Comparação entre Paradigma Orientado a Notificações e Paradigma Imperativo sobre um Simulador de Tráfego

~~Leonardo F. Pordeus[1], Adriano F. Ronszcka[1], Cleverson A. Ferreira[2]~~, Fabio Negrini[2], ~~Robson R. Linhares[2], João A. Fabro[2], Douglas P. B. Renaux[2], Paulo C. Stadzisz[1,2], Jean M. Simão[1,2]~~

[1] Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI),

[2] Programa de Pós-Graduação em Computação Aplicada (PPGCA),

Universidade Tecnológica Federal do Paraná (UTFPR) - Av. Sete de Setembro, 3165. Curitiba-PR, Brasil

{leonardopordeus | ronszcka | cleversonferreira | fabionegrini @alunos.} | {linhares | fabro | douglasrenaux | stadzisz | jeansimao @}utfpr.edu.br

***Resumo*— O Paradigma Orientado a Notificações (PON) apresenta uma nova abordagem para desenvolver software de maneira mais eficiente quando comparado com os paradigmas de programação tradicionais, como o Paradigma Declarativo (PD) e o Paradigma Imperativo (PI). Esses paradigmas apresentam deficiências, como redundâncias de execução e entidades fortemente acopladas, o que degrada o desempenho e gera maior dificuldade de paralelização e distribuição. Este artigo apresenta comparações entre o PON (via materialização recente chamada LingPon) e o PI (via linguagem C++) em termos de desempenho e complexidade no desenvolvimento de software, por meio da implementação, em ambos os paradigmas, de uma aplicação para simulação de tráfego chamado CTA. Os resultados indicam uma melhoria no desempenho quando comparado com a mesma implementação da aplicação em um paradigma tradicional. No entanto, em termos de complexidade de desenvolvimento, o PON ainda apresenta algumas dificuldades relativas às suas materializações atuais.**

Palavras-chaves— Paradigma Orientado a Notificações, Comparações entre PON e IP, Simulação de controle de tráfego.

I. INTRODUÇÃO

O crescimento do desempenho dos processadores utilizados em sistemas computacionais ao longo da história dependeu principalmente, do aumento da frequência de operação (*clock*) e do aumento da densidade de integração de semicondutores. Este último fator, particularmente, continua a ser válido de acordo com os preceitos da Lei de Moore, segundo a qual a densidade de integração aproximadamente dobra a cada 24 meses [1]. No entanto, a frequência do *clock* já não pode ser aumentada nas mesmas proporções históricas, devido a limitações físicas dos materiais semicondutores utilizados, o que traz a necessidade de se explorar cada vez melhor o espaço disponível devido ao aumento da densidade para o aumento do desempenho de computação.

O aumento da densidade de integração tem sido aproveitado pelos fabricantes para a implementação de múltiplos núcleos em uma mesma pastilha (multicore). Embora, em tese, o aumento do número de unidades de processamento em paralelo permita aumentar o desempenho de execução da computação, na prática isto depende de software que explore adequadamente o paralelismo. Este, por sua vez, depende de técnicas adequadas de desenvolvimento, as quais geralmente impõem dificuldades de abstração aos desenvolvedores em função justamente da dinâmica de execução paralela [2][3][4].

O Paradigma Orientado a Notificações (PON) é uma nova abordagem para o desenvolvimento de sistemas computacionais. O PON tende a apresentar melhor desempenho, maior nível de abstração e proporciona facilidades para o paralelismo/distribuição em comparação com sistemas baseados em paradigmas tradicionais, como a Programação Procedimental e a Programação Orientada a Objetos (POO) do Paradigma Imperativo (PI), assim como os Sistemas baseados em Regras (SBR) do paradigma declarativo (PD).

O PON propõe uma solução para os problemas destes paradigmas, que apresentam deficiências com relação a redundâncias estruturais, temporais e forte acoplamento entre suas entidades, diminuindo o desempenho e gerando maior dificuldade de paralelização e distribuição [5][6]. Para o desenvolvimento de softwares fazendo uso do PON, foram realizadas pesquisas com framework C++ [7] e uma segunda versão otimizada do framework C++ [8], permitindo a criação de softwares PON sob abordagem de POO.

Outras pesquisas também exploraram a implementação do PON em hardware com uso de lógica reconfigurável [9] seguindo os conceitos do PON. Peters[10] propôs a implementação em lógica reconfigurável de um co-processador PON (CoPON), uma solução híbrida, na qual a parte da aplicação responsável pelo processamento factual é executada em um núcleo von Neumann e a parte da aplicação responsável pelo cálculo lógico-causal e propagação de notificações é executada por meio de um co-processador baseado nos princípios do PON. Outrossim, uma arquitetura de processador foi desenvolvida de acordo com o modelo do PON, sendo denominada Notification-Oriented Computer Architecture (NOCA) [11]. Ademais, pesquisas recentes visam o desenvolvimento de um compilador e de uma linguagem específica para o PON, denominada LingPon [12][13].

Neste artigo são apresentados os experimentos e resultados de comparações entre os paradigmas PI (na forma de Programação Orientada a Objetos – POO) e PON, sendo efetuados por meio da análise de tempo de execução da implementação de um software simulador de trânsito chamado CTA Simulator [14]. Neste software, os módulos responsáveis por controle são implementados tanto em PI (POO) quanto em LingPon, permitindo assim sua comparação.

As próximas seções estão organizadas da seguinte maneira: a Seção II apresenta os conceitos sobre PON. A Seção III descreve a aplicação CTA Simulator. Na Seção IV os experimentos e os resultados são apresentados. As conclusões e os trabalhos futuros são discutidos na Seção V.

II. PARADIGMA ORIENTADO A NOTIFICAÇÕES (PON)

O PON foi proposto como um novo paradigma de desenvolvimento de software, que apresenta algumas vantagens quando comparado aos paradigmas tradicionais (mais especificamente, o PI – Paradigma Imperativo - e o PD – Paradigma Declarativo) no que diz respeito ao seu modelo lógico. Tais vantagens são constituídas por uma maior facilidade na concepção de sistemas que apresentem paralelismo ou distribuição, além da redução ou eliminação de alguns dos problemas clássicos de software PI e PD, tais como redundâncias de execução e acoplamento excessivo entre entidades computacionais [15].

Estruturalmente, o software PON é representado na forma de Base de Fatos (*FBE – Fact Base Element*) e as Regras (*Rules*). Os elementos *FBE* são utilizados para representar objetos do mundo real em um sistema computacional, por meio de estados (atributos) e serviços (métodos). Os elementos *Rules*, por sua vez, definem o cálculo lógico-causal a ser efetuado sobre os estados dos *FBEs*, controlando a execução dos seus serviços. A colaboração entre estes elementos ocorre por meio de notificações diretas, que é um processo de inferência essencialmente distinto dos processos utilizados em software PI e em Sistemas Baseados em Regras (SBR) do PD [7].

Nos *FBEs* os estados (atributos) são representados por meio de objetos da classe *Attribute*, enquanto os serviços dessas entidades (métodos) são representados pela classe *Method*. O elemento que representa uma *Rule* é composto por uma condição, representada pelo objeto da classe *Condition* e uma Ação, pelo objeto da classe *Action*. O objeto da classe *Condition* efetua um cálculo lógico sobre o valor de uma ou duas premissas, representadas por objetos da classe *Premise*. Estas são responsáveis por efetuar cálculo relacional sobre os valores dos atributos de um *FBE*, encapsulados em objetos da classe *Attribute*. O objeto da classe *Action*, por sua vez referencia um ou mais objetos da classe *Instigation*, por meio dos quais é capaz de disparar métodos dos *FBEs*, encapsulados em objetos da classe *Method* [7].

Para apresentar o modelo dinâmico do PON, faz-se uso de um exemplo de *Rule* (Figura 1) implementado para o sistema CTA apresentado na Seção III.

Do ponto de vista dinâmico, cada vez que o valor de um *Attribute* (p ex. *atSemaphoreState)* é modificado, o próprio *Attribute* notifica as entidades *Premise* que dependem do seu valor (no caso, a *Premise prSemaphoreState*). Cada uma destas *Premises*, por sua vez, tem seu valor lógico reavaliado por meio do cálculo relacional que define, o qual opera sobre o novo valor do *Attribute* que a notificou e também sobre um segundo valor, que pode ser uma constante ou outro *Attribute* (no caso, o valor constante 5).

Cada *Premise* cujo resultado lógico tenha sido alterado notifica um conjunto de entidades *Conditions* que dela dependem. Em seguida, cada uma destas *Conditions* também têm seus estados lógicos reavaliados de acordo com os resultados das *Premises*. No exemplo, caso o atributo *atSeconds* ou o *atSemaphoreState* tenham sido alterados, a *Premise* correspondente notificará a *Condition* definida para a *Rule rlHorizontalTrafficLightGreen,* enviando o seu novo valor lógico. A *Condition,* em seguida, recalculará o seu valor lógico a partir da operação lógica AND que define.

Se o valor lógico da entidade *Condition* é calculado como verdadeiro, esta aprova a execução da sua respectiva *Rule*. Com isso, a entidade *Action* agregada a esta *Rule* notifica as *Instigations* às quais está conectada, as quais disparam os *Methods* correspondentes (*mtHorizontalTrafficLightGreen* do *FBE Semaphore_NOP*) também por meio do envio de notificações. A execução dos *Methods* gera o processamento factual que atualiza *Attributes* dos *FBEs* (no caso, alteração de *atSemaphoreState* para 0), reiniciando o ciclo.

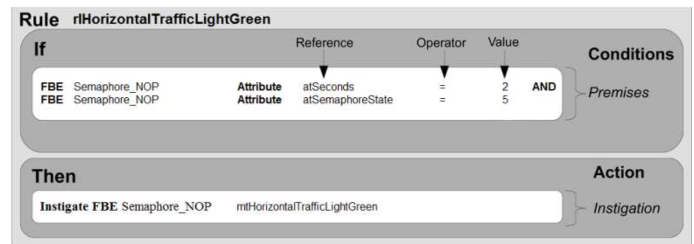

Figura 1. Exemplo de *Rule* para controle independente.

De maneira geral, o PON proporciona uma execução livre de avaliações redundantes e desnecessárias. Porém, ainda existem casos em que uma determinada mudança constante no valor de um *Attribute*, inicia fluxos de notificações, sem que ocorra a aprovação de uma *Rule* efetivamente. Desta forma, estes fluxos de notificações desnecessários impactam negativamente no desempenho de uma aplicação PON [7].

Por exemplo, na Figura 1 são avaliados os *Attributes atSemaphoreState* e *atSeconds*, na qual cada mudança do *Attribute atSemaphoreState* é pertinente para que ocorra a mudança do estado atual, para um próximo estado em sequencia(no caso, alteração de *atSemaphoreState* para 0). No caso do *Attribute atSeconds*, para a *Rule rlHorizontalTrafficLightGreen*, a maior parte do tempo as alterações deste *Attribute* não impactam na ativação da *Rule* em questão, podendo ser chamada de impertinente. Assim, a avaliação de uma *Premise*, em que o *Attribute* seja impertinente (no caso, *atSeconds*) só será realizada quando as *Premises* dos *Attributes* pertinentes (no caso, *atSemaphoreState*) tiverem seus valores lógicos verdadeiros.

A Figura 2 apresenta um exemplo genérico de instanciação das entidades que compõem um software PON, bem como as relações de envio e recebimento de notificação. As 4 linhas em negrito representam o fluxo de notificações para a ativação das *Rules* (cálculo lógico-causal), ao passo que as linhas mais claras representam o fluxo para a execução das *Rules* (cálculo factual).

III. DESCRIÇÃO DO CTA SIMULATOR

Nesta Seção são apresentados os objetivos e características do software CTA Simulator.

Os objetivos do projeto CTA Simulator são desenvolver estratégias de controle de semáforos, simular regiões de tráfego em uma área urbana, comparar o desempenho das estratégias, comparar o desempenho e complexidade quando as estratégias de controles forem implementadas em paradigmas diferentes, como o paradigma imperativo e o paradigma orientado a notificações [14].

Para realizar as comparações entre os paradigmas, o CTA Simulator foi dividido em dois módulos desacoplados, sendo um de simulação da região urbana, e outro módulo para implementação das estratégias de controle em diferentes paradigmas.

A. Simulador

O simulador representa elementos do mundo real, como veículos, ruas, pistas, quadras, sinaleiros, sensores e

cruzamentos em uma região de simulação matricial composta por dez ruas verticais e 10 ruas horizontais de mão única, formando uma matriz 10x10 com um total de 100 interseções [14].

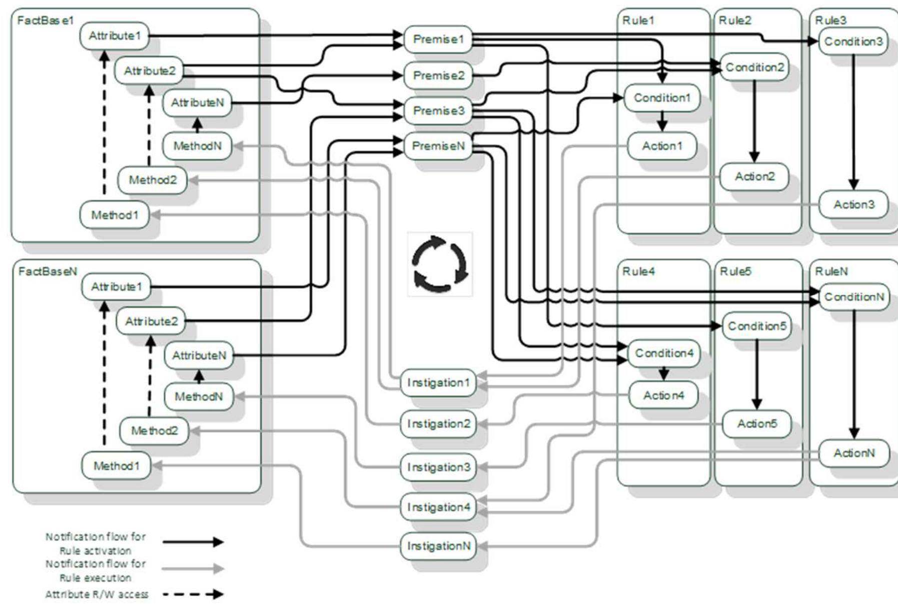

Figura 2. Colaboração por notificações das entidades do PON [12]

Cada interseção da região simulada é composta por dois sinaleiros, um em cada rua, que têm os mesmos estados de um sinaleiro do mundo real: verde, amarelo e vermelho. A união de um sinaleiro para a rua vertical e outro para a rua horizontal é chamada de semáforo e, por questões de segurança, os sinaleiros não podem estar verdes simultaneamente [14].

Cada quadra possui o comprimento de 100 metros e capacidade de 25 veículos por pista, sendo que cada rua pode possuir de 1 a 4 pistas. Os veículos são criados de forma constante, no intervalo de 0.1, 0.2, 0.3, 0.4 ou 0.5 veículos por segundo. Quando um veículo é criado em uma entrada, ele começa a se movimentar a uma velocidade de 20m/s, e deve parar de se movimentar quando o sinal da quadra em que se encontra estiver vermelho ou no caso da próxima quadra estar cheia. Em cada cruzamento, uma porcentagem de veículos poderá virar para uma outra quadra. Esta porcentagem pode variar de 5% até 35% [14].

Em todas as quadras são instalados sensores que monitoram a quantidade de veículos que estão parados num sinal vermelho. Esses sensores apresentam três estados: FEW, MANY e FULL. Para o sensor estar no estado FEW a taxa de ocupação da rua deve ser menor do que 60%, para o estado MANY, a taxa de ocupação deve estar entre 60% e 99% e para o estado FULL, a taxa de ocupação deve ser 100% [14].

B. *Estratégias de controle*

Para o simulador CTA, foram levantadas três alternativas de estratégia de controle de semáforos: controle independente, controle baseado em congestionamento e controle baseado em tráfego facilitado [14].

No controle independente, cada semáforo possui tempos fixos para cada estado do sinaleiro, não sendo considerados os sensores de quantidade de veículos e tempos de semáforos vizinhos.

Na estratégia de controle baseada em congestionamento é avaliado o tempo de cada semáforo e o estado referente à quantidade de veículos parados. Se o sensor detecta que a porcentagem de veículos parados está entre 60% até 100%, e o tempo do sinaleiro em vermelho é menor do que 24 segundos, então o tempo total do sinaleiro no estado vermelho é ajustado para 30 segundos. Caso o sensor detecte que a taxa de ocupação está entre 60% e 100% e o tempo do semáforo vermelho está entre 25 segundos e 39 segundos, o sinaleiro oposto altera imediatamente para o estado amarelo e o tempo restante do sinaleiro no estado vermelho é ajustado para 6 segundos. Se o sensor detectar que a ocupação está entre 60% e 100%, e o tempo do sinaleiro em vermelho for maior do que 39 segundos, não é realizada nenhuma alteração no tempo do sinaleiro.

No controle baseado em tráfego facilitado, um de seus sinaleiros possui uma *flag* sinalizando que a rua possui tráfego facilitado. Neste método de controle, o semáforo conhece o estado dos sensores de quantidade de tráfego e o tempo do semáforo anterior.

Neste caso, se o nível de congestionamento for menor do que 60% o atraso com relação ao semáforo anterior é de 5 segundos, ou seja, o sinaleiro abre 5 segundos após o sinaleiro do semáforo anterior abrir. Se o nível de congestionamento estiver entre 60% e 99% não há diferença no tempo de abertura com os semáforos anteriores. Se o nível de congestionamento for igual a 100% o atraso com relação ao semáforo anterior é de -5 segundos, ou seja, o semáforo abre 5 segundos antes do que o semáforo anterior [14].

C. *Desenvolvimento do CTA*

O software foi dividido em dois módulos: um módulo de simulação e outro de controle de estratégias, de forma que seja possível implementar o controle dos semáforos em paradigmas de programação diferentes. Para o desenvolvimento do simulador, foi utilizada a linguagem C++ com o ambiente de desenvolvimento Visual Studio 2013 Professional. As estratégias de controle foram desenvolvidas em PI, com a mesma linguagem do simulador, e em PON através da geração de código C++ a partir do LingPon.

Para lógica do controle independente foi utilizado o diagrama de estados como mostra na Figura 3, conforme retratado no documento CONOPS [14].

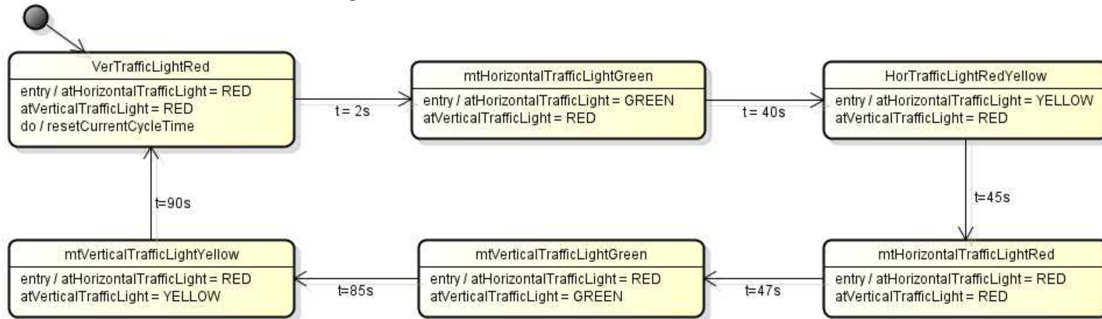

Figura 3. Diagrama de estados para controle independente.

Para implementação da mesma estratégia utilizando PON, utilizou-se da linguagem LingPon [12][13]. Cada transição do diagrama da Figura 4 foi traduzida para uma regra (*Rule*), totalizando seis regras para a estratégia de controle independente, na qual cada regra contém duas *Premises*, o estado e o tempo do semáforo. A Figura 4 e a Figura 5 apresentam as sintaxes de uma *Rule* e um *FBE* escrito através da linguagem LingPon, com base no diagrama da Figura 3, enquanto a Figura 6 apresenta a sintaxe do código desenvolvido em PI, com base no mesmo diagrama.

```
rule rlHorizontalTrafficLightGreen
    condition
        subcondition sbHorizontalTrafficLightGreen
            premise imp prSeconds semaphore_NOP.atSeconds == 2  and
            premise prSemaphoreState semaphore_NOP.atSemaphoreState == 5
        end_subcondition
    end_condition
    action
        instigation inHTLG semaphore_NOP.mtHorizontalTrafficLightGREEN();
    end_action
end_rule
```

Figura 4. Exemplo de *Rule* para controle independente.

```
fbe Semaphore_NOP
    attributes
        integer atSeconds 0
        integer atSemaphoreState 5
    end_attributes

    methods
        method mtResetTimer(atSeconds = 0)
        method mtHorizontalTrafficLightGREEN(atSemaphoreState = 0)
        method mtHorizontalTrafficLightYELLOW(atSemaphoreState = 1)
        method mtHorizontalTrafficLightRED(atSemaphoreState = 2)
        method mtVerticalTrafficLightGREEN(atSemaphoreState = 3)
        method mtVerticalTrafficLightYELLOW(atSemaphoreState = 4)
        method mtVerticalTrafficLightRED(atSemaphoreState = 5)
    end_methods
end_fbe
```

Figura 5. Exemplo de FBE para controle independente.

```
while (...){
    if (semaphore->getCurrentCycleTime() == 2){
        semaphore->getHorizontalTrafficLight()->setState(GREEN);
        semaphore->getVerticalTrafficLight()->setState(RED);
    }
    else if (semaphore->getCurrentCycleTime() == 38){
        semaphore->getHorizontalTrafficLight()->setState(YELLOW);
        semaphore->getVerticalTrafficLight()->setState(RED);
    }
    else if (semaphore->getCurrentCycleTime() == 45){
        semaphore->getHorizontalTrafficLight()->setState(RED);
        semaphore->getVerticalTrafficLight()->setState(RED);
    }
    else if (semaphore->getCurrentCycleTime() == 47){
        semaphore->getHorizontalTrafficLight()->setState(RED);
        semaphore->getVerticalTrafficLight()->setState(GREEN);
    }
    else if (semaphore->getCurrentCycleTime() == 85){
        semaphore->getHorizontalTrafficLight()->setState(RED);
        semaphore->getVerticalTrafficLight()->setState(YELLOW);
    }
    else if (semaphore->getCurrentCycleTime() == 90){
        semaphore->getHorizontalTrafficLight()->setState(RED);
        semaphore->getVerticalTrafficLight()->setState(RED);
        semaphore->setCurrentCycleTime(0);
    }
}
```

Figura 6. Exemplo de código em PI.

O LingPON [12][13] é uma linguagem de descrição de software PON a partir da qual é possível gerar programas, em um nível mais baixo de abstração (p. ex C/C++ ou assembly), cuja estrutura e dinâmica são conformes ao modelo do PON. Assim, como as primeiras versões do C++ que eram baseadas em um código C com classes [16].

Para fazer a integração da estratégia desenvolvida em PON e o simulador desenvolvido em PI, mais especificamente em orientação a objetos, foram adicionadas ao projeto as classes geradas a partir da compilação do código em LingPon para C++. A classe *Semaphore_NOP*, gerada a partir da compilação do código LingPon, foi manualmente alterada para ser derivada da classe *Semphore* presente no simulador e suas *Rules* instanciadas a partir da instância de cada semáforo da região de simulação. Além das alterações no código gerado, foi utilizado o padrão de projeto Factory [17] para instanciar um semáforo corretamente, a partir da escolha da estratégia a ser analisada.

Na implementação da estratégia de controle baseada em congestionamento foi usada como referência a descrição da estratégia de controle descrita na Seção III B, no qual o tempo de abertura de um semáforo passa a depender do estado do sensor de tráfego e do tempo do semáforo. Para separar as estratégias de controle independente com a de controle baseado em congestionamento, foi desenvolvida uma nova aplicação em LingPon, sendo adicionada ao projeto de forma idêntica, através dos padrões de projeto. A Figura 7 mostra a implementação do *FBE* para o controle baseado em congestionamento. O *FBE Semaphore_NOP_CBCL* contém os mesmos *Attributes* do *FBE Semaphore_NOP* e foram adicionados *Attributes* dos sensores de congestionamento para os sinaleiros horizontais e verticais, além dos métodos que alteram seus estados.

```
fbe Semaphore_NOP_CBCL
    attributes
        integer atSeconds 0
        integer atSemaphoreState 5
        integer atHVSS 0
        integer atVVSS 0
    end_attributes

    methods
        method mtRT(atSeconds = 0)
        method mtHTLG(atSemaphoreState = 0)
        method mtHTLY(atSemaphoreState = 1)
        method mtHTLR(atSemaphoreState = 2)
        method mtVTLG(atSemaphoreState = 3)
        method mtVTLY(atSemaphoreState = 4)
        method mtVTLR(atSemaphoreState = 5)
        method mtHTLGCBCL(atSemaphoreState = 6)
        method mtHTLYCBCL(atSemaphoreState = 7)
        method mtVTLGCBCL(atSemaphoreState = 8)
        method mtVTLYCBCL(atSemaphoreState = 9)
    end_methods
end_fbe
```

Figura 7. Exemplo de *FBE* para controle baseado em congestionamento.

```
rule rlCBCL18
    condition
        subcondition sbCBCL18
                premise imp prSeconds10Full semaphore_NOP.atSeconds >= 18 and
                premise imp prSeconds210Full semaphore_NOP.atSeconds < 32 and
                premise prSemaphoreState10Full semaphore_NOP.atSemaphoreState == 3 and
                premise imp prVehicleSensorState10Full semaphore_NOP.atHVSS == 2
        end_subcondition
    end_condition
    action
        instigation inCBCL30 semaphore_NOP.mtVTLYCBCL();
        instigation inCBCL31 semaphore_NOP.mtRT();
    end_action
end_rule
```

Figura 8. Exemplo de *Rule* para controle baseado em congestionamento.

No desenvolvimento desta estratégia também foi necessário criar novas *Rules* para implementação do controle baseado em congestionamento em PON, sendo 18 o total de *Rules* implementadas e 8 dessas *Rules* possuem avaliações de congestionamento através dos sensores. A Figura 8 apresenta a sintaxe de uma *Rule* desenvolvida segundo abordagem de estratégia de controle baseado em congestionamento. Porém, há duas avaliações do *Attribute atSeconds* para que esteja com valor maior ou igual a 18 e menor do que 32 segundos, e o valor do *Attribute atVVSS* igual a 2, representando o sensor de congestionamento vertical igual a FULL e o *Attribute atSemaphoreState* que representa o estado atual, deve possuir valor 3. Quando todas as *Premises* forem verdadeiras, é efetuada a instigação do método *mtVTLYCBCL*, que é responsável pela transição para o estado 9 e do método *mtRT* que zera o tempo do semáforo.

Por questões de desempenho, as *Premises* que avaliam o tempo e o estado dos sensores só são pertinentes [8] caso esteja no estado desejado para que ocorra a transição. Assim, essas *Premises* só serão notificadas e avaliadas, caso a *Premise* que avalia o estado do semáforo esteja verdadeira. Ao compilar as aplicações desenvolvidas em LingPon foi necessário modificar manualmente os códigos equivalentes gerados em C++ para que fizessem uso da impertinência dos *Attributes* de tempo e sensor. Outra alteração necessária para o código PON apresentar melhora de desempenho foi alterar para que cada *Premise* seja única e compartilhada entre *Rules*. Desta forma foi possível eliminar redundâncias na avaliação do estado das *Premises*.

Para implementação da estratégia baseada em tráfego facilitado foi desenvolvido um novo diagrama de estados baseado no diagrama da Figura 3 e na descrição da estratégia na Seção III B. Para esta estratégia, foram implementadas 23 *Rules*, que avaliam o tempo, o estado atual, sensor de congestionamento e a *flag* de tráfego facilitado de cada semáforo. Porém, durante a implementação desta estratégia de controle, foram encontradas dificuldades no desenvolvimento devido à estrutura da implementação do código equivalente em C++. Desta forma, partiu-se então para uma estratégia um pouco diferente: os semáforos identificados pelo flag de tráfico facilitado, ao identificarem um congestionamento através de seus sensores atuam então igual à estratégia baseada em congestionamento e, ao mudarem seu estado para verde no sentido do tráfego facilitado, notificam o próximo semáforo que houve uma abertura antecipada. Este, por sua vez antecipa o fechamento do semáforo para receber então a onda de carros liberados pelo semáforo anterior. Cada semáforo então notifica o próximo que irá agir da mesma forma formando assim uma onda verde.

Na implementação da estratégia de controle descrita acima foi desenvolvida uma nova aplicação em LingPon, sendo adicionada ao projeto de forma idêntica, através dos padrões de projeto. A Figura 9 mostra a implementação do *FBE* para o controle baseado em tráfego faticilitado . O *FBE Semaphore_NOP_CBTF* contém os mesmos *Attributes* do *FBE Semaphore_NOP_CBCL* e foram adicionados *Attributes* de identificação de tráfego facilitado e de reconhecimento que a onda verde precisa ser propagada. Esta segunda utilizada na comunicação entre um semáforo se seu sucessor na sequencia da rua.

```
fbe Semaphore_NOP_CBTF
    attributes
        integer atSeconds 0
        integer atSemaphoreState 5
        integer atHVSS 0
        integer atVVSS 0
        integer atGWType 0
        integer atGWPropagation 0
    end_attributes

    methods
        method mtRT(atSeconds = 0)
        method mtHTLG(atSemaphoreState = 0)
        method mtHTLY(atSemaphoreState = 1)
        method mtHTLR(atSemaphoreState = 2)
        method mtVTLG(atSemaphoreState = 3)
        method mtVTLY(atSemaphoreState = 4)
        method mtVTLR(atSemaphoreState = 5)
        method mtHTLGCBTF(atSemaphoreState = 6)
        method mtHTLYCBTF(atSemaphoreState = 7)
        method mtVTLGCBTF(atSemaphoreState = 8)
        method mtVTLYCBTF(atSemaphoreState = 9)
        method mtRGWPROP(atGWPropagation = 0)
        method mtPHGW() begin_method Semaphore_NOP_CBTF->PropagateHorizontalGreenWave(); end_method
        method mtPVGW() begin_method Semaphore_NOP_CBTF->PropagateVerticalGreenWave(); end_method

    end_methods
end_fbe
```

Figura 9. Exemplo de *FBE* para controle baseado em tráfego facilitado.

```
rule rlCBTF140
    condition
        subcondition sbCBTF140
                premise imp prSeconds8Full semaphore_NOP.atSeconds >= 18 and
                premise imp prSecondsSup8Full semaphore_NOP.atSeconds < 32 and
                premise prSemaphoreState8Full semaphore_NOP.atSemaphoreState == 0 and
                premise prGWType2 semaphore_NOP.atGWType == 2 and
                premise prGWProp2 semaphore_NOP.atGWPropagation == 2
        end_subcondition
    end_condition
    action
        instigation inCBTF33 semaphore_NOP.mtHTLYCBTF();
        instigation inCBTF34 semaphore_NOP.mtRT();
    end_action
end_rule
```

Figura 10. Exemplo de *Rule* para controle baseado em tráfego facilitado.

No desenvolvimento desta estratégia também foi necessário criar novas *Rules* para implementação do controle baseado em tráfego facilitado em PON, sendo 22 o total de *Rules* implementadas, 8 dessas *Rules* possuem avaliações de congestionamento através dos sensores e 4 possuem avaliações de propagação de onda observando sua condição de onda verde e a notificação pelo semáforo anterior. Para implementar a notificação de semáforo foi necessário utilizar-se de codificação em OO, desta forma os métodos de propagação *mtPHGW* e *mtPVGW* foram definidos como chamada de um método OO *PropagateHorizontalGreenWave* e *PropagateVerticalGreenWave* respectivamente. Ambos os métodos são utilizados tanto em OO quanto em PON.

A Figura 10 apresenta a sintaxe de uma *Rule* desenvolvida segundo abordagem de estratégia de controle baseado em em tráfego facilitado. Porém, há duas avaliações do *Attribute atSeconds* para que esteja com valor maior ou igual a 18 e menor do que 32 segundos, e o valor do *Attribute atGWType* igual a 2, representando sua atribuição como onda verde de propagação horizontal e o *Attribute atGWPropagation* que representa que o semáforo anterior o notificou para prosseguir com a onda verde na direção horizontal. Quando todas as *Premises* forem verdadeiras, é efetuada a instigação do método *mtHTLYCBTF*, que é responsável pela transição para o estado 9 e do método *mtRT* que zera o tempo do semáforo.

Tal como na estratégiade controle baseada em congestionamento, as *Premises* que avaliam o tempo, o estado dos sensores e a notificação dos semáforos só são pertinentes

[8] caso esteja no estado desejado para que ocorra a transição. Assim, essas *Premises* só serão notificadas e avaliadas, caso a *Premise* que avalia o estado do semáforo esteja verdadeira. Ao compilar as aplicações desenvolvidas em LingPon, tal como na estratégia anterior, foi necessário modificar manualmente os códigos equivalentes gerados em C++ para que fizessem uso da impertinência dos *Attributes* de tempo e sensor. A unicidade das Premisses também foi preservada nesta estratégia a fim de eliminar redundâncias tal como na estratégia anterior.

IV. RESULTADOS

Os experimentos realizados tiveram como objetivo analisar o desempenho das estratégias de controle implementadas em PON, comparadas as implementações em PI. Para isso, foram desenvolvidas as estratégias de controle independente e baseado em congestionamento descrito anteriormente, em ambos os paradigmas. Para executar os experimentos foi utilizado um notebook com processador core i5 – 4210U 1.70GHz e 4GB de memória RAM, com sistema operacional Windows 8.1 Pro. O sistema operacional Windows 8.1 foi utilizado devido ao ambiente de desenvolvimento escolhido.

Para análise dos experimentos de desempenho, a aplicação foi executada alterando o número de repetições: 100, 200, 500, 1000, 1500 e 2000, na qual cada repetição representa 1 segundo do semáforo. A Tabela I e a Figura 11 apresentam os resultados dos experimentos executados, que correspondem ao tempo de execução em milissegundos das duas estratégias implementadas, alterando o número de repetições.

TABELA I. RESULTADOS RELATIVOS AO EXPERIMENTO.

Strategy	100	200	500	1000	1500	2000
Independent PI	17278	27567	63437	134689	280263	423543
Independent PON	26565	40792	75093	169352	307110	507818
CBCL PI	23998	41889	70984	175990	424132	390994
CBCL PON	36327	53908	102962	181491	455348	495938
CBTF PI	23521	44383	82599	143157	423079	546218
CBTF PON	40766	52559	114735	180282	474813	635587

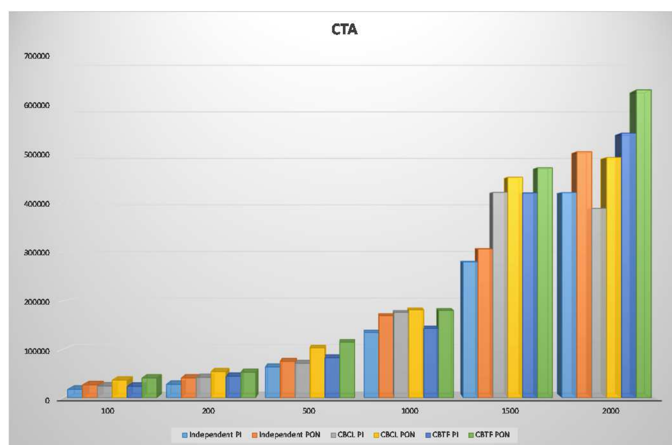

Figura 11. Gráfico dos resultados relativos ao experimento.

A partir dos dados da Tabela I e da Figura 9, observou-se que a implementação em PON teve desempenho inferior quando comparado ao da implementação em PI. Porém atualmente existem nova materializações PON que têm se mostrado muito mais eficientes que a implementação utilizada neste experimento que, apesar de ter controle por impertinências implementado, ainda apresentou um desempenho inferior ao seu equivalente em OO.

V. CONCLUSÕES E TRABALHOS FUTUROS

O PON apresenta uma abordagem para o desenvolvimento mais eficiente de softwares em comparação aos paradigmas tradicionais. Este paradigma se propõe a resolver problemas tais como redundâncias estruturais, temporais e forte acoplamento entre as suas entidades, visando facilitar o desenvolvimento de softwares paralelos e distribuídos.

Este trabalho apresentou o desenvolvimento de uma aplicação para a comparação de desempenho entre software desenvolvido em PON e software desenvolvido em PI (POO). Apesar de comparações de desempenho ainda inferiores, a avaliação dos resultados permite elaborar conclusões a respeito da complexidade de desenvolvimento sob este novo paradigma.

Quando realizados os experimentos comparando a implementação da aplicação CTA Simulator em ambos os paradigmas, a implementação em PI se demonstrou mais eficiente do que a implementação equivalente em PON, porém já existem estudos onde há melhora na materialização do código gerado PON em uma versão C++ estática que apresenta melhoras significativas de performance de execução em comparação com a versão utilizada neste experimento com o LingPon.

Com relação à complexidade de desenvolvimento, no PON as *Rules* do sistema podem ser visualizadas de maneira mais natural, pois o nível de abstração das *Rules* é maior, quando comparado às abstrações das mesmas transições desenvolvidas em linguagem imperativa. Porém, o estado da técnica do compilador LingPon ainda está em desenvolvimento, logo ocorreram algumas dificuldades do ponto de vista expressão das *Rules*, tais como a falta de suporte da linguagem a estruturas de dados e à declaração de um *FBE*, como *Attribute* de outro *FBE*.

Mesmo apresentando algumas dificuldades o uso da linguagem LingPon é promissor, pois com o avanço no estado da técnica desta linguagem, como otimizações do compilador, inclusão de estruturas de dados, determinismo e identificação de erros pelo compilador, será possível gerar código de maneira mais eficiente, com possível distribuição e paralelismo, facilidade de programação e menor tempo de desenvolvimento independentemente da plataforma na qual será executado. No estado da técnica atual já é possível gerar códigos C e C++, seguindo o modelo PON. Além das linguagens já implementadas, é possível futuramente gerar códigos para outras linguagens, como Java, C#, NOCA[11] e PON HD (VHDL) [9].

Ademais o PON mantém características dos Sistemas Orientados a Regras (SOR), mais especificamente o SBR. Dado o fato que os SORs são comumente utilizados no âmbito de aplicações de Inteligência Computacional e Inteligência Artificial. O PON surge como uma nova abordagem para este tipo de aplicação, mantendo o mesmo nível de abstração de programação em alto nível dos SORs e o mesmo desempenho do PI [7]. Neste sentido, o PON começa a ser aplicado a áreas como Sistemas Especialistas, Fuzzy e Redes Neurais [18].

Como sugestão para trabalho futuro fica a possibilidade de criação de método de controle utilizando as materializações mais recentes que têm se mostrado mais eficientes no âmbito de eficiência de execução. Um exemplo deste é a materialização em C++ estático que têm apresentado melhora considerável frente à materialização utilizada neste experimento.

REFERÊNCIAS

[1] G. Moore, Cramming More Components onto Integrated Circuits. Eletronics Magazine , 1965.

[2] F. Eschmann, B. Klauer, R. Moore, and K. Waldschmidt, “SDAARC : An Extended Cache-Only Memory Architecture,” Micro, IEEE, vol. 22, no. 3, pp. 62–70, 2002.

[3] S. Borkar and A. A. Chien, “The Future of Microprocessors”, in: Communications of the ACM, 54(5), p. 67-77, 2011. DOI 10.1145/1941487.

[4] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, “The Landscape of Parallel Computing Research: A View from Berkeley,” 2006.

[5] J. M. Simão and P. C. Stadzisz, “Inference Process Based on Notifications: The Kernel of a Holonic Inference Meta-Model Applied to Control Issues”. IEEE Trans. on Systems, Man and Cybernetics. Part A, Systems and Humans, V.39, I.1, 238-250, 2009. DOI 10.1109/TSMCA.2008.2006371.

[6] J. M. Simão and P. C. Stadzisz, “Paradigma Orientado a Notificações (PON) - Uma Técnica de Composição e Execução de Software Orientada a Notificações”. Patent pending submitted to INPI/Brazil in 2008 and UTFPR Innovation Agency 2007. INPI Number: PI0805518-1. http://www.patentesonline.com.br/paradigma-orientado-anotificacoes--uma-tecnica-de-composicao-e-execucao-de-software-234943.html.

[7] R. F. Banaszewski, "Paradigma Orientado a Notificações: Avanços e Comparações". M. Sc. Thesis, CPGEI/UTFPR. Curitiba- PR, Brazil 2009. http://arquivos.cpgei.ct.utfpr.edu.br/Ano-2009/dissertacoes/Dissertacao-500-2009.pdf

[8] A. F. Ronszcka, Contribuição Para a Concepção de Aplicações no Paradigma Orientado a Notificações (PON) Sob o Viés de Padrões. 2012. Dissertação de Mestrado, CPGEI, UTFPR. Curitiba, Brasil, 2012.

[9] J. M. Simão, R. R. Linhares, F. A. Witt, C. R. E. Lima and P. C. Stadzisz, “Paradigma Orientado a Notificações em Hardware Digital”. Patent pending submitted to INPI/Brazil in 2012 and UTFPR Innovation Agency (2012). INPI Provisory Number: BR 10 2012026429 3.

[10] E. Peters, Co-processor to Speed up of Application developed under the Notification Oriented Paradigm. Original title in Portuguese: Coprocessador para Aceleração de Aplicações Desenvolvidas utilizando Paradigma Orientado a Notificações. M. Sc. Thesis, CPGEI/UTFPR. Curitiba-PR, Brazil 2012

[11] R. R. Linhares, J. M. Simão and P. C. Stadzisz, "NOCA A Notification-Oriented Computer Architecture," Latin America Transactions, IEEE (Revista IEEE America Latina), vol.13, no.5, pp.1593,1604, May 2015. Doi: 10.1109/TLA.2015.7112020

[12] R. D. Xavier, “Paradigmas de desenvolvimento de software: comparação entre abordagens orientada a eventos e orientada a notificações”. M. Sc. Thesis, PPGCA/UTFPR. Curitiba-PR, Brazil 2014

[13] C. A. Ferreira. Linguagem e compilador para o paradigma orientado a notificações (PON): Avanços e comparações. Seminário de Acompanhamento, PPGCA, UTFPR. Curitiba, Brasil, 2014.

[14] D. P. B. Renaux, R. R. Linhares, J. M. Simão, P. C. Stadzisz , “CTA_CONOPS”, Available in: < http://www.dainf.ct.utfpr.edu.br/~douglas/CTA_CONOPS.pdf>, 2014,Accessed in July15th, 2015.

[15] J. G. Brookshear, “Computer Science: An Overview”. Addison Wesley, 2006.

[16] B. Stroustrup, The C++ Programming Language. 4th Edition. Addison-Welsey. 2013.

[17] Gamma, Erich. Padrões de projeto: soluções reutilizáveis de software orientado a objetos. Porto Alegre: Bookman, 2000 xii,364 p. ISBN 8573076100.

[18] L. C. V. Melo, J. M. Simão, and J. A. Fabro, “Adaptation of the Notification Oriented Paradigm (NOP) for the Development of Fuzzy Systems,” Mathw. Soft Comput., vol. 22, no. 1, pp. 40–64, 2015.