LingPON 2.0 & compilador para solução PON em C++ orientado a espaço de nomes

Larissa Keiko Oshiro, (L. F. Pordeus, A. F. Ronszcka), [J. A. Fabro, J. M. Simão]
CPGEI / UTFPR
Avenida Sete de Setembro, 3165
Curitiba-PR - CEP 80.230-910
E-mail: lari.oshiro@gmail.com

***Resumo*—Um novo paradigma de programação é, atualmente, objeto de estudo e seus fundamentos têm sido materializados nas áreas de softwares e hardwares. Trata-se do Paradigma Orientado a Noticações (PON), cuja técnica une algumas das características de dois paradigmas dominantes (flexibilidade de programação do Paradigma Imperativo e facilidade de programação do Paradigma Declarativo), além de trazer uma nova visão de programar, estruturar e executar. Essa nova técnica visa diminuir as redundâncias temporais e estruturais, aumentando o desempenho de execução e obter um acoplamento mínimo entre os módulos do software, podendo ser utilizada em sistemas multiprocessados e distribuídos. Para materializar os princípios do PON, foi desenvolvida uma linguagem de programação específica, denominada LingPON. O LingPON tem sido alvo de diversos estudos e atualmente está no processo de desenvolvimento da nova versão, LingPON 2.0. Para validar o LingPON 2.0, estão sendo realizados estudos de implementação de compilação desta nova linguagem em diversas plataformas, dentre softwares e hardwares. Este trabalho apresenta a implementação de geração de código do LingPON 2.0 em C++, utilizando namespaces. Como resultado, foi possível implementar códigos em namespaces a partir do LingPON 2.0, mantendo tanto a integridade das características do novo paradigma de programação, bem como o desempenho de execução proposto pela nova técnica.**

Palavras chaves - Paradigma orientado a noticações, LingPON 2.0, PON, Namespaces C++.

I. Introdução

No campo do desenvolvimento de software, cada linguagem de programação pode ser classificada, de acordo com suas características (como funcionalidades, estruturação e modo de execução), em um ou mais paradigmas de programação. Atualmente, há dois maiores paradigmas de programação, o Paradigma Imperativo (PI) e o Paradigma Declarativo (PD). No entanto, essas técnicas apresentam alguns aspectos que poderiam ser melhorados, em se tratando de desempenho na execução, a programação um pouco complexa do PI ou as funcionalidades de codificação do PD e a dificuldade em utilizar multiprocessamentos por não haver um desacoplamento entre os módulos do software [1].

Essas deficiências dos atuais paradigmas foram motivações para o desenvolvimento de uma nova técnica chamada Paradigma Orientada a Notificações (PON). O PON propõe uma nova forma de programação, orientada a regras e notificações, visando melhorar o desempenho na execução do programa, facilitar a programação, além de proporcionar o desacoplamento de código [2] [3].

A fim de validar as características fundamentais do PON, foi desenvolvida uma linguagem de programação específica denominada LingPON. Essa linguagem de programação já foi alvo de pesquisas e atualmente sua nova versão se encontra em desenvolvimento (LingPON 2.0).

Nas sessões seguintes, serão apresentadas com mais detalhes abordagens sobre o PON e sua respectiva linguagem de programação, LingPON.

A. Paradigma Orientado a Notificações (PON)

O Paradigma Orientado a Notificações corresponde a um paradigma emergente que foi desenvolvido com o objetivo de amenizar alguns problemas dos paradigmas dominantes, o Paradigma Imperativo (PI) e o Paradigma Declarativo (PD). Não apenas traz soluções para as deficiências desses paradigmas vigentes, como também apresenta algumas de suas vantagens (programação orientada a objetos do PI e sistemas baseados em regras do PD). Assim, é uma técnica que une a facilidade de programação do declarativo e a flexibilidade de programação do imperativo.

O PON é constituído por dois conjuntos de entidades de processamento: o processamento facto-execucional e o processamento lógico-causal. O primeiro conjunto corresponde a entidades notificantes, os chamados Elementos da Base de Fatos (FBE – *Fact Base Elements*, em inglês). Já o segundo conjunto corresponde às entidades denominadas Regras ou Rules, que receberão notificações pelos FBE [3]. Por meio da Figura 1 é possível visualizar melhor a ideia do funcionamento do PON e a interação entre os dois conjuntos de processamento.

No Paradigma Imperativo, o laço de iteração é escrito de forma explícita por meio dos comando *if* e *while*. No entanto, isso acaba proporcionando redundâncias temporais e estruturais, o que acarreta consequentemente em um processamento desnecessário, afetando o desempenho na execução [1] [4]. Em contrapartida, no PON a repetição ocorre apenas se for identificada a mudança de estado em algum *Attribute*, ou seja, a repetição é executada apenas quando for efetivamente necessário.

Como pode-se observar na Figura 1, quando é identificada a mudança de estado de um *Attribute* de um determinado

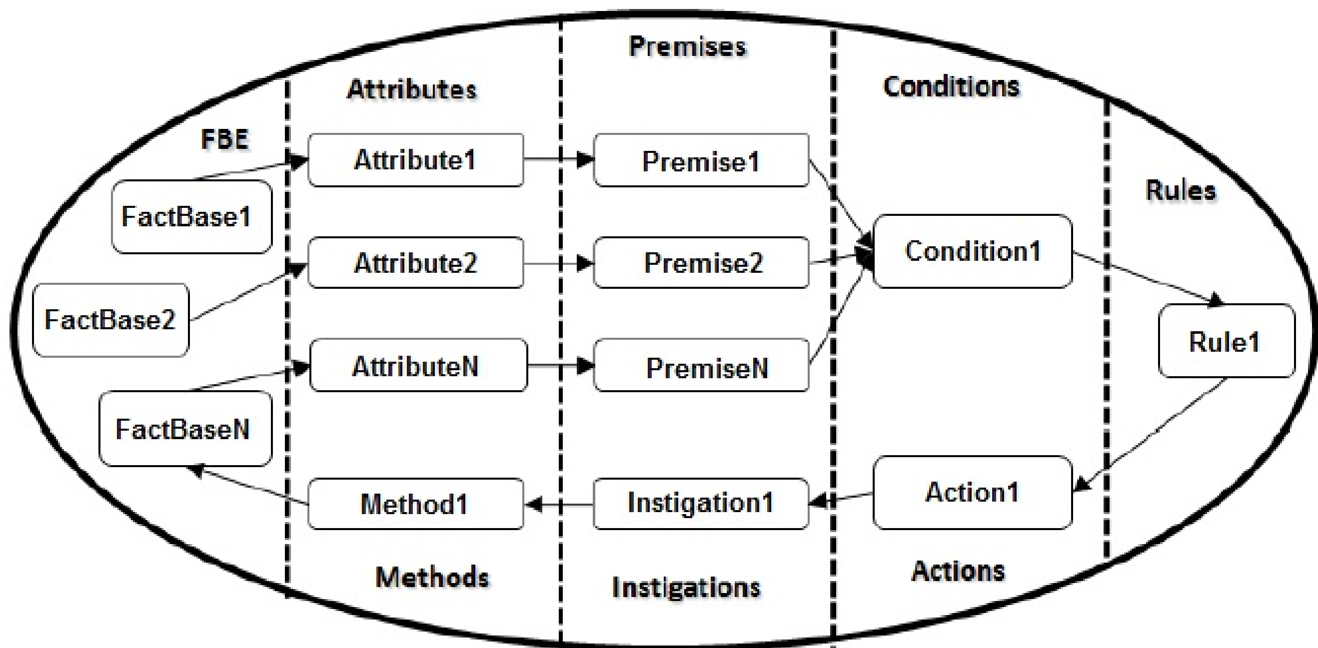

Figura 1. Exemplo da interação entre as entidades do PON. Fonte: Adaptado de [1].

FBE, automaticamente o *Attribute* notifica as *Premises* correspondentes, e apenas essas *Premises*, para que possam ser feitas avaliações de seus estados lógicos. Uma vez que o valor lógico dessa *Premise* é modificada, a *Premise* notifica suas respectivas *Conditions*. A satisfação de uma *Condition* é feita pela avaliação dos valores lógicos de um conjunto de *Premises*. Se os estados de todas as *Premises* correspondentes são satisfeitos, a *Condition* é aprovada, ativando consequentemente sua respectiva Rule. Esta, por sua vez, executará sua *Action*, que é uma entidade computacional conectada a um ou mais *Instigations*. Como o próprio nome sugere, um *Instigation* instiga ou ativa, por meio de seus *Methods*, um serviço ou funcionalidade de um FBE. Geralmente, os *Methods* acabam por alterar os estados dos *Attributes*, reativando, consequentemente, o fluxo de notificações [1]–[6].

O surgimento do PON veio a acrescentar muito no âmbito de sistemas computacionais, visto que se trata de uma nova visão para estruturar, executar e desenvolver software. Por se inspirar no Sistema Baseado em Regras, possibilita a programação em alto nível baseada em regras, além de possibilitar a execução distribuída. O Paradigma Orientado a Notificações já foi foco de algumas pesquisas na área de software e, mais recentemente, tem sido objeto de estudo em sistemas de hardware digital. Isto é um indício de que há um campo amplo de pesquisas sobre o PON, podendo ser utilizado em diversas outras áreas.

Ainda no campo de desenvolvimento de software, para qualificar os fundamentos do Paradigma Orientado a Notificações, uma linguagem de programação denominada LingPON foi criada e atualmente encontra-se na terceira versão. O próximo tópico trata brevemente sobre a evolução do LingPON e sua contribuição para o desenvolvimento de sistemas computacionais.

B. LINGPON

Conforme descrito anteriormente, foram desenvolvidos uma linguagem de programação e um compilador especificamente para o PON. O LingPON, como é denominada tal linguagem de programação, vem sendo estudado e aprimorado a cada versão. A Tabela 1 apresenta uma relação dos avanços alcançados em cada versão do LingPON, os quais serão brevemente descritos a seguir.

Sua evolução iniciou com um protótipo, no qual já eram encontrados nitidamente os fundamentos do PON, (coluna ”Prot.”na Tabela 1). De forma sucinta, a estrutura do código fonte da LingPON é baseada em declarações, sendo o programa constituído por três blocos principais que correspondem à definição de FBEs, às instanciações de FBEs e à declaração das *Rules* [3].

O LingPON 1.0 foi a primeira versão ”oficial”dessa linguagem de programação e com ela já conseguiu-se materializar algumas aplicações do PON, como mostra a Tabela 1. Uma dessas aplicações seria o conceito de *Formation Rules* (Regras de Formação), que consiste na possibilidade de criar regras específicas a partir de uma *Rule* genérica. Em casos em que diversas regras apresentam a mesma estrutura, porém se diferenciam apenas por serem referenciadas por instâncias de FBEs diferentes, o uso da Regra de Formação é muito viável. Outro

avanço dessa versão do LingPON foi a implementação do conceito de Entidades impertinentes. bloqueando as notificações de *Attibute* impertinente para uma determinada *Premise*. Isto impacta no desempenho da execução, uma vez que evita notificações desnecessárias. Notou-se que o LingPON apresentava algumas limitações referentes ao encapsulamento de Rules internas aos FBEs e agregação de FBEs. Para solucionar isso, o LingPON 1.0 possibilita criar agregações entre FBEs, a fim de que esses possam fazer referência a outros FBEs, aumentando, dessa forma, o encapsulamento de *Methods* e *Attributes*.

Porém, mesmo que se tenha alcançado bons resultados na questão de desempenho, em termos de facilidade de programação ainda não trouxe avanços consideráveis [4]. Neste sentido, uma nova versão, LingPON 2.0, foi desenvolvida.

Dentre os avanços que a versão 2.0 do LingPON apresenta, se destaca a possibilidade de paralelismo. Para isso, os elementos que executaam de forma paralela são modelados como *threads*. Por consequência desse paralelismo e do desaclopamento, característico do PON, pode haver conflitos quando, por exemplo, duas *Rules* em execuções diferentes dependem de um recurso em comum (compartilhado) em um mesmo instante, sendo que esse recurso compartilhado deve ser acessado exclusivamente por uma das *Rules* naquele momento. Visando a esse ponto, o LingPON 2.0 apresenta um escalonador de *Rules*, que consiste em uma estrutura de dados linear (como por exemplo, uma lista) que guarda as *Rules* à medida que são aprovados. E essas *Rules* são executadas de acordo com o tipo de estratégia de escalonamento pré-definido pelo desenvolvedor. Além disso, a versão 2.0 permite que um FBE apresente em sua estrutura Rules internas [3] [4].

O PON tem sido estudado para aplicações não somente em software, como também em hardware digital. Dessa forma, surgiu a oportunidade de desenvolver uma linguagem universal que materialize os fundamentos do PON e possibilite uma codificação menos complexa em alto nível [3].

Neste sentido, o LingPON vem sendo atualmente estudado e estruturado em uma nova versão, LingPON 2.0. A proposta dessa nova versão é proporcionar uma linguagem universal, completa e mais efetiva para que possa ser base de qualquer sistema PON, independente da plataforma [3]. Além disso, essa versão segue um modelo uniforme de mapeamento de entidades PON, denominado Grafo PON, possibilitando, assim, a construção de compiladores para plataformas distintas. Esse mapeamento permite traduzir o código fonte por meio de um grafo único, o qual apresenta as conexões baseadas em notificações entre as entidades do PON, bem como as características particulares de cada entidade [3]. Essa versão LingPON 2.0 atualmente se encontra em processo de desenvolvimento e, para validá-la, está sendo alvo de diversos estudos, assim como foi feito com o LingPON 1.0: utilizando o compilador PON é possível gerar código em diversas linguagens-alvo (como Framework C++, C, java e namespaces C++) a partir de um código-fonte em LingPON. Em relação à versão 1.0, o C++ com namespaces foi a linguagem-alvo que obteve

	Prot.	1.0	2.0
Entidades Reativas	X	X	X
Desempenho	X	X	X
Programação de alto nível	X	X	X
Entidades impertinentes		X	X
Regras de Formação		X	X
FBE Agregador		X	X
Propriedades Reativas dos *Attributes*		X	X
Compartilhamento de Entidades			X
Escalonamento de Rules			X
Paralelismo			X

Tabela I
AVANÇOS AO LONGO DA EVOLUÇÃO DO LINGPON. FONTE: ADAPTADO DE [1]

os melhores resultados, por conta da eficiência de execução em ambientes monoprocessados, além da clareza do código gerado. Portanto, o objetivo deste trabalho é ter o namespaces C++ como linguagem alvo para a implementação de geração de código a partir do LingPON 2.0.

II. MATERIAIS E MÉTODOS

A linguagem de programação PON surgiu para materializar o Paradigma Orientado a Notificações. No entanto, desde sua versão preliminar vem sendo melhorada a cada versão e este presente trabalho tem o objetivo de contribuir para o estudo e desenvolvimento da linguagem em questão, implementando a geração de códigos a partir do LingPON 2.0 para o C++ com a utilização de namespaces.

Namespaces seria basicamente uma maneira de organizar itens (classes, enumerações, estruturas, etc.) de forma lógica. O uso de namespaces pode ser bastante útil e positivo em se tratando de programação com orientação a objetos. Isto porque trata-se de funções, estruturas ou classes utilizadas de forma a organizar a estrutura do código em grupos lógicos. À medida que um projeto se desenvolve e cresce, naturalmente aumenta o número de classes e estruturas. Dessa forma, a necessidade de cuidar para evitar a ambiguidade ao nomear esses elementos aumenta. Além disso, o uso de bibliotecas aumenta a possibilidade de ocorrer o problema de ambiguidade.É possível evitar esses conflitos de uma maneira prática usando namespaces. Por exemplo, uma mesma aplicaçao pode fazer referência a duas biblioteca, sendo que cada uma contém uma classe com o mesmo nome ”Símbolo”, porém usada para manipular tipos de elementos (enumerações, funções ou variáveis) diferentes. A probabilidade de gerar conflitos de ambiguidade é grande, caso a gestão do código não seja bem articulada. Nesse sentido, é possível associar cada classe ”Símbolo”a um namespace para evitar tais conflitos.

A. *LingPon 2.0 e Implementação Namespaces C++*

Como já mencionado anteriormente, o PON obedece à lógica de notificações entre suas entidades. Dessa forma, para a geração de códigos em namespace, cada uma dessas entidades do PON foi considerada e materializada em uma classe namespace a fim de preservar a dinâmica da lógica de notificações, conforme mostra a Figura 2.

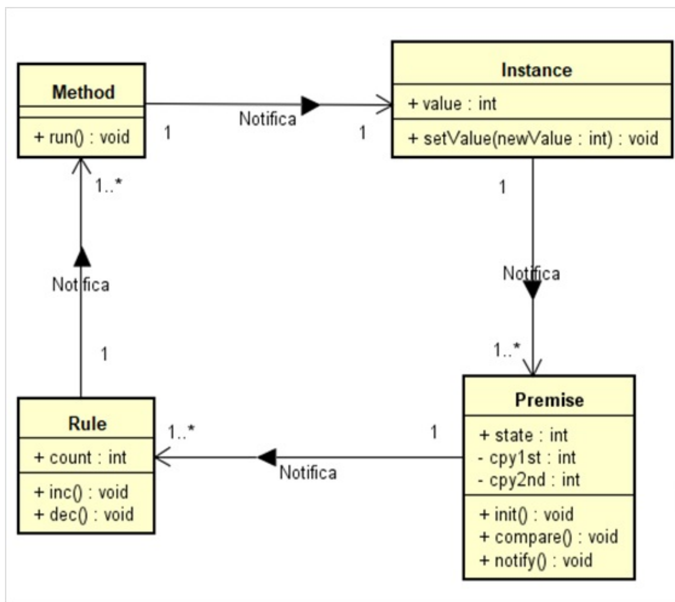

Figura 2. Materialização da geração de código do LingPon em namespaces C++

A Figura 2 mostra o ciclo de notificações entre as entidades *Instance*, *Premise*, *Rule* e *Method*. Reiterando o que foi dito na sessão anterior, o ciclo de notificações é disparado quando é identificada uma alteração no valor de um *Attribute* de um determinado FBE. O elemento *Attribute* notifica todas as suas *Premises* relacionadas para que os estados lógicos destas sejam avaliados. O conjunto de *Premises* correspondem a uma *Condition* ou *Subcondition*. Uma vez que todas as *Premises* de todas as *Subconditions* ou *Condition* existentes em um FBE forem obedecidas, isto significa que uma *Rule* foi satisfeita (e, conforme a lógica PON, notificada). Por conseguinte, automaticamente o *Method* correspondente à *Rule* satisfeita é notificado e executado. Geralmente, a execução dos *Methods* resulta na alteração de algum *Attribute*, reiniciando assim novamente o ciclo de notificações. A implementação da geração de código em namespaces C++ obedeceu a lógica de notificações entre essas entidades do PON. Portanto, em um mesmo arquivo (namespace), chamado "*Instances*", foram agrupadas todas as *Instances* de um FBE (instâncias de atributos e instâncias de outros FBEs), no arquivo (namespace) "*Premises*" se encontram todas as *Premises* dos FBEs, no arquivo (namespace) "*Rules*" se estão todas as *Rules* a serem aprovadas e, por fim, no arquivo (namespace) "*Methods*" foram agrupados todos os *Methods* a serem executados.

O programa .pon que foi alvo para a geração de código em namespaces C++ foi um projeto chamado "Sensors", cuja estrutura apresenta elementos do LingPON 2.0 que não estavam contidos na versão 1.0 da linguagem (como por exemplo, inclusão de bibliotecas externas, encapsulamento de *Rules*, interação entre FBEs). O processo de implementação em namespaces C++ foi executado em duas etapas: na primeira etapa foi implementada a geração de código para apenas um FBE (contendo apenas uma *Instance*, uma *Premise* e um *Method*) e, após a primeira etapa ter sido concluída e seu respectivo código gerado em namespaces C++ estar executando corretamente, o trabalho passou para a segunda etapa, onde foi implementada a geração de código para vários FBEs, contendo assim interação entre eles. Para a implementação da primeira etapa, foi utilizado o FBE "Main.nop" (representado pelo Código Fonte 1) do projeto Sensors e para a segunda etapa, todo o projeto Sensors foi implementado.

```
fbe Main

    includes FRAMEWORK_CPP_2_0
        #include "SMSSender.h"
        #include <iostream>
        using namespace std;
    end_includes

    private Sector sectorA
    private Sector sectorB

    private method mtSendSms
        params
            String cellphone
        end_params
        code FRAMEWORK_CPP_2_0
            SMSSender *sender = new SMSSender
();
            sender->send(cellphone);
        end_code
    end_method

    rule rlInvasionDetection
        condition
            premise prSectorAInvaded
                sectorA.atIntruderDetected ==
true
            end_premise
            or
            premise prSectorBInvaded
                sectorB.atIntruderDetected ==
true
            end_premise
        end_condition
        action sequential
            instigation
                call this.mtSendSms("
41-999999999", "47-999999999")
            end_instigation
        end_action
    end_rule

    properties
        strategy PRIORITY
    end_properties

end_fbe
```

Código Fonte 1. Arquivo fonte Main.nop em LingPON 2.0 utilizado para a geração de código em namespaces C++.

Os tópicos seguintes descrevem com mais detalhes a implementação e o funcionamento de cada namespace, bem como sua implementação na etapa final do processo.

B. *Instances*

O namespace Instances (.h e .cpp) agrupa todas as instâncias dos FBEs presentes em um projeto em LingPON. Instâncias de FBEs abrangem tanto as instâncias de atributos, como instâncias de outros FBEs (e, consequentemente, seus respectivos atributos).

```
fbe Main

    includes FRAMEWORK_CPP_2_0
        #include "SMSSender.h"
        #include <iostream>
        using namespace std;
    end_includes

    private Sector sectorA
    private Sector sectorB
```

Código Fonte 2. Declaração de Instances do FBE Main.nop

No Código Fonte 2, as linhas 9 e 10 apresentam declarações de duas instâncias do FBE Sector no FBE Main. Por sua vez, o Código Fonte 3 mostra que no FBE Sector há instância de um *Attribute* (linhas 3), denominado "atIntruderDetected", e instâncias de outros FBEs (FBE Alarm, FBE Sensor e FBE Siren), declaradas nas linhas 5 a 14.

```
fbe Sector

    private Boolean atIntruderDetected = false

    private Alarm alarmA
    private Alarm alarmB

    private Siren sirenA1
    private Siren sirenA2
    private Siren sirenB1

    private Sensor sensorA1
    private Sensor sensorA2
    private Sensor sensorB1
```

Código Fonte 3. Declaração de Instances do FBE Sector.nop

O código gerado em namespaces C++ pode ser visualizado no Código Fonte 4. É possível observar que, dentro do namespace global denominado "*Instance*"é gerado um namespace próprio para cada instância de FBE com seus respectivo *Attribute*. A atribuição de valores dos *Attributes* é feita acessando o namespace do método SetValue() de cada atributo. Para alterar o valor do *Attribute* da instância "AlarmA", por exemplo, deve-se executar a chamada *instance::SectorA::AlarmA::at::atStatus::setValue()*.

```
#pragma once
#include <string>
namespace instance{
  namespace sectorA{
    namespace at{
      namespace atIntruderDetected{
        extern bool value;
        extern void setValue(bool newValue);
      }
    }
    namespace alarmA{
      namespace at{
        namespace atStatus{
          extern bool value;
          extern void setValue(bool newValue);
        }
      }
    }
    namespace alarmB{
      namespace at{
        namespace atStatus{
          extern bool value;
          extern void setValue(bool newValue);
        }
      }
    }
    namespace sensorA1{
      namespace at{
        namespace atState{
          extern bool value;
          extern void setValue(bool newValue);
        }
      }
    }
    namespace sensorA2{
```

Código Fonte 4. Trecho do arquivo Instance.h gerado pela implementação em namespaces

C. *Premises*

O arquivo namespace *Premise* (.h e .cpp) apresenta as *Premises* de todos os FBEs de um projeto implementado em LingPON. Conforme mostra o Código Fonte 5, cada *Premise* contém duas vaiáveis (cpy1st e cpy2nd) utilizadas para fazer a comparação (dentro do método compare()) e averiguar se a *Premise* em questão foi satisfeita (variável *state*). Em caso afirmativo, é feita uma chamada do método notify(), o qual é responsável por notificar a *Condition* (ou *SubCondition*) correspondente.

```
#include "premises.h"
#include "rules.h"
#include <string>

        #include "SMSSender.h"
        #include <iostream>
        using namespace std;

namespace premise{
  namespace sectorA{
    namespace prAlarmAOn{
      bool state = false;
      bool cpy1st, cpy2nd;
      void init(){
        cpy1st = 0;
        cpy2nd = 1;
      }
      void compare(){
        if(cpy1st  ==  cpy2nd){
          if(state == false){
            state = true;
            rule::sectorA::rlFireAlarmA::incl
    ();
          }
        }else{
          if(state == true){
```

```
            state = false;
            rule::sectorA::rlFireAlarmA::dec1
();
          }
        }
      }
      void notify_alarmA_atStatus(bool
newValue){
        cpy1st = newValue;
        compare();
      }
      // Notified by attributes: [ atStatus ]
```

Código Fonte 5. Trecho do arquivo Premise.h gerado pela implementação em namespaces

D. Rules

A compilação da lógica do namespace *Rule* foi a mais complexa. Isto porque uma *Rule* é composta por uma *Condition*. Porém uma *Condition* pode ser composta por diversas *Subconditions* (que corresponde a um conjunto de *Premises*), ou pode ser composta diretamente por um conjunto de *Premises* ("pulando"o nível *Subcondition*).

O Código Fonte 6 mostra um exemplo de *Rule* composta por apenas *Condition* e *Premises* (*rule* rlInvasionDetection do FBE Main) e um exemplo de *Rule* composta também de *Subcondition*, além de *Condition* e *Premises* (*rule* rlFireAlarmA do FBE Sector).

```
fbe Main
    rule rlInvasionDetection
        condition
            premise prSectorAInvaded
                sectorA.atIntruderDetected ==
    true
            end_premise
            or
            premise prSectorBInvaded
                sectorB.atIntruderDetected ==
    true
            end_premise
        end_condition

    end_rule


fbe SectorA
    rule rlFireAlarmA
        condition
            subcondition
                premise prSectorInPeaceA
                    this.atIntruderDetected ==
     false
                end_premise
                and
                premise prAlarmAOn
                    alarmA.atStatus == true
                end_premise
            end_subcondition
            and
            subcondition
                premise prSensorA1State
                    sensorA1.atState == true
                end_premise
                or
                premise prSensorA2State
                    sensorA2.atState == true
                end_premise
            end_subcondition
        end_condition

    end_rule
```

Código Fonte 6. Implementação da Rule "rlInvasionDetection"do FBE Main e da Rule "rlFireAlarmA"do FBE Sector do projeto Sensors em LingPON 2.0

```
        #include "SMSSender.h"
        #include <iostream>
        using namespace std;

namespace rule{
  namespace main{
    namespace rlInvasionDetection{
      int count = 0;
      void inc(){
        count++;
        if (count >= 1){
          method::main::mtSendSms::mtSendSms("
    41-999999999");
          method::main::mtSendSms::mtSendSms("
    47-999999999");
        }
      }
      void dec(){
        count--;
      }
    }
  }
  namespace sectorA{
    namespace rlFireAlarmA{
      int count1 = 0;
      bool status1;
      int count2 = 0;
      bool status2;
      void inc1(){
        count1++;
        status1 = false;
        if (count1 = 2){
          status1 = true;
          compareStatusSubConditions();
        }
      }
      void dec1(){
        count1--;
      }
      void inc2(){
        count2++;
        status2 = false;
        if (count2 >= 1){
          status2 = true;
          compareStatusSubConditions();
        }
      }
      void dec2(){
        count2--;
      }
      void compareStatusSubConditions(){
        if((status1 = true) && (status2 = true
    )){
          //method::sirenA1::mtFire::mtFire
    (10);
```

Código Fonte 7. Trecho do arquivo Rule.cpp gerado pela implementação em namespaces

O código gerado do namespace *Rule* é apresentado pelo Código Fonte 7. Como pode-se observar, a *rule* ”rlInvasionDetection”possui apenas uma *Condition* composta por *Premises*. Por isso, no código gerado em namespaces, essa *rule* apresenta um método chamado inc(), o qual incrementa o valor da variável ”count”. Essa variável contabiliza o número de *Premises* que foram obedecidas, ou seja, a cada *Premise* satisfeita, o método inc() é executado para incrementar o valor de ”count”. Quando essa variável atinge o número mínimo de *Premises* necessárias para que a *rule* em questão seja satisfeita, é feita a chamada do namespace *Method* para a execução do correspondente *Method*. No caso de *rules* que possuem *Subconditions*, a lógica é parecida. Porém, para cada *Subcondition* há um método inc() e uma variável ”count”, responsáveis por averiguar os estados apenas de suas *Premises* correspondentes. Quando todas as *Premises* da *Subcondition* em questão são obedecidas, sua respectiva variável ”status”recebe o valor *true*, o que significa que essa *Subcondition* foi satisfeita. Quando todas as *Subconditions* da *Rule* forem satisfeitas (verificação realizada pelo método compareStatusSubConditions()), é feita a chamada para a execução do correspondente *Method*.

E. Methods

O arquivo namespace *Method* (.h e .cpp) apresenta os *Methods* de todos os FBEs do projeto. O Código Fonte 8 mostra o código gerado referente ao namespace *Method*. Este exemplo contém a inclusão de uma biblioteca externa (declarada pelas linhas 5 a 7), a classe ”SMSSender.h”, cujo método é utilizado dentro do escopo do *Method* ”mtSendSms”.

Visando à questão da facilidade de programação proposta para o LingPON 2.0, a ideia é possibilitar a integração com bibliotecas de linguagens imperativas, como o C e C++ por exemplo. Ocorre que essas linguagens imperativas podem ser encapsuladas em *Methods* PON para que possam seguir a execução orientada a notificações [3].

```
#include "methods.h"
#include "instances.h"
#include <string>

        #include "SMSSender.h"
        #include <iostream>
        using namespace std;

namespace method{
  namespace main{
    namespace mtSendSms{
      void mtSendSms(std::string cellphone){

            SMSSender *sender = new SMSSender
    ();
            sender->send(cellphone);

      }
    }
  }
  namespace sectorA{
    namespace mtNotifyInvasion{
      void mtNotifyInvasion(){
        instance::sectorA::at::
    atIntruderDetected::setValue(1);
      }
    }
  }
  namespace sectorB{
    namespace mtNotifyInvasion{
      void mtNotifyInvasion(){
        instance::sectorB::at::
    atIntruderDetected::setValue(1);
      }
    }
  }
}
```

Código Fonte 8. Arquivo Method.cpp gerado pela implementação em namespaces

A possibilidade de integração com bibliotecas de terceiros é uma inovação da versão 2.0 do LingPON e foi possível implementá-la adequadamente na compilação em namespaces C++ com a utilização da entidade *IncludeBlock* do PON. Para isso, foi preciso criar uma classe externa ”SMSSender.h”, a qual é apresentada pelo Código Fonte 9.

```
#include <iostream>
#include <string>
using namespace std;

class SMSSender
{
    std::string telephone;

public:
    //inicializa();
    void send(std::string tel);
};
```

Código Fonte 9. Exemplo de Classe externa para a inclusão de bibliotecas em IncludeBlock

III. Resultados e Discussão

Como anteriormente dito, esse projeto foi implementado em duas etapas. A primeira foi a implementação da geração de código utilizando um projeto básico em LingPON 2.0, o qual apresentava apenas um FBE, uma *Instance*, uma *Premise* e um *Method*. A estratégia de iniciar com um projeto básico .pon e, somente depois que essa primeira etapa estiver compilando coerentemente, partir para a interação entre FBEs foi muito importante para compreender primeiro as funcionalidades do PON na prática. Uma vez que as características básicas do PON foram claramente compreendidas, implementar a geração de código de um projeto um pouco mais complexo, como o ”Sensors”, flui de forma mais natural.

```
int main() {
  premise::sectorA::prAlarmAOn::init();
  premise::sectorA::prSectorInPeaceA::init();
  premise::sectorA::prSensorA1State::init();
  premise::sectorA::prSensorA2State::init();
  premise::sectorA::prAlarmBOn::init();
  premise::sectorA::prSectorInPeaceB::init();
  premise::sectorA::prSensorB1State::init();
  premise::sectorB::prAlarmAOn::init();
  premise::sectorB::prSectorInPeaceA::init();
  premise::sectorB::prSensorA1State::init();
  premise::sectorB::prSensorA2State::init();
```

```
premise::sectorB::prAlarmBOn::init();
premise::sectorB::prSectorInPeaceB::init();
premise::sectorB::prSensorB1State::init();
premise::main::prSectorAInvaded::init();
premise::main::prSectorBInvaded::init();

timeval time;
double initial;
double final;
gettimeofday(&time,0);
initial = (time.tv_sec * 1000.0) + (time.
  tv_usec / 1000.0);

instance::sectorB::at::atIntruderDetected::
  setValue(0);
instance::sectorB::alarmB::at::atStatus::
  setValue(1);
instance::sectorB::sensorB1::at::atState::
  setValue(1);
```

Código Fonte 10. Implementação da Main.cpp em namespaces

Para testar o funcionamento do código gerado, foi preciso implementar a ”Main.cpp”e atribuir manualmente os valores de *Attributes*, como verificado pelas linhas 33 a 35 do Código Fonte 10. Os *Attributes* e seus respectivos valores foram escolhidos de forma a satisfazer uma determinada *Premise* e, assim, possibilitar o fluxo de notificações proposto pelo PON. O ponto de partida das notificações é a alteração no valor de *Attributes*, os quais notificam todas as *Premises* correspondentes. Caso todas as *Premises* de uma *Rule* forem obedecidas, significa que esta foi satisfeita e deve notificar para executar seu respectivo *Method*. No caso do projeto ”Sensors”, o *Method* final executa o método da classe externa ”SMSSender.h”, o qual apresenta uma mensagem de texto alertando o envio de SMS para um determinado número de celular. Esse resultado esperado foi atingido, conforme mostra a Figura 3.

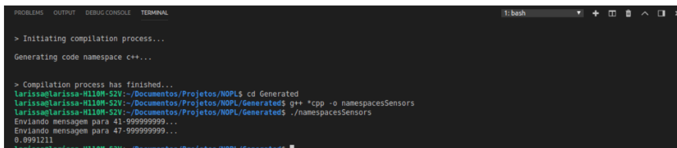

Figura 3. Resultados da geração de código de LingPon 2.0 em namespaces C++ (Projeto Sensors)

A. *Considerações Finais*

Este trabalho foi fruto do projeto final da disciplina Linguagens e Compiladores, lecionada na fase 3 do ano 2018, como parte do programa de Mestrado do CPGEI (Programa de Pós Graduação em Engenharia Elétrica e Infomática Industrial). A experiência de trabalhar com uma nova linguagem e um novo paradigma de programação foi muito produtiva e positiva.

Especificamente sobre a versão 2.0 do LingPON, há duas sugestões a serem colocadas. A primeira é que durante a implementação do projeto, sentiu-se a necessidade de poder obter a *Instance* correspondente a uma determinada *Premise*, ou seja, se a entidade *Premise* obtivesse a função getInstance(), seria muito útil para a compilação em namespaces C++. Isto impactou no arquivo namespace Instance.cpp do código gerado. Depois de compilado, foi preciso alterar a palavra ”sectorA”(e ”sectorB”) para ”main”, conforme o Código Fonte 11.

```
namespace instance{
  namespace sectorA{
    namespace at{
      namespace atIntruderDetected{
        bool value = 0;
        void setValue(bool newValue){
          if (value != newValue){
            value = newValue;
            //foi preciso alterar a palavra "
    sectorA" por "main"
            premise::main::prSectorAInvaded::
    notify_sectorA_atIntruderDetected(newValue
    );
            premise::sectorA::prSectorInPeaceA
    ::notify_sectorB_atIntruderDetected(
    newValue);
            premise::sectorA::prSectorInPeaceB
    ::notify_sectorB_atIntruderDetected(
    newValue);
          }
        }
      }
    }
```

Código Fonte 11. Exemplo de alteração no código gerado em namespace C++

Outra consideração é a possibilidade de haver um bloco ”Main”, para que não seja necessário atribuir o valor de *Attributes* manualmente, como foi descrito na sessão anterior. Ademais, vale ressaltar que o LingPON 2.0 apresenta uma evolução nítida em relação à sua versão anterior. O Grafo PON é uma inovação muito útil e intuitivo, de forma a facilitar muito a compreensão do funcionamento do PON. Foi um elemento essencialmente importante para a implementação deste projeto.

A evolução do LingPON para a nova versão e a sua implementação de compilação em diversas plataformas demonstram que o novo paradigma emergente PON é promissor.

IV. Conclusão

Este artigo tem o objetivo de apresentar uma contribuição para o desenvolvimento da versão 2.0 do LingPON. O LingPON é uma nova linguagem de programação que foi desenvolvida a fim de validar fundamentos do paradigma emergente chamado Paradigma Orientado a Notificações (PON). Esta nova técnica se baseia no Sistema Baseado em Regras do Paradigma Declarativo e na Programação Orientada a Objetos do Paradigma Imperativo. Ademais, o PON traz uma nova visão de implementar e executar programas a fim de solucionar algumas deficiências que os paradigmas atuais ainda apresentam.

Portanto, o surgimento do PON e sua respectiva linguagem de programação específica, LingPON, podem contribuir no campo computacional em termos de aumento do desempenho de execução, facilidade de programação em alto nível e busca pelo desacoplamento entre os módulos do software, podendo

ser, portanto, aproveitada também em sistemas multiprocessados e distribuídos.

Neste sentido, surgiu a necessidade de uma linguagem de programação universal e mais efetiva que proporcionasse a facilidade de programação em alto nível, quaisquer que fossem as plataformas utilizadas. Para solucionar essas adversidades o LingPON 2.0 vem sendo desenvolvido por um grupo de pesquisadores.

O LingPON é baseado fortemente nos fundamentos do PON e desde a sua versão preliminar vem sendo aprimorada ao longo dos anos. Espera-se que o trabalho apresentado possa contribuir para o desenvolvimento da versão LingPON 2.0. Esta nova versão da linguagem de programação tem o objetivo de trazer uma melhoria em termos de paralelismo na execução e possibilidade de integrar bibliotecas de terceiros na codificação PON, sem perder a essência das propriedades do PON.

Uma vez que esses objetivos sejam alcançados, os princípios do PON serão materializados de forma mais eficaz. Além disso, outros horizontes se abrirão para o PON, visto que há indícios de ser útil para diversas outras áreas, além dos sistemas computacionais.

REFERÊNCIAS

[1] R. F. Banaszewski, "Paradigma orientado a notificações: avanços e comparações," Ph.D. dissertation, Master's thesis, Universidade Tecnológica Federal do Paraná (UTFPR), Curitiba, PR, 2009.

[2] A. F. Ronszcka, "Contribuição para a concepção de aplicações no paradigma orientado a notificações (pon) sob o viés de padrões," Master's thesis, Universidade Tecnológica Federal do Paraná, 2012.

[3] "Contribuição para a concepção de aplicações no paradigma orientado a notificações (pon) sob o viés de padrões," Ph.D. dissertation.

[4] L. A. Santos *et al.*, "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," Master's thesis, Universidade Tecnológica Federal do Paraná, 2017.

[5] J. M. Simão and P. C. Stadzisz, "Paradigma orientado a notificações (pon)–uma técnica de composição e execução de software orientado a notificações," *Pedido de Patente submetida ao INPI/Brazil (Instituto Nacional de Propriedade Industrial) em*, 2008.

[6] C. A. Ferreira, "Linguagem e compilador para o paradigma orientado a notificações (pon): Avanços e comparações," Master's thesis, Universidade Tecnológica Federal do Paraná, 2015.