Linguagem e Compilador para o Paradigma Orientado a Notificações: uma solução *performante* orientada a regras

Larissa K. Oshiro[1], Adriano F. Ronszcka[1], João A. Fabro[2], Jean M. Simão[1]

[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) - Curitiba - PR – Brasil
{larissaoshiro, ronszcka}@alunos.utfpr.edu.br, {fabro, jeansimao}@utfpr.edu.br

Abstract. *The Notification-Oriented Paradigm (NOP) is a software development solution that allows, among other characteristics, excellent performance by means of lean entities that collaborate by precise notifications. The present work proposes a materialization for NOP in the scope of NOPL 2.0 Technology by means of a code generator for the 'notifying C ++ oriented to namespaces' target, called NPCPP 2.0, from the rule-oriented language NOPL. A comparative study between the same solution in usual C++ and LingPON/NPCPP 2.0 is also presented, showing its proper performance.*

Resumo. *O Paradigma Orientado a Notificações (PON) é uma solução de desenvolvimento de software que permite, entre outras características, excelente desempenho computacional via entidades enxutas que colaboram por notificações precisas. Este trabalho propõe uma materialização do PON via Tecnologia LingPON 2.0 na forma de um gerador de código para 'C++ notificante via namespaces', denominado de NPCPP 2.0, a partir da linguagem orientada a regras LingPON. É apresentado também um estudo comparativo entre uma mesma solução em C++ usual e LingPON/NPCPP 2.0 mostrando a performance apropriada deste.*

1. Introdução

O novo Paradigma Orientado a Notificações (PON) apresenta propriedades que unem a flexibilidade de programação do Paradigma Imperativo e a facilidade de programação do Paradigma Declarativo. Ademais, o PON proporciona uma nova visão de desenvolver, estruturar e executar software por meio de entidades facto-execucionais e lógico-causais que colaboram por notificações precisas e pontuais [Ronszcka 2019].

O PON apresenta três propriedades elementares, que consistem em: (a) facilidade de programação orientada a regras em alto nível, (b) evitar redundâncias de código que viabiliza alto desempenho de execução e (c) desacoplamento implícito de construtos que viabiliza paralelismo e distribuição [Ronszcka 2019]. Esse artigo apresenta uma nova implementação para o PON, via a chamada Tecnologia LingPON 2.0, na forma de um compilador que permite código 'performante' (de excelente desempenho computacional) a partir de linguagem de regras em alto nível.

2. Paradigma Orientado a Notificações - PON

Em suma, o PON é constituído por dois conjuntos de entidades: o facto-execucional e o lógico-causal. A relação entre as entidades constituintes do PON, são ilustradas, no contexto de um sistema de correlação de sensores, pela Figura 1 [Ronszcka 2019].

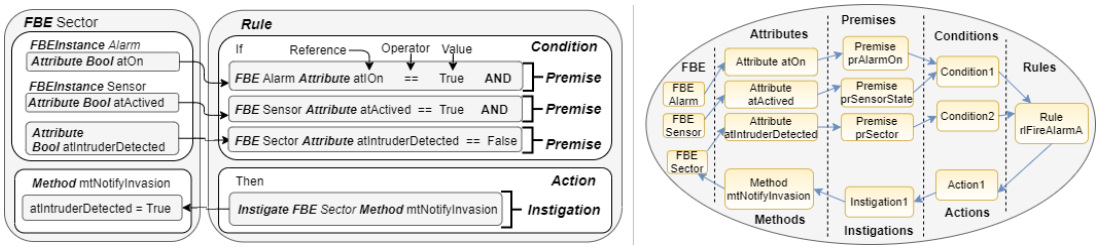

Figura 1. Interação entre as entidades do PON e ciclo de notificações delas.

Conforme a Figura 1, há entidades facto-execucionais notificantes, os *Fact Base Elements* (*FBE*), que representam entidades do mundo, compostas de sub-entidades *Attributes,* que tratam seus estados, e sub-entidades *Methods,* que tratam seus serviços/comportamento. Há também entidades lógico-causais notificáveis, as *Rules*, cada qual composta de uma sub-entidade *Condition,* que se associa a sub-entidades *Premises,* e uma sub-entidade *Action,* que se associa a sub-entidades *Instigations.*

Em suma, o ciclo de notificações funciona assim: cada *Attribute* de uma instância de um FBE que mudar de estado notifica apenas as *Premises* pertinentes, o que faz com que essas refaçam seus cálculos lógicos. Cada *Premise* que mudar de estado lógico notifica apenas as *Conditions* pertinentes, fazendo com que essas refaçam seus cálculos lógicos pelos estados notificados contabilizados. Se a *Condition* é aprovada, ela pode aprovar sua respectiva *Rule*. Esta, quando aprovada, ativa sua *Action,* que notifica suas *Instigations*, as quais instigam os *Methods*. Estes últimos geralmente alteram os estados dos *Attributes*, reativando o fluxo de notificações [Ronszcka 2019].

Para materializar os princípios do PON, primeiramente foram desenvolvidos *frameworks* sobre linguagens de programação orientadas a objetos (POO), fazendo estas trabalharem de uma nova forma que é justamente orientada a notificações. Entretanto, as estruturas de dados nos *frameworks* sobre outras linguagens eram computacionalmente custosas. Assim, foram desenvolvidas linguagens de programação e seus respectivos compiladores, próprios ao paradigma, via solução denominada de Tecnologia LingPON. Esta tecnologia se encontra em sua versão 2.0, que deu origem à LingPON 2.0, uma linguagem de programação mais completa em relação à precedente LingPON 1.0. O código na Figura 2 exemplifica uma implementação em LingPON 2.0 [Ronszcka 2019].

A concepção e evolução da Tecnologia LingPON permitiu a concomitante criação do MCPON, um método de compilação de linguagens próprias ao PON, que possibilita o mapeamento de cada programa em LingPON para um grafo de entidades notificantes, denominado de GrafoPON, exemplificado na Figura 2. A partir deste, via navegação em suas instâncias por meio de biblioteca e API pertinentes, viabiliza-se a construção de compiladores para diversas plataformas distintas (*targets*), como C++ notificante para plataforma *monocore*, Elixir/Erlang notificante para plataforma *multicore*, VHDL notificante para tecnologia FPGA e assembly PON para a ArqPON, uma arquitetura de computação própria ao PON [Negrini *et al*. 2019] [Ronszcka 2019].

No âmbito da Tecnologia LingPON/MCPON, este artigo propõe a implementação de um novo compilador para a LingPON 2.0, o qual gera códigos para o *target* 'C++ notificante orientado a *namespaces*' (NPCPP 2.0). Este é uma versão mais 'performante' que *targets* precedentes, nos quais cada aplicação é um código (C++ ou outro) notificante em específico [Ronszcka 2019]. Ainda, o artigo traz um estudo comparativo de desempenho do NPCPP 1.0, NPCPP 2.0 e implementação equivalente em POO C++.

```
fbe Sector
  private Boolean atIntruderDetected = false

  private Alarm alarmA
  private Sensor sensorA1
  private Sensor sensorA2

  private method mtNotifyInvasion
    attribution
      this.atIntruderDetected = true
    end_attribution
  end_method

  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
    action
      instigation
        call this.mtNotifyInvasion()
      end_instigation
    end_action
  end_rule
end_fbe
```

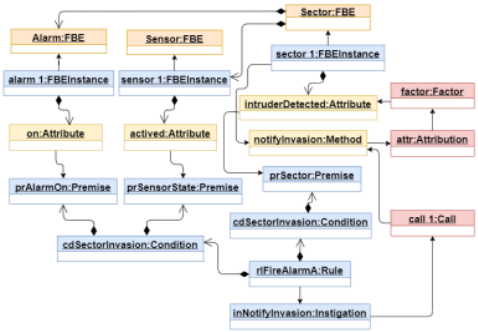

Figura 2. Exemplo em LingPON 2.0 e seu mapeamento para GrafoPON.

3. Desenvolvimento do Trabalho

A Tecnologia LingPON passou por uma evolução e atualmente se encontra em sua versão 2.0. Estudos precedentes mostraram que dentre os geradores de código construídos para a Tecnologia LingPON 1.0, o compilador que gerava código para o *target* NPCPP 1.0 foi o que obteve o melhor resultado de desempenho em arquitetura Von Neumann *monocore*.

Assim sendo, a construção do gerador de código para esse *target* NPCPP utilizando a versão atual 2.0 da Tecnologia LingPON (NPCPP 2.0) pode contribuir para a materialização do PON. Isto porque a LingPON 2.0 apresenta o PON de forma mais completa e variada em termos de conceitos de programação (*i.e.*, mais tipos de *Attributes*, maior variedade conectivos lógicos etc.) que a LingPON 1.0 e elimina-se as imperfeições (*bugs*) do prototipal NPCPP 1.0 [Ronszcka 2019].

Namespaces em C++ POO seria uma maneira de organizar itens (classes, objetos e enumerações) de forma coesa. Em C++ notificante, *namespaces* podem organizar *Rules*, *Premises* e afins. No NPCPP 2.0, os códigos resultantes são apresentados na forma de módulos com procedimentos e afins, justamente por meio de *namespaces* e não de classes, mantendo coesão e evitando sobrecargas dessas (como tabelas de polimorfismo).

Pelo fato de o PON obedecer a lógica de notificações entre suas entidades, para a geração de códigos para NPCPP 2.0, as entidades notificantes são materializadas em módulos *namespace* específicos, de acordo com o programa fonte implementado em LingPON 2.0. O Código 1 apresenta o código resultante para o chamado *namespace Instance*, no qual estão contidos cada instância de *FBE* com seus respectivos *Attributes*.

```
#include "instances.h"
#include "premises.h"
#include <string>
namespace instance{
  namespace sector{
    namespace at{
      namespace atIntruderDetected{
        bool value = 0;
        void setValue(bool newValue){
          if (value != newValue){
            value = newValue;
            premise::main::prSectorInvaded::
              notify_sector_atIntruderDetected(newValue);
          }}}}
    namespace alarmA{
      namespace at{
        namespace atStatus{
          bool value = 0;
          void setValue(bool newValue){
            if (value != newValue){
              value = newValue;
              premise::sector::prAlarmAOn::
                notify_alarmA_atStatus(newValue);
            }}}}}}
```

```
#include "premises.h"
#include "rules.h"
#include <string>
namespace premise{
  namespace sector{
    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::inc1();
          }
        }else{
          if(state == true){
            state = false;
            rule::sectorA::rlFireAlarmA::dec1();
          }}}
      void notify_alarmA_atStatus(bool newValue){
        cpy1st = newValue;
        compare();
      }}}}
```

Código 1. Código gerado pelo compilador para NPCPP 2.0.

De forma geral, o processo de notificações entre as entidades em *namespaces* ocorre da seguinte maneira: quando um *Attribute* tem seu valor alterado, é feita uma

chamada, via *namespace* pela função *setValue*, das *Premises* interessadas; a cada *Premise* satisfeita, é feita uma chamada, via *namespace*, das respectivas *Rules* via funções pertinentes. Para cada *Rule* é realizada uma verificação de quantas de suas respectivas *Premises* foram satisfeitas pela contabilização dos estados advindos via notificações. Se a quantidade de *Premises* necessárias para aprovar a *Rule* for atingida, ocorre uma chamada via *namespace* dos *Methods* interessados, os quais acabam por alterar o valor de um *Attribute*, notificando-os via *namespace* e realimentando o fluxo de notificações.

4. Testes e Resultados

Com o objetivo de validar o NPCPP 2.0, foram realizados testes com a aplicação de "Rede de Sensores", que simula um sistema de monitoramento de alarmes. Basicamente, este sistema de monitoramento é composto por um conjunto de sensores e alarmes. Quando é detectada a presença de um invasor, o alarme é acionado e o sistema simula a notificação da invasão via mensagem de texto (Figuras 1 e 2). Os resultados foram comparados para com o desempenho de solução em Paradigma Imperativo, mais especificamente implementado em POO C++, bem como para com o do NPCPP 1.0.

Para a realização dos testes foi utilizado um ambiente Linux Mint 19 (64 bits) em uma máquina com 12 GB RAM, Intel Core i3 – 7100 CPU @ 3.90 GHz. O sistema apresenta um total de 10 *Rules* e foi executado para quatro cenários, nos quais 10%, 40%, 70% e 100% das *Rules* são aprovadas. A Figura 3 apresenta os resultados, obtidos em tempo de execução em milissegundos, das implementações em OO C++, do NPCPP 1.0 e do NPCPP 2.0, para 2000 iterações nos quatro cenários. Os resultados mostram que o tempo de execução do NPCPP 1.0 e do NPCPP 2.0 são menores que o da implementação em OO C++, uma vez que esta é regida por um paradigma em que há redundâncias temporais, um dos problemas que o PON propõe eliminar.

Resultado de desempenho para Sistema de Sensores (ms)								
Porcentagem de *Rules* aprovadas	10%		40%		70%		100%	
	média	σ	média	σ	média	σ	média	σ
C++ OO usual	74,78106	22,897	837,7578	27,535	1552,65	52,049	2213,6	**103,3**
NPCPP 1.0	63,73804	15,562	754,7156	21,212	1506,496	72,896	2134,964	93,547
NPCPP 2.0	**51,069**	16,103	**694,2398**	29,894	**1417,338**	68,506	**2098,668**	97,107

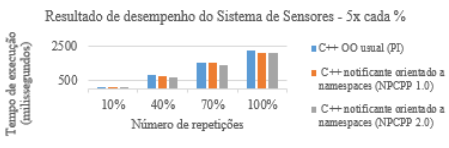

Figura 3. Resultado de desempenho do Sistema de Sensores (2000 iterações).

5. Conclusão e Trabalhos Futuros

Os resultados demonstram que NPCPP 2.0 é um caminho viável para implementar o PON segundo suas propriedades elementares, já alcançando performance apropriada e desenvolvimento em alto nível. A próxima evolução do NPCPP 2.0 é sua aplicação em *multicore*, buscando explorar a propriedade de desacoplamento de entidades do PON.

Referências

Negrini F., Linhares R. R., Fabro J. A., Stadzisz P. C. e Simão J. M. (2019) "NOPL-Erlang: Programação Multicore Transparente em Linguagem de Alto Nível", V Escola Regional de Alto Desempenho do Rio de Janeiro (ERAD-RJ 2019), Brasil.

Ronszcka, A. F. (2019) "Método para a Criação de Linguagens de Programação e Compiladores para o Paradigma Orientado a Notificações em Plataformas Distintas", Tese de Doutorado – CPGEI/UTFPR, Curitiba, Brasil.

Agradecimento: J. M. Simão agradece à Fundação Araucária pela bolsa PQ – Edital 15/2017.