LINGPON 2.0 & COMPILADOR PARA FRAMEWORK PON C++ 3.0 & PON-IP

Christian Carlos Souza Mendes, Cleverson Avelino Ferreira, (A. F. Ronszczka), [J. A. Fabro, J. M. Simão]

Abstract— This article presents the activities carried in a concise way, reflecting the final project of the Language and Compiler discipline, which included the adaptation of Framework 2.0 and Framework 3.0, for transparent adoption of PONIP and Multicore in the applications generated in the target code in C ++.

Index Terms— PON, PONIP, Framework, C++

---

1 INTRODUÇÃO

O projeto desenvolvido durante a disciplina de Linguagens e Compiladores contempla a necessidade de integração de funcionalidades não disponíveis nos frameworks atuais para geração de código-fonte na linguagem C++. Desta forma, neste artigo são apresentadas as adaptações implementadas nos *Frameworks 2.0 e 3.0* para geração de código-fonte utilizando a LingPON (linguagem de programação PON), com os objetivos específicos de prover suporte de forma integrada ao PONIP e ao uso de aplicações *multicore*.

A adaptação junto aos *Frameworks* fez-se necessária pois trata-se de uma forma de simplificar o processo de geração de código-alvo na linguagem C++. Assim, o código gerado para compilação possuirá diretivas especificas para atendimento às funcionalidades de envio de estado de atributo através de uma rede de computadores e o uso de *multicore* para a execução da aplicação de interesse.

Este artigo está organizado na seguinte forma: a seção 2 apresenta os fundamentos do PON, LingPON, PONIP e Multicore. Em seguida, na seção 3, são detalhadas as implementações realizadas nos *Frameworks*. Na seção 4 são descritos os testes realizados. A seção 5 detalha as principais dificuldades encontradas durante a realização do projeto. Na seção 6 são descritas as principais considerações sobre o uso do *Grafo PON*. Por fim, na seção 7 são apresentadas as conclusões finais.

2 CONTEXTUALIZAÇÃO

2.1 PON

O paradigma orientado a notificações (PON) é um paradigma atualmente voltado para o desenvolvimento de softwares, que foi inicialmente proposto por Simão [1] em sua tese de doutorado.

O PON se inspira no Paradigma Orientado a Objetos (POO) e no Paradigma Lógico (PL), com ênfase nos conceitos dos Sistemas Baseados em Regras (SBR). Unificando as principais características e vantagens do SBR como a representação do conhecimento na forma de regras e do POO a flexibilidade de expressão e nível apropriado de abstração, propondo resolver várias de suas deficiências em aplicações de software monoprocessado e multiprocessado. Entre essas deficiências pode-se salientar as redundâncias estruturais/temporais e o acoplamento forte entre as entidades computacionais. [2]

Pode-se dizer que o PON permite desenvolvimento orientado a regras em alto nível, ao mesmo tempo em que utiliza elementos de programação reativa e orientada a eventos para alcançar um arranjo particular de inferência. Neste arranjo, os elementos factuais (usualmente variáveis, atributos-objetos, frame-slots e afins) notificam elementos lógico-causais (regras, expressões se, então e afins) para alcançar uma nova forma de inferência. No PON, entretanto, tudo ocorre por notificações (similares a eventos, interrupções e afins) nos construtos mais elementares de programação, o que se constitui na sua contribuição ao estado da arte enquanto nova técnica de programação[3].

No PON, as entidades computacionais que possuem entidades atributos (*Attribute*) e entidades métodos (*Methods*) são genericamente chamadas de entidades *FBEs (Fact Base Elements*). Por meio de *seus Attributes e Methods, as FBEs* são passíveis de correlação lógico-causal por meio de entidades Rules, as quais constituem elementos fundamentais do PON [4][5][6][7].

Diferentemente do que ocorre no PI, incluindo a programação orientada a objetos, no qual o desenvolvedor informa de maneira explícita o laço de iteração através de comandos como while e for, no PON a repetição ocorre de forma natural na perspectiva de execução da aplicação a partir da mudança de estado de um *Attribute* [4].

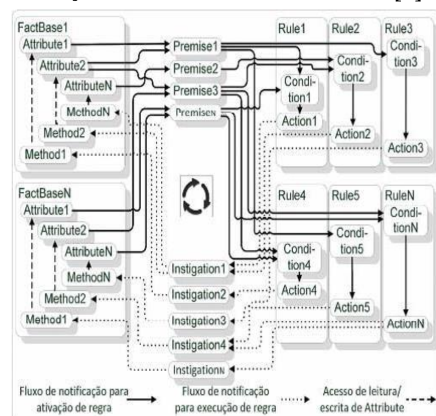


Fig.1 - Cadeia de notificações do PON [2].

O PON possui como principal característica o desacoplamento de entidades que podem comunicar-se através do uso de notificações, gerando inferências, conforme ilustrado na figura 1.

Diversos estudos realizados comprovam o aumento de performance e uma maior facilidade para a criação de um software através do uso do PON [8][9][10].

2.2 LingPON

Para o desenvolvimento e maior adoção do PON, foi criada a linguagem PON (denominada doravante "LingPon") e seu compilador (denominado doravante "compilador PON"), o que torna possível o desenvolvimento de aplicações específicas em uma linguagem conformada ao PON e gerar resultados, em termos de código-alvo, sem a adição de estruturas de dados caras [11].

De modo geral, o código fonte da linguagem PON (doravante denominada LingPON) segue um padrão de declarações. Primeiramente, o desenvolvedor precisa definir os FBEs de seu programa. Em seguida, o desenvolvedor precisa declarar as instâncias de tais FBEs, bem como definir a estratégia de escalonamento das Rules. Subsequentemente, é necessário definir as Rules para fins de avaliação lógico causal dos estados do FBEs por meio de notificações.

Por fim, é possível adicionar código específico da linguagem alvo escolhida no processo de compilação (e.g. C ou C++) com a utilização do bloco de código main.

2.3 PONIP

Devido a característica de desacoplamento das entidades e computação distribuída, fez-se necessário o desenvolvimento de uma plataforma que permitisse o envio de dados através da rede de forma integrada ao PON, sendo criado por Talau [12]. O PONIP pode ser considerado um plugin para uso do PON em ambientes distribuídos permitindo o envio dos estados dos atributos e premissas através de uma rede de computadores. Foi implementado em linguagem C ANSI e é utilizado na forma de biblioteca compartilhada por aplicações geradas em PON. Inicialmente foi desenvolvido permitindo a utilização em conjunto com o framework C++ 2.0, exigindo a adaptação de algumas entidades e parâmetros.

A implementação na forma de biblioteca traz ainda vantagens, como a possibilidade de comunicação de aplicações feitas diferentes materializações de forma transparente ao programador [12].

2.4 Multicore

Assim como o uso do PON e PONIP, a adaptação de aplicações para o uso de ambientes *multicore* são essenciais atualmente. Os sistemas operacionais convencionais podem fazer uso da capacidade dos processadores *multicore*, realizando o chamado thread scheduling e dividindo a carga de processamento dos softwares. Entretanto, tal particionamento do processamento entre os núcleos ainda não é satisfatório, porque o problema não está no escalonamento de tarefas realizado pelo sistema operacional e sim nos softwares a serem escalonados por ele [13].

Como a programação paralela exige maior esforço na sua concepção, o desenvolvimento de software para ambientes *multicore* deve utilizar recursos de programação, tais como APIs, bibliotecas, diretivas de pré-compilação, linguagens de programação, compiladores e técnicas de programação voltadas à questão da concorrência. Tal implementação encontra-se disponível para uso no Framework 3, exigindo uma integração para que seja possível a geração de códigos na linguagem-alvo.

3 Implementação

O projeto tem como principal objetivo a adaptação da LingPON junto aos *Frameworks 2.0 e 3.0*, além da integração com o PONIP e disponibilização do uso de features para uso de aplicações *multicore*.

De maneira geral, a estrutura do PON, definida por meio das entidades notificantes do paradigma, possibilitou uma certa inovação no processo de compilação, no qual é possível traduzir o código fonte em um grafo único, composto por instâncias das entidades fundamentais do PON, com as características mapeadas de cada qual e suas respectivas conexões baseadas em notificações. Para essa estrutura foi dado o nome de Grafo PON [14].

O processo foi iniciado com a análise do Grafo PON, no qual são armazenados cada um dos elementos de forma distinta, permitindo mapear suas particularidades, bem como suas conexões com os elementos a serem notificados [14].

Após o estudo e entendimento do uso do Grafo PON, efetivamente foi iniciado o processo de leitura e navegação dentro da estrutura. Assim, através da navegabilidade funcional, os *Frameworks* foram sendo adaptados para a linguagem alvo, que neste caso trata-se do C++.

A geração dos códigos-alvo passa pela criação de forma automatizada dos arquivos .cpp e .h, após a disponibilização dos arquivos .NOP, que devem ser criados através do uso da LingPON.

Após a fase inicial realizada, foi implementada a adaptação do PONIP para que fosse integrado de forma transparente durante o processo de geração do código-alvo, permitindo ao desenvolvedor habilitar ou não a distribuição do estado de um atributo através de uma rede de computadores.

Por fim, no *Framework 3.0* foram adaptadas as funcionalidades referentes para geração do código-alvo que contemplam a ativação do uso de processamento distribuído *multicore* de forma instrínseca à aplicação.

3.1 Framework 2.0 e Framework 3.0

A geração de código para o *Framework* PON basicamente recria os componentes do *Framework* de acordo com o código PON criado. Cada elemento do código PON tem a sua respectiva correspondência vinculada ao Grafo e é inserido no *Framework*.

Para simular a criação de código através do *Framework* via compilador PON, uma aplicação simples foi gerada

com nome de *Sensor*. A aplicação gerada através da LingPON pode ser observada no Algoritmo 01, onde é representada uma *FBE* chamada *Door*.

Algoritmo 01: Código em LingPON para *Door*

```
1    fbe Door
2        includes FRAMEWORK_CPP_2_0
3        #include <iostream>
4        using namespace std;
5        using std::cout;
6        using std::endl;
7        end_includes
8
9      public integer atDoorState = 0
10     public String atDoorName = "Door"
11
12     private method mtDoorOpen
13        params
14           integer atDoorState;
15        end_params
16        code FRAMEWORK_CPP_2_0
17                    this->atDoorState->set-
18 Value(atDoorState);
19        end_code
20     end_method
21 end_fbe
```

Após a compilação da aplicação apresentada no Algoritmo 01 pelo *Framework 2.0*, é possível visualizar o código .h e .cpp equivalentes a *FBE Door*, representados no Algoritmo 02. É necessário citar que o código gerado realiza as importações necessárias de forma direta, sem a necessidade de intervenção externa.

Algoritmo 02: Classe *Door* em *Framework*

```
1     /*
2      * Door.h
3      *
4      *  Created on: <Month> <Day>, <Year>
5      *  Author: add your name
6      *  @CopyRight LingPon
7      */
8     #ifndef Door_H_
9     #define Door_H_
10    #include            "framework/elements/ab-
11 stract/MethodPointer.h"
12    #include "framework/utils/SingleInclude.h"
13
14       #include <iostream>
15       using namespace std;
16       using std::cout;
17       using std::endl;
18
19    class Door : public FBE {
20    public:
21                Door();
22                ~Door();
23                //Atributes
24                Integer * atDoorState;
25                String * name;
26                //rules e premises
27                //methods
28                void   mtDoorClose(int   atDoor-
29 State);
30                void   mtDoorOpen(int   atDoor-
31 State);
32    };
33    #endif
34
35
36
37
38    /*
39     * Door.cpp
40     *
41     *  Author: add your name
42     *  @CopyRight LingPon
43     */
44
45    #include "Door.h"
46    #include <iostream>
47    using std::cout;
48    using std::cout;
49    Door::Door(void) {
50          INTEGER(this,this->atDoorState,0);
51          NET_ATTR_OUT_INT("atDoorState",
52 atDoorState, ATTR_ONLY_CHANGE, "127.0.0.8",
53 ATTR_TYPE_STANDARD);
54          STRING(this,this->name,"Door");
55          NET_ATTR_OUT_INT("atDoorName",
56 atoDoorName,ATTR_ONLY_CHANGE,
57 "127.0.0.8", ATTR_TYPE_STANDARD);
58    }
59    Door::~Door(void) {
60    }
61    void Door::mtDoorClose(int atDoorState) {
62

                        this->atDoorState->set-
   Value(atDoorState);

      }
      void Door::mtDoorOpen(int atDoorState) {

                        this->atDoorState-
   >setValue(atDoorState);

   }
```

Através do arquivo .cpp, apresentado no Algoritmo 02, é possível identificar a pré-configuração para a transmissão do estado do atributo *atDoorState e atDoorName* através da rede, sendo realizada pelo uso da função NET_ATTR_OUT_INT, implementada pelo PONIP.

O Algoritmo 03 apresenta a criação de uma *Rule* através do uso da LingPON.

Algoritmo 03: Código PON para criação de uma *Rule*

```
1  rule rlInvasionDetection
2     condition
3           premise prSectorAInvaded
4           sectorA.atIntruderDetected == true
```

5	end_premise
6	or
7	premise prSectorBInvaded
8	sectorB.atIntruderDetected == true
9	end_premise
10	end_condition
11	action sequential
12	instigation
13	call this.mtSendSms("41-999999999")
14	end_instigation
15	end_action
16	end_rule

No Algoritmo 03 é apresentado o código PON referente à criação de uma *Rule* com a sua *Condition*, suas *SubConditions* e *Actions*. Cada *SubCondition* é composta por uma ou mais *Premises*. Cada *Premise*, neste caso, é constituída de um identificador, uma referência para uma *FBE*, o atributo da *FBE* que é utilizado pela avaliação da *Premise* e o operador de avaliação com o resultado esperado. Ainda, o elemento *Action* da *Rule* pode conter uma ou mais *Instigations*. Cada *Instigation* é constituída de um identificador, de uma referência para uma *FBE* e do método que será chamado quando a *Rule* estiver ativa.

O Algoritmo 04 apresenta o código *Framework* gerado, que compõe o arquivo principal utilizado para executar a aplicação. Neste arquivo, é possível visualizar as referências para os elementos PON como o conjunto de *Rules*, conjunto de *Premises* e o conjunto de *Instigations*. Como é de conhecimento, o *Framework* possui a implementação de todos esses elementos e o código alvo realiza as importações necessárias de forma automática.

Algoritmo 04: ***MainApplication.h e MainApplication.cpp com Framework 2.0***	
1	/*
2	* MainApplication.h
3	*
4	*/
5	
6	#ifndef MAINAPPLICATION_H_
7	#define MAINAPPLICATION_H_
8	#include <iostream>
9	using std::cout;
10	using std::endl;
11	#include <cstdio>
12	#include "framework/NOPApplication.h"
13	#include "framework/utils/SingleInclude.h"
14	#include "Door.h"
15	#include "Main.h"
16	
17	class MainApplication : public NOPApplication
18	{
20	
21	public:
22	MainApplication();
23	virtual ~MainApplication();
24	
25	public:
26	void initStartApplicationComponents();
27	void initFactBase();
28	void initRules();
29	void initSharedEntities();
30	void codeApplication();
31	};
32	#endif
33	
34	/*
35	* MainApplication.cpp
36	*
37	*/
38	#include "MainApplication.h"
39	#include <iostream>
40	using namespace std;
41	#include <sys/time.h>
42	MainApplication::MainApplication() {
42	initStartApplicationComponents();
43	}
44	MainApplication::~MainApplication() {
45	}
46	void MainApplication::initStartApplica-
47	tionComponents() {
48	SingletonFactory::changeStructure(Sin-
49	gletonFactory::NOPVECTOR);
50	SingletonLog::changeStream(Singleton-
51	Log::CONSOLE);
52	this->startApplication();
53	}
54	void MainApplication::initFactBase() {
55	}
56	void MainApplication::initSharedEntities() {
57	}
58	void MainApplication::initRules() {
59	}
60	void MainApplication::codeApplication() {
61	}

Ainda, o Algoritmo 05 apresenta o trecho de código que realiza a configuração de uma *Rule* com as suas *Premises* e *Instigations*. Esse código é baseado no código PON apresentado no Algoritmo 03, sendo possível observar que o código-alvo é capaz de associar cada elemento pertencente a *Rule* de forma correta.

Algoritmo 05: Configuração de uma *Rule*	
1	PREMISE(prSectorAInvaded,this->sec-
2	torA.atIntruderDetected,1,Prem-
3	ise::EQUAL,Premise::STANDARD, false);
4	PREMISE(prSectorBInvaded,this->sec-
5	torB.atIntruderDetected,1,Premise::EQUAL,Prem-
6	ise::STANDARD, false);
7	
8	RULE (rlInvasionDetection,scheduler,Condi-
9	tion::CONJUNCTION);
10	rlInvasionDetection->addPremise(prSectorAIn-
11	vaded);
12	rlInvasionDetection->addPremise(prSec-
13	torBInvaded);
14	
15	

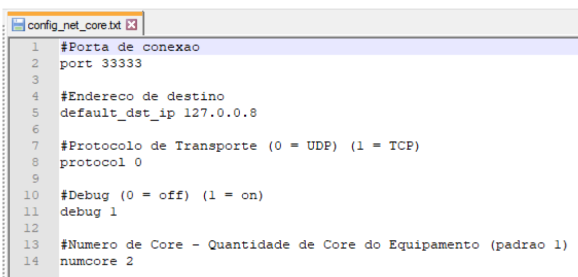
```
16  rlInvasionDetection->addInstigation(INSTIGA-
    TION(this.mtSendSms("41-999999999")));
17  rlInvasionDetection->end();
```

O *Framework* 2.0 inicialmente foi integrado ao PONIP e para o *Framework 3.0,* além da integração com o PONIP, foi realizada a integração para disponibilizar o uso de *multicore*.

O PONIP exige a utilização de um arquivo que contenha a pré-configuração de dados para envio dos estados de atributos via rede, como endereço de ip de destino, porta para comunicação, tipo de protocolo, entre outros. Já para ativação do uso de *multicore* é necessário que esta informação seja disponibilizada no momento da compilação do código, através do *Framework 3.0,* para a geração do código-alvo, habilitando ou não o uso de *multicore*.

Considerando a necessidade de que as duas funcionalidades sejam integradas aos *Frameworks* exigindo a inserção de dados essencias, fez-se necessária a criação de um arquivo de configuração, config_net_core.txt, no qual estão listadas as configurações específicas para a ativação do PONIP e uso do *multicore*.

No momento da compilação do código em LingPON pelos *Frameworks,* é necessário que seja realizada a importação e validação do arquivo de configuração, config_net_core.txt, que deve estar localizado no diretório raiz do compilador. Este arquivo de configuração, conforme apresentado na figura 2, contém as informações necessárias para configurar o *Framework* com relação ao envio do estado dos atributos através da rede de computadores e a alocação de processadores (*cores)* na execução de aplicações PON.



```
config_net_core.txt
1   #Porta de conexao
2   port 33333
3
4   #Endereco de destino
5   default_dst_ip 127.0.0.8
6
7   #Protocolo de Transporte (0 = UDP) (1 = TCP)
8   protocol 0
9
10  #Debug (0 = off) (1 = on)
11  debug 1
12
13  #Numero de Core - Quantidade de Core do Equipamento (padrao 1)
14  numcore 2
```

Fig.2 – Arquivo de Configuração para ativação da comunicação via rede e ativação de *multicore*.

Para que seja possível realizar o envio do estado do atributo através da rede de computadores é necessária a utilização do PONIP. Este, por sua vez, exige a adequação de estruturas nos respectivos *Frameworks,* para que assim o uso de suas funcionalidades esteja disponível. Os seguintes arquivos, referentes as estruturas, devem ser adequados:

-Premisse.h
-Premisse.cpp
-Integer.h
-Integer.cpp
-FBE.h

Após a realização das alterações, o PONIP é compilado utilizando o *Framework* de interesse, considerando que este foi adaptado, e a partir deste momento poderá ser utilizado como uma biblioteca a ser vinculada à aplicação no momento de compilação do código-alvo, que nos casos dos *Frameworks* abordados neste artigo são na linguagem C++.

A figura 3 apresenta um exemplo de vinculação do PONIP durante o processo de compilação de uma aplicação em C++.

```
#!/bin/sh
# location of the framework
FRAMEWORK_DIR="/home/ccsm/framework_cpp_2.0"
# location of app dir
APP_DIR="sensors"
# executable name
EXE_NAME="distributed_door"
# bin ponlib dir
BIN_LIBPON="/home/ccsm/ponip"

g++ -g -o $EXE_NAME $APP_DIR/*.cpp -I$APP_DIR -I$FRAMEWORK_DIR/Include -I$FRAMEWORK_DIR/Source
$FRAMEWORK_DIR/Source/framework/*.cpp
$FRAMEWORK_DIR/Source/framework/utils/*.cpp $FRAMEWORK_DIR/Source/framework/elements/nop_list/*.cpp
$FRAMEWORK_DIR/Source/framework/elements/nop_list/attributes/*.cpp
$FRAMEWORK_DIR/Source/framework/elements/nop_hashmap/*.cpp
$FRAMEWORK_DIR/Source/framework/elements/nop_hashmap/attributes/*.cpp
$FRAMEWORK_DIR/Source/framework/elements/stl_list/*.cpp
$FRAMEWORK_DIR/Source/framework/elements/stl_list/attributes/*.cpp
$FRAMEWORK_DIR/Source/framework/elements/nop_vector/*.cpp
$FRAMEWORK_DIR/Source/framework/elements/nop_vector/attributes/*.cpp
$FRAMEWORK_DIR/Source/framework/elements/abstract/*.cpp
$FRAMEWORK_DIR/Source/framework/elements/abstract/operations/*.cpp
$FRAMEWORK_DIR/Source/framework/elements/abstract/operations/binary/*.cpp
$FRAMEWORK_DIR/Source/framework/elements/abstract/operations/unary/*.cpp
$FRAMEWORK_DIR/Source/framework/elements/abstract/operators/*.cpp
$FRAMEWORK_DIR/Source/framework/elements/abstract/attributes/*.cpp
$FRAMEWORK_DIR/Source/framework/elements/abstract/schedulers/*.cpp
$FRAMEWORK_DIR/Source/framework/factories/*.cpp
$FRAMEWORK_DIR/Source/framework/modules/LibPONIP/*.cpp
-L$BIN_LIBPON -lponip -lpthread
```

Fig.3 – Arquivo utilizado para compilação da aplicação em C++.

Após as alterações já informadas, foi necessária a realização de intervenções no código fonte do PONIP, para o uso do protocolo TCP durante a comunicação via rede de computadores e ativação do debug (para acompanhamento das transmissões através da rede). Estas alterações foram necessárias para que o arquivo de configuração, específico para uso do PONIP, não seja mais necessário, sendo apenas o arquivo config_net_core.txt utilizado durante a compilação da aplicação em LingPON para o código-alvo em C++ e não sendo mais exigido durante a execução da aplicação compilada em C++, algo até então definido pelo PONIP de forma intrínseca de acordo com a sua concepção.

Para o envio dos estados dos atributos através da rede de computadores, foi realizada uma adaptação no PONIP e nos *Frameworks,* permitindo que o destino do envio das transmissões possa ser múltiplo, ou seja, não sendo limitado a um único destino.

Desta forma, as adaptações necessárias para o *Framework 2* foram finalizadas e inseridas também no *Framework 3.0.*

O *Framework 3.0* possui como principal necessidade a adoção do uso de *multicore* para a execução das aplicações. Considerando isto, uma premissa básica neste processo é realizar a checagem de quantidade de processadores disponíveis para uso durante o processo de validação do arquivo de configuração. Caso o arquivo apresente dois processadores e na realidade não existam dois processadores para utilização, isto é identificado no momento de execução do *Framework 3.0,* quando será gerada a aplicação no código-alvo em C++. Esta validação faz-se necessária, não permitindo que tal situação seja desconsiderada e que apresente problemas durante a execução da aplicação compilada em C++.

Conforme apresentado na figura 2, a aplicação PON a ser gerada utilizando o *Framework 3.0* deve ser *multicore* e, neste caso especifico, apresentar a alocação dinâmica de

dois *cores* para processamento. O Algoritmo 06 apresenta o trecho de código em *Framework* 3.0, que adiciona o código necessário para alocação dos recursos de processamento, assim como em relação aos demais componentes do *Framework* que são necessários para execução da aplicação PON em *multicore*.

Algoritmo 6: *MainApplication.h e MainApplication.ccp* com *Framework 3.0*

```
1     /*
2      * MainApplication.h – Framework 3.0
3      *
4      */
5     #ifndef MAINAPPLICATION_H_
6     #define MAINAPPLICATION_H_
7     #include <iostream>
8     using std::cout;
9     using std::endl;
10    #include <cstdio>
11    #include "framework/NOPApplication.h"
12    #include "framework/utils/SingleInclude.h"
13    #include "framework/multicore/CoreControl-
14  lersManager.h"
15    #include "framework/multicore/facto-
16  ries/NOPVectorThreadElementsFactory.h"
17    #include "framework/proposedMethod/Pro-
18  posedMethod.h"
19    #include "framework/system/SystemControl-
20  ler.h"
21    #include "Door.h"
22
23    #include "Main.h"
24
25    class MainApplication : public NOPApplication
26  {
27    public:
28        MainApplication();
29        virtual ~MainApplication();
30    public:
31        void initStartApplicationComponents();
32        void startApplication();
33        void initFactBase();
34        void initRules();
35        void initSharedEntities();
36        void codeApplication();
37    };
38
39  #endif }
40
41    /*
42     * MainApplication.cpp – Framework 3.0
43     *
44     */
45    #include "MainApplication.h"
46    #include <iostream>
47    #include <sys/time.h>
48    MainApplication::MainApplication() {
49        initStartApplicationComponents();
50    }
51    MainApplication::~MainApplication() {
52    }
53    void        MainApplication::initStartApplica-
54  tionComponents() {
55        Thread::init();;
56        CoreControllersManager::createCoreCon-
57  trollers(1);
58        SingletonFactory::setStructure(new
59  NOPVectorThreadElementsFactory());
60        SingletonLog::changeStream(Singleton-
61  Log::NO_ONE);
62        SingletonScheduler::changeSched-
63  uler(SchedulerStrategy::NO_ONE);
64        ProposedMethod::initialAlocation();
65        this->startApplication();
66        Thread::destroy();
67    }
68    void MainApplication::initFactBase() {
69    }
70    void MainApplication::initSharedEntities() {
71    }
72    void MainApplication::initRules() {
73    }
74    void MainApplication::codeApplication() {
75    }
76    void MainApplication::startApplication(){
77        Door * door = new Door();
78        cout << "****** APP Start ****" << endl;
79        cout << " 10 sec to start" << endl;
80        sleep(10);
81        cout << "Door is CLOSE" << endl;
82        door ->atDoorState->setValue(0);
83        cout << endl;
84  }
```

De forma simplificada, pode-se considerar que as principais mudanças na geração do código entre os *Frameworks* estão vinculadas às linhas 13 a 20 e 55 a 66, representadas no Algoritmo 06.

Para disponibilizar o uso dos *Frameworks* de forma simples ao usuário, foram alterados os arquivos abaixo:

- bizon.y;
- flex.l;
- target.h;
- target.cpp;

Desta forma, o uso dos *Frameworks* dar-se-á da seguinte maneira: ao utilizar o número 1, será executado o *Framework 2.0* e ao utilizar o número 2, será executado o F*ramework 3.0.*

./NOPL 1 Application/NomedaAplicacao
./NOPL 2 Application/NomedaAplicacao

4 Estudos de Casos

Para realização efetiva de testes e apresentação de resultados referentes às implementações realizadas no *Framework 2.0* e *Framework 3.0*, foram desenvolvidas algumas aplicações básicas.

A primeira aplicação trata-se de um sensor que repre-

senta a situação de uma porta. Estando aberta, é representada com o termo OPEN ou estado 1, e estando fechada é representanda com o termo CLOSED ou estado 0.

Esta aplicação envia uma notificação contendo o estado do atributo da *FBE Door* através da rede, para que seja recebida pela central que irá realizar a alteração ou não do estado pré-estabelecido anteriormente.

Já a segunda aplicação desenvolvida corresponde ao uso de duas *FBEs,* sendo criadas a *FBE Fire e FBE People,* simulando sensores que representam em um ambiente de simulação a detecção ou não de fogo, assim como a detecção de presença ou ausência de pessoas no ambiente.

Esta aplicação envia notificações contendo o estado do atributo da *FBE Fire e FBE People* através da rede, para que sejam recebidas pela central (ou centrais) que irá analisar os estados recebidos e de acordo com as suas *Rules*, definir a abertura ou fechamento da porta fictícia.

4.1 Framework 2.0

Conforme representado nos Algoritmos 01, 02 e 04, a aplicação que possui apenas a *FBE Door,* que foi gerada através da LingPON pelo *Framework 2.0,* obteve como resultado a representação próxima de 100% do código-alvo para a linguagem C++, exigindo a alteração de uma linha de código para que fosse realizada a compilação de forma direta.

Apenas o arquivo Main.cpp exigiu que a linha referente ao método principal fosse alterada, pois por padrão não retorna valores. Foi realizada a alteração listada abaixo:

- Original
  void main(int argc, char argv){
- Alteração realizada
  int main(int argc, char *argv[]) {

Para ativação do envio do estado do atributo via rede, foi necessário realizar a configuração no arquivo apresentado na figura 2, no qual foram definidos os destinos para envio das notificações.

Após as alterações citadas, o processo de geração do código-alvo é realizado através da execução do código LingPON pelo *Framework 2.0,* conforme apresentado na figura 4.

```
csm@PON:~/NOPL$ ./NOPL 1 Applications/RedeAtrib/Door/
ompiler()
rameworkCPP20Compiler()
```

Fig.4 – Execução do comando de geração do código-alvo pelo *Framework 2.0*.

Dentro do diretório criado para conter o código fonte da aplicação em C++, foi criado um script para compilação que simplifica este processo.

Conforme apresentado na figura 3, este script é executado e cria-se o arquivo *distributed_door,* que será responsável pelo envio do estado do atributo da *FBE Door* através da rede de computadores para uma central, que estará localizada em outro equipamento com objetivo de receber a notificação e analisá-la.

A figura 5 apresenta o processo de execução da aplicação *distributed_door,* através da qual é possível identificar o estado do atributo *FBE Door*, CLOSE que será enviado via rede.

```
csm@PON:~/NOPL/aplicacoes_geradas/appDoor_2$ ./distributed_door
[ponip] WARNING: no config (ponip_config.txt) file, using defaults.
************* App started *************
10 secs to start sensors
Door CLOSE

csm@PON:~/NOPL/aplicacoes_geradas/appDoor_2$ _
```

Fig.5 – Execução da aplicação.

Já a figura 6, trata-se da central que está localizada em outro equipamento que receberá a notificação enviada pelo sensor *FBE Door,* e de acordo com o estado recebido, irá informar o estado da porta, sendo aberta ou fechada. No inicio do processo, o sensor apresenta a porta aberta e após o recebimento da notificação informando que a porta está fechada, irá alterar o estado para porta fechada.

```
csm@PON:~/aplicacoes/app_Door_Atrib$ ./distributed_fire-central
************* App started *************
Door is open
sleeping for 20 sec
[ponip] DEBUG: received: [RAW] PKT ATTR = *11111atDoorState0*
[ponip] DEBUG: cheking attr [atDoorState]
[ponip] DEBUG: attribute [atDoorState] found!
** at the end of DistributedFire
Door is closed
```

Fig.6 – Recebimento da notificação do estado da *FBE Door* pela central.

4.1 Framework 3.0

Conforme representado no Algoritmo 06, a aplicação gerada pelo *Framework 3.0* obteve como resultado um sucesso inferior ao gerado pelo *Framework 2.0*, pois além da mesma alteração no método principal, foi necessário mover o código de execução da aplicação do arquivo Main.cpp para o arquivo MainApplication.cpp e inserir algumas entradas no arquivo MainApplication.h.

O Algoritmo 07 apresenta o código Main.ccp gerado diretamente pelo *Framework 3.0* e o Algoritmo 08 apresenta as alterações necessárias para que seja possível realizar a compilação em C++.

	Algoritmo 07: Main.cpp
1	#include "Main.h"
2	#include <iostream>
3	using std::cout;
4	using std::cout;
5	Main::Main(void) {
6	}
7	Main::~Main(void) {
8	}
9	void Main::main(int argc, char argv) {
10	
11	MainApplication *s = new MainApplica-
12	tion();
13	Door * door = new Door();
14	cout << "************* App started
15	*************" << endl;
16	cout << "10 secs to start sensors" << endl;
17	sleep(10);
18	cout << "Door CLOSE" << endl;
19	door->atDoorState->setValue(0);
20	cout << endl;
21	}

Algoritmo 08: Main.cpp adaptado	
1	#include "Main.h"
2	#include <iostream>
3	using std::cout;
4	using std::cout;
5	Main::Main(void) {
6	}
7	Main::~Main(void) {
8	}
9	int main(int argc, char *argv[]) {
10	MainApplication *s = new MainApplica-
11	tion();
12	}
13	

O código removido do arquivo Main.cpp foi movido para o arquivo MainApplication.cpp, conforme apresentado no Algoritmo 06.

A ativação do envio do estado do atributo via rede foi igual ao processo realizado no código-alvo gerado pelo *Framework* anterior, mas neste caso, exige-se a inserção da informação relacionada ao número de processadores que serão utilizados para execução da aplicação, pois cabe lembrar que o *Framework* 3.0 possui como principal característica a possibilidade de utilização de *multicore*.

Após as alterações citadas, o processo de geração do código-alvo é realizado através da execução do código LingPON pelo *Framework 3.0,* conforme apresentado na figura 7.

```
ccsm@PON:~/NOPL$ ./NOPL 2 Applications/RedeAtrib/Door/
Compiler()
FrameworkCPP30Compiler()
```

Fig.7 – Execução do comando de geração do código-alvo pelo *Framework 3.0*.

Dentro do diretório criado para o código fonte da aplicação em C++, foi criado um script para compilação assim como o utilizado na versão anterior para simplificar o processo de compilação, mas para este arquivo foram necessárias adaptações para inclusão de funções adicionais, conforme apresentado na figura8.

```
$FRAMEWORK_DIR/Source/framework/multicore/*.cpp
$FRAMEWORK_DIR/Source/framework/multicore/entities/*.cpp
$FRAMEWORK_DIR/Source/framework/multicore/entities/attributes/*.cpp
$FRAMEWORK_DIR/Source/framework/multicore/factories/*.cpp
$FRAMEWORK_DIR/Source/framework/proposedMethod/*.cpp
$FRAMEWORK_DIR/Source/framework/system/*.cpp
```

Fig.8 – Inserções adicionais ao arquivo apresentado na figura 3.

Ao executar este script, cria-se o arquivo *distributed_door,* que será responsável pelo envio do estado do atributo da *FBE Door* através da rede de computadores para a central, que estará localizada em outro equipamento.

A figura 9 apresenta o processo de execução da aplicação *distributed_door,* através da qual é possível identificar a verificação por parte da aplicação da quantidade de processadores disponíveis no equipamento e a alocação conforme necessária.

```
ccsm@PON:~/NOPL/aplicacoes_geradas/appDoor_3$ ./make_framework3.sh
ccsm@PON:~/NOPL/aplicacoes_geradas/appDoor_3$ ./distributed_door
Core 1: 100.00% available.
Core 2: 100.00% available.
Allocating NOP software on core 1
****** APP Start ****
 10 sec to start
Door is CLOSE
[ponip] DEBUG: destination ip = *192.168.1.113*
[ponip] DEBUG: last value=0
```

Fig.9 – Execução da aplicação.

Já a figura 10, trata-se da central que está localizada em outro equipamento na rede de computadores, que receberá a notificação enviada pelo sensor *FBE Door* e, de acordo com o estado recebido, irá informar o estado da porta aberta ou fechada. No inicio do processo, o sensor apresenta a porta aberta e após o recebimento da notificação, irá alterar o estado para porta fechada.

```
ccsm@PON:~/aplicacoes/app_Door_Atrib$ ./distributed_fire-central
************* App started *************
Door is open
sleeping for 20 sec
[ponip] DEBUG: received: [RAW] PKT ATTR = *11111atDoorState0*
[ponip] DEBUG: cheking attr [atDoorState]
[ponip] DEBUG: attribute [atDoorState] found!
** at the end of DistributedFire
Door is closed
```

Fig.10 – Recebimento da notificação do estado da *FBE Door* pela central.

Para apresentação da alocação e uso de múltiplos processadores por uma aplicação, foi desenvolvida uma aplicação mais complexa que possui mais de uma *FBE,* mas que de forma similar a anterior, envia o estado dos atributos de *FBE Fire e FBE People* para uma central. Esta, de acordo com a *Rule* pré-estabelecida, analisa o estado recebido via rede e define qual será a ação a ser realizada.

```
ccsm@PON:~/NOPL/aplicacoes_geradas/appSensores_3_adap/fire$ ./distributed_fire
Core 1: 98.99% available.
Core 2: 100.00% available.
Allocating NOP software on core 2
[ponip] WARNING: no config (ponip_config.txt) file, using defaults.
************ App started ************
10 secs to start sensors
Fire YES
```

Fig.11 – Aplicação para envio do estado do sensor Fire.

Conforme apresentado na figura 11, é possível identificar a alocação da aplicação em execução ao segundo processador do equipamento, pois trata-se do processador com menor carga no instante da execução.

Já na figura 12, pode-se verificar que a aplicação para envio do estado da *FBE People* está sendo executada pelo primeiro processador.

```
ccsm@PON:~/NOPL/aplicacoes_geradas/appSensores_3_adap/people$ ./distributed_people
Core 1: 100.00% available.
Core 2: 100.00% available.
Allocating NOP software on core 1
[ponip] WARNING: no config (ponip_config.txt) file. using defaults.
************ App started ************
10 secs to start sensors
People IN
```

Fig.12 – Aplicação para envio do estado do sensor People.

Nesta situação, a central (ou centrais) receberá os estados dos dois sensores e de acordo com a regra pré-definida irá realizar uma instigação do método para abertura ou fe-

chamento da porta. O algoritmo 09 apresenta as *Rules* existentes.

Algoritmo 09: Rule existente na Central	
1	RULE(rlDoorOpen, scheduler, Condi-
2	tion::CONJUNCTION);
3	rlDoorOpen->addPremise(prPeopleIn);
4	rlDoorOpen->addPremise(prFireYes);
5	rlDoorOpen->addPremise(prDoorIsClose);
6	rlDoorOpen->addInstigation(INSTIGA-
7	TION(door->mtDoorOpen));
8	RULE(rlDoorClose, scheduler, Condi-
9	tion::CONJUNCTION);
10	rlDoorClose->addPremise(prPeopleOut);
11	rlDoorClose->addPremise(prFireYes);
12	rlDoorClose->addPremise(prDoorIsOpen);
13	rlDoorClose->addInstigation(INSTIGA-
14	TION(door->mtDoorClose));

A *Rule* rlDoorOpen, possui como premissas a existência de pessoas e a existência de fogo no local, além da porta encontrar-se fechada. Assim, caso todas as premissas sejam atendidas, será realizada a instigação do método *mtDoorOpen,* que irá realizar a abertura da porta.

Já a *Rule* rlDoorClose, possui como premissas a ausêncoa de pessoas, a ausência de fogo no local e a porta deve encontrar-se aberta. Caso todas as premissas sejam atendidas será realizada a instigação do método *mtDoorClose,* que irá realizar o fechamento da porta.

```
./distributed_fire-central
************* App started *************
eople are out
ire = NO
oor is close
leeping for 60 sec
[ponip] DEBUG: received: [RAW] PKT ATTR = *11111atFireState1*
[ponip] DEBUG: cheking attr [atPeopleState]
[ponip] DEBUG: cheking attr [atFireState]
[ponip] DEBUG: attribute [atFireState] found!
[ponip] DEBUG: received: [RAW] PKT ATTR = *11113atPeopleState1*
[ponip] DEBUG: cheking attr [atPeopleState]
[ponip] DEBUG: attribute [atPeopleState] found!
[ponip] DEBUG: cheking attr [atFireState]
** at the end of DistributedFire
eople are in
ire = YES
oor is open
```

Fig.13 – Recebimento da notificação do estado das *FBEs* pela central e análise das regras pré-definidas.

A figura 13 apresenta o recebimento do estado dos atributos *Fire e People* por parte da central e, devido as notificações recebidas dos sensores informarem a confirmação de presença de fogo, a presença de pessoas no local, além de considerar que a porta está fechada, a central realiza uma análise das regras que possui e realiza a ativação da *Rule rlDoorOpen,* que irá instigar a execução do *mtDoorOpen* e realizar a abertura da porta.

5 Dificuldades

A nova linguagem do PON se apresenta como uma evolução natural da linguagem previamente criada, denominada versão 2.0. Essa nova versão é considerada mais completa e organizada quando comparada a versão anterior, no entanto, alguns itens ainda apresentam a necessidade de alterações ou melhorias para maior evolução.

Para geração de código-alvo em *Framework*, o qual se utiliza da linguagem C++, faz-se necessário em determinadas situações o uso de constantes, como enum, e esta funcionalidade ainda não faz parte da LingPON atual.

Outra dificuldade apresentada está no tocante ao envio do estado de um atributo através de uma rede de computadoresAté o presente momento não existe alguma forma de configuração, seja através do uso de arquivos ou através da linguagem para que se possa determinar qual atributo deverá enviar seu estado através da rede utilizando o PONIP.

Além disto, o uso e configuração de métodos não está claro na linguagem, e assim, faz-se necessária a definição de uma alternativa para retorno de valores do método, que ainda está ausente. Este foi o motivo pelo qual a aplicação mais simples através do *Framework 2.0* obteve seu código-alvo próximo de 100%. Caso esta funcionalidade estivesse contemplada, não seria necessária nenhuma intervenção para compilação da aplicação na linguagem C++. Atualmente é necessário realizar uma intervenção direta no código-alvo gerado para que seja executado pela aplicação.

Finalmente, a linguagem se faz um pouco confusa quando é necessário gerar o aquivo *Main* para qualquer um dos *Frameworks* utilizados neste artigo. Uma vez que na linguagem o *Main* é constituído de uma *Fbe,* não é possível copiar o bloco principal para a classe principal do *Framework.* Assim, exige-se uma intervenção direta nos códigos-alvo gerados, de forma similar ao citado no caso dos métodos, para que a aplicação possa ser compilada com sucesso.

Devido a falta de mais exemplos e experiência, ocorreram dificuldades relacionadas à execução e testes de aplicações que apresentassem em execução as diretrizes referentes à ativação da propriedade *multicore*. Assim, a apresentação deste quesito foi simples, mas funcional.

6 Considerações Gerais

Conforme foi possível observar no decorrer deste artigo, a nova implementação da linguagem e compilador PON, a qual faz uso do Grafo PON, foi pensada para expor de maneira simples os conceitos relativos ao PON e permitir a criação de código de uma maneira mais pragmática no que se refere aos seus conceitos fundamentais.

De modo geral, esta nova implementação mostra-se coesa, pois os elementos foram criados com um nível de abstração coerente, permitindo que a geração do código pelo compilador aconteça, de certa forma de maneira simples, exigindo um baixo nível de conhecimento prévio dos conceitos fundamentais do paradigma.

7 Conclusão

Após as implementações realizadas junto aos *Frameworks,* é possível identificar que a possibilidade de uma pessoa

que seja responsável pelo desenvolvimento de uma aplicação em PON atingir o seu objetivo principal em gerar o código-alvo em C++ é algo viável e possível.

As implementações de integração do PONIP e Multicore nos *Frameworks* permitem uma maior versatilidade dos códigos-alvo através de tais funcionalidades, sendo que deve ser considerado que tais funcionalidades são disponibilizadas nas aplicações alvo através da realização de uma pré-configuração por parte responsável pela aplicação de forma simplificada.

As contribuições apresentadas neste trabalho podem ser consiradas mais um avanço para a materialização das funcionalidades do paradigma PON

Para trabalhos futuros, sugere-se:

- Implementação de uma técnica que permita a especificação de quais são os atributos que deverão ter seus estados enviados através da rede de computadores;
- Implementação de aplicações mais complexas que possam apresentar melhorias com o uso do processamento *multicore*.

References

[1] J. M. Simão, "A Contribution To The Development Of A HMS Simulation Tool And Proposition Of A Meta-Model For Holonic Control," Tese. Doutorado em Engenharia Elétrica e Informática Industrial - CPGEI. Universidade Tecnológica Federal do Paraná (UTFPR), Curitiba, p. 168, 2005

[2] R. R. Linhares. A Contribution to the Development of a Computer Architecture Proper to the Notification Oriented Paradigm",Tese. Doutorado em Engenharia Elétrica e informática Industrial – CPGEI. Universidade Tecnológica Federal do Paraná (UTFPR), Curitiba

[3] A. F. Ronszcka, G. Z. Valença; R. R. Linhares; P. C. Stadzisz, J. M. Simão. *Notification-Oriented Paradigm Framework 2.0: An Implementation Based On Design Patterns*. IEEE LA - IEEE Latin America Transactions, Vol. 15, Issue 11, Nov. 2017. ISSN: 1548-0992.

[4] L. A. Santos, J. M. Simão, J. A. Fabro. Linguagem e Compilador para o Paradigma Orientado a Notificações Avanços para a Redução de Complexidade de Código, 2017. VII SBESC - Brazilian Symposium on Computing Systems Engineering - November 07 - 10, 2017 - Curitiba - Paraná – Brazil.

[5] R. R. Linhares, A. F. Ronszcka, G. Z. Valença, M. V. Batista, F. A. Witt, C. R. E. Lima, J. M. Simão, and P. C. Stadzisz, "Comparações entre o paradigma orientado a objetos e o paradigma orientado a notificações sob o contexto de um simulador de sistema telefônico," in III Congresso Internacional de Computación y Telecom.-COMTEL, Lima, Peru, 2011.

[6] J. M. Simão, C. A. Tacla, P. C. Stadzisz, R. F. Banaszewski et al., "Notification oriented paradigm (nop) and imperative paradigm: A comparative study," 2012, journal of Software Engineering and Applications (JSEA), p.402-416, v.5, n.6, 2012. ISSN: 1945-3116. DOI 10.4236/jsea.2012.59083.

[7] J. M. Simão, D. L. Belmonte, A. F. Ronszcka, L. R. R., G. Z. Valença, R. F. Banaszewski, J. A. Fabro, C. A. Tacla, P. C. Stadzisz, and M. V. Batista, "Notification oriented and object oriented paradigm comparison via sale system," Journal of Software Engineering and Applications, vol. 5, no. 09, pp. 695–710, 2012, iSSN 1945-3116. DOI 10.4236/jsea.2012.56047.

[8] J. M. Simão, C. A. Tacla, P. C. Stadzisz, and R. F. Banaszewski, "Notification oriented paradigm (nop) and imperative paradigm: A comparative study," *Journal of Software Engineering and Applications*, vol. 5, no. 6, pp. 402–416, 2012.

[9] D. L. Belmonte, A. F. Ronszcka, R. R. Linhares, R. F. Banaszewski, C. A. Tacla, P. C. Stadzisz, and M. Batista, "Notification oriented and object oriented paradigms comparison via sale system," *Journal of Software Engineering and Applications*, vol. 5, no. 9, pp. 695–710, 2012.

[10] D. L. Belmonte, M. V. Batista, R. R. Linhares, R. F. Banaszewski, C. A.Tacla, P. C. Stadzisz, and A. F. Ronszcka, "A game comparative study: Object-oriented paradigm and notification-oriented paradigm," *Journalof Software Engineering and Applications*, vol. 5, no. 9, pp. 722–736, 2012.

[11] C. A. Ferreira, *Linguagem e Compilador para o Paradigma Orientado a Notificações (PON): Avanços e Comparações*, Dissertação de Mestrado, PPGCA/UTFPR, 2015.

[12] M. Talau, A. F. Ronszcka, J. M. Simão. PONIP: Uso do Paradigma Orientado a Notificações em Redes IP. Aplicação em Framework PON 2.0 C++. Doutorando CPGEI/UTFPR, 2016. Disciplina sobre Paradigma Orientado a Notificações (PON), CPGEI-PPGCA/UTFPR, Curitiba - PR, Brasil, 2016.

[13] D. L. Belmonte, R. R. Linhares, P. C. Stadzisz; J. M. Simão A new Method for Dynamic Balancing of Workload and Scalability in Multicore Systems. IEEE Latin America Transactions, Vol. 14, Issue 7, Jul 2016 Pg. 3335-3344. ISSN: 1548-0992. 2016. DOI: 10.1109/TLA.2016.7587639 .

[14] A. F. Ronszcka, LingPON- linguagem de programação e compiladorpara o paradigma orientado a notificações (pon) – uma materialização efetiva para a validação das propriedades elementares do pon. Qualificação de Doutorado, CPGEI/UTFPR, 2018. Prof. J. M. Simão, Prof. J.A. Fabro.

Christian Carlos Souza Mendes. nasceu em Curitiba-PR, em 1982. Possui mestrado em Engenharia Elétrica e Informática Industrial pela Universidade Tecnológica Federal do Paraná (2008). Tem experiência na área de Ciência da Computação, com ênfase em Redes de Computadores e Segurança da Informação. Atualmente, desde 2008, Professor na UTFPR.
http://lattes.cnpq.br/8012110786536981.

Cleverson Avelino Ferreira. nasceu em Curitiba-PR, em 1981. Possui mestrado em Computação Aplicada pela Universidade Tecnológica Federal do Paraná (2015). Tem experiência na área de Engenharia da Computação, com ênfase em Análise e Desenvolvimento de Sistemas. Atualmente, desde 2012, Professor na Universidade Positivo.
http://lattes.cnpq.br/0436181767990412.