildRule(
        NOP::BuildCondition<NOP::Conjunction>(
            NOP::BuildPremise(atIsActivated, true, NOP::Equal()),
            NOP::BuildPremise(atIsRead, false, NOP::Equal())
            ),
        NOP::BuildAction(
            NOP::BuildInstigation(mtProcess)
            )
        )
    };
};
```

APÊNDICE F

A aplicação mira ao alvo tem como objetivo ser uma aplicação que realce os problemas de redundâncias estruturais e temporais, permitindo assim comparações de desempenho do PON com os demais paradigmas. Neste cenário, ilustrado na Figura 34, um conjunto de arqueiros deve atirar em um conjunto de maçãs, de acordo com o estado das maçãs e da arma de fogo que tem a função de sinalizar o disparo de início desta interação.

Figura 34 – Aplicação mira ao alvo



Fonte: Banaszewski (2009)

O Código 11 apresenta o desenvolvimento deste cenário com o *Framework* PON C++ 4.0. A maçã, arqueiro e arma de fogo são representados, respectivamente, por *NOP4Apple*, *NOP4Archer* e *NOP4Gun*, enquanto a *Rule* é impplementada em *NOP4GUn*.

Código 11 – Código da estrutura do mira ao alvo com o *Framework* PON C++ 4.0

```
struct NOP4Apple : NOP::FBE {
   NOP::SharedAttribute<std::string> atColor =
      NOP::BuildAttribute<std::string>("GREEN");
   NOP::SharedAttribute<bool> atStatus = NOP::BuildAttribute(true);
   NOP::SharedAttribute<bool> atIsCrossed = NOP::BuildAttribute(false);
};

struct NOP4Archer : NOP::FBE {
```

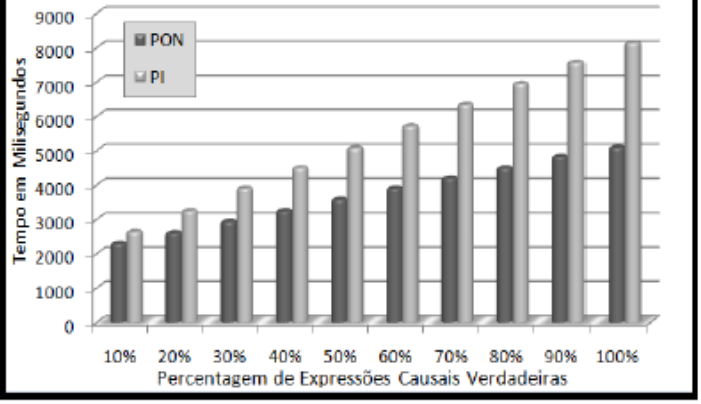
```
   NOP::SharedAttribute<bool> atStatus = NOP::BuildAttribute(true);
};

struct NOP4Gun : NOP::FBE {
   NOP::SharedAttribute<bool> atStatus = NOP::BuildAttribute(false);
};


struct NOP4ArcherRule {
   NOP::SharedRule rule;
   NOP4ArcherRule(std::shared_ptr<NOP4Apple>& apple,
                  std::shared_ptr<NOP4Archer>& archer,
                  NOP::SharedPremise& prGunIsTrue) {
      rule = NOP::BuildRule(
         NOP::BuildCondition<NOP::Conjunction>(
            NOP::BuildPremise<std::string>(apple->atColor, "RED",
                                           NOP::Equal()),
            NOP::BuildPremise(apple->atStatus, true, NOP::Equal()),
            NOP::BuildPremise(archer->atStatus, true, NOP::Equal()),
            prGunIsTrue),
         NOP::BuildAction(NOP::BuildInstigation(
            [=]() { apple->atIsCrossed->SetValue(true); })));
   }
};
```

Fonte: Autoria própria

Durante os experimentos realizados por Banaszewski (2009), a aplicação desenvolvida com o *Framework* PON C++ 1.0 apresentou desempenho um pouco inferior à mesma aplicação desenvolvida com o POO, conforme apresentado na Figura 35. Porém, nos novos testes realizados com o *Framework* PON C++ 4.0, a aplicação em PON não conseguiu chegar perto dos mesmos resultados, conforme apresentado na Figura 36.

Figura 35 – Resultado do experimento mira ao alvo com o *Framework PON C++ 1.0*

Exp. Verdad.	PON (ms)	PI (ms)
10%	2297	2641
20%	2609	3250
30%	2937	3906
40%	3250	4500
50%	3578	5079
60%	3907	5719
70%	4203	6344
80%	4500	6953
90%	4828	7563
100%	5094	8141



Fonte: Banaszewski (2009)

A dificuldade em se reproduzir os mesmos resultados pode ser justificada pela dificuldade em se determinar as condições originais do teste, visto que a versão do compilador e opções de otimização podem influenciar de forma significa o desempenho da aplicação. Além disso, processadores com arquiteturas mais modernas, com os utilizados nos novos testes[8], apresentam

[8] Teste executado em um computador com processor Ryzen 5 3600 a 3.6GHz e 16 GB de RAM DDR4 a 3000MHz (dual channel) e sistema operacional Windows 10 64 bits.

recursos que otimizam a execução de operações condicionais e laçoes de repetição, de modo que são capazes de compensar os efeitos das redundâncias temporais e estruturais.

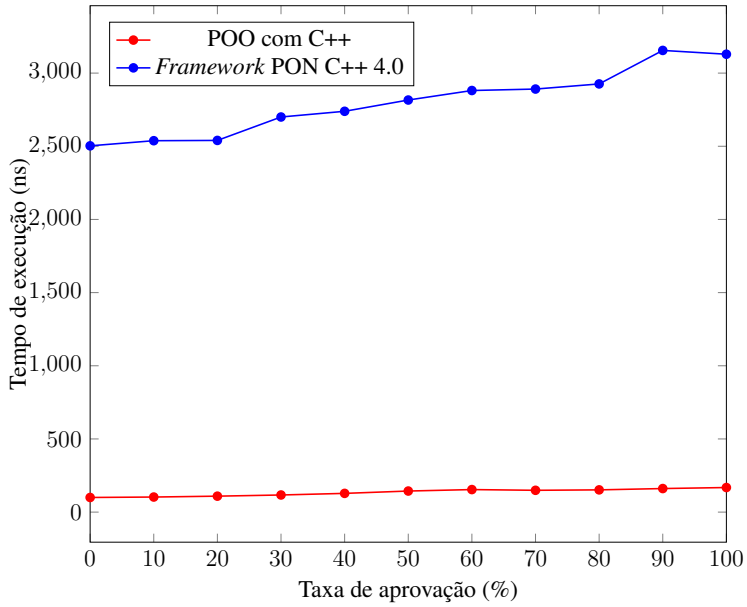


Figura 36 – Resultado do experimento mira ao alvo com o *Framework PON C++ 4.0*
Fonte: Autoria própria

APÊNDICE G

CÓDIGO-FONTE DA APLICAÇÃO BITONIC SORT EM PON NO *FRAMEWORK* PON C++ 4.0

```
template <typename T = int>
struct NOPBitonicSorter
{
   std::map<int, std::vector<NOP::SharedAttribute<T>>> elements;
   std::vector<NOP::SharedPremise> premises;
   std::vector<NOP::SharedRule> rules;
   int stages{0};
   std::vector<T> out;
   size_t size;
   std::vector<int> indexes;

   NOPBitonicSorter(size_t len) : size{len}
   {
      out.resize(size);
      indexes.resize(len);
      std::iota(indexes.begin(), indexes.end(), 0);

      Init();
   }

   auto Sort(const std::vector<T>& input)
   {
      for (auto i = 0; i < input.size(); i++)
      {
         elements[0][i]->SetValue(input[i], (i % 2));
      }
      return out;
   }

   auto ParallelSort(const std::vector<T>& input)
   {
      std::for_each(
         std::execution::par_unseq, indexes.begin(), indexes.end(),
         [&](int i) {
            elements[0][i]->template SetValue<NOP::ReNotify>(input[i]);
         });
      return out;
   }

   void MoveAttributes(NOP::SharedAttribute<T>& at1,
                  NOP::SharedAttribute<T>& at2, int stage, int x, int y)
   {
      if (stage < stages - 1)
      {
         elements[stage + 1][x]->template SetValue<NOP::ReNotify>(
            at1->GetValue());
         elements[stage + 1][y]->template SetValue<NOP::ReNotify>(
            at2->GetValue());
      }
      else
      {
         out[x] = at1->GetValue();
         out[y] = at2->GetValue();
      }
   }

   void CreateComparatorFBE(int x, int y, int stage, bool reverse = false)
   {
      auto& at1 = elements[stage][x];
      auto& at2 = elements[stage][y];
      NOP::SharedPremise pr1;
      NOP::SharedPremise pr2;
      if (!reverse)
      {
         pr1 = NOP::BuildPremise(at2, at1, NOP::Greater());
         pr2 = NOP::BuildPremise(at2, at1, NOP::LessEqual());
```

```
    }
    else
    {
       pr2 = NOP::BuildPremise(at2, at1, NOP::Greater());
       pr1 = NOP::BuildPremise(at2, at1, NOP::LessEqual());
    }
    premises.push_back(pr1);
    premises.push_back(pr2);

    // Rule to move
    rules.push_back(NOP::BuildRule(
       NOP::BuildCondition(CONDITION(*pr1), pr1),
       NOP::BuildAction(NOP::BuildInstigation([&, stage, x, y]() {
          MoveAttributes(at1, at2, stage, x, y);
       }))));
    // Rule to swap
    rules.push_back(NOP::BuildRule(
       NOP::BuildCondition(CONDITION(*pr2), pr2),
       NOP::BuildAction(NOP::BuildInstigation([&, stage, x, y]() {
          MoveAttributes(at1, at2, stage, y, x);
       }))));
}

void Init()
{
    elements.clear();
    premises.clear();
    rules.clear();

    const size_t half = size / 2;
    const size_t quarter = size / 4;

    int n = 1;
    for (int i = 1; i < log2(size); i++)
    {
       stages += n++;
    }
    for (int s = 0; s < stages; s++)
    {
       elements[s].resize(size);
       for (int i = 0; i < size; i++)
       {
          elements[s][i] = NOP::BuildAttribute(T());
       }
    }

    int stage = 0;
    for (auto div = half; div > 1; div = div / 2)
    {
       bool reverse{false};
       int internal_stage = stage;
       for (auto start = 0; start < size; start += size / div)
       {
          internal_stage = stage;
          for (auto step = half / div; step > 0; step = step / 2)
          {
             for (auto i = 0; i < step; i++)
             {
                for (auto j = 0; j < size / div; j = j + 2 * step)
                {
                   int x = start + i + j;
                   int y = start + i + j + step;
                   CreateComparatorFBE(x, y, internal_stage, reverse);
                }
             }
             internal_stage++;
          }
          reverse = !reverse;
       }
       stage += (internal_stage - stage);
    }

    int sort_stages = log2(size);
    int curr_stage = stages;
    for (int s = 0; s < sort_stages; s++)
    {
       stages++;
       elements[curr_stage + s].resize(size);
       for (int i = 0; i < size; i++)
       {
          elements[curr_stage + s][i] = NOP::BuildAttribute(T());
```

```
         }
      }

      stage = curr_stage;
      for (auto start = 0; start < size; start += size)
      {
         for (auto step = half; step > 0; step = step / 2)
         {
            for (auto i = 0; i < step; i++)
            {
               for (auto j = 0; j < size; j = j + 2 * step)
               {
                  auto x = start + i + j;
                  auto y = start + i + j + step;
                  CreateComparatorFBE(x, y, stage);
               }
            }
            stage++;
         }
      }
   }
};


template <typename T = int>
struct NOPBitonicSorterStages
{
   std::vector<NOP::SharedAttribute<bool>> stages;
   std::map<int, NOP::SharedPremise> stage_premises;

   std::vector<NOP::SharedAttribute<T>> elements;
   std::vector<NOP::SharedPremise> premises;
   std::vector<NOP::SharedRule> rules;

   std::vector<int> out{};
   size_t size;

   NOPBitonicSorterStages(size_t len) : size{len}
   {
      Init();
      out.resize(len);
   }

   void Input(std::vector<T>& input)
   {
      for (size_t i = 0; i < input.size(); i++)
      {
         elements[i]->SetValue(input[i], NOP::NoNotify);
      }
   }

   auto ParallelSort()
   {
      for (const auto& stage : stages)
      {
         stage->SetValue<NOP::Parallel>(true);
         stage->SetValue<NOP::Parallel>(false);
      }
      for (auto i = 0; i < elements.size(); i++)
      {
         out[i] = elements[i]->GetValue();
      }
      return out;
   }

   auto Sort()
   {
      for (const auto& stage : stages)
      {
         stage->SetValue(true);
         stage->SetValue(false);
      }
      for (auto i = 0; i < elements.size(); i++)
      {
         out[i] = elements[i]->GetValue();
      }
      return out;
   }

   void Init()
   {
```

```
elements.clear();
premises.clear();
rules.clear();

for (auto i = 0; i < size; i++)
{
   elements.push_back(NOP::BuildAttribute(T()));
}
const size_t half = size / 2;
const size_t quarter = size / 4;

int n = 1;
int total_stages = 0;
for (int i = 1; i < log2(size); i++)
{
   total_stages += n++;
}

for (int s = 0; s < total_stages; s++)
{
   NOP::SharedAttribute<bool> atStage{NOP::BuildAttribute(false)};
   stages.push_back(atStage);
   NOP::SharedPremise prStage =
      NOP::BuildPremise(atStage, true, NOP::Equal());
   stage_premises[s] = prStage;
}

int stage = 0;
for (auto div = half; div > 1; div = div / 2)
{
   bool reverse{false};
   int internal_stage = stage;
   for (auto start = 0; start < size; start += size / div)
   {
      internal_stage = stage;
      for (auto step = half / div; step > 0; step = step / 2)
      {
         for (auto i = 0; i < step; i++)
         {
            for (auto j = 0; j < size / div; j = j + 2 * step)
            {
               auto x = start + i + j;
               auto y = start + i + j + step;
               auto& at1 = elements[x];
               auto& at2 = elements[y];
               NOP::SharedPremise pr;
               if (reverse)
               {
                  pr = NOP::BuildPremise(at1, at2, NOP::Less());
               }
               else
               {
                  pr =
                     NOP::BuildPremise(at1, at2, NOP::Greater());
               }
               premises.push_back(pr);

               rules.push_back(NOP::BuildRule(
                  NOP::BuildCondition<NOP::Conjunction>(
                     pr, stage_premises[internal_stage]),
                  NOP::BuildAction(NOP::BuildInstigation(
                     METHOD(SwapAttributes(at1, at2);)))));
            }
         }
         internal_stage++;
      }
      reverse = !reverse;
   }
   stage += (internal_stage - stage);
}

int sort_stages = log2(size);
int curr_stage = total_stages;
for (int s = 0; s < sort_stages; s++)
{
   NOP::SharedAttribute<bool> atStage{NOP::BuildAttribute(false)};
   stages.push_back(atStage);
   NOP::SharedPremise prStage =
      NOP::BuildPremise(atStage, true, NOP::Equal());
   stage_premises[curr_stage + s] = prStage;
}
```

```
      stage = curr_stage;
      for (auto start = 0; start < size; start += size)
      {
         for (auto step = half; step > 0; step = step / 2)
         {
            for (auto i = 0; i < step; i++)
            {
               for (auto j = 0; j < size; j = j + 2 * step)
               {
                  auto x = start + i + j;
                  auto y = start + i + j + step;
                  auto& at1 = elements[x];
                  auto& at2 = elements[y];
                  NOP::SharedPremise pr =
                     NOP::BuildPremise(at1, at2, NOP::Greater());
                  premises.push_back(pr);
                  rules.push_back(NOP::BuildRule(
                     NOP::BuildCondition<NOP::Conjunction>(
                        pr, stage_premises[stage]),
                     NOP::BuildAction(NOP::BuildInstigation(
                        METHOD(SwapAttributes(at1, at2);))));
               }
            }
            stage++;
         }
      }
   }
};
```

APÊNDICE H

CÓDIGO-FONTE DA APLICAÇÃO BITONIC SORT NO PP EM LINGUAGEM DE PROGRAMAÇÃO C

```
/*
 bitonic.c

 This file contains two different implementations of the bitonic sort
      recursive version
      imperative version : impBitonicSort()


 The bitonic sort is also known as Batcher Sort.
 For a reference of the algorithm, see the article titled
 Sorting networks and their applications by K. E. Batcher in 1968


 The following codes take references to the codes avaiable at

 http://www.cag.lcs.mit.edu/streamit/results/bitonic/code/c/bitonic.c

 http://www.tools-of-computing.com/tc/CS/Sorts/bitonic_sort.htm

 http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonicen.htm
 */

/*
------- ----------------------
  Nikos Pitsianis, Duke CS
-----------------------------
*/


#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

struct timeval startwtime, endwtime;
double seq_time;


int N; // data array size
int *a; // data array to be sorted

const int ASCENDING = 1;
const int DESCENDING = 0;


void init(void);
void print(void);
void sort(void);
void test(void);
inline void exchange(int i, int j);
void compare(int i, int j, int dir);
void bitonicMerge(int lo, int cnt, int dir);
void recBitonicSort(int lo, int cnt, int dir);
void impBitonicSort(void);


/** the main program **/
int main(int argc, char **argv) {

  if (argc != 2) {
    printf("Usage: %s n\n where n is problem size (power of two)\n", argv[0]);
    exit(1);
  }

  N = atoi(argv[1]);
  a = (int *) malloc(N * sizeof(int));

  init();
```

```
  gettimeofday (&startwtime, NULL);
  impBitonicSort();
  gettimeofday (&endwtime, NULL);

  seq_time = (double)((endwtime.tv_usec - startwtime.tv_usec)/1.0e6
                + endwtime.tv_sec - startwtime.tv_sec);

  printf("Imperative wall clock time = %f\n", seq_time);

  test();

  init();
  gettimeofday (&startwtime, NULL);
  sort();
  gettimeofday (&endwtime, NULL);

  seq_time = (double)((endwtime.tv_usec - startwtime.tv_usec)/1.0e6
                + endwtime.tv_sec - startwtime.tv_sec);

  printf("Recursive wall clock time = %f\n", seq_time);

  test();

  // print();
}

/** -------------- SUB-PROCEDURES ----------------- **/

/** procedure test() : verify sort results **/
void test() {
  int pass = 1;
  int i;
  for (i = 1; i < N; i++) {
    pass &= (a[i-1] <= a[i]);
  }

  printf(" TEST %s\n",(pass) ? "PASSed" : "FAILed");
}


/** procedure init() : initialize array "a" with data **/
void init() {
  int i;
  for (i = 0; i < N; i++) {
    // a[i] = rand() % N; // (N - i);
    a[i] = (N - i);
  }
}

/** procedure print() : print array elements **/
void print() {
  int i;
  for (i = 0; i < N; i++) {
    printf("%d\n", a[i]);
  }
  printf("\n");
}


/** INLINE procedure exchange() : pair swap **/
inline void exchange(int i, int j) {
  int t;
  t = a[i];
  a[i] = a[j];
  a[j] = t;
}



/** procedure compare()
   The parameter dir indicates the sorting direction, ASCENDING
   or DESCENDING; if (a[i] > a[j]) agrees with the direction,
   then a[i] and a[j] are interchanged.
**/
void compare(int i, int j, int dir) {
  if (dir==(a[i]>a[j]))
    exchange(i,j);
}
```

```
/** Procedure bitonicMerge()
   It recursively sorts a bitonic sequence in ascending order,
   if dir = ASCENDING, and in descending order otherwise.
   The sequence to be sorted starts at index position lo,
   the parameter cbt is the number of elements to be sorted.
 **/
void bitonicMerge(int lo, int cnt, int dir) {
  if (cnt>1) {
    int k=cnt/2;
    int i;
    for (i=lo; i<lo+k; i++)
      compare(i, i+k, dir);
    bitonicMerge(lo, k, dir);
    bitonicMerge(lo+k, k, dir);
  }
}



/** function recBitonicSort()
    first produces a bitonic sequence by recursively sorting
    its two halves in opposite sorting orders, and then
    calls bitonicMerge to make them in the same order
 **/
void recBitonicSort(int lo, int cnt, int dir) {
  if (cnt>1) {
    int k=cnt/2;
    recBitonicSort(lo, k, ASCENDING);
    recBitonicSort(lo+k, k, DESCENDING);
    bitonicMerge(lo, cnt, dir);
  }
}


/** function sort()
   Caller of recBitonicSort for sorting the entire array of length N
   in ASCENDING order
 **/
void sort() {
  recBitonicSort(0, N, ASCENDING);
}



/*
  imperative version of bitonic sort
*/
void impBitonicSort() {

  int i,j,k;

  for (k=2; k<=N; k=2*k) {
    for (j=k>>1; j>0; j=j>>1) {
      for (i=0; i<N; i++) {
        int ij=i^j;
        if ((ij)>i) {
          if ((i&k)==0 && a[i] > a[ij])
             exchange(i,ij);
          if ((i&k)!=0 && a[i] < a[ij])
             exchange(i,ij);
        }
      }
    }
  }
}
```

APÊNDICE I

CÓDIGO-FONTE DA APLICAÇÃO RANDOM FOREST EM PON NO *FRAMEWORK* PON C++ 4.0

```
#pragma once
#include "libnop/framework.h"

std::vector<std::vector<int>> test_data{
   {580, 280, 509, 240}, {600, 220, 400, 100}, {550, 420, 140, 20},
   {730, 290, 630, 180}, {500, 340, 150, 20}, {630, 330, 600, 250},
   {500, 350, 130, 30}, {670, 310, 470, 150}, {680, 280, 480, 140},
   {610, 280, 400, 130}, {610, 260, 560, 140}, {640, 320, 450, 150},
   {610, 280, 470, 120}, {650, 280, 459, 150}, {610, 290, 470, 140},
   {490, 360, 140, 10}, {600, 290, 450, 150}, {550, 260, 440, 120},
   {480, 300, 140, 30}, {540, 390, 130, 40}, {560, 280, 490, 200},
   {560, 300, 450, 150}, {480, 340, 190, 20}, {440, 290, 140, 20},
   {620, 280, 480, 180}, {459, 360, 100, 20}, {509, 380, 190, 40},
   {620, 290, 430, 130}, {500, 229, 330, 100}, {500, 340, 160, 40}};
std::vector<int> result_data{2, 1, 0, 2, 0, 2, 0, 1, 1, 1, 2, 1, 1, 1, 1,
                     0, 1, 1, 0, 0, 2, 1, 0, 0, 2, 0, 0, 1, 1, 0};

struct Forest
{
   int result_class{-1};

   NOP::SharedAttribute<int> attr0 = NOP::BuildAttribute<int>(0);
   NOP::SharedAttribute<int> attr1 = NOP::BuildAttribute<int>(0);
   NOP::SharedAttribute<int> attr2 = NOP::BuildAttribute<int>(0);
   NOP::SharedAttribute<int> attr3 = NOP::BuildAttribute<int>(0);
   NOP::SharedAttribute<int> count_setosa = NOP::BuildAttribute<int>(0);
   NOP::SharedAttribute<int> count_versicolor = NOP::BuildAttribute<int>(0);
   NOP::SharedAttribute<int> count_virginica = NOP::BuildAttribute<int>(0);
   NOP::SharedAttribute<bool> trigger_tree_0 =
      NOP::BuildAttribute<bool>(false);
   NOP::SharedAttribute<bool> trigger_tree_1 =
      NOP::BuildAttribute<bool>(false);
   NOP::Method mt_count_setosa = [&]() {
      count_setosa->SetValue(count_setosa->GetValue() + 1);
   };
   NOP::Method mt_rst_count_setosa = [&]() { count_setosa->SetValue(0); };
   NOP::Method mt_count_versicolor = [&]() {
      count_versicolor->SetValue(count_versicolor->GetValue() + 1);
   };
   NOP::Method mt_rst_count_versicolor = [&]() {
      count_versicolor->SetValue(0);
   };
   NOP::Method mt_count_virginica = [&]() {
      count_virginica->SetValue(count_virginica->GetValue() + 1);
   };
   NOP::Method mt_rst_count_virginica = [&]() {
      count_virginica->SetValue(0);
   };
   NOP::Method mtTrigger0 = [&]() { trigger_tree_0->SetValue(true); };
   NOP::Method mtRstTrigger0 = [&]() { trigger_tree_0->SetValue(false); };
   NOP::Method mtTrigger1 = [&]() { trigger_tree_1->SetValue(true); };
   NOP::Method mtRstTrigger1 = [&]() { trigger_tree_1->SetValue(false); };

   NOP::SharedRule rlTree_0_1 = NOP::BuildRule(
      NOP::BuildCondition<NOP::Conjunction>(
         NOP::BuildPremise(attr3, 75, NOP::LessEqual()),
         NOP::BuildPremise(trigger_tree_0, true, NOP::Equal())),
      NOP::BuildAction(
         NOP::BuildInstigation(mt_count_setosa, mtTrigger1, mtRstTrigger0)));

   NOP::SharedRule rlTree_0_5 = NOP::BuildRule(
      NOP::BuildCondition<NOP::Conjunction>(
         NOP::BuildPremise(attr3, 75, NOP::Greater()),
         NOP::BuildPremise(attr2, 495, NOP::LessEqual()),
         NOP::BuildPremise(attr3, 165, NOP::LessEqual()),
         NOP::BuildPremise(trigger_tree_0, true, NOP::Equal())),
      NOP::BuildAction(NOP::BuildInstigation(mt_count_versicolor, mtTrigger1,
```

```
                                    mtRstTrigger0)));

NOP::SharedRule rlTree_0_7 = NOP::BuildRule(
   NOP::BuildCondition<NOP::Conjunction>(
      NOP::BuildPremise(attr3, 75, NOP::Greater()),
      NOP::BuildPremise(attr2, 495, NOP::LessEqual()),
      NOP::BuildPremise(attr3, 165, NOP::Greater()),
      NOP::BuildPremise(attr1, 310, NOP::LessEqual()),
      NOP::BuildPremise(trigger_tree_0, true, NOP::Equal())),
   NOP::BuildAction(NOP::BuildInstigation(mt_count_virginica, mtTrigger1,
                                    mtRstTrigger0)));

NOP::SharedRule rlTree_0_8 = NOP::BuildRule(
   NOP::BuildCondition<NOP::Conjunction>(
      NOP::BuildPremise(attr3, 75, NOP::Greater()),
      NOP::BuildPremise(attr2, 495, NOP::LessEqual()),
      NOP::BuildPremise(attr3, 165, NOP::Greater()),
      NOP::BuildPremise(attr1, 310, NOP::Greater()),
      NOP::BuildPremise(trigger_tree_0, true, NOP::Equal())),
   NOP::BuildAction(NOP::BuildInstigation(mt_count_versicolor, mtTrigger1,
                                    mtRstTrigger0)));

NOP::SharedRule rlTree_0_11 = NOP::BuildRule(
   NOP::BuildCondition<NOP::Conjunction>(
      NOP::BuildPremise(attr3, 75, NOP::Greater()),
      NOP::BuildPremise(attr2, 495, NOP::Greater()),
      NOP::BuildPremise(attr2, 505, NOP::LessEqual()),
      NOP::BuildPremise(attr3, 185, NOP::LessEqual()),
      NOP::BuildPremise(trigger_tree_0, true, NOP::Equal())),
   NOP::BuildAction(NOP::BuildInstigation(mt_count_versicolor, mtTrigger1,
                                    mtRstTrigger0)));

NOP::SharedRule rlTree_0_12 = NOP::BuildRule(
   NOP::BuildCondition<NOP::Conjunction>(
      NOP::BuildPremise(attr3, 75, NOP::Greater()),
      NOP::BuildPremise(attr2, 495, NOP::Greater()),
      NOP::BuildPremise(attr2, 505, NOP::LessEqual()),
      NOP::BuildPremise(attr3, 185, NOP::Greater()),
      NOP::BuildPremise(trigger_tree_0, true, NOP::Equal())),
   NOP::BuildAction(NOP::BuildInstigation(mt_count_virginica, mtTrigger1,
                                    mtRstTrigger0)));

NOP::SharedRule rlTree_0_15 = NOP::BuildRule(
   NOP::BuildCondition<NOP::Conjunction>(
      NOP::BuildPremise(attr3, 75, NOP::Greater()),
      NOP::BuildPremise(attr2, 495, NOP::Greater()),
      NOP::BuildPremise(attr2, 505, NOP::Greater()),
      NOP::BuildPremise(attr1, 275, NOP::LessEqual()),
      NOP::BuildPremise(attr3, 175, NOP::LessEqual()),
      NOP::BuildPremise(trigger_tree_0, true, NOP::Equal())),
   NOP::BuildAction(NOP::BuildInstigation(mt_count_versicolor, mtTrigger1,
                                    mtRstTrigger0)));

NOP::SharedRule rlTree_0_16 = NOP::BuildRule(
   NOP::BuildCondition<NOP::Conjunction>(
      NOP::BuildPremise(attr3, 75, NOP::Greater()),
      NOP::BuildPremise(attr2, 495, NOP::Greater()),
      NOP::BuildPremise(attr2, 505, NOP::Greater()),
      NOP::BuildPremise(attr1, 275, NOP::LessEqual()),
      NOP::BuildPremise(attr3, 175, NOP::Greater()),
      NOP::BuildPremise(trigger_tree_0, true, NOP::Equal())),
   NOP::BuildAction(NOP::BuildInstigation(mt_count_virginica, mtTrigger1,
                                    mtRstTrigger0)));

NOP::SharedRule rlTree_0_14 = NOP::BuildRule(
   NOP::BuildCondition<NOP::Conjunction>(
      NOP::BuildPremise(attr3, 75, NOP::Greater()),
      NOP::BuildPremise(attr2, 495, NOP::Greater()),
      NOP::BuildPremise(attr2, 505, NOP::Greater()),
      NOP::BuildPremise(attr1, 275, NOP::Greater()),
      NOP::BuildPremise(trigger_tree_0, true, NOP::Equal())),
   NOP::BuildAction(NOP::BuildInstigation(mt_count_virginica, mtTrigger1,
                                    mtRstTrigger0)));

NOP::SharedRule rlEnd = NOP::BuildRule(
   NOP::BuildCondition<NOP::Single>(
      NOP::BuildPremise(trigger_tree_1, true, NOP::Equal())),
   NOP::BuildAction(NOP::BuildInstigation(
      [&]() {
         result_class =
            count_setosa->GetValue() < count_versicolor->GetValue() ? 1
```

```
                                                                : 0;
            result_class =
               count_setosa->GetValue() < count_virginica->GetValue() &&
                     count_versicolor->GetValue() <
                        count_virginica->GetValue()
                  ? 2
                  : result_class;
         },
         mt_rst_count_setosa, mt_rst_count_versicolor,
         mt_rst_count_virginica, mtRstTrigger1)));

   int Classify(int a0, int a1, int a2, int a3)
   {
      attr0->SetValue(a0);
      attr1->SetValue(a1);
      attr2->SetValue(a2);
      attr3->SetValue(a3);
      trigger_tree_0->SetValue(true);
      return result_class;
   }
};
```

CÓDIGO-FONTE DA APLICAÇÃO RANDOM FOREST EM PP NA LINGUAGEM DE PROGRAMAÇÃO C

```
int randomForestClassifier(int attr[4]);
static int sumOfResult[3];
static int result[1][3];
static void tree0(int attr[4]);
static int voting();
void tree0(int attr[4])
{
   if (attr[3] <= 75)
   {
      result[0][0] += 100;
   }
   else
   {
      if (attr[2] <= 495)
      {
         if (attr[3] <= 165)
         {
            result[0][1] += 100;
         }
         else
         {
            if (attr[1] <= 310)
            {
               result[0][2] += 100;
            }
            else
            {
               result[0][1] += 100;
            }
         }
      }
      else
      {
         if (attr[2] <= 505)
         {
            if (attr[3] <= 185)
            {
               result[0][1] += 100;
            }
            else
            {
               result[0][2] += 100;
            }
         }
         else
         {
            if (attr[1] <= 275)
```

```
            {
               if (attr[3] <= 175)
               {
                  result[0][1] += 100;
               }
               else
               {
                  result[0][2] += 100;
               }
            }
            else
            {
               result[0][2] += 100;
            }
         }
      }
   }
}
int voting()
{
   for (int t = 0; t < 1; t++)
   {
      for (int c = 0; c < 3; c++)
      {
         sumOfResult[c] += result[t][c];
      }
   }
   int maxIndex, maxValue = 0;
   for (int i = 0; i < 3; i++)
   {
      if (sumOfResult[i] >= maxValue)
      {
         maxValue = sumOfResult[i];
         maxIndex = i;
      }
   }
   return maxIndex;
}
void rst()
{
   for (int i = 0; i < 3; i++)
   {
      sumOfResult[i] = 0;
   }
   for (int t = 0; t < 1; t++)
   {
      for (int c = 0; c < 3; c++)
      {
         result[t][c] = 0;
      }
   }
}
int randomForestClassifier(int attr[4])
{
   rst();
   tree0(attr);
   return voting();
}
```

APÊNDICE J

CÓDIGO-FONTE DA APLICAÇÃO CTA EM LINGPON

```
fbe Semaphore_CTA

   private Integer atSemaphoreState = 5
   public Integer atSeconds = 0

   private method mtResetTimer
      assignment
         this.atSeconds = 0
      end_assignment
   end_method

   private method mtHorizontalTrafficLightGREEN
      assignment
         this.atSemaphoreState = 0
      end_assignment
   end_method

   private method mtHorizontalTrafficLightYELLOW
      assignment
         this.atSemaphoreState = 1
      end_assignment
   end_method

   private method mtHorizontalTrafficLightRED
      assignment
         this.atSemaphoreState = 2
      end_assignment
   end_method

   private method mtVerticalTrafficLightGREEN
      assignment
         this.atSemaphoreState = 3
      end_assignment
   end_method

   private method mtVerticalTrafficLightYELLOW
      assignment
         this.atSemaphoreState = 4
      end_assignment
   end_method

   private method mtVerticalTrafficLightRED
      assignment
         this.atSemaphoreState = 5
      end_assignment
   end_method

   rule rlHorizontalTrafficLightGreen
      condition
            premise prSeconds
               this.atSeconds == 2
            end_premise
            and
            premise prSemaphoreState
               this.atSemaphoreState == 5
            end_premise
      end_condition
      action sequential
         instigation parallel
            call this.mtHorizontalTrafficLightGREEN()
         end_instigation
      end_action
   end_rule

   rule rlHorizontalTrafficLightYellow
      condition
         premise prSeconds2
            this.atSeconds == 40
         end_premise
         and
```

```
         premise prSemaphoreState2
            this.atSemaphoreState == 0
         end_premise
      end_condition
      action sequential
         instigation parallel
            call this.mtHorizontalTrafficLightYELLOW()
         end_instigation
      end_action
   end_rule

   rule rlHorizontalTrafficLightRed
      condition
         premise prAtSeconds3
            this.atSeconds == 45
         end_premise
         and
         premise prSemaphoreState3
            this.atSemaphoreState == 1
         end_premise
      end_condition
      action sequential
         instigation parallel
            call this.mtHorizontalTrafficLightRED()
         end_instigation
      end_action
   end_rule

   rule rlVerticalTrafficLightGreen
      condition
         premise prAtSeconds4
            this.atSeconds == 47
         end_premise
         and
         premise prSemaphoreState4
            this.atSemaphoreState == 2
         end_premise
      end_condition
      action sequential
         instigation parallel
            call this.mtVerticalTrafficLightGREEN()
         end_instigation
      end_action
   end_rule

   rule rlVerticalTrafficLightYellow
      condition
         premise prAtSeconds5
            this.atSeconds == 85
         end_premise
         and
         premise prSemaphoreState5
            this.atSemaphoreState == 3
         end_premise
      end_condition
      action sequential
         instigation parallel
            call this.mtVerticalTrafficLightYELLOW()
         end_instigation
      end_action
   end_rule

   rule rlVerticalTrafficLightRed
      condition
         premise prAtSeconds6
            this.atSeconds == 90
         end_premise
         and
         premise prSemaphoreState6
            this.atSemaphoreState == 4
         end_premise
      end_condition
      action sequential
         instigation parallel
            call this.mtVerticalTrafficLightRED()
            call this.mtResetTimer()
         end_instigation
      end_action
   end_rule

end_fbe
```

```
fbe Semaphore_CBCL

   private Integer atSemaphoreState = 5
   public Integer atSeconds = 0
      private integer atHVSS = 0
      private integer atVVSS = 0

   private method mtRT
      assignment
         this.atSeconds = 0
      end_assignment
   end_method

   private method mtHTLG
      assignment
         this.atSemaphoreState = 0
      end_assignment
   end_method

   private method mtHTLY
      assignment
         this.atSemaphoreState = 1
      end_assignment
   end_method

   private method mtHTLR
      assignment
         this.atSemaphoreState = 2
      end_assignment
   end_method

   private method mtVTLG
      assignment
         this.atSemaphoreState = 3
      end_assignment
   end_method

   private method mtVTLY
      assignment
         this.atSemaphoreState = 4
      end_assignment
   end_method

   private method mtVTLR
      assignment
         this.atSemaphoreState = 5
      end_assignment
   end_method

      private method mtHTLGCBCL
      assignment
         this.atSemaphoreState = 6
      end_assignment
   end_method

   private method mtHTLYCBCL
      assignment
         this.atSemaphoreState = 7
      end_assignment
   end_method

   private method mtVTLGCBCL
      assignment
         this.atSemaphoreState = 8
      end_assignment
   end_method

   private method mtVTLYCBCL
      assignment
         this.atSemaphoreState = 9
      end_assignment
   end_method

   rule rlCBCL1
      condition
            premise prSeconds
               this.atSeconds == 2
            end_premise
            and
            premise prSemaphoreState
```

```
            this.atSemaphoreState == 5
         end_premise
   end_condition
   action sequential
      instigation parallel
         call this.mtHTLG()
                     call this.mtRT()
      end_instigation
   end_action
end_rule

rule rlCBCL2
   condition
      premise prSeconds2
         this.atSeconds == 38
      end_premise
      and
      premise prSemaphoreState2
         this.atSemaphoreState == 0
      end_premise
   end_condition
   action sequential
      instigation parallel
         call this.mtHTLY()
                     call this.mtRT()
      end_instigation
   end_action
end_rule

rule rlCBCL3
   condition
      premise prSecondsCBCL2
         this.atSeconds == 30
      end_premise
      and
      premise prSemaphoreStateCBCL2
         this.atSemaphoreState == 6
      end_premise
   end_condition
   action sequential
      instigation parallel
         call this.mtHTLY()
                     call this.mtRT()
      end_instigation
   end_action
end_rule

rule rlCBCL4
   condition
      premise prSeconds3
         this.atSeconds == 5
      end_premise
      and
      premise prSemaphoreState3
         this.atSemaphoreState == 1
      end_premise
   end_condition
   action sequential
      instigation parallel
         call this.mtHTLR()
                     call this.mtRT()
      end_instigation
   end_action
end_rule

rule rlCBCL5
   condition
      premise prSecondsCBCL3
         this.atSeconds == 6
      end_premise
      and
      premise prSemaphoreStateCBCL3
         this.atSemaphoreState == 7
      end_premise
   end_condition
   action sequential
      instigation parallel
         call this.mtHTLR()
                     call this.mtRT()
      end_instigation
   end_action
```

```
end_rule

rule rlCBCL6
   condition
      premise prSeconds4
         this.atSeconds == 2
      end_premise
      and
      premise prSemaphoreState4
         this.atSemaphoreState == 2
      end_premise
   end_condition
   action sequential
      instigation parallel
         call this.mtVTLG()
         call this.mtRT()
      end_instigation
   end_action
end_rule

   rule rlCBCL7
   condition
         premise prSeconds5
            this.atSeconds == 38
         end_premise
         and
         premise prSemaphoreState5
            this.atSemaphoreState == 3
         end_premise
   end_condition
   action sequential
      instigation parallel
         call this.mtVTLY()
                     call this.mtRT()
      end_instigation
   end_action
end_rule

rule rlCBCL8
   condition
      premise prSecondsCBCL5
         this.atSeconds == 30
      end_premise
      and
      premise prSemaphoreStateCBCL5
         this.atSemaphoreState == 8
      end_premise
   end_condition
   action sequential
      instigation parallel
         call this.mtVTLY()
                     call this.mtRT()
      end_instigation
   end_action
end_rule

rule rlCBCL9
   condition
      premise prSeconds6
         this.atSeconds == 5
      end_premise
      and
      premise prSemaphoreState6
         this.atSemaphoreState == 4
      end_premise
   end_condition
   action sequential
      instigation parallel
         call this.mtVTLR()
                     call this.mtRT()
      end_instigation
   end_action
end_rule

rule rlCBCL10
   condition
      premise prSecondsCBCL6
         this.atSeconds == 6
      end_premise
      and
      premise prSemaphoreStateCBCL6
```

```
            this.atSemaphoreState == 9
         end_premise
      end_condition
      action sequential
         instigation parallel
            call this.mtVTLR()
                        call this.mtRT()
         end_instigation
      end_action
   end_rule

   rule rlCBCL11
      condition
         premise prSeconds7
            this.atSeconds <= 17
         end_premise
         and
         premise prSemaphoreState7
            this.atSemaphoreState == 0
         end_premise
                  and
                  premise prVehicleSensorState7
                        this.atVVSS == 1
                  end_premise
      end_condition
      action sequential
         instigation parallel
            call this.mtHTLGCBCL()
         end_instigation
      end_action
   end_rule

   rule rlCBCL12
      condition
         premise prSeconds7Full
            this.atSeconds <= 17
         end_premise
         and
         premise prSemaphoreState7Full
            this.atSemaphoreState == 0
         end_premise
                  and
                  premise prVehicleSensorState7Full
                        this.atVVSS == 2
                  end_premise
      end_condition
      action sequential
         instigation parallel
            call this.mtHTLGCBCL()
         end_instigation
      end_action
   end_rule

      rule rlCBCL13
      condition
         premise prSeconds8
            this.atSeconds >= 18
         end_premise
         and
                  premise prSecondsSup8
            this.atSeconds < 32
         end_premise
         and
         premise prSemaphoreState8
            this.atSemaphoreState == 0
         end_premise
                  and
                  premise prVehicleSensorState8
                        this.atVVSS == 1
                  end_premise
      end_condition
      action sequential
         instigation parallel
            call this.mtHTLYCBCL()
                        call this.mtRT()
         end_instigation
      end_action
   end_rule

   rule rlCBCL14
      condition
```

```
      premise prSeconds8Full
         this.atSeconds >= 18
      end_premise
      and
               premise prSecondsSup8Full
         this.atSeconds < 32
      end_premise
      and
      premise prSemaphoreState8Full
         this.atSemaphoreState == 0
      end_premise
               and
               premise prVehicleSensorState8Full
                     this.atVVSS == 2
               end_premise
   end_condition
   action sequential
      instigation parallel
         call this.mtHTLYCBCL()
                     call this.mtRT()
      end_instigation
   end_action
end_rule

rule rlCBCL15
   condition
      premise prSeconds9
         this.atSeconds <= 17
      end_premise
      and
      premise prSemaphoreState9
         this.atSemaphoreState == 3
      end_premise
      and
               premise prVehicleSensorState9
         this.atSemaphoreState == 1
      end_premise
   end_condition
   action sequential
      instigation parallel
         call this.mtVTLGCBCL()
      end_instigation
   end_action
end_rule

rule rlCBCL16
   condition
      premise prSeconds9Full
         this.atSeconds <= 77
      end_premise
      and
      premise prSemaphoreState9Full
         this.atSemaphoreState == 3
      end_premise
      and
               premise prVehicleSensorState9Full
                     this.atSemaphoreState == 2
               end_premise
   end_condition
   action sequential
      instigation parallel
         call this.mtVTLGCBCL()
      end_instigation
   end_action
end_rule

rule rlCBCL17
   condition
      premise prSeconds10
         this.atSeconds >= 18
      end_premise
      and
      premise prSecondsSup10
         this.atSeconds < 32
      end_premise
      and
               premise prSemaphoreState10
         this.atSemaphoreState == 3
      end_premise
      and
      premise prVehicleSensorState10
```

```
            this.atHVSS == 1
         end_premise
      end_condition
      action sequential
         instigation parallel
            call this.mtVTLYCBCL()
                        call this.mtRT()
         end_instigation
      end_action
   end_rule

   rule rlCBCL18
      condition
         premise prSeconds10Full
            this.atSeconds >= 18
         end_premise
         and
         premise prSecondsSup10Full
            this.atSeconds < 32
         end_premise
         and
                  premise prSemaphoreState10Full
            this.atSemaphoreState == 3
         end_premise
         and
         premise prVehicleSensorState10Full
            this.atHVSS == 2
         end_premise
      end_condition
      action sequential
         instigation parallel
            call this.mtVTLYCBCL()
            call this.mtRT()
         end_instigation
      end_action
   end_rule

end_fbe
```

CÓDIGO-FONTE DA APLICAÇÃO CTA EM PON NO *FRAMEWORK* PON C++ 4.0

```
#include "libnop/framework.h"

class SemaphoreCTA
{
  public:
   NOP::SharedAttribute<int> atSeconds;

  private:
   NOP::SharedAttribute<int> atSemaphoreState;
   const NOP::SharedPremise prSeconds;
   const NOP::SharedPremise prSemaphoreState;
   NOP::SharedRule rlHorizontalTrafficLightGreen;
   const NOP::SharedPremise prAtSeconds3;
   const NOP::SharedPremise prSemaphoreState3;
   NOP::SharedRule rlHorizontalTrafficLightRed;
   const NOP::SharedPremise prSeconds2;
   const NOP::SharedPremise prSemaphoreState2;
   NOP::SharedRule rlHorizontalTrafficLightYellow;
   const NOP::SharedPremise prAtSeconds4;
   const NOP::SharedPremise prSemaphoreState4;
   NOP::SharedRule rlVerticalTrafficLightGreen;
   const NOP::SharedPremise prAtSeconds6;
   const NOP::SharedPremise prSemaphoreState6;
   NOP::SharedRule rlVerticalTrafficLightRed;
   const NOP::SharedPremise prAtSeconds5;
   const NOP::SharedPremise prSemaphoreState5;
   NOP::SharedRule rlVerticalTrafficLightYellow;

  public:
   SemaphoreCTA();

  private:
```

```
   void mtHorizontalTrafficLightGREEN();
   void mtHorizontalTrafficLightRED();
   void mtHorizontalTrafficLightYELLOW();
   void mtResetTimer();
   void mtVerticalTrafficLightGREEN();
   void mtVerticalTrafficLightRED();
   void mtVerticalTrafficLightYELLOW();
};

SemaphoreCTA::SemaphoreCTA()
   : atSeconds{NOP::BuildAttribute<int>(0)},
     atSemaphoreState{NOP::BuildAttribute<int>(5)},
     prSeconds{NOP::BuildPremise<>(atSeconds, 2, NOP::Equal())},
     prSemaphoreState{
        NOP::BuildPremise<int>(atSemaphoreState, 5, NOP::Equal())},
     prAtSeconds3{NOP::BuildPremise<int>(atSeconds, 45, NOP::Equal())},
     prSemaphoreState3{
        NOP::BuildPremise<int>(atSemaphoreState, 1, NOP::Equal())},
     prSeconds2{NOP::BuildPremise<int>(atSeconds, 40, NOP::Equal())},
     prSemaphoreState2{
        NOP::BuildPremise<int>(atSemaphoreState, 0, NOP::Equal())},
     prAtSeconds4{NOP::BuildPremise<int>(atSeconds, 47, NOP::Equal())},
     prSemaphoreState4{
        NOP::BuildPremise<int>(atSemaphoreState, 2, NOP::Equal())},
     prAtSeconds6{NOP::BuildPremise<int>(atSeconds, 90, NOP::Equal())},
     prSemaphoreState6{
        NOP::BuildPremise<int>(atSemaphoreState, 4, NOP::Equal())},
     prAtSeconds5{NOP::BuildPremise<int>(atSeconds, 85, NOP::Equal())},
     prSemaphoreState5{
        NOP::BuildPremise<int>(atSemaphoreState, 3, NOP::Equal())}
{
   rlHorizontalTrafficLightGreen = NOP::BuildRule(
      NOP::BuildCondition(CONDITION(*prSeconds && *prSemaphoreState),
                     prSeconds, prSemaphoreState),
      NOP::BuildAction(NOP::BuildInstigation<NOP::Parallel>(
         METHOD(mtHorizontalTrafficLightGREEN();)))

   );
   rlHorizontalTrafficLightRed = NOP::BuildRule(
      NOP::BuildCondition(CONDITION(*prAtSeconds3 && *prSemaphoreState3),
                     prAtSeconds3, prSemaphoreState3),
      NOP::BuildAction(NOP::BuildInstigation<NOP::Parallel>(
         METHOD(mtHorizontalTrafficLightRED();)))

   );
   rlHorizontalTrafficLightYellow = NOP::BuildRule(
      NOP::BuildCondition(CONDITION(*prSeconds2 && *prSemaphoreState2),
                     prSeconds2, prSemaphoreState2),
      NOP::BuildAction(NOP::BuildInstigation<NOP::Parallel>(
         METHOD(mtHorizontalTrafficLightYELLOW();)))

   );
   rlVerticalTrafficLightGreen = NOP::BuildRule(
      NOP::BuildCondition(CONDITION(*prAtSeconds4 && *prSemaphoreState4),
                     prAtSeconds4, prSemaphoreState4),
      NOP::BuildAction(NOP::BuildInstigation<NOP::Parallel>(
         METHOD(mtVerticalTrafficLightGREEN();)))

   );
   rlVerticalTrafficLightRed = NOP::BuildRule(
      NOP::BuildCondition(CONDITION(*prAtSeconds6 && *prSemaphoreState6),
                     prAtSeconds6, prSemaphoreState6),
      NOP::BuildAction(NOP::BuildInstigation<NOP::Parallel>(
         METHOD(mtVerticalTrafficLightRED();), METHOD(mtResetTimer();)))

   );
   rlVerticalTrafficLightYellow = NOP::BuildRule(
      NOP::BuildCondition(CONDITION(*prAtSeconds5 && *prSemaphoreState5),
                     prAtSeconds5, prSemaphoreState5),
      NOP::BuildAction(NOP::BuildInstigation<NOP::Parallel>(
         METHOD(mtVerticalTrafficLightYELLOW();)))

   );
}

void SemaphoreCTA::mtHorizontalTrafficLightGREEN()
{
   atSemaphoreState->SetValue<NOP::Parallel>(0);
}

void SemaphoreCTA::mtHorizontalTrafficLightRED()
```

```
{
   atSemaphoreState->SetValue<NOP::Parallel>(2);
}

void SemaphoreCTA::mtHorizontalTrafficLightYELLOW()
{
   atSemaphoreState->SetValue<NOP::Parallel>(1);
}

void SemaphoreCTA::mtResetTimer() { atSeconds->SetValue<NOP::Parallel>(0); }

void SemaphoreCTA::mtVerticalTrafficLightGREEN()
{
   atSemaphoreState->SetValue<NOP::Parallel>(3);
}

void SemaphoreCTA::mtVerticalTrafficLightRED()
{
   atSemaphoreState->SetValue<NOP::Parallel>(5);
}

void SemaphoreCTA::mtVerticalTrafficLightYELLOW()
{
   atSemaphoreState->SetValue<NOP::Parallel>(4);
}

class Semaphore_CBCL
{
  public:
   NOP::SharedAttribute<int> atSeconds;

  private:
   NOP::SharedAttribute<int> atHVSS;
   NOP::SharedAttribute<int> atSemaphoreState;
   NOP::SharedAttribute<int> atVVSS;
   const NOP::SharedPremise prSeconds;
   const NOP::SharedPremise prSemaphoreState;
   NOP::SharedRule rlCBCL1;
   const NOP::SharedPremise prSecondsCBCL6;
   const NOP::SharedPremise prSemaphoreStateCBCL6;
   NOP::SharedRule rlCBCL10;
   const NOP::SharedPremise prSeconds7;
   const NOP::SharedPremise prSemaphoreState7;
   const NOP::SharedPremise prVehicleSensorState7;
   NOP::SharedRule rlCBCL11;
   const NOP::SharedPremise prSeconds7Full;
   const NOP::SharedPremise prSemaphoreState7Full;
   const NOP::SharedPremise prVehicleSensorState7Full;
   NOP::SharedRule rlCBCL12;
   const NOP::SharedPremise prSeconds8;
   const NOP::SharedPremise prSecondsSup8;
   const NOP::SharedPremise prSemaphoreState8;
   const NOP::SharedPremise prVehicleSensorState8;
   NOP::SharedRule rlCBCL13;
   const NOP::SharedPremise prSeconds8Full;
   const NOP::SharedPremise prSecondsSup8Full;
   const NOP::SharedPremise prSemaphoreState8Full;
   const NOP::SharedPremise prVehicleSensorState8Full;
   NOP::SharedRule rlCBCL14;
   const NOP::SharedPremise prSeconds9;
   const NOP::SharedPremise prSemaphoreState9;
   const NOP::SharedPremise prVehicleSensorState9;
   NOP::SharedRule rlCBCL15;
   const NOP::SharedPremise prSeconds9Full;
   const NOP::SharedPremise prSemaphoreState9Full;
   const NOP::SharedPremise prVehicleSensorState9Full;
   NOP::SharedRule rlCBCL16;
   const NOP::SharedPremise prSeconds10;
   const NOP::SharedPremise prSecondsSup10;
   const NOP::SharedPremise prSemaphoreState10;
   const NOP::SharedPremise prVehicleSensorState10;
   NOP::SharedRule rlCBCL17;
   const NOP::SharedPremise prSeconds10Full;
   const NOP::SharedPremise prSecondsSup10Full;
   const NOP::SharedPremise prSemaphoreState10Full;
   const NOP::SharedPremise prVehicleSensorState10Full;
   NOP::SharedRule rlCBCL18;
   const NOP::SharedPremise prSeconds2;
   const NOP::SharedPremise prSemaphoreState2;
   NOP::SharedRule rlCBCL2;
   const NOP::SharedPremise prSecondsCBCL2;
```

```
   const NOP::SharedPremise prSemaphoreStateCBCL2;
   NOP::SharedRule rlCBCL3;
   const NOP::SharedPremise prSeconds3;
   const NOP::SharedPremise prSemaphoreState3;
   NOP::SharedRule rlCBCL4;
   const NOP::SharedPremise prSecondsCBCL3;
   const NOP::SharedPremise prSemaphoreStateCBCL3;
   NOP::SharedRule rlCBCL5;
   const NOP::SharedPremise prSeconds4;
   const NOP::SharedPremise prSemaphoreState4;
   NOP::SharedRule rlCBCL6;
   const NOP::SharedPremise prSeconds5;
   const NOP::SharedPremise prSemaphoreState5;
   NOP::SharedRule rlCBCL7;
   const NOP::SharedPremise prSecondsCBCL5;
   const NOP::SharedPremise prSemaphoreStateCBCL5;
   NOP::SharedRule rlCBCL8;
   const NOP::SharedPremise prSeconds6;
   const NOP::SharedPremise prSemaphoreState6;
   NOP::SharedRule rlCBCL9;

  public:
   Semaphore_CBCL();

  private:
   void mtHTLG();
   void mtHTLGCBCL();
   void mtHTLR();
   void mtHTLY();
   void mtHTLYCBCL();
   void mtRT();
   void mtVTLG();
   void mtVTLGCBCL();
   void mtVTLR();
   void mtVTLY();
   void mtVTLYCBCL();
};

Semaphore_CBCL::Semaphore_CBCL()
   : atSeconds{NOP::BuildAttribute<int>(0)},
     atHVSS{NOP::BuildAttribute<int>(0)},
     atSemaphoreState{NOP::BuildAttribute<int>(5)},
     atVVSS{NOP::BuildAttribute<int>(0)},
     prSeconds{NOP::BuildPremise<int>(atSeconds, 2, NOP::Equal())},
     prSemaphoreState{
        NOP::BuildPremise<int>(atSemaphoreState, 5, NOP::Equal())},
     prSecondsCBCL6{NOP::BuildPremise<int>(atSeconds, 6, NOP::Equal())},
     prSemaphoreStateCBCL6{
        NOP::BuildPremise<int>(atSemaphoreState, 9, NOP::Equal())},
     prSeconds7{NOP::BuildPremise<int>(atSeconds, 17, NOP::LessEqual())},
     prSemaphoreState7{
        NOP::BuildPremise<int>(atSemaphoreState, 0, NOP::Equal())},
     prVehicleSensorState7{NOP::BuildPremise<int>(atVVSS, 1, NOP::Equal())},
     prSeconds7Full{NOP::BuildPremise<int>(atSeconds, 17, NOP::LessEqual())},
     prSemaphoreState7Full{
        NOP::BuildPremise<int>(atSemaphoreState, 0, NOP::Equal())},
     prVehicleSensorState7Full{
        NOP::BuildPremise<int>(atVVSS, 2, NOP::Equal())},
     prSeconds8{NOP::BuildPremise<int>(atSeconds, 18, NOP::GreaterEqual())},
     prSecondsSup8{NOP::BuildPremise<int>(atSeconds, 32, NOP::Less())},
     prSemaphoreState8{
        NOP::BuildPremise<int>(atSemaphoreState, 0, NOP::Equal())},
     prVehicleSensorState8{NOP::BuildPremise<int>(atVVSS, 1, NOP::Equal())},
     prSeconds8Full{
        NOP::BuildPremise<int>(atSeconds, 18, NOP::GreaterEqual())},
     prSecondsSup8Full{NOP::BuildPremise<int>(atSeconds, 32, NOP::Less())},
     prSemaphoreState8Full{
        NOP::BuildPremise<int>(atSemaphoreState, 0, NOP::Equal())},
     prVehicleSensorState8Full{
        NOP::BuildPremise<int>(atVVSS, 2, NOP::Equal())},
     prSeconds9{NOP::BuildPremise<int>(atSeconds, 17, NOP::LessEqual())},
     prSemaphoreState9{
        NOP::BuildPremise<int>(atSemaphoreState, 3, NOP::Equal())},
     prVehicleSensorState9{
        NOP::BuildPremise<int>(atSemaphoreState, 1, NOP::Equal())},
     prSeconds9Full{NOP::BuildPremise<int>(atSeconds, 77, NOP::LessEqual())},
     prSemaphoreState9Full{
        NOP::BuildPremise<int>(atSemaphoreState, 3, NOP::Equal())},
     prVehicleSensorState9Full{
        NOP::BuildPremise<int>(atSemaphoreState, 2, NOP::Equal())},
     prSeconds10{NOP::BuildPremise<int>(atSeconds, 18, NOP::GreaterEqual())},
```

```
    prSecondsSup10{NOP::BuildPremise<int>(atSeconds, 32, NOP::Less())},
    prSemaphoreState10{
       NOP::BuildPremise<int>(atSemaphoreState, 3, NOP::Equal())},
    prVehicleSensorState10{NOP::BuildPremise<int>(atHVSS, 1, NOP::Equal())},
    prSeconds10Full{
       NOP::BuildPremise<int>(atSeconds, 18, NOP::GreaterEqual())},
    prSecondsSup10Full{NOP::BuildPremise<int>(atSeconds, 32, NOP::Less())},
    prSemaphoreState10Full{
       NOP::BuildPremise<int>(atSemaphoreState, 3, NOP::Equal())},
    prVehicleSensorState10Full{
       NOP::BuildPremise<int>(atHVSS, 2, NOP::Equal())},
    prSeconds2{NOP::BuildPremise<int>(atSeconds, 38, NOP::Equal())},
    prSemaphoreState2{
       NOP::BuildPremise<int>(atSemaphoreState, 0, NOP::Equal())},
    prSecondsCBCL2{NOP::BuildPremise<int>(atSeconds, 30, NOP::Equal())},
    prSemaphoreStateCBCL2{
       NOP::BuildPremise<int>(atSemaphoreState, 6, NOP::Equal())},
    prSeconds3{NOP::BuildPremise<int>(atSeconds, 5, NOP::Equal())},
    prSemaphoreState3{
       NOP::BuildPremise<int>(atSemaphoreState, 1, NOP::Equal())},
    prSecondsCBCL3{NOP::BuildPremise<int>(atSeconds, 6, NOP::Equal())},
    prSemaphoreStateCBCL3{
       NOP::BuildPremise<int>(atSemaphoreState, 7, NOP::Equal())},
    prSeconds4{NOP::BuildPremise<int>(atSeconds, 2, NOP::Equal())},
    prSemaphoreState4{
       NOP::BuildPremise<int>(atSemaphoreState, 2, NOP::Equal())},
    prSeconds5{NOP::BuildPremise<int>(atSeconds, 38, NOP::Equal())},
    prSemaphoreState5{
       NOP::BuildPremise<int>(atSemaphoreState, 3, NOP::Equal())},
    prSecondsCBCL5{NOP::BuildPremise<int>(atSeconds, 30, NOP::Equal())},
    prSemaphoreStateCBCL5{
       NOP::BuildPremise<int>(atSemaphoreState, 8, NOP::Equal())},
    prSeconds6{NOP::BuildPremise<int>(atSeconds, 5, NOP::Equal())},
    prSemaphoreState6{
       NOP::BuildPremise<int>(atSemaphoreState, 4, NOP::Equal())}
{
   rlCBCL1 = NOP::BuildRule(
      NOP::BuildCondition(CONDITION(*prSeconds && *prSemaphoreState),
                     prSeconds, prSemaphoreState),
      NOP::BuildAction(NOP::BuildInstigation<NOP::Parallel>(METHOD(mtHTLG();),
                                             METHOD(mtRT();)))

   );
   rlCBCL10 =
      NOP::BuildRule(NOP::BuildCondition(
                     CONDITION(*prSecondsCBCL6 && *prSemaphoreStateCBCL6),
                     prSecondsCBCL6, prSemaphoreStateCBCL6),
                  NOP::BuildAction(NOP::BuildInstigation<NOP::Parallel>(
                     METHOD(mtVTLR();), METHOD(mtRT();)))

      );
   rlCBCL11 = NOP::BuildRule(
      NOP::BuildCondition(CONDITION(*prSeconds7 && *prSemaphoreState7 &&
                              *prVehicleSensorState7),
                     prSeconds7, prSemaphoreState7,
                     prVehicleSensorState7),
      NOP::BuildAction(
         NOP::BuildInstigation<NOP::Parallel>(METHOD(mtHTLGCBCL();)))

   );
   rlCBCL12 = NOP::BuildRule(
      NOP::BuildCondition(
         CONDITION(*prSeconds7Full && *prSemaphoreState7Full &&
                 *prVehicleSensorState7Full),
         prSeconds7Full, prSemaphoreState7Full, prVehicleSensorState7Full),
      NOP::BuildAction(
         NOP::BuildInstigation<NOP::Parallel>(METHOD(mtHTLGCBCL();)))

   );
   rlCBCL13 = NOP::BuildRule(
      NOP::BuildCondition(
         CONDITION(*prSeconds8 && *prSecondsSup8 && *prSemaphoreState8 &&
                 *prVehicleSensorState8),
         prSeconds8, prSecondsSup8, prSemaphoreState8,
         prVehicleSensorState8),
      NOP::BuildAction(NOP::BuildInstigation<NOP::Parallel>(
         METHOD(mtHTLYCBCL();), METHOD(mtRT();)))

   );
   rlCBCL14 = NOP::BuildRule(
      NOP::BuildCondition(
```

```
      CONDITION(*prSeconds8Full && *prSecondsSup8Full &&
             *prSemaphoreState8Full && *prVehicleSensorState8Full),
      prSeconds8Full, prSecondsSup8Full, prSemaphoreState8Full,
      prVehicleSensorState8Full),
   NOP::BuildAction(NOP::BuildInstigation<NOP::Parallel>(
      METHOD(mtHTLYCBCL();), METHOD(mtRT();)))

);
rlCBCL15 = NOP::BuildRule(
   NOP::BuildCondition(CONDITION(*prSeconds9 && *prSemaphoreState9 &&
                        *prVehicleSensorState9),
                  prSeconds9, prSemaphoreState9,
                  prVehicleSensorState9),
   NOP::BuildAction(
      NOP::BuildInstigation<NOP::Parallel>(METHOD(mtVTLGCBCL();)))

);
rlCBCL16 = NOP::BuildRule(
   NOP::BuildCondition(
      CONDITION(*prSeconds9Full && *prSemaphoreState9Full &&
             *prVehicleSensorState9Full),
      prSeconds9Full, prSemaphoreState9Full, prVehicleSensorState9Full),
   NOP::BuildAction(
      NOP::BuildInstigation<NOP::Parallel>(METHOD(mtVTLGCBCL();)))

);
rlCBCL17 = NOP::BuildRule(
   NOP::BuildCondition(
      CONDITION(*prSeconds10 && *prSecondsSup10 && *prSemaphoreState10 &&
             *prVehicleSensorState10),
      prSeconds10, prSecondsSup10, prSemaphoreState10,
      prVehicleSensorState10),
   NOP::BuildAction(NOP::BuildInstigation<NOP::Parallel>(
      METHOD(mtVTLYCBCL();), METHOD(mtRT();)))

);
rlCBCL18 = NOP::BuildRule(
   NOP::BuildCondition(
      CONDITION(*prSeconds10Full && *prSecondsSup10Full &&
             *prSemaphoreState10Full && *prVehicleSensorState10Full),
      prSeconds10Full, prSecondsSup10Full, prSemaphoreState10Full,
      prVehicleSensorState10Full),
   NOP::BuildAction(NOP::BuildInstigation<NOP::Parallel>(
      METHOD(mtVTLYCBCL();), METHOD(mtRT();)))

);
rlCBCL2 = NOP::BuildRule(
   NOP::BuildCondition(CONDITION(*prSeconds2 && *prSemaphoreState2),
                  prSeconds2, prSemaphoreState2),
   NOP::BuildAction(NOP::BuildInstigation<NOP::Parallel>(METHOD(mtHTLY();),
                                          METHOD(mtRT();)))

);
rlCBCL3 =
   NOP::BuildRule(NOP::BuildCondition(
                  CONDITION(*prSecondsCBCL2 && *prSemaphoreStateCBCL2),
                  prSecondsCBCL2, prSemaphoreStateCBCL2),
               NOP::BuildAction(NOP::BuildInstigation<NOP::Parallel>(
                  METHOD(mtHTLY();), METHOD(mtRT();)))

   );
rlCBCL4 = NOP::BuildRule(
   NOP::BuildCondition(CONDITION(*prSeconds3 && *prSemaphoreState3),
                  prSeconds3, prSemaphoreState3),
   NOP::BuildAction(NOP::BuildInstigation<NOP::Parallel>(METHOD(mtHTLR();),
                                          METHOD(mtRT();)))

);
rlCBCL5 =
   NOP::BuildRule(NOP::BuildCondition(
                  CONDITION(*prSecondsCBCL3 && *prSemaphoreStateCBCL3),
                  prSecondsCBCL3, prSemaphoreStateCBCL3),
               NOP::BuildAction(NOP::BuildInstigation<NOP::Parallel>(
                  METHOD(mtHTLR();), METHOD(mtRT();)))

   );
rlCBCL6 = NOP::BuildRule(
   NOP::BuildCondition(CONDITION(*prSeconds4 && *prSemaphoreState4),
                  prSeconds4, prSemaphoreState4),
   NOP::BuildAction(NOP::BuildInstigation<NOP::Parallel>(METHOD(mtVTLG();),
                                          METHOD(mtRT();)))
```

```
   );
   rlCBCL7 = NOP::BuildRule(
      NOP::BuildCondition(CONDITION(*prSeconds5 && *prSemaphoreState5),
                     prSeconds5, prSemaphoreState5),
      NOP::BuildAction(NOP::BuildInstigation<NOP::Parallel>(METHOD(mtVTLY();),
                                              METHOD(mtRT();)))

   );
   rlCBCL8 =
      NOP::BuildRule(NOP::BuildCondition(
                     CONDITION(*prSecondsCBCL5 && *prSemaphoreStateCBCL5),
                     prSecondsCBCL5, prSemaphoreStateCBCL5),
                  NOP::BuildAction(NOP::BuildInstigation<NOP::Parallel>(
                     METHOD(mtVTLY();), METHOD(mtRT();)))

      );
   rlCBCL9 = NOP::BuildRule(
      NOP::BuildCondition(CONDITION(*prSeconds6 && *prSemaphoreState6),
                     prSeconds6, prSemaphoreState6),
      NOP::BuildAction(NOP::BuildInstigation<NOP::Parallel>(METHOD(mtVTLR();),
                                              METHOD(mtRT();)))

   );
}

void Semaphore_CBCL::mtHTLG() { atSemaphoreState->SetValue<NOP::Parallel>(0); }

void Semaphore_CBCL::mtHTLGCBCL()
{
   atSemaphoreState->SetValue<NOP::Parallel>(6);
}

void Semaphore_CBCL::mtHTLR() { atSemaphoreState->SetValue<NOP::Parallel>(2); }

void Semaphore_CBCL::mtHTLY() { atSemaphoreState->SetValue<NOP::Parallel>(1); }

void Semaphore_CBCL::mtHTLYCBCL()
{
   atSemaphoreState->SetValue<NOP::Parallel>(7);
}

void Semaphore_CBCL::mtRT() { atSeconds->SetValue<NOP::Parallel>(0); }

void Semaphore_CBCL::mtVTLG() { atSemaphoreState->SetValue<NOP::Parallel>(3); }

void Semaphore_CBCL::mtVTLGCBCL()
{
   atSemaphoreState->SetValue<NOP::Parallel>(8);
}

void Semaphore_CBCL::mtVTLR() { atSemaphoreState->SetValue<NOP::Parallel>(5); }

void Semaphore_CBCL::mtVTLY() { atSemaphoreState->SetValue<NOP::Parallel>(4); }

void Semaphore_CBCL::mtVTLYCBCL()
{
   atSemaphoreState->SetValue<NOP::Parallel>(9);
}
```

APÊNDICE K

Avaliação do Framework PON C++ 4.0

Gostaria de receber feedback referente a utilização dos frameworks do PON, de modo a permitir comparações Frameworks PON C++ 2.0, que será utilizado na composição da minha dissertação. Suas respostas podem ser baseadas tanto na sua experiência pessoal utilizando os frameworks ou de conhecimento adquirido através de apresentações e leitura de materiais como artigos, dissertações e teses.

*Obrigatório

1. Assinale as versões de Framework (ou NOPL) do PON que você já utilizou para o desenvolvimento de programas *

Marque todas que se aplicam.

- [ ] C++ Prototipal
- [ ] C++ 1.0
- [ ] C++ 2.0
- [ ] C++ 3.0
- [ ] C++ 4.0
- [ ] Java / C#
- [ ] C# IoT
- [ ] Elixir/Erlang
- [ ] Akka
- [ ] JuNOC++
- [ ] LingPON
- [ ] NOPLite
- [ ] Outro: _____

2. Como você avalia o Framework PON C++ 2.0 nas seguintes características *

Marcar apenas uma oval por linha.

	Fraco	Moderado	Satisfatório	Muito bom	Excelente
Facilidade de aprendizado	◯	◯	◯	◯	◯
Facilidade de uso	◯	◯	◯	◯	◯
Verbosidade	◯	◯	◯	◯	◯
Versatilidade	◯	◯	◯	◯	◯
Desempenho	◯	◯	◯	◯	◯
Aderência aos fundamentos do PON	◯	◯	◯	◯	◯

3. Como você avalia o Framework PON C++ 4.0 nas seguintes características *

Marcar apenas uma oval por linha.

	Fraco	Moderado	Satisfatório	Muito bom	Excelente
Facilidade de aprendizado	◯	◯	◯	◯	◯
Facilidade de uso	◯	◯	◯	◯	◯
Verbosidade	◯	◯	◯	◯	◯
Versatilidade	◯	◯	◯	◯	◯
Desempenho	◯	◯	◯	◯	◯
Aderência aos fundamentos do PON	◯	◯	◯	◯	◯

4. Numa escala de 0 a 5, o quanto você considera que o Framework PON C++ 4.0 apresenta melhorias sobre o Framework PON C++ 2.0, sendo que 0 significa que não houve melhoria. *

Marcar apenas uma oval.

0	1	2	3	4	5
◯	◯	◯	◯	◯	◯

5. Numa escala de 0 a 5, o quanto você considera viável dar manutenção ao código apresentado pelo Framework PON C++ 4.0, sendo que 0 significa que não seria viável. Nessa avaliação considere a complexidade e quantidade de código apresentada. *

Marcar apenas uma oval.

0	1	2	3	4	5
◯	◯	◯	◯	◯	◯

6. Qual a probabilidade de você escolher utilizar os seguintes Frameworks no desenvolvimento de uma aplicação em PON *

Marcar apenas uma oval por linha.

	Improvável	Pouco provável	Provável	Muito provável
C++ Prototipal	◯	◯	◯	◯
C++ 1.0	◯	◯	◯	◯
C++ 2.0	◯	◯	◯	◯
C++ 3.0	◯	◯	◯	◯
C++ 4.0	◯	◯	◯	◯
Java / C#	◯	◯	◯	◯
C# IoT	◯	◯	◯	◯
Elixir/Erlang	◯	◯	◯	◯
Akka	◯	◯	◯	◯
JuNOC++	◯	◯	◯	◯
LingPON	◯	◯	◯	◯
NOPLite	◯	◯	◯	◯

7. Considerando que o Framework PON C++ 4.0 ainda se encontra em desenvolvimento ativo, possuí alguma sugestão para o desenvolvimento do mesmo? Considere aqui sugestões que de forma geral podem melhorar a experiência de desenvolvimento de programas com os frameworks em PON, sejam estas questões de arquitetura, interface e até mesmo de compilação.