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

CLEVERSON AVELINO FERREIRA

LINGUAGEM E COMPILADOR PARA O PARADIGMA ORIENTADO A NOTIFICAÇÕES (PON): AVANÇOS E COMPARAÇÕES

DISSERTAÇÃO

CURITIBA
2015

CLEVERSON AVELINO FERREIRA

LINGUAGEM E COMPILADOR PARA O PARADIGMA ORIENTADO A NOTIFICAÇÕES (PON): AVANÇOS E COMPARAÇÕES

Dissertação apresentada como requisito parcial à obtenção do título de Mestre em Computação, do Programa de Pós-Graduação em Computação Aplicada, da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. Jean Marcelo Simão
Co-orientador: Prof. Dr. João Alberto Fabro

CURITIBA

2015

Dados Internacionais de Catalogação na Publicação

F383L Ferreira, Cleverson Avelino
2015 Linguagem e compilador para o paradigma orientado a notificações (PON) : avanços e comparações / Cleverson Avelino Ferreira.-- 2015.
xvii, 227 f.: il.; 30 cm

Texto em português, com resumo em inglês.

Dissertação (Mestrado) - Universidade Tecnológica Federal do Paraná. Programa de Pós-graduação em Computação Aplicada, Curitiba, 2015.
Bibliografia: f. 134-139.

1. Paradigma orientado a notificações. 2. Linguagem de programação (Computadores). 3. Compiladores (Programa de computador). 4. Software - Desenvolvimento. 5. C++ (Linguagem de programação de computador). 6. Framework (Programa de computador). 7. Métodos de simulação. 8. Computação - Dissertações. I. Simão, Jean Marcelo, orient. II. Fabro, João Alberto, coorient. III. Universidade Tecnológica Federal do Paraná - Programa de Pós-graduação em Computação Aplicada. IV. Título.

CDD 22 -- 621.39

Biblioteca Central da UTFPR, Câmpus Curitiba

ATA DE DEFESA DE DISSERTAÇÃO DE MESTRADO Nº 36

Aos 28 dias do mês de agosto de 2015 realizou-se na sala C-301 a sessão pública de Defesa da Dissertação de Mestrado intitulada "Linguagem e Compilador para o Paradigma Orientado a Notificações (PON): Avanços e Comparações", apresentada pelo aluno **Cleverson Avelino Ferreira** como requisito parcial para a obtenção do título de Mestre em Computação Aplicada, na área de concentração "Engenharia de Sistemas Computacionais", linha de pesquisa "Engenharia de Software".

Constituição da Banca Examinadora:

Prof. Dr. Jean Marcelo Simão, UTFPR - CT (Presidente) _____

Prof. Dr. Roni Fabio Banaszewski, UTFPR – Guarapuava _____

Prof. Dr. Paulo Cézar Stadzisz, UTFPR – CT _____

Profª. Drª. Inali Wisniewski Soares, UNICENTRO _____

Prof. Dr. Robson Ribeiro Linhares, UTFPR – CT _____

Em conformidade com os regulamentos do Programa de Pós-Graduação em Computação aplicada e da Universidade Tecnológica Federal do Paraná, o trabalho apresentado foi considerado _____ (aprovado/reprovado) pela banca examinadora. No caso de aprovação, a mesma está condicionada ao cumprimento integral das exigências da banca examinadora, registradas no verso desta ata, da entrega da versão final da dissertação em conformidade com as normas da UTFPR e da entrega da documentação necessária à elaboração do diploma, em até _____ dias desta data.

Ciente (assinatura do aluno): _____

(para uso da coordenação)

A Coordenação do PPGCA/UTFPR declara que foram cumpridos todos os requisitos exigidos pelo programa para a obtenção do título de Mestre.

Curitiba PR, _____/_____/_____ _____

"A Ata de Defesa original está arquivada na Secretaria do PPGCA".

Dedico este trabalho à minha família,
professores e amigos que me ajudaram
ao longo dessa caminhada.

AGRADECIMENTOS

Certamente estes parágrafos não irão atender a todas as pessoas que fizeram parte dessa importante fase de minha vida. Portanto, desde já peço desculpas àquelas que não estão presentes entre essas palavras, mas elas podem estar certas que fazem parte do meu pensamento e de minha gratidão.

Agradeço ao meu orientador Prof. Dr. Jean Marcelo Simão, pela sabedoria com que me guiou nesta trajetória.

Agradeço ao meu co-orientador Prof. Dr. João Alberto Fabro, pelo compartilhamento do seu conhecimento e ajuda neste trabalho, assim como a co-orientação inicial do Prof. Dr. P. C. Stadzisz.

Agradeço pelo apoio do meu amigo Glauber Z. Valença pela motivação na entrada ao mestrado e pela indicação dada ao Prof. Dr. Jean Marcelo Simão.

Agradeço pela ajuda do meu amigo Adriano Francisco Ronszcka que desde o início do mestrado tem me ajudado nas mais complicadas tarefas.

Agradeço a grupo de pesquisa do PON, em especial aos alunos Clayton Kossoski, a Priscila Loris e ao A. F. Ronszcka pela ajuda na composição da primeira versão prototipal da linguagem e compilador PON, no âmbito da disciplina de Linguagens e Compiladores dos Profs. J. A. Fabro e J. M. Simão no segundo trimestre 2014.

Aos meus colegas de sala.

A Secretaria do Curso, pela cooperação.

Gostaria de deixar registrado também, o meu reconhecimento à minha família, pois acredito que sem o apoio deles seria muito difícil vencer esse desafio.

Enfim, a todos os que por algum motivo contribuíram para a realização desta pesquisa.

"O bom combate é aquele que é travado em nome de nossos sonhos." (Paulo Coelho, 1997)

RESUMO

FERREIRA, Cleverson A. **LINGUAGEM E COMPILADOR PARA O PARADIGMA ORIENTADO A NOTIFICAÇÕES (PON): AVANÇOS E COMPARAÇÕES.** 139f. Dissertação – Programa de Pós-Graduação em Computação Aplicada - Universidade Tecnológica Federal do Paraná. Curitiba, 2015.

Atuais paradigmas correntes de programação de *software*, mais precisamente o Paradigma Imperativo (PI) e o Paradigma Declarativo (PD), apresentam deficiências que afetam o desempenho das aplicações e a obtenção de "desacoplamento" (ou acoplamento mínimo) entre elementos de *software*. Com o objetivo de amenizar essas deficiências, foi desenvolvido o Paradigma Orientado a Notificações (PON). O PON se inspira nos conceitos do PI (e.g. objetos) e do PD (e.g. base de fatos e regras), mas altera a essência da execução ou inferência lógica-causal. Basicamente, o PON usa objetos para tratar de fatos e regras na forma de composições de outros objetos menores que, entretanto, apresentam características comportamentais de certa autonomia, independência, reatividade e colaboração por meio de notificações pontuais para fins de inferência. Isto dito, salienta-se que a materialização dos conceitos do PON se deu por meio de um arquétipo ou *Framework* elaborado em linguagem de programação C++. Tal materialização do PON vem sendo utilizada como uma alternativa para o desenvolvimento de aplicações sob o domínio desse paradigma e possibilitou, de fato, a criação de aplicações para ambientes computacionais usuais baseados na chamada arquitetura Von Neumann. Apesar destas contribuições para com a sua materialização, o desenvolvimento de aplicações no PON ainda não apresentava resultados satisfatórios em termos de desempenho tal qual deveria a luz do seu cálculo assintótico, nem a facilidade de programação que seria uma das suas características principais. Nesse âmbito, o presente trabalho propõe como evolução para o estado da técnica do PON a criação de uma linguagem e compilador para o paradigma. Sendo assim, este trabalho apresenta a definição da linguagem criada com a utilização de exemplos práticos guiados pelo desenvolvimento de aplicações. Subsequentemente são apresentados detalhes do compilador bem como sua estrutura. Para demonstrar a evolução do estado da técnica do paradigma, no tocante a desempenho (e.g. tempo de processamento) e facilidade de programação foram realizados estudos comparativos com a utilização da linguagem e compilador. Os estudos comparativos foram guiados com a elaboração de dois *softwares* denominados aplicação Mira ao Alvo e aplicação de Vendas. Essas aplicações foram desenvolvidas com base na linguagem PON e foram realizados experimentos simulando sequências de execução com o intuito de avaliar o tempo de processamento para o resultado gerado pelo compilador PON. Ainda, tais experimentos possibilitaram a avaliação de maneira subjetiva da linguagem de programação PON no tocante a facilidade de programação. Deste modo, foi possível observar com tais estudos comparativos que os resultados apresentados pelo compilador PON foram satisfatórios quando comparados aos resultados obtidos pelo

Framework e por aplicações equivalentes desenvolvidas baseadas no Paradigma Orientado a Objetos (POO).

Palavras-chave: Paradigma Orientado a Notificação (PON), Linguagem e Compilador PON, Comparações entre materializações do PON.

ABSTRACT

FERREIRA, Cleverson A. PROGRAMMING LANGUAGE AND COMPILER TO THE NOTIFICATION ORIENTED PARADIGM (NOP): ADVANCES AND COMPARISONS. 139p. Dissertação – Programa de Pós-Graduação em Computação Aplicada - Universidade Tecnológica Federal do Paraná. Curitiba, 2015.

The current software development paradigms, specifically the Imperative Paradigm (IP) and the Declarative Paradigm (DP), have weaknesses that affect the applications performance and decoupling (or minimal coupling) between the software modules. In order to provide a solution regarding these weaknesses, the Notification Oriented Paradigm (NOP) was developed. NOP is inspired by the concepts of the IP (e.g. objects) and DP (e.g. base of facts and *Rules*). Basically, NOP uses objects to deal with facts and *Rules* as compositions of other, smaller, objects. These objects have the following behavioral characteristics: autonomy, independence, responsiveness and collaboration through notifications. Thus, it's highlighted that the realization of these concepts was firstly instantiated through a *Framework* developed in C++. Such NOP materialization has been used as an alternative for *Application* development in the domain of this paradigm and made possible, in fact, the creation of applications for typical computing environments based on Von Neumann architecture. The development of the C++ materialization of NOP has not presented satisfactory results in terms of performance as it should when taking into account its asymptotic calculation and programming facility. In this context, this work presents an evolution of NOP by creating a specific programming language, and its respective compiler, for this paradigm. Therefore, this work presents the language definition and the details of the development of its compiler. To evaluate the evolution regarding to performance (e.g. processing time) and programming facility, some comparative studies using the NOP language and compiler are presented. These comparative studies were performed by developing two software applications called Target and Sales *Application*. These applications have been developed based on NOP language, and the experiments were performed simulating sequences of execution in order to evaluate the processing time for the generated results by NOP compiler. Still, these experiments allowed the evaluation of NOP programming language, in a subjective way, regarding to ease programming. Thus, with such comparative studies, it was possible to observe that the results presented by the compiler NOP were satisfactory when compared to the results achieved via *Framework* and for equivalent applications developed based on the Oriented Object Paradigm (OOP).

Keywords: Notification Oriented Paradigm (NOP), NOP Language and Compiler, Comparisons of NOP materializations.

LISTA DE ILUSTRAÇÕES

LISTA DE ALGORITMOS

LISTA DE SIGLAS

SIGLA	***Original***	***Tradução***
BNF	*Backus-Naur Form*	Forma de Backus-Naur
CPLD	*Complex Programmable Logic Device*	Dispositivo Lógico Complexo Programável
FBE	*Fact Base Element*	Elemento da base de fatos
FIFO:	*First In*, *First Out*	Primeira a entrar, primeiro a sair
LIFO	*Last In*, *First Out*	Último a entrar, primeiro a sair
LR	*Left to right*	Esquerda para direita
OO	*Object Oriented*	Orientado a Objetos
PC	*Personal Computer*	Computador Pessoal
PD	Paradigma Declarativo	*Declarative Paradigm*
PI	Paradigma Imperativo	*Imperative Paradigm*
PON	Paradigma Orientado a Notificações	*Notification Oriented Paradigm*
POO	Programação Orientada a Objetos	*Object Oriented Programming*
PPGCA	Programa de Pós-Graduação em Computação Aplicada	*Postgraduate Program in Applied Computing*
SBR	Sistema Baseado em Regras	*Rule Based Systems*
VHDL	*VHSIC Hardware Description Language*	Linguagem de Descrição de Hardware VHSIC
VHSIC	*Very High Speed Integrated Circuit*	Circuito Integrado de Velocidade Muito Alta
XML	*Extensible Markup Language*	Linguagem de Marcação Extensível

SUMÁRIO

1 INTRODUÇÃO

O chamado Paradigma Orientado a Notificações (PON) vem sendo desenvolvido por um grupo de pesquisadores da Universidade Tecnológica Federal do Paraná (UTFPR), como uma alternativa para o desenvolvimento de aplicações em plataforma de *software* e *hardware* [LINHARES et al., 2014; SIMÃO e STADZISZ, 2008; 2009a]. Em suma, o PON se propõe a resolver certos problemas existentes nos paradigmas usuais de programação, nomeadamente o Paradigma Declarativo (PD) e o Paradigma Imperativo (PI) [LINHARES et al., 2014; SIMÃO e STADZISZ, 2008; 2009a].

Na verdade, o PON unifica as principais vantagens do PD (e.g. representação do conhecimento em regras) e do PI (e.g. flexibilidade de expressão e nível apropriado de abstração), ao mesmo tempo em que resolve (em termos de modelo) várias de suas deficiências e inconvenientes relativos ao cálculo lógico-causal em aplicações de *software* e também de *hardware*, supostamente desde ambientes monoprocessados a completamente multiprocessados [SIMÃO e STADZISZ, 2008; 2009a; BANASZEWSKI, 2009; SIMÃO et al., 2012a].

O PON resolve problemas destes paradigmas no tocante ao cálculo lógico-causal, como sintaxe inapropriada (e.g. miscelânea de comandos de regras causais e comandos para outros processamentos), redundâncias estruturais e temporais (i.e. redundâncias de códigos lógico-causais) e, particularmente, o acoplamento forte de entidades computacionais (i.e. difícil dissociabilidade em granularidade fina).

Justamente, o PON apresenta outra maneira de realizar estas avaliações ou inferências lógico-causais. Isto é feito por meio de entidades computacionais de pequeno porte, reativas (ou até mesmo ativas conforme implementação ou entendimento) e desacopladas que colaboram por meio de notificações pontuais e que são criadas a partir do 'conhecimento' de regras lógico-causais [SIMÃO e STADZISZ, 2008; 2009a; BANASZEWSKI, 2009; SIMÃO et al., 2012a].

Isto dito, salienta-se que a materialização destes conceitos foi primeiramente implementada na forma de arquétipo ou *Framework* usando a linguagem de programação C++, sendo que tal *Framework* se encontra na sua terceira versão [VALENÇA, 2012; RONSZCKA et al., 2011]. Tais materializações possibilitaram certa demonstração de conceito pela criação de aplicações sob o domínio desse

novo paradigma para ambientes computacionais usuais, baseados na chamada arquitetura *Von Neumann*.

Não obstante aos esforços aplicados na otimização do *Framework* PON [VALENÇA, 2012] e na criação de padrões de projeto para o desenvolvimento em PON [RONSZCKA, 2012], dentre outras melhorias a cada versão do *Framework*, de fato observou-se em [SIMÃO et al., 2012a] e [FERREIRA et al., 2013] que tais materializações não proporcionaram um resultado de todo satisfatório em termos de desempenho, tal qual seria esperado no PON à luz do seu cálculo assintótico [BANASZEWSKI, 2009].

Tal desempenho insatisfatório está relacionado inclusive ao uso de estruturas de dados computacionalmente “caras” utilizadas no desenvolvimento de cada versão do *Framework*. Exemplos de tais estruturas de dados computacionalmente “caras” são listas duplamente encadeadas da biblioteca *Standard Template Library (STL)* e tabelas virtuais utilizadas para prover a capacidade de polimorfismo da linguagem C++ [BANASZEWSKI, 2009, VALENÇA, 2012].

Enfim, chegou-se à conclusão que existe necessidade de melhorias no estado da técnica para que o PON atinja o que fora vislumbrado em seu estado da arte. Nesse âmbito, uma proposta para a evolução do estado da técnica é o desenvolvimento de uma linguagem de programação específica, acompanhada do respectivo compilador, que traduza o código PON para código-alvo mais puro e menos dependente dos conceitos de outros paradigmas, bem como evite o uso de estruturas de dados que sejam computacionalmente caras.

No tocante a esta tecnologia de linguagem e compilador para o PON, a parte linguagem visa se tornar uma alternativa para a codificação de aplicações PON com relação ao *Framework* atual. Nesse sentido, uma linguagem didática fortemente baseada nos conceitos do estado da arte do PON é aqui proposta e detalhada. Com o surgimento de tal linguagem, há a necessidade subsequente da construção de um compilador que gere o código-alvo mais puro e menos dependente de estruturas de dados e sobrecargas (*overheads*) presentes na atual implementação genérica de *Framework* PON. Isto posto, tal linguagem e compilador constituem o objeto de estudo deste trabalho e serão apresentados e discutidos ao longo dos capítulos.

Neste capítulo introdutório, a Seção 1.1 apresenta a motivação para a execução deste trabalho detalhando de forma sucinta o PON, bem como as

considerações sobre o estado atual do paradigma. A seção 1.2, por sua vez, apresenta a justificativa da criação de linguagem e compilador para o PON. A Seção 1.3, particularmente, apresenta os objetivos pretendidos com este trabalho. Por fim, a Seção 1.4 apresenta a organização dos capítulos subsequentes que compõem este trabalho.

1.1 MOTIVAÇÃO

Como já salientado nesta introdução, o PON, teve seu estado da técnica materializado na forma de um *Framework*. Este *Framework* foi desenvolvido usando a linguagem de programação C++ e se encontra na sua terceira versão, sendo denominado *Framework* otimizado [VALENÇA, 2012]. Conforme foi apresentado em trabalhos anteriores, o *Framework* PON permite a construção de aplicações sob o domínio desse novo paradigma e tornou-se uma alternativa para o desenvolvimento de aplicações no âmbito da arquitetura Von Neumann [VALENÇA, 2012, RONSZCKA, 2012].

Não obstante, acredita-se que o PON ainda pode evoluir, a luz do seu cálculo assintótico, no tocante a questões de desempenho. Ainda, vislumbra-se também a evolução do PON no tocante à facilidade de programação. Para tal, neste trabalho, são apresentados uma linguagem e compilador específicos ao PON. Com a linguagem PON (denominada doravante “LingPon”) e seu compilador (denominado doravante “compilador PON”), torna-se possível desenvolver aplicações específicas em uma linguagem conformada ao PON e gerar resultados, em termos de código, sem a adição de estruturas de dados caras.

Isto se difere da programação por macros e/ou instanciação de objetos e também do código com estruturas de dados consideradas caras, como relatado no desenvolvimento com o *Framework* PON em suas três versões [BANASZEWSKI, 2009, VALENÇA, 2012]. Portanto, a LingPon e seu compilador se propõem a ser uma alternativa para composição de aplicações sob o viés deste paradigma orientado a notificações.

Neste contexto, a Subseção 1.1.1 apresenta maiores detalhes sobre o PON. A Subseção 1.1.2 destaca as considerações sobre o *Framework* PON e a motivação para a criação de uma linguagem e compilador. A Subseção 1.1.3, por sua vez,

apresenta as considerações sobre a versão prototipal da linguagem e compilador PON.

1.1.1 Paradigma Orientado a Notificações

Em linhas gerais, o Paradigma Orientado a Notificações (PON) se propõe a resolver certos problemas existentes nos paradigmas usuais de programação, nomeadamente o Paradigma Declarativo (PD) e o Paradigma Imperativo (PI). Na verdade, o PON unifica as principais características e as vantagens do PD (e.g. representação do conhecimento em regras) e do PI (e.g. flexibilidade de expressão e nível apropriado de abstração). Ademais, o PON supostamente resolve, já em termos de modelo, várias das deficiências e inconvenientes de PI e PD em aplicações de *software* e mesmo de hardware, possivelmente desde ambientes monoprocessados a completamente multiprocessados [SIMÃO e STADZISZ, 2008, 2009a; BANASZEWSKI, 2009; SIMÃO, et al., 2012a].

De fato, o PON permite desacoplar expressões causais do código-fonte, ao considerar cada uma destas e seus fatos relacionados como entidades computacionais, as quais são objetos nas atuais implementações em *software* e módulo de circuito nas atuais implementações em *hardware*. Estas entidades são notificantes, permitindo assim uma escalabilidade de desempenho em ambientes de processamento paralelo ou não, bem como melhor aproveitamento de recursos em ambiente distribuído, tanto em implementações de *software* quanto de *hardware* [BELMONTE, et al., 2012].

Particularmente no tocante a *software*, isto é diferente dos programas usuais do PI (salientando os Orientados a Objetos - OO) e do PD (salientado os chamados Sistemas Baseados em Regras – SBR). Nestes, as expressões causais são passivas e acopladas (senão fortemente acopladas) a outras partes do código, além de haver algum ou mesmo muito desperdício de processamento, conforme o caso [GABBRIELLI e MARTINI, 2010; SIMÃO, et al., 2012a].

No PON, cada entidade computacional que trata de uma expressão causal é chamada de *Rule*. As *Rules* gerenciam o conhecimento sobre qualquer comportamento causal no sistema. O conhecimento causal de uma *Rule* provém normalmente de uma regra (se-então), o que é uma maneira natural de expressão

deste tipo de conhecimento. Não obstante, este conhecimento causal pode ser representado em outro formalismo equivalente quando se fizer pertinente, salientando-se o formalismo de Redes de Petri [SIMÃO e STADZISZ, 2008, 2009a; BANASZEWSKI, 2009; WIECHETECK, 2011; WIECHETECK et al., 2011; SIMÃO et al., 2012a].

A Figura 1 apresenta um exemplo de *Rule*, justamente na forma de uma regra causal. Uma *Rule* é composta por uma *Condition* (ou Condição) e por uma *Action* (ou Ação). A *Condition* trata da decisão da *Rule*, enquanto a *Action* trata da execução das ações dela. Assim sendo, *Condition* e *Action* trabalham para realizar o conhecimento causal da *Rule*. Na verdade, tanto a *Condition* quanto a *Action* são entidades computacionais tidas como agregadas em uma *Rule* [SIMÃO e STADZISZ, 2008, 2009].

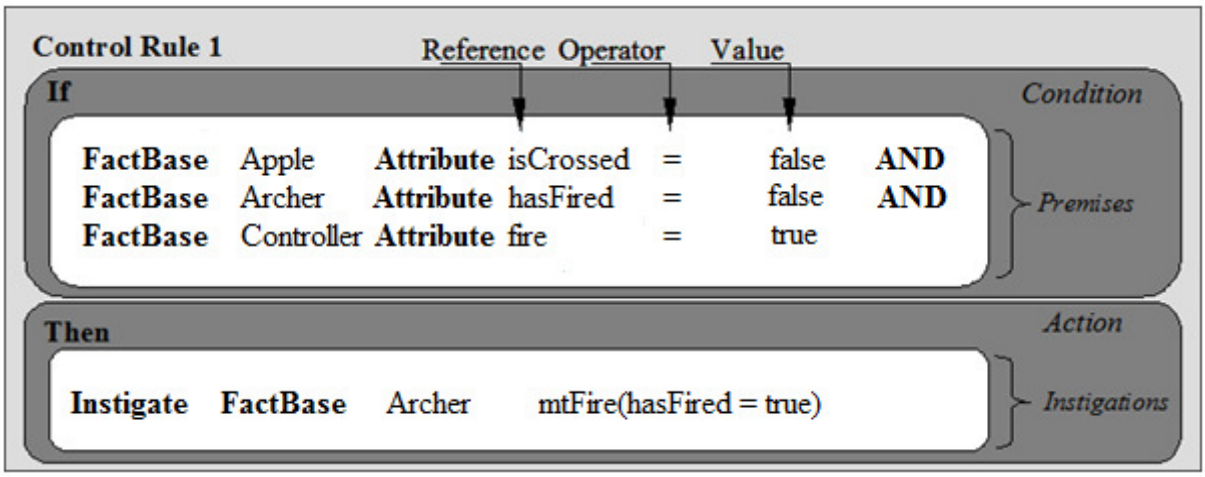


Figura 1 - Exemplo de uma *Rule* [adaptado de SIMÃO e STADIZISZ, 2008, 2009]

A *Rule* apresentada na Figura 1 faz parte de um aplicativo denominado Mira ao Alvo, que consiste na implementação de um jogo de mira ao alvo, onde entidades do tipo *Arqueiro* interagem com as entidades do tipo *Alvo*. A *Condition* desta *Rule* lida com a decisão de um dado Arqueiro ou *Archer* (a *Mira*) flechar uma determinada Maçã ou *Apple* (o *Alvo*) quando um controlador determina o disparo. Na verdade, cada um destes elementos computacionais, analisáveis por *Conditions*, são chamados de *FactBaseElements* (*FBE*) no PON [SIMÃO e STADZISZ, 2008, 2009].

Conforme ilustrado na Figura 1, a *Condition* daquela *Rule* em questão é composta por três *Premises* (ou Premissas) que se constituem em outro tipo de entidade computacional. Estas *Premises* em questão fazem as seguintes verificações sobre os *FBEs*: a) A *Apple* já foi flechada? b) O *Archer* já realizou o tiro?

c) O *Controller* permitiu o tiro? Assim sendo, conclui-se (em geral) que os estados dos atributos dos *FBEs* compõem os fatos a serem avaliados pelas *Premises* [SIMÃO e STADZISZ, 2008, 2009].

Ainda, os estados de cada um dos atributos de um *FBE* são tratados por meio de uma entidade chamada *Attribute* (ou Atributo). Além do mais, e principalmente, para cada mudança de estado de um *Attribute* de um *FBE*, ocorrem automaticamente avaliações (lógicas) somente nas *Premises* relacionadas com eventuais mudanças nos seus estados. Igualmente, a partir da mudança de estado das *Premises*, ocorrem automaticamente avaliações (causais) somente nas *Conditions* relacionadas com eventuais mudanças de seus estados [SIMÃO e STADZISZ, 2008, 2009a; BANASZEWSKI, 2009; SIMÃO et al., 2012a].

Isto tudo se dá por meio de uma cadeia de notificações entre entidades computacionais, conforme ilustra a Figura 2, o que se constitui no ponto central da inovação inferencial do PON. Em suma, cada *Attribute* notifica as *Premises* relevantes sobre seus estados somente quando se fizer efetivamente necessário. Cada *Premise*, por sua vez, notifica as *Conditions* relevantes dos seus estados usando o mesmo princípio. Baseado nestes estados notificados é que a *Condition* pode ser aprovada ou não. Se a *Condition* é aprovada, a respectiva *Rule* pode ser ativada executando sua *Action* [SIMÃO e STADZISZ, 2008, 2009a; BANASZEWSKI, 2009; SIMÃO et al., 2012a].

Uma *Action* também é uma entidade computacional que se conecta a entidades computacionais de outro tipo, as *Instigations* (ou Instigações). No exemplo dado, a *Action* contém uma *Instigation* a qual é utilizada para modificar o *status* do *Archer* configurando que o mesmo já realizou o seu tiro. Efetivamente, o que uma *Instigation* faz é instigar um ou mais métodos responsáveis por realizar serviços ou habilidades de um *FBE* [SIMÃO e STADZISZ, 2008, 2009a; BANASZEWSKI, 2009; SIMÃO et al., 2012a].

Certamente, cada método de *FBE* é também tratado por uma entidade computacional, a qual é chamada de *Method* (ou Método). Geralmente, a execução de um *Method* muda o estado de um ou mais *Attributes*. Na verdade, os conceitos de *Attribute* e *Method* representam uma evolução dos conceitos de atributos e métodos do POO. A diferença é o desacoplamento implícito da classe proprietária e a "inteligência" colaborativa pontual para com *Premises* e *Instigations* do PON [SIMÃO e STADZISZ, 2008, 2009a; BANASZEWSKI, 2009; SIMÃO et al., 2012a].

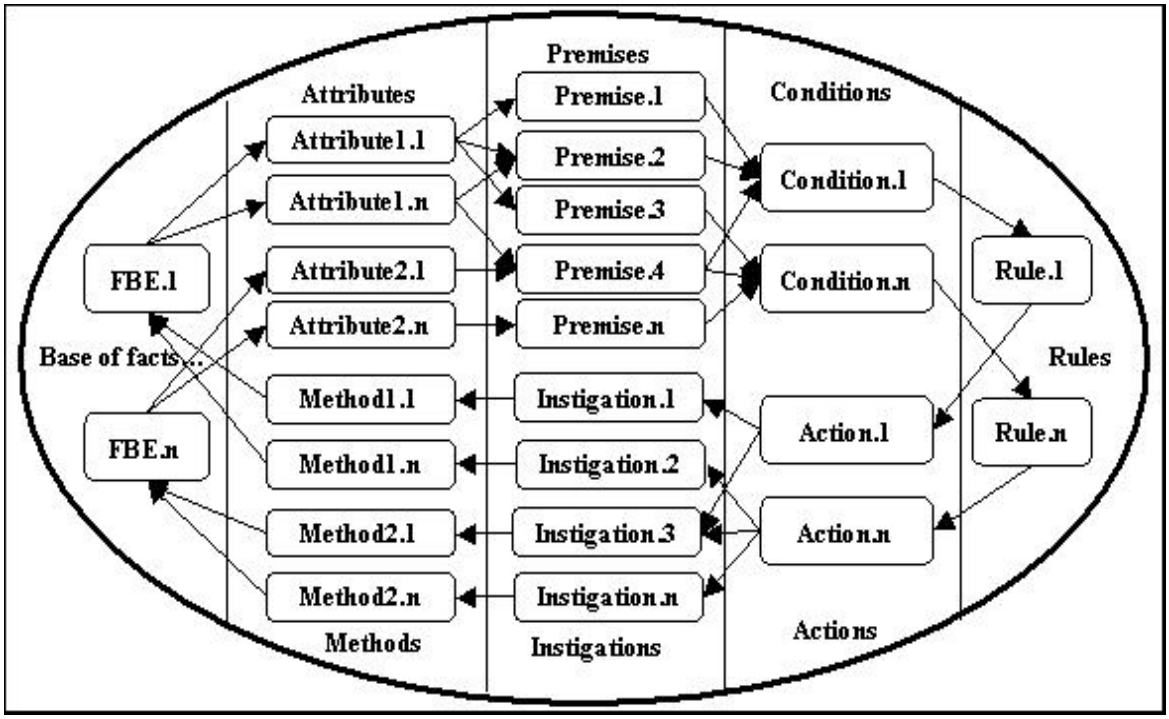


Figura 2 - Cadeia de Notificações [SIMÃO e STADZISZ, 2008, 2009a]

Com isto considerado, salienta-se que a ciência de qual elemento deve notificar qual se dá na própria composição das *Rules*, o que poderia ser feito em um ambiente amigável na forma de regras causais. Em suma, cada vez que um elemento referenciar outro, o referenciado o considera como elemento a ser notificado quando houver mudanças em seu estado. Por exemplo, quando uma *Premise* faz menção a um dado *Attribute*, este considera tal *Premise* como elemento a ser notificado. Isto permite criar o mecanismo de inferência por notificações sem esforços para tal por parte do desenvolvedor do *software* ou sistema computacional [SIMÃO e STADZISZ, 2008, 2009a; BANASZEWSKI, 2009; SIMÃO et al., 2012a].

A natureza do PON leva a uma nova maneira de compor sistemas computacionais (*software* inclusive), na qual os fluxos de execução são distribuídos e colaborativos nas entidades. De fato, aplicações ou sistemas PON apresentam um fluxo de execução não convencional, o que difere o desenvolvimento de aplicações e sistemas no PON dos demais paradigmas. Assim sendo, o PON permite uma nova maneira de estruturar, executar e pensar os artefatos de sistemas computacionais, focando neste trabalho em *software* [SIMÃO e STADZISZ, 2008, 2009a; BANASZEWSKI, 2009; SIMÃO et al., 2012a].

1.1.2 Considerações sobre o *Framework* PON

O PON foi materializado inicialmente como um arquétipo ou *Framework* de *software* que foi evoluindo ao longo do tempo. Uma versão prototipal do *Framework* foi inicialmente construída por Simão [SIMÃO e STADZISZ, 2009a; SIMÃO et al., 2012b] e uma nova versão, mais completa, foi desenvolvida por Banaszewski (2009) inclusive com o propósito de melhor estruturar o *Framework* e melhorar seu desempenho [BANASZEWSKI et al., 2007; BANASZEWSKI, 2009; SIMÃO et al., 2012a].

De maneira geral, a materialização do *Framework* PON concretizada no trabalho de Banaszewski (2009) apresentou resultados satisfatórios em comparação a programas do POO (em C++) sendo que os programas estudados apresentam considerável quantidade de redundância estrutural e temporal no tocante a cálculo lógico-causal. Tal comparação levou em conta principalmente o desempenho do ponto de vista de tempo de execução. Em contrapartida, em experimentos subsequentes, a materialização original do *Framework* PON não demonstrou vantagens satisfatórias ou mesmo demonstrou desvantagens (do ponto de vista de desempenho) em comparação a implementações POO C++ para programas com pouca ou mesmo mediana quantidade de redundância [BANASZEWSKI, 2009].

Entretanto, o cálculo assintótico do processo de inferência do PON sugere que o PON deve apresentar resultados melhores do que os obtidos com o uso do *Framework* implementado em linguagem C++ [BANASZEWSKI, 2009. A própria materialização do PON através deste *Framework* apresentava sinais explícitos de que eram passíveis evoluções, como o uso de estrutura de dados mais enxutas. Neste âmbito, melhorias na estrutura geral do *Framework* PON eram desejáveis para um melhor uso do proposto na teoria que cerca o PON.

Deste modo, o trabalho de Valença [VALENÇA et al., 2011] apresentou alternativas para otimizações e refatorações de código em toda a estrutura do *Framework*, acumulando ganhos de desempenho, que de fato resultaram em sua melhoria como um todo. Essa versão também proporcionou uma ferramenta *Wizard* a fim de facilitar o desenvolvimento das *Rules* e suas entidades colaborativas na forma de conhecimento lógico causal “Se-Então” [VALENÇA, 2012]. Outro avanço no *Framework* atual ainda foi realizado, onde pode ser constatada a agregação de

padrões de projeto para PON e facilidade de programação por meio de macros [RONSZCKA, 2012; RONSZCKA et al., 2011; SIMÃO et al., 2012a].

Ainda relacionado ao trabalho de Valença (2012), observou-se que o processo de notificações, baseado em estruturas de dados que comportam os endereços das entidades responsáveis por tal processo, representava o ponto concentrador da maior parte do custo de processamento envolvido na execução de um programa PON. Sendo assim, os esforços de otimização foram concentrados principalmente na proposta de novas estruturas de dados para minimizar os custos envolvidos nesse processo. Igualmente, os experimentos comparativos realizados demonstraram que a nova versão do *Framework* PON obteve ganhos substanciais (em torno de 30%) de desempenho em relação à versão precedente. Tal versão do *Framework* foi denominada *Framework* otimizado.

No entanto, foram observados em [SIMÃO et al., 2012a] e [FERREIRA et al., 2013] que tal materialização do PON ainda não apresenta resultados satisfatórios, em termos de desempenho, tal qual seria esperado, à luz do seu cálculo assintótico [BANASZEWSKI, 2009]. Tal desempenho se deve ao uso no *Framework* de estruturas de dados ainda computacionalmente “caras” para realizar o processo de inferência [BANASZEWSKI, 2009, VALENÇA, 2012]. Deste modo, o presente trabalho motiva-se por tal constatação e apresenta como alternativa para evolução do estado da técnica do PON uma linguagem e compilador específicos ao PON com o objetivo de eliminar ou minimizar tais problemas apresentados.

1.1.3 Considerações sobre a versão prototipal da Linguagem e Compilador para o PON

Existe um esforço prévio a esta presente dissertação no tocante a uma versão prototipal da Linguagem e Compilador para o PON. Esta versão foi objeto de estudo de uma disciplina de Linguagens e Compiladores, a qual foi ofertada no segundo trimestre de 2014 e ministrada pelo Prof. Dr. João Alberto Fabro e pelo Prof. Dr. Jean Marcelo Simão. Em suma, a versão prototipal foi apenas uma primeira demonstração sobre a viabilidade de se desenvolver uma linguagem e compilador para o PON.

Com relação aos resultados alcançados pela versão prototipal, destacam-se uma definição inicial da linguagem e a criação do compilador para a mesma. Este

compilador prototipal gera código em três linguagens alvos, nominalmente código específico em C sob o viés do PON, código específico em C++ sob o viés do PON e código no *Framework* do PON Otimizado. Ainda, houve a criação de uma aplicação intitulada Mira ao Alvo cujo objetivo foi realizar uma primeira validação do funcionamento deste protótipo de linguagem e compilador PON.

No entanto, essa versão prototipal não contou com a inclusão de todas as características presentes no estado da arte do PON (detalhadas na Seção 2.2), sendo possível destacar a falta da inclusão das/de:

- estratégias de escalonamento;
- prioridade em *Rules*;
- questão de *Attributes* impertinentes;
- questão da *Rules* dependentes;
- variações nas construções dos *Methods* das *FBEs*;
- inclusão de todos os tipos de dados disponíveis do PON; e
- a estratégia de resolução de conflitos *Keeper*.

Além disso, tal versão demonstrava a necessidade ser mais estável (i.e. livre de erros), ser avaliada com a criação de mais aplicações e ser utilizada por mais desenvolvedores. Também havia a necessidade de compará-la com outras formas de desenvolvimento, particularmente a programação imperativa.

Isto considerado, a partir dos esforços desta versão prototipal, o presente trabalho tem o intuito de alcançar uma linguagem e compilador para o PON que inclua as características aqui citadas e presentes no estado da arte do PON, bem como crie um artefato estável neste âmbito.

Por fim, pertinente salientar que o autor deste trabalho contribuiu no desenvolvimento da versão prototipal, de maneira equânime com os outros três colegas e os dois professores da disciplina, participando da definição da sintaxe da linguagem, na construção das etapas de análise léxica e da análise sintática e, ainda, na criação da aplicação Mira ao Alvo. Ademais, de maneira solo, o presente autor foi o responsável pelo desenvolvimento do módulo de geração de código em *Framework* PON Otimizado e em código C++ específico sob o viés do PON.

Em tempo, os detalhamentos sobre esta versão prototipal e o tipo de participação de cada autor no desenvolvimento da versão da linguagem e compilador do PON são apresentados no apêndice A

1.2 JUSTIFICATIVA

Na atual versão do PON, os conceitos foram materializados na forma de um *Framework* desenvolvido usando a linguagem de programação C++. Conforme salientado na Subseção 1.1.2, o *Framework* PON se encontra na sua terceira versão, sendo denominado *Framework* Otimizado [VALENÇA, 2012]. De fato, tal materialização se apresenta como uma alternativa para o desenvolvimento de aplicações sob o viés deste paradigma. No entanto, conforme apresentado em trabalhos anteriores, tal materialização ainda não alcançou resultados satisfatórios em termos de desempenho [SIMÃO et al., 2012a; FERREIRA et al., 2013].

Tal constatação se dá pelo fato da atual estrutura do *Framework* ser genérica e dinâmica. Isto aumenta o custo de processamento da execução do processo de inferência do PON e culmina nos resultados apresentados até o momento, os quais não são ainda de todo satisfatório a luz do cálculo assintótico do PON, dito de ordem polinomial para o pior caso e constante para o caso médio . [SIMÃO, 2005; RONSZCKA, 2012]. Deste modo, de maneira a evitar o uso de estruturas de dados caras, seria pertinente a criação de uma implementação ou materialização do PON que permitisse criar os objetos notificantes e notificados do PON e conectá-los diretamente em tempo de compilação sem uso de estruturas de dados.

Isto seria possível com a criação de uma linguagem e compilador próprio para o PON. Este compilador se encarregaria de materializar de forma otimizada as relações entre os diversos elementos componentes do mecanismo de notificações (e.g *Attributes*, *Premises*, *Conditions*, *Rules*) ao mesmo tempo em que se poderia continuar fazendo uso da linguagem imperativa pertinente na programação dos métodos [BANASZEWSKI, 2009].

No tocante a facilidade de programação, por sua vez, o PON apresenta uma forma de compor *software* em alto nível que é chamado de Programação Orientada a Regras. Isto se dá pela representação do conhecimento lógico-causal com a utilização de regras lógico-causais, inspirando-se nos Sistemas Baseados em Regras (SBRs) que facilitam a programação por deixá-la em alto nível. Assim sendo, uma das características fundamentais do PON é a facilidade de programação, de certa forma herdada dos SBRs. No entanto, com a utilização do *Framework*, as regras podem ficar dispersas no código fonte, tornando difícil o entendimento do

software composto. Ainda, a utilização do *Framework*, de certa forma, é uma tarefa difícil, pois o desenvolvedor necessita do conhecimento prévio e técnico do *Framework* (com seus componentes) para ser possível compor o *software*.

Sendo assim, com o advento da linguagem e compilador PON o desenvolvedor poderá compor as regras em alto nível utilizando elementos da linguagem para tal. Com isto, espera-se que seja possível facilitar a localização das regras compostas e as suas conexões entre os elementos. Além disso, com a utilização da linguagem PON o desenvolvedor deverá somente entender os conceitos relacionados ao paradigma e a intuitiva linguagem PON para compor tal *software*. Portanto, vislumbra-se tornar a linguagem PON uma alternativa concreta para composição de *software* baseado em tal paradigma.

1.3 OBJETIVOS

Tendo em vista o apresentado até agora em termos de motivação e justificativa, este trabalho apresenta como objetivo principal:

(i) Evoluir o estado da técnica do Paradigma Orientado a Notificações (PON) no tocante a questões de desempenho e facilidade de programação por meio da criação e validação de uma linguagem para o PON e de um compilador para tal linguagem.

Para atingir este objetivo geral, o presente trabalho de pesquisa tem os seguintes objetivos específicos:

- Desenvolver a especificação da linguagem de programação para o PON a luz de seu estado da arte e da técnica;
- Criar uma versão estável do compilador para esta linguagem, com geração de código para o atual *Framework* PON com o intuito de validar o funcionamento do compilador;
- Criar uma versão estável do compilador para a linguagem do PON com geração de código específico em C++ e C sob o viés do PON a fim de avaliar questões de desempenho;

- Realizar estudos comparativos entre os resultados obtidos por duas aplicações, desenvolvidas diretamente na linguagem de programação e no compilador PON, mas que funcionam em três versões: (1) código específico em C sob o viés do PON, (2) código específico em C++ sob o viés do PON e (3) código PON em Framework PON Otimizado;
- Realizar estudos comparativos entre os resultados obtidos por duas aplicações, desenvolvidas diretamente na linguagem de programação e no compilador PON com versões das mesmas aplicações desenvolvidas em POO C++, a fim de validar questões de desempenho;
- Realizar estudos no tocante a facilidade de programação com versões das mesmas aplicações desenvolvidas em Linguagem PON, puramente em *Framework* PON e em POO C++; e
- Analisar relatos de outrem sobre a utilização da Linguagem e Compilador PON para avaliar questões relacionadas a facilidade de programação e co-validar o funcionamento do compilador.

1.4 ESTRUTURA DO TRABALHO

O capítulo 1 apresenta o problema que motivou o desenvolvimento do trabalho, contextualizou a justificativa para sua elaboração e, por fim, definiu seu objetivo geral e seus objetivos específicos.

No capítulo 2 é apresentada a revisão da literatura relacionada ao presente trabalho no que se refere aos paradigmas consolidados e emergentes, onde ao final fazem-se reflexões comparativas entre os paradigmas. Ainda, este capítulo dá ênfase em maiores detalhes ao Paradigma Orientado a Notificações (PON), tendo como premissa a melhor elucidação de sua origem, concepção e materialização. Este capítulo também conta com uma revisão de conceitos e ferramentas para construção de compiladores aqui utilizadas.

No capítulo 3 são apresentados detalhes sobre a linguagem e compilador desenvolvidos para o PON. Essa apresentação destaca as características da linguagem, bem como define a forma de utilização da mesma com exemplos práticos. No tocante ao compilador, são apresentados detalhes da sua implementação, bem como toda estrutura criada para o seu funcionamento. Ainda,

neste capítulo são apresentados os resultados do compilador no que diz respeito a geração dos códigos intermediários para código PON no *Framework PON*, código específico PON em linguagem C++ e código específico PON em linguagem C.

No capítulo 4 são apresentados os estudos comparativos entre os códigos gerados pelo compilador PON, bem como código equivalentes em POO. O intuito destes estudos comparativos foi avaliar, em termos de tempo de processamento e memória, o desempenho entre as abordagens. Para realizar os estudos foi necessária a criação das aplicações denominadas “Mira ao Alvo” e “Aplicação de Vendas”, ambas desenvolvidas com a linguagem PON proposta. Ainda, foram realizados estudos no tocante a facilidade de programação a fim de observar se a linguagem PON se apresenta com uma alternativa amigável para o desenvolvimento de software.

Finalmente, no capítulo 5 são apresentadas as conclusões relacionadas ao presente trabalho e são também discutidas as possibilidades de trabalhos futuros. Ainda, os apêndices e anexos apresentam temas pertinentes a este trabalho .

2 REVISÃO DA LITERATURA

Este capítulo apresenta os principais conceitos para estruturar este trabalho. Primeiramente, a Seção 2.1 apresenta reflexões sobre o atual estado da arte dos paradigmas e linguagens de programação usuais da computação. A Seção 2.2 apresenta o estado da arte do PON em uma revisão mais aprofundada àquela apresentada na Subseção 1.1.1, bem como detalha as atuais materializações do PON no tocante a *software* e *hardware*. Ainda, a Seção 2.2 apresenta as reflexões sobre a evolução do PON e sobre as vantagens da utilização de uma linguagem e compilador. Por sua vez, a Seção 2.3 apresenta uma revisão da literatura relacionada a linguagens e compiladores. Por fim, a Seção 2.4 apresenta as reflexões sobre a revisão da literatura apresentada neste capítulo.

2.1 PARADIGMAS

A utilização de técnicas de Paradigma Imperativo (PI), como o Paradigma de Programação Orientada a Objetos (POO), costuma atrair os desenvolvedores devido a questões como inércia cultural, riqueza de abstração e flexibilidades algorítmicas. Em PI, em suma, concebem-se programas como sequências de instruções utilizando-se para tanto de buscas ou percorrimentos sobre entidades passivas (dados e comandos) organizadas segundo uma lógica de execução que envolve, inclusive, a avaliação de expressões lógico-causais (e.g. se-então) [BANASZEWSKI et al., 2007; BROOKSHEAR, 2006; GABBRIELLI e MARTINI, 2010].

Particularmente, estas expressões lógico-causais são frequentemente avaliadas desnecessariamente, degradando desempenho. Isto pode ser exemplificado considerando um conjunto de expressões se-então que avaliam os estados de objetos dentro de um laço de repetição dito ‘infinito’. Cada expressão condicional avalia certos estados dos atributos de objetos e, se aprovada, chama alguns métodos dos objetos que podem mudar os estados dos atributos. Isto é apresentado no Algoritmo 1 [BROOKSHEAR, 2006; GABBRIELLI e MARTINI, 2010; SIMÃO e STADZISZ, 2008; SIMÃO e STADZISZ, 2009a].

Algoritmo 1: Pseudocódigo para o Paradigma Imperativo

```
while (true) do
        if((objeto1.atributo1 = 1) AND (objeto2.atributo1 = 1)) then
                objeto1.método1();
                objeto2.método2();
        end_if
        if( (objeto1.atributoN = N) AND (objeto2.atributoN = N)) then
                objeto1.métodoN();
                objeto2.métodoN();
        end_if
end_while
```

Neste exemplo, é observado que o laço de repetição força a avaliação (ou inferência) de todas as condições de maneira sequencial. Entretanto, muitas delas são desnecessárias porque somente alguns objetos têm o valor de seus atributos modificado e, mesmo se modificados, não necessariamente os novos valores permitem aprovar uma porção das condições. Isto até pode ser considerado não importante neste exemplo simples e pedagógico, sobretudo se o número N de expressões causais for pequeno. Entretanto, se for considerado um sistema complexo, integrando muitas partes como aquela (particularmente com condições dispersas no código), pode-se ter uma grande diferença de desempenho [FORGY, 1982; SIMÃO e STADZISZ, 2008; SIMÃO e STADZISZ, 2009a].

Por sua vez, em Paradigma Declarativo (PD), salientando Sistemas Baseados em Regras (SBR), existe a vantagem da programação em alto nível. Primeiramente, define-se uma Base de Fatos composta por entidades como objetos com atributos/métodos. Subsequentemente define-se a Base de Regras com relações lógico-causais relativas aos elementos da Base de Fatos. Estas duas bases são processadas por meio de uma Máquina ou Motor de Inferência. Esta automaticamente compara regras lógico-causais (e.g. regra se então) e fatos (e.g. estados de atributos) gerando novos fatos e, portanto, um ciclo de inferência. Não obstante a organização e mesmo eficientes algoritmos de inferência (e.g. Rete), a programação em PD normalmente é computacionalmente cara em termos de estruturas de dados a serem processadas [SCOTT, 2000; SIMÃO e STADZISZ, 2008; SIMÃO e STADZISZ, 2009a].

Independentemente, em uma análise mais profunda, PI e PD são similares no tocante à inferência, que normalmente se dá por entidades monolíticas baseadas

em pesquisas sobre entidades passivas ou voltadas a passividade (i.e. fracamente reativas) que conduzem a programas com passos de execução interdependentes. Estas características contribuem para a existência de desperdício de processamento e forte acoplamento entre expressões causais e estrutura de fatos/dados, o que dificulta a execução dos programas de maneira otimizada, bem como paralela ou distribuída [GABBRIELLI e MARTINI, 2010; SIMÃO e STADZISZ, 2008, SIMÃO e STADZISZ, 2009a]. Ainda que existam outras alternativas de programação, como orientações a eventos e mesmo orientação a dados, elas apenas atenuam ou fatoram o problema, não o resolvendo conforme discutido em [BANASZEWSKI, 2009; SIMÃO e STADZISZ, 2008; SIMÃO e STADZISZ, 2009a; XAVIER, 2014].

Revisões aprofundadas sobre paradigmas de desenvolvimento-programação podem ser encontrados nos livros [ROY e HARIDI, 2004; SCOTT, 2000; GABBRIELLI e MARTINI, 2010], bem como nos seguintes trabalhos de dissertação de mestrado [BANASZEWSKI, 2009; RONSZCKA, 2012; VALENCA, 2012; XAVIER, 2014].

2.2 PARADIGMA ORIENTADO A NOTIFICAÇÕES (PON)

Em linhas gerais, o Paradigma Orientado a Notificações (PON) encontra inspirações no PI, tais como flexibilidades algorítmicas e a abstração assaz similar a classes/objetos da POO e mesmo a reatividade da programação dirigida a eventos. O PON também aproveita conceitos próprios do PD, como facilidade de programação em alto nível e a representação do conhecimento em regras dos SBR. Assim, o PON provê a possibilidade de uso (de parte) de ambos os estilos de programação em seu modelo, ainda que os evolua e mesmo os revolucione (de certa maneira) no tocante ao processo de inferência ou cálculo lógico-causal [SIMÃO e STADZISZ, 2008a; BANASZEWSKI, 2009; RONSZCKA et al., 2011; SIMÃO et al., 2012a; XAVIER, 2014].

Neste âmbito, o PON apresenta resposta aos problemas desses paradigmas, como repetição de expressões lógicas e reavaliações desnecessárias delas (i.e. redundâncias estruturais e temporais) e, particularmente, o acoplamento forte de entidades no tocante as avaliações ou cálculo lógico-causal. Justamente, o PON apresenta outra maneira de realizar tais avaliações ou inferências por meio de

entidades computacionais de pequeno porte, ativas e desacopladas e que colaboram por meio de notificações pontuais e são criadas a partir do ‘conhecimento’ de regras [SIMÃO et al., 2012a].

Neste sentido, esta seção detalha o Paradigma Orientado a Notificações (PON), o qual foi introduzido brevemente na Subseção 1.1.1. Mais precisamente, a Subseção 2.2.1 apresenta o mecanismo de notificação do PON. Por sua vez, a Subseção 2.2.2 detalha a função assintótica do PON em relação ao seu processo de resolução do cálculo lógico-causal. A Subseção 2.2.3, aborda sobre o mecanismo de resolução de conflitos e garantias de determinismo em aplicações PON. Ainda, a Subseção 2.2.4, particularmente, reflete e contextualiza sobre as propriedades inerentes ao PON. A Subseção 2.2.5, define as características de utilização e compreensão do PON. A Subseção 2.2.6 apresenta os detalhes sobre a propriedade do PON denominada *Attributes* Impertinentes. A Subseção 2.2.7 apresenta a descrição sobre a propriedade do PON denominada Dependência entre *Rules*. Subsequentemente, a Subseção 2.2.8 descreve de maneira sucinta a sua materialização via *Software*. A Subseção 2.2.9 apresenta de maneira sucinta a composição de programas em PON. A Subseção 2.2.10 descreve de maneira sucinta a sua materialização via *Hardware*. Por fim, a Subseção 2.2.11 apresenta algumas reflexões sobre o PON no tocante ao seu estado da técnica atual.

2.2.1 Mecanismo de Notificação do PON

O Paradigma Orientado a Notificações (PON) introduz um novo conceito para a concepção, construção e execução de aplicações computacionais, salientando aqui as de *software*. As aplicações PON são compostas por pequenas entidades reativas e desacopladas, que colaboram por meio de notificações precisas e pontuais, ditando assim o fluxo de execução dessas [SIMÃO e STADZISZ, 2008; 2009a; SIMÃO et al., 2012a]. Essa nova maneira de tratar o desenvolvimento de *software* tende a proporcionar uma melhora no desempenho das aplicações e, potencialmente, tende a facilitar suas concepções, tanto para ambientes não distribuídos como para ambientes distribuídos [BELMONTE et al., 2012; SIMÃO, STADZISZ, 2008; 2009a; SIMÃO, et al., 2012a; SIMÃO et al., 2012b].

O fluxo das iterações das aplicações do PON é realizado de maneira transparente ao desenvolvedor, graças ao orquestramento da cadeia de notificações pontuais entre os elementos PON. Isto é diferente do fluxo de iterações encontrado em aplicações do PI, mesmo no subparadigma OO, onde o desenvolvedor deve de maneira explícita informar o laço de iterações através de comandos, como *while* e *for*. No PON a repetição ocorre de forma natural na perspectiva de execução da aplicação, conforme exemplificado na Figura 2, página 7, e esboçado no diagrama de classes conceitual da Figura 3 da presente seção.

O fluxo de execução ocorre em função da mudança de estado de um objeto *Attribute* de um respectivo *FBE*. Após a mudança de estado do objeto *Attribute*, ele notifica todas as *Premises* pertinentes, para que estas reavaliem seus estados lógicos. Se o valor lógico da *Premise* se altera, a *Premise* colabora com a avaliação lógica de uma ou de um conjunto de *Conditions* conectadas, o que ocorre por meio da notificação sobre a mudança relacionada ao seu estado lógico [BANASZEWSKI, 2009].

Consequentemente, cada *Condition* notificada avalia o seu valor lógico de acordo com as notificações da *Premise* e com o operador lógico (de conjunção ou disjunção) utilizado. Assim, no caso de uma conjunção, quando todas as *Premises* que integram uma *Condition* são satisfeitas (em estado verdadeiro), a *Condition* também é satisfeita, resultando na aprovação da sua respectiva *Rule* que pode ser executada [BANASZEWSKI, 2009].

Ainda, quando uma dada *Rule* aprovada está pronta para executar (i.e. com conflitos resolvidos conforme discute a próxima subseção), a sua *Action* é ativada. Uma *Action* é conectada a um ou vários *Instigations*. Os *Instigations* colaboram com as atividades das *Actions*, acionando a execução de algum serviço de um objeto *FBE* por meio dos seus objetos *Methods*. Geralmente, as chamadas para os *Methods* mudam os estados dos *Attributes* e o ciclo de notificação recomeça [BANASZEWSKI, 2009].

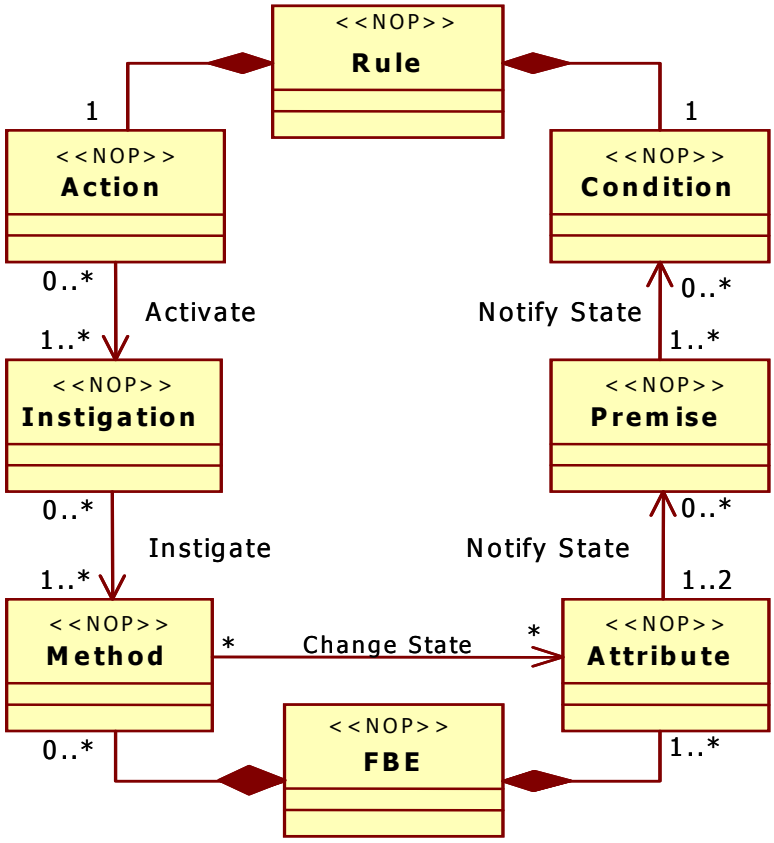


Figura 3 – Principais entidades do PON e seus relacionamentos [BANASZEWSKI, 2009]

Oportunamente, as conexões entre os objetos notificantes são estabelecidas em tempo de criação e por emergência. Por exemplo, na criação de um objeto *Premise* pelo menos um objeto *Attribute* é considerado. Uma vez que um *Attribute* é referenciado em uma *Premise*, o *Attribute* considera automaticamente esta *Premise* como sendo interessada em receber notificações sobre o seu estado. Assim, o *Attribute* identifica todas as *Premises* interessadas e notifica-as quando o seu estado muda. Ainda, mecanismo similar ocorre em relação as *Premises* e as *Conditions*, bem como em relação às *Conditions* e às *Rules* [BANASZEWSKI, 2009].

2.2.2 Cálculo Assintótico da Inferência do PON

A complexidade assintótica polinomial do PON, no pior cenário, é representada por $O(n^3)$ ou $O(FactBaseSize * nPremises * nRules)$, onde *FactBaseSize* corresponde à quantidade máxima de objetos *Attributes*, *nPremises* corresponde à quantidade máxima de objetos *Premises* notificados por estes

Attributes e *nRules* corresponde à quantidade máxima de objetos *Conditions* notificados por estas *Premises* [SIMÃO, 2005; BANAZEWSKI, 2009].

A função assintótica apresentada para o PON, no pior cenário, demonstra uma solução bastante similar ao mecanismo de inferência do algoritmo HAL [BANAZEWSKI, 2009]. Ainda, a função temporal polinomial do HAL1 ($O(n^3)$) se apresenta mais eficiente do que os algoritmos de inferência RETE, TREAT e LEAPS [BANAZEWSKI, 2009].

Esta função assintótica do PON representa a quantidade de notificações entre as instâncias colaboradoras que também corresponde à quantidade de avaliações lógicas. A constatação desta função assintótica pode ser realizada pela análise da Figura 4, a qual demonstra as relações por notificações entre instâncias colaboradoras. Nesta, os *Attributes*, *Premises*, *Conditions* e *Rules* correspondem, respectivamente, aos símbolos com abreviações *Att*, *Pr*, *Cd* e *Rl* [BANAZEWSKI, 2009].

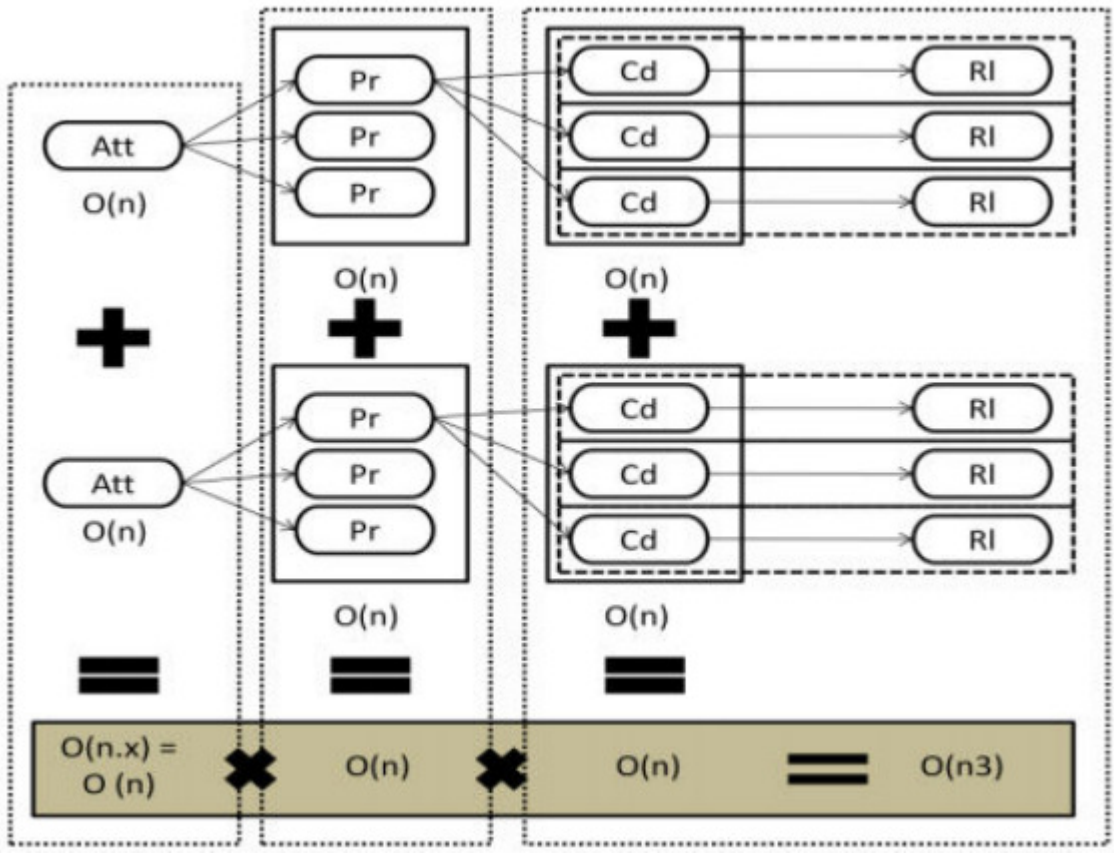


Figura 4 – Cálculo assintótico do mecanismo de notificações [BANAZEWSKI, 2009]

Outra forma mais adequada de analisar a complexidade polinomial do PON é considerar, em vez do pior caso, o caso médio. A análise da complexidade do caso médio é iniciada analisando-se o começo do processo de notificação do PON através da entidade *Attribute*. Assim, as principais variáveis envolvidas em uma notificação de um *Attribute* são demonstradas pela equação da Figura 5.

$$FB_{at}() = NumPremises + NumRules$$

Figura 5 – Complexidade da Notificação *Attribute* [SIMÃO, 2005]

A variável "*NumPremises*" é a soma de entidades *Premises* ao respectivo *Attribute* e a variável "*NumRules*" é a soma das entidades *Rules* a cada entidade *Premise* contada em "*NumPremises*". Portanto, se for considerado simplesmente cada ciclo de inferência como a instigação de um *Attribute*, uma média possível seria: TMedium(x) = (FBAT.1()+ … + FBAT.w()) / w, onde (w) é o número de todos os *Attributes* existentes. Assim, o resultado desta média seria uma ordem de (n), o que implicaria uma complexidade linear O(n) [SIMÃO, 2005].

2.2.3 Resolução de Conflitos no PON

Um conflito ocorre quando duas ou mais *Rules* referenciam um mesmo *FBE* e demandam exclusividade de acesso a este *FBE*. Deste modo, as *Rules* concorrem para adquirir acesso exclusivo a este *FBE*, sendo que somente uma destas *Rules* em conflito pode executar por vez, a qual obteve o acesso exclusivo. Neste âmbito, para resolver as questões de resolução de conflitos entre as *Rules*, basicamente, o fluxo de sua execução é determinado segundo uma estratégia pré-estabelecida. Estas estratégias podem variar para alcançar o fluxo de execução pretendido pelo desenvolvedor, tanto em ambientes monoprocessados quanto em ambientes multiprocessados [BANASZEWSKI, 2009].

Em um ambiente monoprocessado, a resolução de conflitos ocorre para estabelecer a ordem de execução das *Rules* (enquanto elas estiverem aprovadas), de forma que apenas uma *Rule* pode executar por vez. Em um ambiente multiprocessado, a resolução de conflitos ocorre para evitar o acesso concorrente a

um recurso referenciado por várias *Rules* a fim de manter a consistência da aplicação PON [BANASZEWSKI, 2009].

Em se tratando de ambientes monoprocessados, basicamente é empregado um escalonador de *Rules* formado por uma estrutura de dados do tipo linear (e.g. pilha, fila ou lista) [BANASZEWSKI, 2009]. Estas estruturas guardam referências para as *Rules* aprovadas, conforme ilustra a Figura 6. Assim, tais estruturas recebem as *Rules* na ordem em que elas são aprovadas, podendo reorganizá-las de acordo com os preceitos de cada estratégia adotada [BANASZEWSKI, 2009].

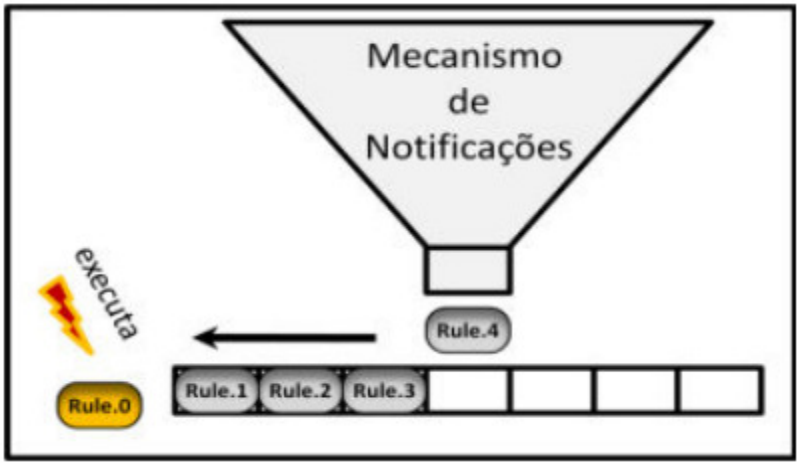


Figura 6 – Modelo Centralizado de Resolução de Conflitos [BANASZEWSKI, 2009]

Desta forma, conforme a estratégia de resolução de conflitos pré-determinada pelo desenvolvedor, as *Rules* em questão serão efetivamente executadas. Neste âmbito, os modelos de resolução de conflitos empregados para o PON em ambientes monoprocessados são:

1. *BREADTH*: se baseia no escalonamento *First In*, *First Out* (*FIFO*), ou seja, refere-se à execução de entidades *Rule*, seguindo uma estrutura de dados do tipo fila;
2. *DEPTH*: se baseia no escalonamento *Last In*, *First Out* (*LIFO*), ou seja, refere-se à execução de entidades *Rule*, seguindo uma estrutura de dados do tipo pilha;
3. *PRIORITY*: organiza as entidades *Rule* de acordo com as prioridades definidas nas mesmas;

Quando nenhuma estratégia for definida pelo desenvolvedor, utiliza-se a estratégia *NO_ONE*, a qual faz com que as entidades *Rules* não sejam enviadas ao

escalonador/estrutura de dados e sejam aprovadas e executadas imediatamente. Basicamente, em aplicações monoprocessadas, a definição do tipo de escalonamento pelo desenvolvedor para a execução das *Rules* da aplicação define o tipo da resolução de conflito adotada (e mesmo a maneira de garantia de determinismo adotada) [BANASZEWSKI, 2009].

Em tempo, uma nova estratégia para resolução de conflitos no PON foi criada [XAVIER, 2014]. Essa estratégia é denominada KEEPER e tem a finalidade de manter uma *Rule* aprovada na fila de execução, até que a mesma seja desaprovada.

Outrossim, a definição de resolução de conflitos de *Rules* deve ser adotada não só em aplicações monoprocessadas mas também e talvez principalmente em aplicações distribuídas e/ou concorrentes. Entretanto, as soluções apresentadas para evitar os conflitos descritos aqui, são particularmente aplicáveis a soluções PON monoprocessadas, ainda que até possam ser úteis em soluções multiprocessadas e distribuídas com a ressalva de serem centralizadoras em alguma sorte [BANASZEWSKI, 2009].

Na verdade, a definição de resolução de conflitos de *Rules* deve ser adotada em aplicações concorrentes e/ou distribuídas com soluções que lhe sejam mais apropriadas. Neste sentido, ainda que este presente trabalho não seja relativo à aplicação de PON em sistemas concorrentes ou distribuídos, nestes trabalhos que seguem aqui citados [BANASZEWSKI, 2009; SIMÃO, 2005; SIMÃO e STADZISZ, 2009a; SIMÃO, et al., 2012a] encontram-se soluções úteis ao PON para resolução de conflitos em aplicações PON distribuídas, bem como soluções correlatas para a garantia de determinismo.

Isto dito, com os conflitos solucionados (e determinismo garantido), uma dada *Rule* aprovada está pronta para executar o conteúdo da sua *Action*. Uma *Action* é conectada a uma ou várias *Instigations*. As *Instigations* colaboram com as atividades das *Actions*, acionando a execução de algum serviço de um *FBE* por meio dos seus *Methods*. Geralmente, as chamadas para os *Methods* mudam os estados dos *Attributes* e o ciclo de notificação é realimentado [BANASZEWSKI, 2009].

2.2.4 Propriedades Inerentes ao PON

Nota-se que a essência da computação no PON está organizada e distribuída entre entidades autônomas e reativas que colaboram por meio de notificações pontuais. Este arranjo forma o mecanismo de notificações, o qual determina o fluxo de execução das aplicações. Por meio deste mecanismo, as responsabilidades de um programa são divididas entre as instâncias do modelo, o que permitiria execução otimizada e 'desacoplada' (i.e. minimamente acoplada) supostamente útil para o aproveitamento correto de monoprocessamento, bem como para o processamento distribuído [BANASZEWSKI, 2009].

Neste sentido, toda esta colaboração por meio de notificações pontuais e precisas representaria a solução para as principais deficiências dos atuais paradigmas de programação. Ao evitar buscas sobre entidades passivas e pelo compartilhar de colaborações por notificação, o PON implicitamente evita as redundâncias temporais e estruturais que tanto afetam o desempenho das aplicações no PI e mesmo no PD [BANASZEWSKI, 2009; SIMÃO, et al., 2012a]. Neste sentido, o PON poderia ser considerado como uma abordagem "verde" que evitaria uso desnecessário de recurso, como processamento e energia.

Ademais, observa-se que os objetos ou entidades participantes da cadeia de notificação do PON se apresentam desacoplados, devido a comunicação realizada por meio de notificações pontuais. Neste âmbito, pode-se dizer que aplicações no PON possuem características apropriadas para a execução em ambientes multiprocessados, uma vez que 'somente' se faz necessário os objetos notificantes conhecerem os endereços dos objetos a serem notificados para a inferência por notificação ocorrer [BANASZEWSKI, 2009].

2.2.5 PON – Utilização x Compreensão

A natureza do PON leva a uma nova maneira de compor *software*, na qual os fluxos de execução são distribuídos e colaborativos nas entidades. Assim sendo, o PON permite uma nova maneira de estruturar, executar e pensar os artefatos de *software*. Em tempo, muito embora o PON permita compor software em alto nível na forma de regras, sem o conhecimento desta sua essência de inferência colaborativa

por notificações, ainda assim este conhecimento é importante [SIMÃO, et al., 2012a].

Por exemplo, é importante saber dos impactos de desempenho, das estratégias de resolução de conflitos e das estratégias de distribuição. Neste último caso, um exemplo mais preciso seria a forma de agrupamento de elementos de maior fluxo de notificações juntos, evitando assim comunicações desnecessárias nos canais de comunicação (e.g. redes) [BELMONTE et al., 2012; SIMÃO, et al., 2012a].

Ainda, a compreensão dos princípios do PON é importante para aplicações complexas, nas quais o fluxo de notificações pode ser intenso e pode precisar de mais formalismo e rastreabilidade, como em algumas aplicações em tempo real e/ou de controle discreto. Na verdade, esse tipo de aplicação pode exigir apoio de ferramentas formais para elaboração do projeto [SIMÃO, et al., 2012a].

Um exemplo particular de formalismo é a rede de Petri. Na verdade, redes de Petri são assaz compatíveis com os sistemas baseados em regras, em geral, em termos de expressão de relações causais [SHEN, JUANG, 2008]. Além disso, redes de Petri são particularmente mais compatíveis com os princípios do PON em termos da sua essência dado que aquelas trabalham com sensibilizações de transições causais que encontrariam uma melhor implementação computacional por meio de notificações [SIMÃO, STADZISZ, 2009]. Neste contexto, seria necessário conhecer o PON e os princípios de rede de Petri, compreendendo que ambos são naturalmente compatíveis [SIMÃO e STADZISZ, 2009a].

2.2.6 *Attributes* Impertinentes

De maneira geral, a reatividade presente nos *Attributes* proporcionaria uma execução livre de avaliações redundantes e desnecessárias, comuns aos paradigmas de programação usuais. Entretanto, existem casos em que a variação de um *Attribute* encadearia sequências de notificações indesejáveis [RONSZCKA et al., 2011].

Isso ocorreria em situações nas quais um dado *Attribute* apresentaria frequentes mudanças de estado, disparando o fluxo de notificações a cada variação, sem afetar efetivamente na aprovação da *Rule* a qual pertence. Assim, tais

notificações desnecessárias impactariam negativamente no desempenho de execução de uma aplicação PON.

Por exemplo, pequenas mudanças em um *Attribute* referente à temperatura interna (*atTemperature*) de uma casa que seria pertinente a uma dada *Rule*. Esta *Rule* seria responsável por ativar o ar condicionado se a temperatura interna atingir um dado valor e se alguém estiver na casa (*atStatus true*). Entretanto, a casa passa a maior parte do tempo vazia (*atStatus false*). Assim, a maior parte das notificações do *Attribute* em questão (*atTemperature*) seriam 'impertinentes'.

De modo a elucidar o problema em questão, considera-se o exemplo ilustrado na Figura 7. Neste exemplo são apresentados dois *Attributes* distintos, um do tipo *Boolean* (*atStatus*) e outro do tipo Double (*atTemperature*) em uma *Condition*/*Rule* composta por duas *Premises*. Uma *Premise* avaliaria se o estado de *atStatus* é verdadeiro, enquanto a outra *Premise* avaliaria se o estado de *atTemperature* é maior que um dado valor [RONSZCKA et al., 2011].

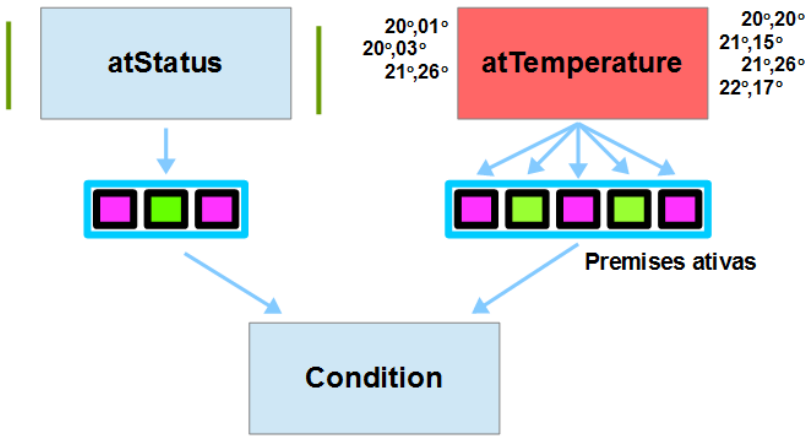


Figura 7 – Impacto nas alterações de estado de *Attributes* ativos [Ronszcka, 2012]

O *Attribute atStatus* apresentaria poucas mudanças em seu estado, permanecendo a maior parte do tempo com o estado *false*, disparando o fluxo de notificações esporadicamente. O *Attribute atTemperature*, por sua vez, apresentaria alterações constantes em seu estado, que no cenário em questão raramente impactariam na aprovação da sua *Condition*.

Neste sentido, um *Attribute* como *atTemperature* poderia ser categorizado como 'impertinente'. Sendo assim, de modo a evitar tal cenário de notificações inúteis, cada *Attribute* impertinente em dado contexto deveria ter suas funções reativas desabilitadas temporariamente para com as *Premises*-*Conditions*-*Rules*

afetadas pela impertinência. Neste âmbito, a Figura 8 considera o mesmo cenário anterior, agora com a inativação temporária das *Premises* contendo *Attributes*' impertinentes'.

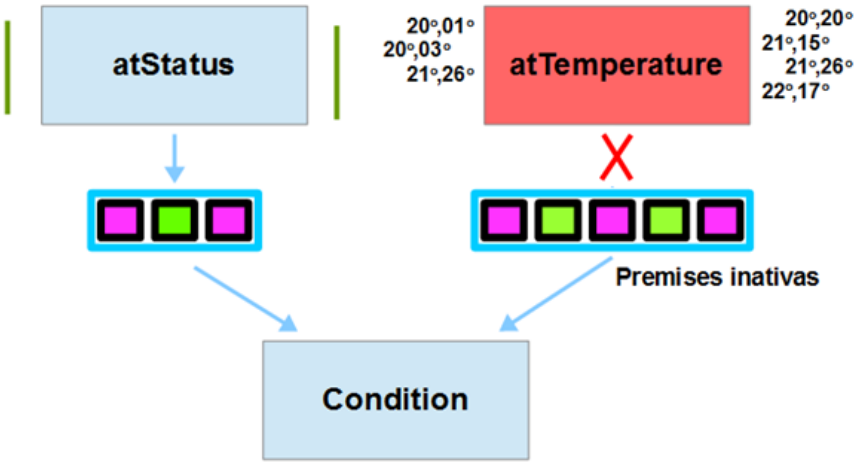


Figura 8 – Impacto nas alterações de estado de *Attributes* 'impertinentes' [Ronszcka, 2012]

Conforme apresenta o cenário ilustrado na Figura 8, ao inativar temporariamente algumas das *Premises* compostas pelo *Attribute* impertinente *atTemperature*, as variações de estado desse não impactariam no disparo do fluxo de notificações para tais entidades. Neste âmbito, quando o conjunto dos *Attributes* tivesse aprovado 'suas' *Premises* em uma dada *Condition-Rule*, essa deveria solicitar a reativação das notificações para a *Premise* composta pelo *Attribute* impertinente. Assim, uma vez que o *Attribute atStatus* apresentasse estado verdadeiro, a *Condition-Rule* ilustrada solicitaria a reativação da *Premise* correspondente ao *Attribute atTemperature*, conforme ilustra a Figura 9.

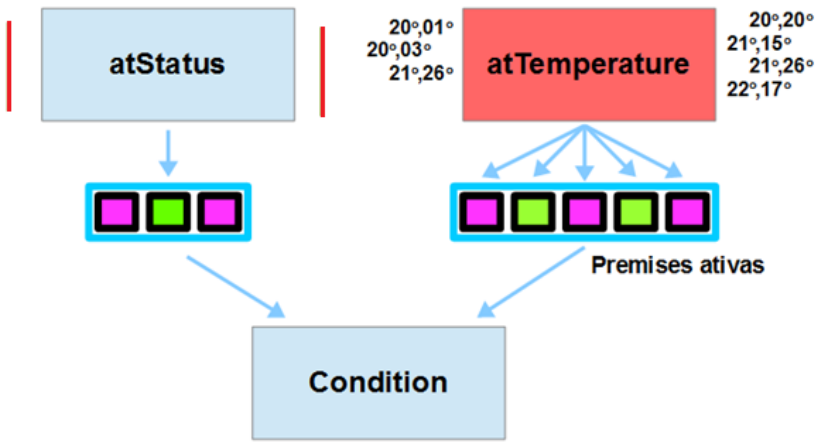


Figura 9 – Exemplo de reativação de uma entidade desativada [Ronszcka, 2012]

Neste sentido, conforme o cenário ilustrado na Figura 9, o *Attribute atStatus*, ao apresentar o estado verdadeiro, colocaria tal *Condition* em "ponto de aprovação", o que reativaria as funções reativas da *Premise* temporariamente desconsiderada. Assim, a entidade *Premise* em questão permitiria ser notificada (e mesmo demandaria notificações) novamente pelo *Attribute atTemperature*, atuando normalmente como uma entidade (re)ativa. Por fim, após a devida aprovação e execução da *Rule* em questão, o *Attribute* impertinente voltaria a ignorar tal *Premise* até que seja requisitado novamente por outra *Conditition*.

2.2.7 Dependência entre *Rules*

De maneira geral, existem casos que um conjunto considerável de *Rules* dependeria de *Premises* semelhantes para suas aprovações/execuções. A criação de entidades *Premises* únicas (com o mesmo teste lógico-causal) e seus respectivos compartilhamentos seriam a alternativa mais apropriada nesse cenário, evitando desta forma a presença de entidades redundantes. Entretanto, existem outros casos que tais *Rules* dependeriam de duas ou mais *Premises* compartilhadas, as quais notificariam todas as entidades interessadas em mudanças ocorridas em seu estado [RONSZCKA et al., 2011][1].

Neste sentido, notificações desnecessárias poderiam ser evitadas caso *Premises* comuns a todas as *Rules* notificassem apenas uma *Rule*, ao invés de todas elas. Assim, no momento que todas as *Premises* dessa *Rule* única apresentassem estado verdadeiro, a ponto de aprovar a execução dessa, ela notificaria as demais *Rules* interessadas. Nesse âmbito, de modo a evitar tal sobrecarga de processamento e mesmo dificuldades de implementação (i.e. compartilhamento de entidades semelhantes e passividade a erros), a funcionalidade de *Rules* dependentes permite criar uma dependência entre *Rules* no PON, onde uma ou mais *Rules* dependeriam da execução de uma determinada *Rule* para, só então, executarem.

[1] Os conceitos de *Attributes* Impertinentes e dependência entre *Rules* tem como base a Dissertação de Mestrado de Ronszcka [RONSZCKA, 2012].

Neste caso, tais *Premises* fariam parte de uma única *Rule*, a qual faria o papel de *Rule* mestre. As demais *Rules* fariam um vínculo com essa *Rule*, de tal forma que dependeriam de sua aprovação e respectiva notificação para só então executarem. Essa dependência traria benefícios em questões de desempenho e facilidades na composição de aplicações.

No âmbito de eficiência na execução, tais *Rules* atuariam na redução de notificações geradas pelas *Premises* em questão, que direcionariam suas notificações apenas a *Rule* mestre. Em relação a facilidade de composição de aplicações, tal abordagem simplificaria a essência de todas as demais *Rules*, tornando o código mais legível e consequentemente mais manutenível.

2.2.8 Materializações do PON em Software

Os conceitos do PON propriamente dito foram primeiramente materializados sobre o POO, através de um arquétipo ou *Framework* desenvolvido com a linguagem de programação C++. A versão dita prototipal (versão 0) concebida por Simão em 2007 evoluindo tecnologia de controle precedente (vide [SIMÃO, 2001, 2005]) e a versão dita original (versão 1) do *Framework* PON desenvolvida por Banaszewski [2009] e da versão otimizada (Versão 2) Valença [2012] Ronszcka [2012] foram implementadas especificamente para ambientes monoprocessados, contemplando considerável parte dos conceitos do paradigma PON apresentados nas Subseções anteriores.

Estruturalmente, o *Framework* PON materializa as entidades colaboradoras do paradigma em forma de classes/instâncias relacionadas através de estruturas de dados com referências às entidades interessadas em seus estados. Neste âmbito, uma estrutura de pacotes foi projetada, conforme mostra a Figura 10, para modelar (e explicar) esta materialização do PON.

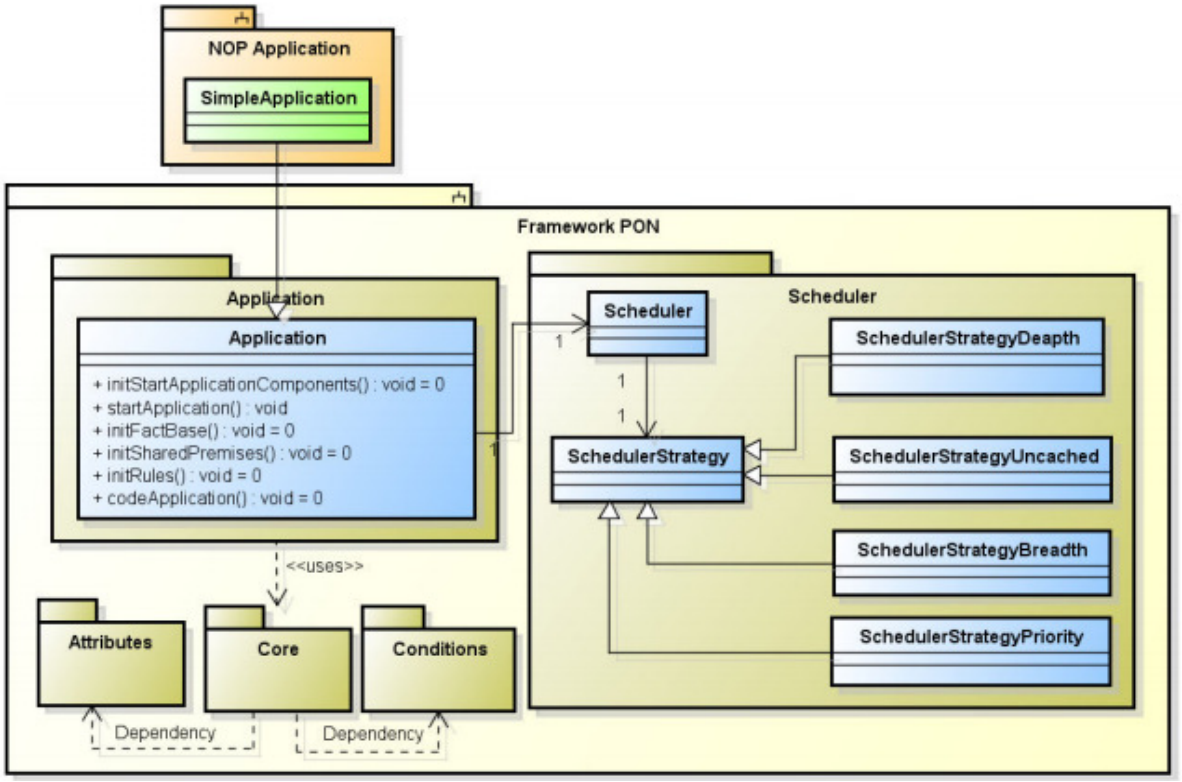


Figura 10 – Estrutura do *Framework* PON [LINHARES et al., 2011]

Conforme ilustrado na Figura 10, o *Framework* PON é subdividido em três pacotes principais. Dentre esses, o pacote *Application* é formado exclusivamente pela classe *Application*, que representa a ponte de ligação entre uma aplicação PON e as demais classes do *Framework*. O pacote *Scheduler*, por sua vez, representa as implementações das classes de resolução de conflitos, conforme descrito na Subseção 2.2.3. O pacote *Core*, particularmente, é formado pelas classes colaboradoras cujas instâncias que realizam o processo de notificação em aplicações feitas a partir do *Framework* PON. A Figura 11 ilustra o diagrama de classes que melhor ilustra os componentes do pacote *Core*.

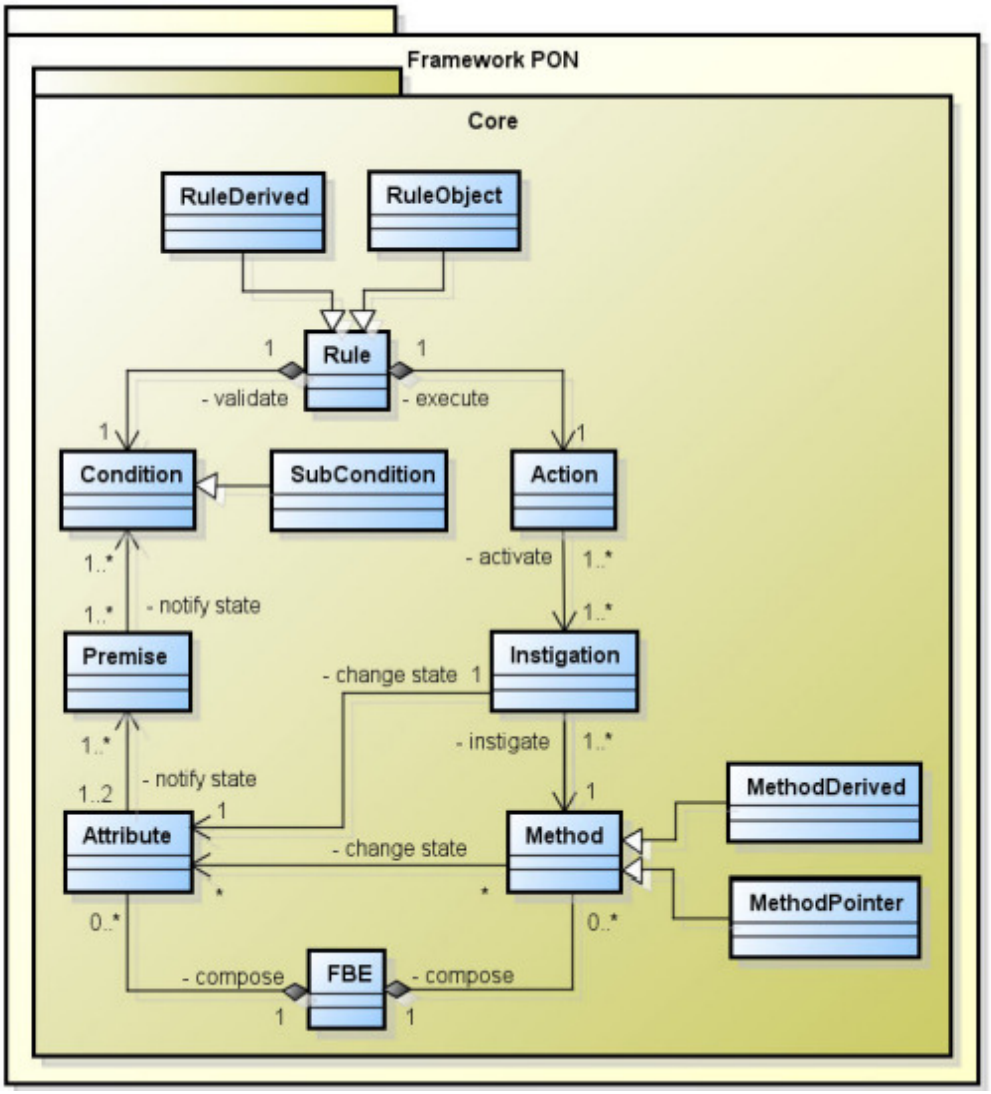


Figura 11 – Estrutura do pacote *Core* [LINHARES et al., 2011]

As classes *Rule* e *FBE* se apresentam nas extremidades opostas e se relacionam por meio de suas classes ditas colaboradores – *Attribute*, *Premise*, *Condition*, *Action*, *Instigation* e *Method* – sendo que a colaboração entre as instâncias destas classes determina o fluxo de execução de aplicação do PON. Ademais, as classes *Method* e *Rule* que definem as entidades puras do PON são estendidas de modo a proporcionar funcionalidades adicionais.

Ainda, o pacote *Core* é composto também pelos subpacotes *Attributes* e *Conditions*. Conforme apresenta a Figura 12, o subpacote *Attribute* é formado pelas classes responsáveis por encapsular os tipos primitivos do POO. Estas classes (*Boolean*, *Char*, *Double*, *Integer* e *String*) introduzem reatividade aos tipos primitivos, ao permitir que estes façam parte de estruturas causais do PON.

Ademais, conforme a Figura 12, o subpacote *Conditions* é formado pelas classes que fazem parte da composição de uma respectiva *Condition*. Particularmente, a classe *LogicalOperator* e suas derivadas (*Conjunction*, *Disjunction* e *Single*), definem a operação lógica utilizada pela *Condition* de maneira a aprovar uma determinada *Rule*.

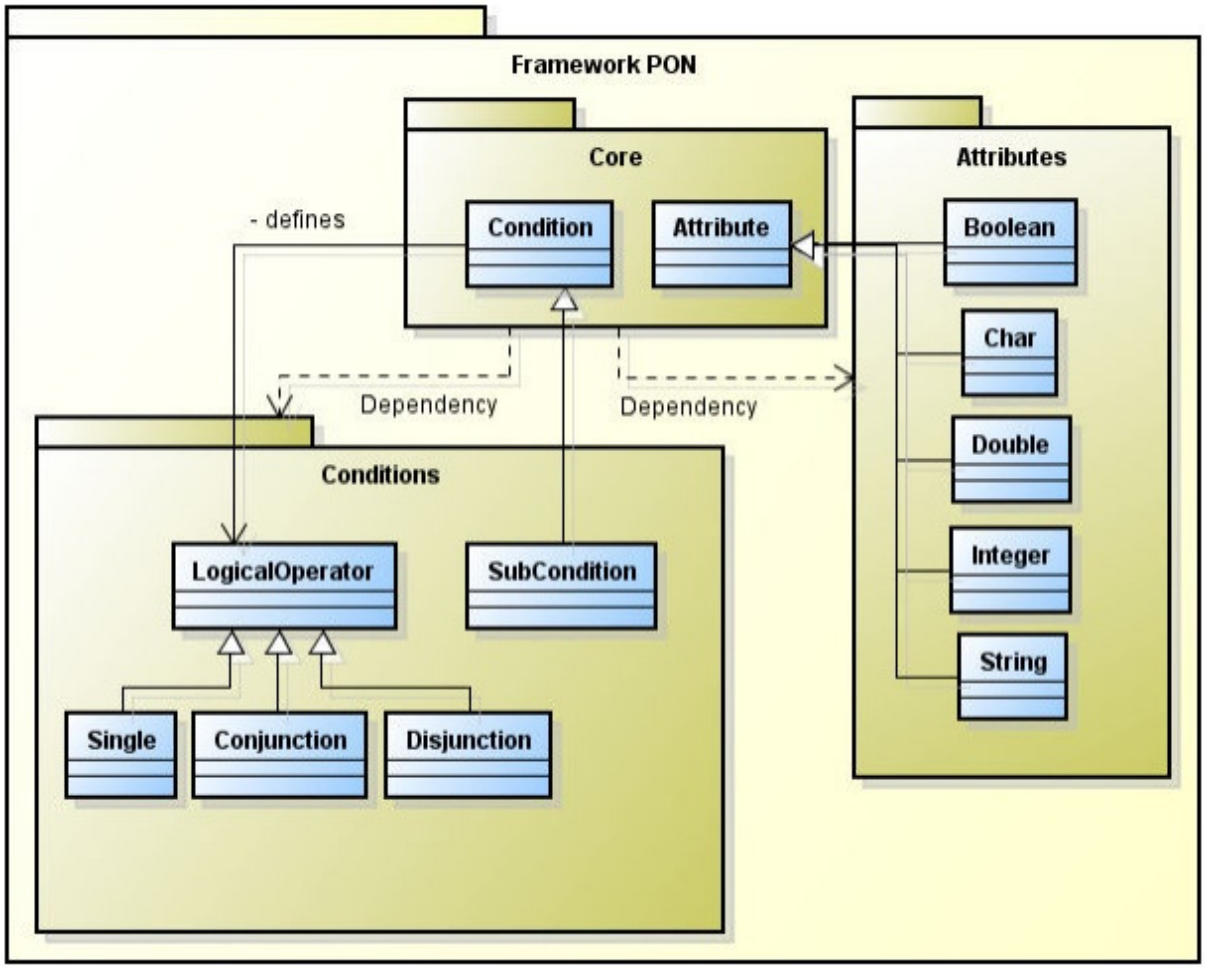


Figura 12 – Estrutura dos subpacotes *Attributes* e *Conditions* [LINHARES et al., 2011]

Essa materialização do PON convenientemente possibilitou a validação dos conceitos relacionados a este paradigma, principalmente em questões da resolução do cálculo lógico-causal e assim a eliminação das redundâncias temporais e estruturais. Em testes comparativos apresentados em [BANASZEWSKI, 2009; SIMÃO et al., 2012a], o PON já se mostrou efetivo nesta materialização, entretanto conforme discutido em [BATISTA et al., 2011; LINHARES et al., 2011; RONSZCKA et al., 2011; VALENÇA et al., 2011; SIMÃO et al., 2012a], tal materialização ainda

demonstra deficiências em alguns domínios de aplicações (i.e. com relação ao desempenho e facilidade de programação).

Os problemas descritos nos trabalhos são referentes principalmente sobre a camada extra de execução das aplicações PON, conforme esboçado na Figura 13.

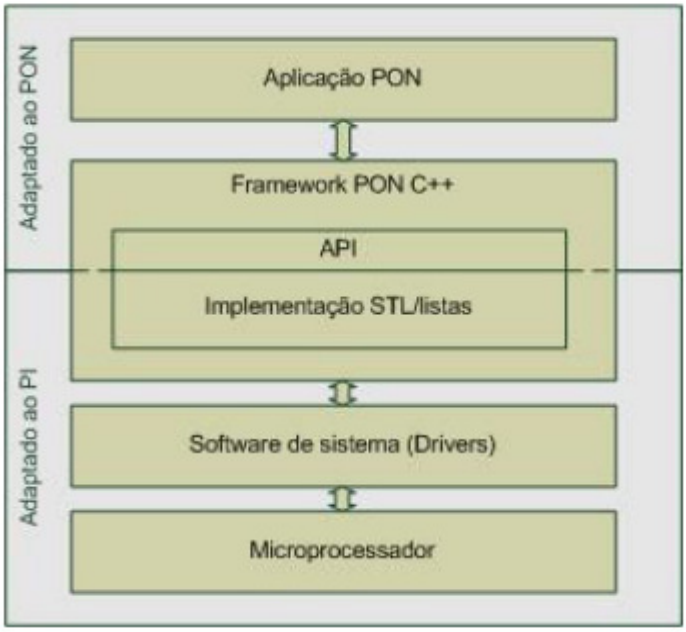


Figura 13 – Camadas da arquitetura do *Framework* PON [LINHARES et al., 2011]

É possível observar na Figura 13 que as aplicações concebidas com o auxílio do *Framework* PON são dependentes de camadas extras, o que contribui para a degradação de seus desempenhos. Entretanto, a implementação do *Framework*, desenvolvido sob os princípios da linguagem de programação C++, apresenta-se passível de otimização, algo que foi feito por [VALENÇA, 2012; VALENÇA et al., 2011]. Ainda, conforme demonstrado por [FERREIRA, 2013], o PON otimizado por [VALENÇA, 2012; VALENÇA et al., 2011] apresentou melhoras em relação aos problemas de desempenho do anterior. No entanto, o seu desempenho não foi o que fora vislumbrado em seu estado da arte a luz do seu cálculo assintótico.

2.2.9 Composição de programas em PON

Esta Subseção tem por objetivo explicar de maneira sucinta a composição de programas utilizando o *Framework* PON. Deste modo, a composição de

programas em tal *Framework* apresenta particularidades que o distinguem das demais implementações existentes. As particularidades dessa versão do *Framework* são descritas na Subseção anterior. Isto dito, para desenvolver aplicações com o *Framework* PON em questão, inicialmente é necessário que o desenvolvedor estenda a classe *NOPApplication*. Tal classe proporciona uma ponte entre a aplicação PON e o cerne do *Framework* PON. A Figura 14 ilustra a classe que representa o procedimento inicial de criação de uma aplicação PON.

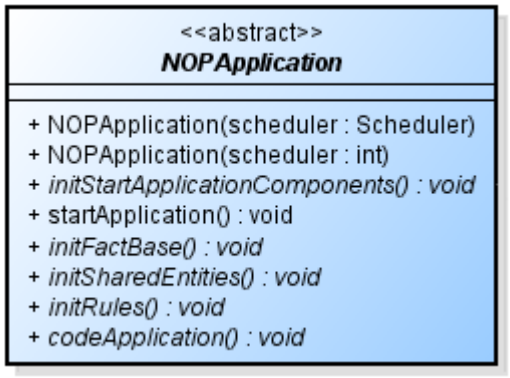


Figura 14 - Diagrama de classes do procedimento inicial de uma aplicação PON [RONSZCKA, 2012]

Conforme apresenta a Figura 14, a classe *NOPApplication* apresenta métodos virtuais puros (i.e. abstratos) que devem ser implementados pela classe responsável pela inicialização de uma aplicação PON. Em tais métodos, o desenvolvedor deve se concentrar na inicialização e criação de *FBEs* e entidades compartilhadas, bem como na concepção de *Rules*.

A composição de programas PON se dá por meio da criação e encadeamento de entidades PON, de modo que tais entidades formem um fluxo coeso e bem distribuído de notificações pontuais. Essencialmente, esse fluxo dita a maneira com que o programa se comporta e responde a eventos ocorridos no mesmo.

Ademais, a criação de entidades PON é dada pelo uso de uma fábrica de entidades, a qual possui como responsabilidade principal instanciar tais entidades adaptadas a uma estrutura de dados específica.

O Algoritmo 2 demonstra a inicialização dos componentes iniciais de uma aplicação PON, em especial, a classe *SingletonFactory*, responsável pela instanciação de entidades PON.

Algoritmo 2: Inicialização dos componentes iniciais de uma aplicação PON

```
1  void Main::initStartApplicationComponents() {
2        SingletonFactory::changeStructure(SingletonFactory::NOPVECTOR);
3        SingletonLog::changeStream(SingletonLog::CONSOLE);
4        SingletonScheduler::changeScheduler(SchedulerStrategy::NO_ONE);
5        this->startApplication();
6  }
```

Conforme ilustra o Algoritmo 2, o método *initStartApplicationComponents* pode ser utilizado para a inicialização da fábrica de entidades concretas (linha 2). As possíveis fábricas a serem utilizadas na criação de entidades PON são *NOP_LIST*, *NOP_VECTOR*, *NOP_HASH* e *STL_LIST*. Ademais, esse método pode ser utilizado na inicialização do gerador de logs (linha 3). É importante observar, que os mesmos métodos *changeStructure* e *changeStream* podem ser utilizados em outras partes do código, caso o desenvolvedor ache pertinente e necessário. Por fim, conforme linha 4, o desenvolvedor deveria chamar o método *startApplication* que é responsável por chamar ordenadamente os métodos de criação de uma aplicação PON. Tal método é implementado na classe *NOPApplication*, conforme apresenta o Algoritmo 3.

Algoritmo 3: Implementação do método *startApplication*

```
1  void NOPApplication::startApplication() {
2     initFactBase();
3     initSharedEntities();
4     initRules();
5     codeApplication();
6  }
```

Nota-se que a essência da classe *NOPApplication* é bastante simples, podendo ser inclusive ignorada na criação de uma aplicação PON. Entretanto, a extensão dessa classe é aconselhável, uma vez que segue bons princípios, ao dividir o processo de inicialização, configuração e construção de uma aplicação PON em métodos coesos e intuitivos.

A composição de entidades *FBEs* no *Framework* PON se dá por meio da criação e instanciação de classes. Inicialmente, o desenvolvedor deve criar as classes que representam os modelos para criação de entidades *FBE* pertinentes à sua aplicação. Tais classes podem conter *Attributes*, *Methods*, referências para outras instâncias de *FBEs* e inclusive *Rules*.

De acordo com a sequência de criação de aplicações, o próximo método herdado da classe *NOPApplication* a ser implementado é o método *initFactBase*. Sendo assim, o Algoritmo 4 apresenta a configuração do método *initFactBase*.

Algoritmo 4: Implementação do método *initFactBase*

```
1  void Main::initFactBase() {
2        apple = new Apple();
3        archer = new Archer();
4        controller = new Controller();
5  }
```

Conforme ilustra o código do Algoritmo 4, entre as linhas 2 e 4 são criadas três instâncias de *FBEs* do tipo *Apple*, *Archer* e *Controller* respectivamente. Salienta-se que essas *FBEs* fazem parte da aplicação denominada Mira ao Alvo que será discutida na Seção 4.1.1.

A composição de *Rules,* por sua vez, é a etapa que define o fluxo de execução de uma aplicação PON. Apesar de todas as entidades PON contribuírem ativamente para gerar o fluxo de execução de tal aplicação, são as *Rules* que definem os relacionamentos entre essas. Neste sentido, o Algoritmo 5 apresenta o trecho de código que define a criação da *Rule* para a aplicação Mira ao Alvo.

Algoritmo 5: Implementação do método *startApplication*

```
1    void Main::initRules() {
2          Scheduler * scheduler =
3    SingletonScheduler::getInstance();
4          RULE (rlRlTurnOn1,scheduler,Condition::CONJUNCTION);
5            rlRlTurnOn1->addPremise(prPrIsCrossed);
6            rlRlTurnOn1->addPremise(prPrFire);
7            rlRlTurnOn1->addPremise(prPrHasFired);
8            rlRlTurnOn1->addPremise(prPrIsCrossed);
9            rlRlTurnOn1->addInstigation(ininFire);
10           rlRlTurnOn1->end();
11   }
```

É possível visualizar no Algoritmo 5 (linha 1) que a configuração das *Rules* se dá pela criação do método *initRules*. Neste método é configurado a *Rule* da aplicação Mira ao Alvo. Esta *Rule* foi discutida na Subseção 1.1.1 e tem por objetivo validar se um *Archer* tem a permissão de flechar uma *Apple.* Ainda, neste algoritmo (linha 2), é possível perceber a configuração do *Scheduler* que neste caso é configurado com a estratégia de escalonamento a ser utilizada pela *Rule* em questão. A linha 4 apresenta a configuração de um *Rule*. Nesta linha de código é

possível perceber que a *Rule* recebe três argumentos sendo: a variável de referência, o *Scheduler,* e o tipo de operação lógica (i.e. *Conjunction).* Na linha 5 é possível visualizar a configuração de uma *Premise* a respectiva *Rule*, e finalmente, na linha 9 é possível visualizar a adição de uma *Instigation* a *Rule*.

2.2.10 Materializações do PON em Hardware

Além das materializações em *software* do PON (três versões do *Framework*), houve também avanços na materialização do paradigma no tocante ao desenvolvimento de *hardware*. Umas das propostas do PON para tal constitui-se em uma implementação em *hardware* (i.e. circuitaria eletrônica) das entidades computacionais que colaboram para o funcionamento da cadeia de notificações no PON [SIMÃO et al., 2012c].

Esta implementação está focada particularmente no subconjunto das entidades computacionais da cadeia de notificações do PON que apresenta funcionamento genérico (i.e. independente da aplicação), os quais podem ser chamados conjuntamente de meta-modelo PON. Tal implementação do PON em *Hardware* Digital (PONHD) é realizável inclusive em dispositivos de lógica reconfigurável, que têm servido para sua prototipação [SIMÃO et al., 2012c].

Um exemplo particular de dispositivo de lógica reconfigurável para a implementação do PONHD seria os *Field Programmable Gate Arrays* (FPGAs). Entretanto, a implementação do PONHD não se limita ao uso desta família de dispositivos lógicos reconfiguráveis apenas, podendo ser feita em outras como a família de *Complex Programmable Logic Device* (CPLD) [SIMÃO et al., 2012c].

Outro exemplo, conforme apresentado por [JASINSKI, 2012], constitui-se de uma solução de desenvolvimento de um interpretador de XML e um associado *Framework* em VHSIC (*Very High Speed Integrated Circuit*) *Hardware Description Language* (VHDL). No mais, tal solução foi capaz de gerar código VHDL para duas aplicações distintas, a partir de código XML, permitindo obter respectivo *hardware* em FPGA [JASINSKI, 2012]. A primeira aplicação, denominada *Wire*, consistiu da reprodução em sua saída de estímulos aplicados a sua entrada. Por sua vez, a segunda aplicação, estimulou a extensão e generalização da solução para suportar

a sua execução, sendo esta aplicação chamada de *eletronic_gate*, a qual tinha por objetivo o controle de um portão eletrônico [JASINSKI, 2012].

Ainda, ao seu turno, Peters realizou a construção de um *hardware* periférico, o CO-PON, capaz de acelerar a execução de *software* em PON criados seguindo uma adaptação do *Framework* de Banaszewski tornando, dessa forma, viável inclusive o uso do PON em sistemas embarcados construídos sob processadores genéricos [PETERS, 2012].

Essas materializações melhoram o desempenho em termos de tempo de execução, tanto em termos de baixo processamento quanto distribuição. No entanto, devem ser programadas em *hardware* (via Linguagem de Descrição de *Hardware* ou esquemático ou afins). Ainda outro problema decorrente é que para cada nova aplicação que surge, uma nova reconfiguração do *hardware* deve ser realizada dado que cada aplicação se constitui em um circuito específico ainda que baseado em elementos genéricos do PON [LINHARES, 2015].

Nesse sentido foi desenvolvido por Linhares (2013) um processador baseado no PON, o ArqPON. O intuito desse processador é prover sempre o mesmo *hardware* independente da aplicação. Ainda, assaz inspirado em processadores usuais, o ArqPON provê uma linguagem *Assembly-like* para realizar sua programação [LINHARES, 2015; LINHARES et al., 2014].

Entretanto, de antemão, um problema do ArqPON é a dificuldade de programação, tal como encontrado nas demais soluções em PON em *Hardware*. Aqui excetua-se a solução de Jasinski [JASINSKI, 2012], mas que é muito limitada, não contemplando questões como resolução de conflitos e determinismo, dentre outras. Em suma, a verdade é que estas soluções em *hardware* também demandariam de alguma solução de linguagem e compiladores que facilitassem o desenvolvimento de PON deste âmbito.

2.2.11 Reflexão Sobre o PON

O Paradigma Orientado a Notificações (PON) é uma alternativa aos atuais paradigmas de programação (e.g. imperativo, declarativo e orientado a eventos), inspirando-se e evoluindo a partir dos mesmos, com ênfase nos conceitos dos Sistemas Baseados em Regras (SBR), os quais oferecem (até então) modelos de

programação com maior proximidade à cognição humana [NEWELL e SIMON, 1972].

O PON visa eliminar algumas das principais deficiências dos atuais paradigmas, como a existência de avaliações causais desnecessárias e fortemente acopladas. Na verdade, o PON objetiva-se a proporcionar uma facilidade maior de desenvolvimento com ganho de desempenho, no tocante a processamento, e facilidade na distribuição dos seus componentes.

Ainda, conforme mencionado no estado da arte e particularmente detalhado na Subseção 2.2.8, as materializações do PON não alcançam as propriedades dele. Além disso, conforme apresentado em [FERREIRA, 2013], o PON ainda perde, em tempo de processamento, em aplicações sem grande quantidade de relações causais e com muitas variações de estados de variáveis, quando comparado ao Paradigma Orientado a Objetos (POO).

Embora o PON apresente uma estabilidade maior em tempo de processamento devido a evitar processamento inútil com avaliações desnecessárias de expressões causais, o fato é que existe esta citada desvantagem técnica nas atuais materializações em *Framework* sobre o C++ inclusive devido ao uso de estruturas de dados que têm o seu preço computacional de processamento.

Como uma alternativa para tal, este trabalho vislumbra a apresentação e aperfeiçoamento de uma linguagem específica para o PON e de um respectivo compilador que possa gerar código alvo em linguagens de assaz baixo nível. Tal linguagem para PON visaria principalmente se tornar uma alternativa para implementação de programas baseados no paradigma. Ainda, com a construção de um compilador específico seria possível gerar código-alvo mais puro e menos dependente de estruturas de dados e *overheads* presentes na implementação genérica do *Framework* PON.

2.3 LINGUAGENS E COMPILADORES

Esta seção explica a essência do estado da técnica de compiladores, baseando-se no livro de Aho [2008] e Grune et al [2012]. Isto é pertinente dado o objetivo deste trabalho, que é o desenvolvimento de uma linguagem e um respectivo compilador que materialize o PON de maneira a melhor respeitar suas propriedades

elementares, sendo: minimização do processamento lógico-causal, desacoplamento entre si dos elementos envolvidos no processamento lógico-causal e a facilidade de desenvolvimento de *software*.

Nesse contexto, faz-se necessário salientar que um compilador é um programa que recebe como entrada um programa escrito em uma linguagem de programação – a linguagem fonte – e o traduz para um programa equivalente em outra linguagem – a linguagem alvo [AHO et al., 2008].

Independente da linguagem fonte ou alvo, em geral, o processo de compilação é dividido em duas etapas, cada uma responsável por transformar o código fonte de uma representação para outra. Dentre essas etapas é possível destacar as etapas de análise e de síntese [AHO et al., 2008].

A etapa de análise compreende as sub-etapas (ou fases) de análise léxica, análise sintática e análise semântica. A etapa de análise avalia se o programa fonte está lexicamente correto (todos os símbolos que são utilizados no código são válidos), se está sintaticamente correto (os símbolos estão encadeados corretamente de acordo com a especificação sintática da linguagem) e se está semanticamente correta (se as atribuições de variáveis, por exemplo, são dos tipos corretos, e outras verificações semânticas simples). Caso sejam encontrados erros, o compilador deve apresentar mensagens esclarecedoras ao programador, de modo que este possa tomar as ações corretivas [AHO et al., 2008].

Na verdade, a etapa de análise coleta informações sobre o programa fonte e as armazena em uma estrutura de dados chamada tabela de símbolos. A tabela de símbolos é responsável pelo armazenamento das informações sobre todo o programa fonte e pode ser utilizada por todas as etapas do compilador. Ao final da etapa de análise, é criada uma representação intermediária do código fonte, que serve de base para a realização da etapa de síntese. Neste âmbito, é comum se ter uma estrutura de dados em árvore que representa a estrutura sintática do código, a qual é denominada usualmente árvore sintática ou árvore sintática abstrata [AHO et al., 2008].

A etapa de síntese ao seu turno é responsável por converter a representação intermediária em um código alvo desejado, podendo utilizar as informações contidas na tabela de símbolos. A etapa de síntese pode ter mais variações de um compilador para outro, podendo ser composta pelas etapas de otimização de código e geração de código final (e.g. código de máquina se este for o

código alvo) [AHO et al., 2008]. A Figura 15 apresenta as etapas de análise e síntese, destacando cada uma das suas fases. Ainda, é possível visualizar em tal Figura o relacionamento entre as etapas com a tabela de símbolos, como salientado anteriormente, responsável por armazenar as informações pertinentes sobre o processo de compilação.

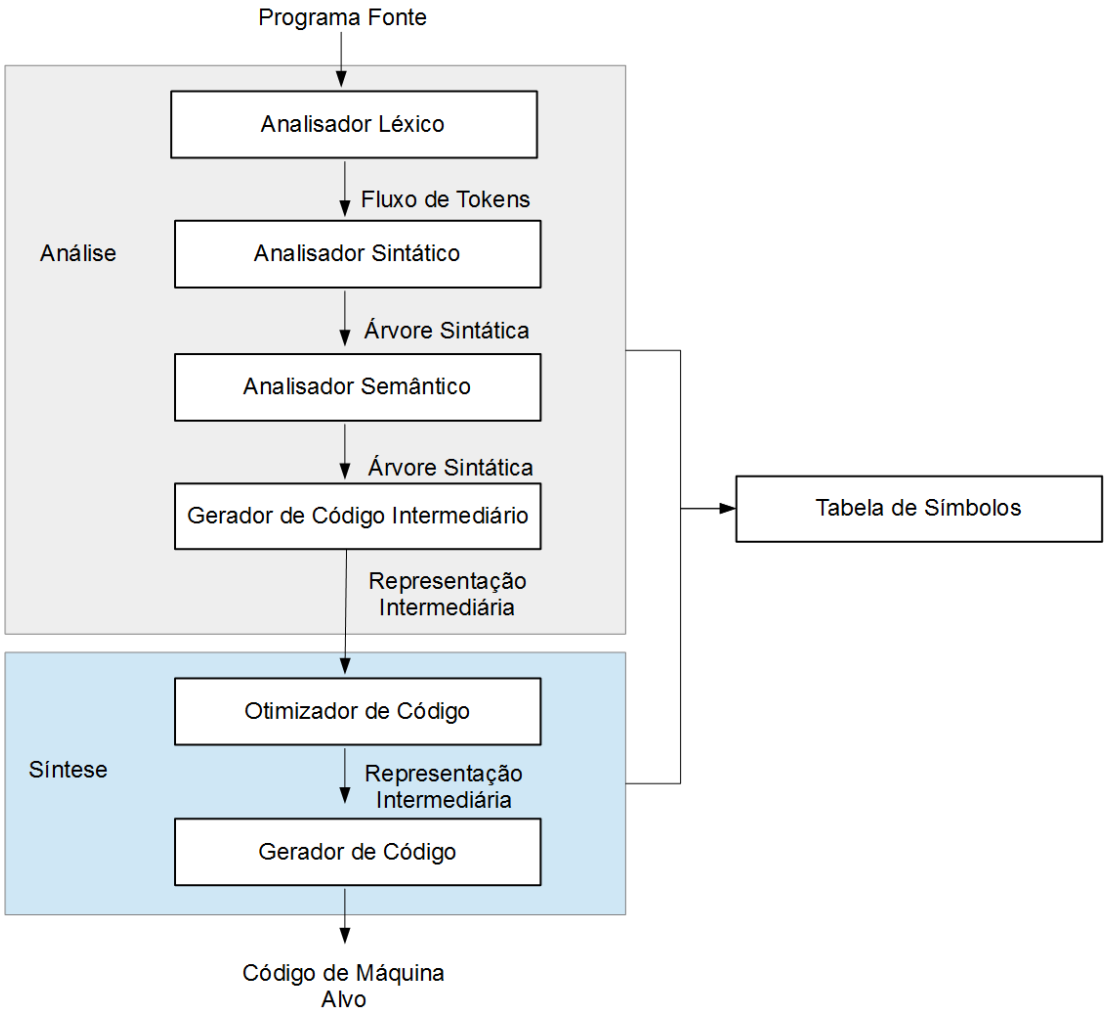


Figura 15 – Fases de um compilador [adaptado de GRUNE et al., 2012]

Deste modo, as próximas Subseções serão responsáveis pelas explicações das fases do compilador de Análise Léxica, Análise sintática e Análise Semântica, sendo estas fases a que apresentam maior pertinência ao trabalho exposto. Subsequentemente é apresentado como o processo de compilação ocorre, por meio de exemplos da configuração necessária para as ferramentas Flex/Lex e Bison [GRUNE et al., 2012], utilizadas para criar a ferramenta de compilação.

2.3.1 Análise Léxica

Como a primeira fase de um compilador, a tarefa principal do analisador léxico é ler os caracteres de entrada do programa fonte, agrupá-los em lexemas (ou *tokens*) e produzir como saída uma sequência de *tokens*, um para cada palavra ou símbolo especial no programa fonte.

De acordo com [AHO et al., 2008] um *token* é um par consistindo de um nome de *token* e um valor de atributo opcional (em caso de identificadores). O nome do *token* é um símbolo abstrato representando um tipo de unidade léxica (e.g. palavra-chave, identificador, número etc). Um lexema é a sequência de caracteres no programa fonte que associa um padrão a um *token* e pode ser considerada uma instância do *token.* Por sua vez, um padrão de *token* pode ser descrito como a forma que os lexemas de um *token* podem tomar. No caso de uma palavra-chave da linguagem, o padrão é a sequência de caracteres que formam a palavra-chave.

Para entender como esses exemplos são usados na prática, o Algoritmo 6 apresenta um trecho de código escrito em linguagem C no qual alguns lexemas são definidos.

Algoritmo 6: Exemplo de lexemas

```
1  printf(“total = %d\n”, score);
```

Como exemplo, no algoritmo acima é possível identificar o lexema *printf* como sendo um padrão para o *token* PRINTF definido na linguagem. O símbolo de parêntesis “*(*“ pode ser considerado um lexema para o padrão do *token* ABREPAR (e.g. Abertura de Parênteses). Seguindo tal linha de raciocínio, *“total = %d\n”* pode ser considerado um lexema para o padrão do *token* descrito como literal. Por sua vez, o *score* é o lexema para o padrão de *token* definido como ID (i.e. identificador de variável). Desta forma, a Figura 16 apresenta os *tokens* e lexemas que foram retirados do trecho de código apresentado no Algoritmo 6.

Token	Lexema
PRINTF	printf
ABREPAR	(
LITERAL	“total = %d\n”
ID	score
FECHAPAR	)
VIRGULA	,
PONTOEVIRG	;

Figura 16 – *Tokens* e lexemas [adaptado de AHO et al., 2008]

O fluxo de *tokens* é enviado ao analisador sintático para que a análise seja efetuada. É comum que o analisador léxico interaja com a tabela de símbolos também. Como exemplo, quando o analisador léxico descobre que um lexema é um identificador de variável, ele precisa inserir esse lexema na tabela de símbolos. Ademais, em alguns casos, as informações referentes ao identificador devem ser lidas da tabela de símbolos pelo analisador léxico para determinar o *token* apropriado que precisa ser repassado ao analisador sintático.

Essas interações são apresentadas na Figura 17. Normalmente, a interação é implementada fazendo-se com que o analisador sintático chame o analisador léxico. A chamada, sugerida pelo comando *getNextToken*, faz com que o analisador léxico leia caracteres de sua entrada até que seja possível identificar o próximo *token* que será retornado ao analisador sintático.

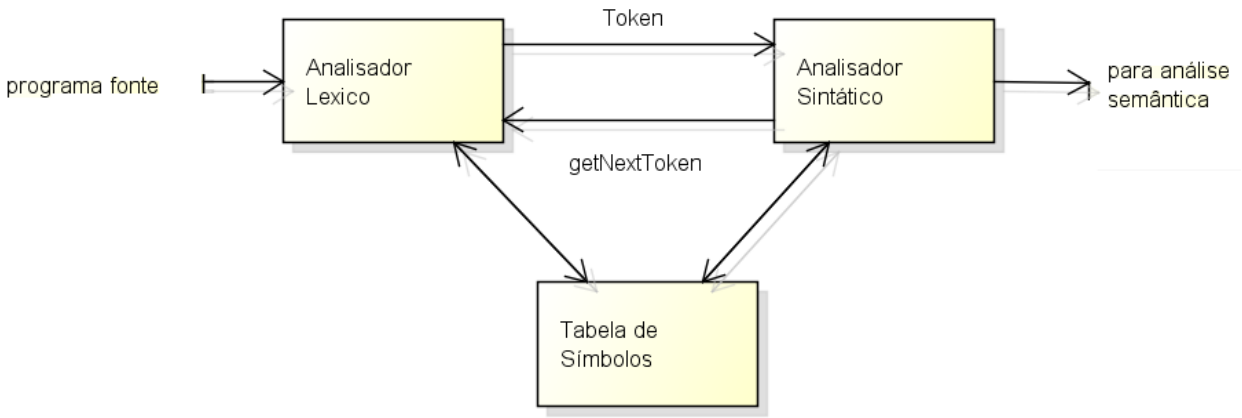


Figura 17 – Interações entre o analisador léxico e o analisador sintático [AHO et al., 2008]

Como o analisador léxico é a parte do compilador que lê o texto fonte, ele pode realizar outras tarefas além da identificação de lexemas. Uma dessas tarefas é remover os comentários e os espaços em branco (i.e. espaço, quebra de linha,

tabulação e talvez outros caracteres que são usados para separar os lexemas na entrada). Outra tarefa é correlacionar as mensagens de erro geradas pelo compilador com o programa fonte. Por exemplo, o analisador léxico pode registrar o número de caracteres de quebra de linha vistos, de modo que possa associar um número de linha a cada mensagem de erro. Em alguns compiladores, o analisador léxico faz uma cópia do programa fonte com as mensagens de erro inseridas nas posições apropriadas. Se o programa fonte utilizar um pré-processador, a expansão de macro-comandos também pode ser realizada pelo analisador léxico.

Para que a fase de análise léxica ocorra, o analisador léxico faz uso, como alternativa, de expressões regulares para o reconhecimento de *tokens.* As expressões regulares são uma importante notação para especificar os padrões de *tokens*. Expressões Regulares provêm uma forma concisa e flexível de identificar cadeias de caracteres que casem com padrões previamente estabelecidos para a linguagem definida. Expressões Regulares são baseadas em uma linguagem formal (e.g. linguagens regulares) que pode ser interpretada por um processador de expressão regular, um programa, neste caso o analisador léxico.

Em suma, um analisador léxico, neste contexto, é responsável pela leitura do fluxo de caracteres que compõem o programa fonte e pela realização do reconhecimento de *tokens* contidos nessa cadeia de caracteres. Como salientado anteriormente, para realizar tal reconhecimento o analisador léxico se utiliza de expressões regulares definidas para linguagem para realizar tal tarefa.

2.3.2 Análise Sintática

A segunda sub-etapa do processo de compilação é denominada análise sintática. A análise sintática tem por objetivo analisar se uma cadeia de símbolos léxicos, lida de um código fonte, pertence a estrutura gramatical determinada por uma gramática formal (ou simplesmente gramática). Para exemplificar tal processo a Figura 18 apresenta uma sequência de análises que são realizadas durante o processo de compilação.

- Programa:

```
if x == y then return 1 else return 2 end
```

- Tokens:

```
IF ID EQ ID THEN RETURN NUM ELSE RETURN NUM END
```

- Árvore:

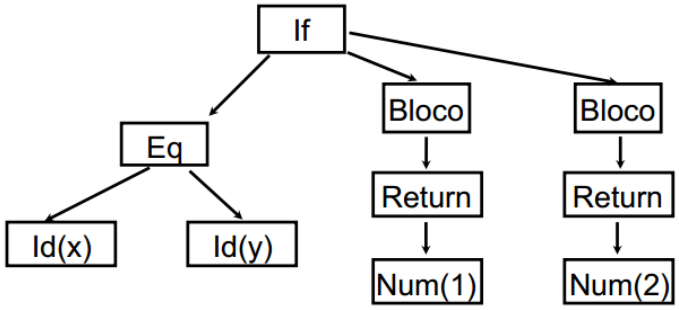


Figura 18 – Sequência de análises do compilador [AHO et al., 2008]

Na Figura 18 é possível observar como um código fonte do programa é analisado pelas sub-etapas de análise léxica e análise sintática. Primeiramente, o código fonte (ou programa) é submetido ao analisador léxico, o que resulta em uma sequência de *tokens*. Essa sequência é submetida ao analisador sintático que cria a árvore sintática baseada na gramática gerada para a linguagem. Nem todas as sequências de *tokens* são programas válidos e, deste modo, o analisador sintático tem que distinguir entre sequências válidas e inválidas baseado na gramática definida. Sendo assim, o analisador sintático verifica se tal cadeia (ou sequência) de *tokens* pertence a linguagem definida fazendo uso de uma gramática.

De fato, a estrutura sintática das construções de uma linguagem de programação é especificada pelas regras gramaticais de uma gramática ou, no caso em questão, uma gramática livre de contexto. Uma gramática livre de contexto é um formalismo que descreve a estrutura de programas na forma de uma linguagem de programação. A princípio, a gramática de uma linguagem descreve somente a estrutura da linguagem, no entanto, uma vez que a semântica da linguagem é definida em termos da sua sintaxe, a gramática também auxilia na definição da sua semântica.

Existem outros tipos de gramáticas além da gramática livre de contexto, conforme definido pela hierarquia de Chomsky [GRUNE et al., 2012]. A Hierarquia de Chomsky é a classificação de gramáticas formais descrita pelo linguista Noam

Chomsky. Esta classificação possui quatro níveis (nível 0 - linguagens recursivamente enumeráveis, nível 1 - linguagens sensíveis ao contexto, nível 2 - linguagens livres do contexto, e nível 3 - linguagens regulares), sendo que os dois últimos níveis (os níveis 2 e 3) são amplamente utilizados na descrição de linguagens de programação e na implementação de interpretadores e compiladores. Mais especificamente, o nível dois é utilizado em análise sintática (computação) e o nível três em análise léxica

Neste sentido, para este trabalho a preocupação ficará voltada somente para gramática livre de contexto e para a gramática regular, sendo ambas necessárias para definição e criação da linguagem aqui visada. Como visto na Subseção anterior, existem vários formalismos de especificação para linguagens regulares, como a gramática regular e a "expressão regular".

Uma gramática consiste em símbolos terminais, símbolos não-terminais, um símbolo inicial e produções. Símbolos terminais são os símbolos básicos a partir dos quais as cadeias são formadas. O termo “nome de *token*” é um sinônimo para “símbolo terminal” e frequentemente a palavra “*token*” é utilizada em vez de símbolo terminal. Assumimos que os terminais são os primeiros componentes dos *tokens* gerados pelo analisador léxico. Os símbolos não-terminais são variáveis sintáticas que representam conjuntos de cadeias. Os conjuntos de cadeias denotando não-terminais auxiliam na definição da linguagem gerada pela gramática. Os não-terminais impõem uma estrutura hierárquica sobre a linguagem que é a chave para a análise sintática e tradução.

Uma produção, por sua vez, é um par contendo um símbolo não-terminal e uma cadeia de símbolos terminais e não-terminais. Produções podem ser consideradas como regras, sendo o não-terminal como lado esquerdo da regra e a cadeia o lado direito. Basicamente, uma gramática livre de contexto especifica quais cadeias são válidas para alguma linguagem. Este processo deve iniciar com a definição de um símbolo inicial chamado de ‘start’. Esse símbolo é substituído pelo lado direito de uma das suas regras. Essas substituições são feitas até ter uma ‘string’ apenas de terminais. As produções de uma gramática especificam a forma como os terminais e os não-terminais podem ser combinados para formar cadeias.

Como exemplo de gramática a Figura 19 apresenta um modelo de gramática definida para o programa representado pela Figura 18.

```
S           -> if_clause
               ;
if_clause   -> IF exp THEN code_block END
             | IF exp THEN code_block ELSE code_block END
               ;
exp         -> ID fator ID
               ;
code_block -> RETURN NUMBER
               ;
fator       -> EQ
             | NE
               ;
```

Figura 19 – Exemplo de gramática livre de contexto [AHO et al., 2008]

Como é possível observar na Figura 19, a gramática deve iniciar com um *token* de *START* no caso referenciado pelo símbolo não-terminal S. Este símbolo não-terminal tem associado a ele uma regra gramatical com dois símbolos: o não-terminal "*if_clause*", e o terminal ";". O não-terminal "*if_clause*" tem a ele associadas duas regras gramaticais (duas produções). A diferença entre as produções se dá pelo fato de no código fonte ser possível montar um *if_clause* com ou sem a opção ELSE. Cada símbolo não-terminal apresenta sua própria regra e produção neste modelo, como exemplo, o símbolo não-terminal "*exp*" que é utilizado para avaliar uma expressão dentro da cláusula "*if_clause*". Ainda, o formato apresentado na Figura 19 é descrito como BNF, o que pode ser considerado como uma abreviação para "Forma de *Backus-Naur*". Uma BNF é adequada para representar uma gramática livre de contexto, ou seja, um modo formal para descrever linguagens formais.

Como salientado anteriormente, o objetivo da análise sintática é validar se uma dada sequência de *tokens* é ou não é válida e uma gramática definida. Essa validação é possível por meio da construção de uma árvore sintática demonstrada na Figura 18. Existem duas estratégias possíveis para a construção da árvore sintática, sendo a construção ascendente e construção descendente [RICARTE, 2008]. Na construção ascendente, o analisador sintático varre a sentença e procura aplicar as produções que permitam substituir sequências de símbolos da sentença pelo lado esquerdo das produções. A sentença é reconhecida quando, na raiz da árvore sintática, o único símbolo restante é o símbolo inicial.

Analisadores Sintáticos Ascendentes são uma categoria de analisadores sintáticos que constroem a árvore de derivação da direita para a esquerda. Dentre os analisadores desta categoria, destacam-se os *LR* (acrônimo de *left to right parsing producing rightmost derivation*). Um analisador sintático *LR* (também chamado *parser LR*) é um algoritmo de análise sintática para gramáticas livres de contexto. Ele lê a entrada de texto da esquerda para a direita e produz uma derivação mais à direita.

Como exemplo, para o processo de construção ascendente será analisado a sequência de *tokens* resultante do programa fonte descrito na Figura 18. A gramática definida para este programa fonte pode ser lida a partir da Figura 19. Deste modo é possível observar na Figura 20 o processo de construção ascendente onde cada *token* encontrado é associado a um símbolo-terminal. O processo de construção ascendente finaliza quando o último *token* é associado e as regras de produção aplicadas são substituídas pelo seu lado esquerdo. No caso, as produções aplicadas foram as que possuem símbolos-terminais associados destacados com o sublinhado.

- Tokens:

```
IF ID EQ ID THEN RETURN NUM ELSE RETURN NUM END
```

```
S           -> if_clause
               ;
if_clause   -> IF exp THEN code_block END
             | IF exp THEN code_block ELSE code_block END
               ;
exp         -> ID fator ID
               ;
code_block -> RETURN NUMBER
               ;
fator       -> EQ
             | NE
               ;
```

Figura 20 – Exemplo de construção ascendente [AHO et al., 2008]

2.3.3 Análise Semântica

A sub-etapa de análise semântica utiliza o processo de construção da análise sintática e as informações armazenadas na tabela de símbolos para verificar

a consistência semântica do programa fonte com a definição da linguagem [AHO et al., 2008]. A semântica, em sua definição, se difere da sintaxe pelo fato de sua preocupação maior estar ligada a significância enquanto a sintaxe se debruça em estruturas ou padrões de como algo pode ser expresso. Como exemplo, muitas linguagens de programação definem tipos de variáveis, por exemplo, uma variável do tipo inteiro. Deste modo, a análise semântica se preocupa em validar se um valor do tipo inteiro foi atribuído a variável do tipo inteiro. Em contrapartida, a análise sintática se preocupa em validar sua estrutura gramatical, ou seja, os símbolos estão encadeados corretamente de acordo com a especificação sintática da linguagem. Para ilustrar tal conceito a Figura 21 apresenta um trecho de código onde é possível perceber as diferenças entre a análise semântica e a análise sintática.

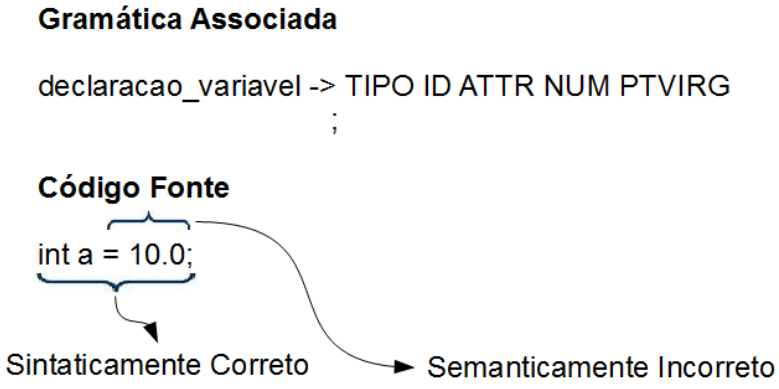


Figura 21 – Avaliação da análise semântica [adaptado de AHO et al., 2008]

Como foi possível observar na Figura 21, primeiramente é apresentada uma gramática que representa a declaração de uma variável. Nesta gramática é possível observar uma produção que consiste de um símbolo não-terminal descrito como *declaração_variavel* e uma cadeia com cinco *tokens* que definem a estrutura de uma declaração de uma variável sendo: o tipo, a identificação, a atribuição, o número associado e o sinal de ponto e vírgula para fechar a declaração. Na imagem, também, é possível observar o código fonte utilizado para declarar uma variável. De acordo com a gramática definida, tal código fonte está sintaticamente correto, no entanto semanticamente ele está incorreto. O *token* que define o valor que pode ser atribuído a variável é o *token NUM*. Para exemplificar, este *token* pode ser associado a qualquer valor numérico (e.g. literal de inteiro e literal de ponto flutuante). No exemplo acima, o valor associado a variável foi um literal de ponto flutuante o que casa com o padrão do *token NUM.* Portanto, para o código fonte os

símbolos associados estão encadeados corretamente de acordo com a gramática definida. No entanto, a análise semântica preocupa-se com a significância e, portanto, um literal de ponto flutuante não pode ser alocado a uma variável de um tipo inteiro o que constitui a inconsistência semântica da expressão.

Por fim, conforme salientado anteriormente, nesta Subseção, a análise semântica faz uso da etapa de construção da análise sintática para realizar as validações. Deste modo, não existe uma ferramenta específica ou um módulo necessário a ser criado para tal análise. Sendo assim, para realizar tal análise é necessário durante o processo de construção da análise sintática realizar as validações semânticas que se façam pertinentes.

2.3.4 Processo de Compilação

Nesta Subseção será apresentado como funciona o processo de compilação utilizando ferramentas automatizadas para isto. Os exemplos aqui apresentados serão guiados pela gramática livre de contexto definida na Subseção 2.3.2, mais precisamente pela Figura 19, que define as regras de uma expressão do tipo *IF.* A Subseção 2.3.4.1 será responsável por apresentar a configuração necessária para criar o módulo de compilação para a fase de análise léxica. A Subseção 2.3.4.2 será responsável por apresentar a configuração necessária para criação do módulo de análise sintática e semântica. A Subseção 2.3.4.3, por sua vez, irá descrever como se dá o processo de geração de código por parte de um compilador.

2.3.4.1 Módulo de Análise Léxica

A ferramenta utilizada neste trabalho para a geração do analisador léxico é o Flex/Lex. O Flex/Lex é um analisador léxico desenvolvido por M. E. Lesk and E. Schmidt, ambos do laboratório AT&T [GRUNE et al., 2012]. A entrada do programa Flex/Lex é um arquivo contendo *tokens* definidos e reconhecidos com a utilização de expressões regulares. Deste modo, a ferramenta Flex/Lex produz um módulo de *software* completo para escaneamento que pode ser compilado e acoplado em conjunto com outros módulos de compilação, como exemplo, o módulo de *software* utilizado para análise Sintática de um código fonte específico.

O arquivo de entrada do Flex/Lex é configurado da seguinte forma:

Algoritmo 7: Modelo do arquivo Flex/Lex

```
1   %{
2
3         #include "if_clause.tab.h"
4
5         int line_num = 1;
6
7   %}
8
9   %%
10
11  if                      return IF;
12  then                    return THEN;
13  else                    return ELSE;
14  return                  return RETURN;
15  end                     return END;
16  "=="                    return EQ;
17  "!="                    return NE;
18
19  [0-9]+                        yylval.sval=strdup(yytext); return
20  NUM;
21  [a-zA-Z\_]([a-zA-Z0-9\_])*    yylval.sval = strdup(yytext);
22        return ID;
23
24  [\t\f ""]               ;
25
26  \n                      line_num++;
27
28  "//".*                        ;
29
30  %%
31
32  int yywrap()
33  {
34        return 1;
35  }
```

Como apresentado no Algoritmo 7, o arquivo Flex/Lex é dividido em três partes separadas pela dupla ocorrência do caractere de porcentagem, i.e. %%. A primeira seção do arquivo Flex/Lex contem uma declaração em linguagem C para incluir o arquivo *if_clause.tab.h*. Nesta seção, também é possível importar bibliotecas específicas outras que se façam necessárias e declarar variáveis que possa ser utilizadas durante a fase de análise léxica, i.e. a variável *line_number* utilizada para armazenar a contagem de linhas de código. Quanto ao arquivo *if_clause.tab.h*, este é produzido pelo Bison [GRUNE et al., 2012], ferramenta para geração de analisadores sintáticos, e é constituído de definições dos *tokens* que são definidos pela linguagem.

A segunda seção do arquivo Flex/Lex apresenta expressões regulares para cada *token* a ser reconhecido e uma ação correspondente. O trecho de código no Algoritmo 8 apresenta como os *tokens* definidos para a expressão *IF* são configurados no arquivo do Flex/Lex. Na verdade, cada *token* é associado com um código *return* seguida de uma Constante que representa o *token*. Essa String será convertida em um valor inteiro que será utilizado posteriormente pelo analisador sintático para reconhecer um item analisado.

Algoritmo 8: *Tokens* definidos para a cláusula *IF*

```
1    %%
2
3    if                 return IF;
4    then               return THEN;
5    else               return ELSE;
6    return             return RETURN;
7    end                return END;
8    "=="               return EQ;
9    "!="               return NE;
10
11   [0-9]+             yylval.sval=strdup(yytext); return NUM;
12
13   [a-zA-Z\_]([a-zA-Z0-9\_])* yylval.sval = strdup(yytext);
14   return ID;
```

.

Ainda, existe uma variável global no analisador sintático denominada *yylval* que é acessada tanto pelo analisador léxico como pelo analisador sintático para adicionar informações adicionais sobre o *token* reconhecido. O código no Algoritmo 9 apresenta a utilização dessa variável global para reconhecer um literal de Inteiro pelo compilador.

Algoritmo 9: Expressão regular para um literal de Inteiro

```
1    %%
2
3    ...
4    [0-9]+             yylval.sval=strdup(yytext); return NUM;
5    ...
```

No trecho de código do Algoritmo 9, mais precisamente na linha 4, é possível observar uma expressão regular correspondente a um literal de Inteiro (e.g. “exemplo de valor Inteiro”). Caso, durante o processo de análise léxica, seja encontrada alguma correspondência no código fonte para esse literal de inteiro, o bloco de código correspondente a essa expressão regular será acionado e o texto

associado ao literal (ou lexema) do inteiro será armazenado na variável *yylval.sval*. Posteriormente, o processo de análise léxica retornará a constante *NUM*, que é o *token* que identifica que uma correspondência ao literal de inteiro foi encontrada no código fonte.

A terceira seção do arquivo Flex/Lex, por sua vez, poderá conter código em C ou C++. No exemplo apresentado no Algoritmo 7 é possível observar a função em C denominada *yywrap*. Esta função é chamada quando o processo de análise léxica encontra o final do arquivo de código fonte. No exemplo do algoritmo 7 a função retorna 1 o que indica que o processo de análise léxica chegou ao fim, no entanto, seria possível apontar para um novo arquivo a ser analisado e continuar com o processo de compilação, como exemplo, compilar vários arquivos de código fonte. Ainda, para isto seria necessário retornar o valor 0 pela função *yywrap.*

Finalmente, o arquivo Flex/Lex criado para o compilador apresenta o nome *if_clause.l*. Para compilar esse arquivo o comando `lex if_clause.l` deve ser executado em linha de comando. O resultado dessa compilação fornece um arquivo chamado *lex.yy.c*, o qual define uma função denominada *yylex().* A cada invocação da função *yylex()* a mesma realiza um *scan* no arquivo de entrada e retorna o próximo *token* encontrado.

2.3.4.2 Módulo de Análise Sintática e Análise Semântica

A segunda fase do processo de compilação é denominada fase de análise sintática. Um analisador sintático é considerado um programa que determina se uma entrada, arquivo contendo código fonte, está sintaticamente correta. Analisadores sintáticos podem ser escritos manualmente ou podem ser gerados automaticamente por um gerador de analisador sintático baseado na descrição de uma estrutura sintática. Essa descrição vem na forma de uma gramática livre de contexto, detalhada na Subseção 2.3.2.

Geradores de analisadores sintáticos podem ser utilizados para desenvolver uma grande variedade de analisadores de linguagem, desde simples analisadores para calculadoras ate linguagens complexas. Um exemplo de gerador de analisador sintático bem estabelecido no estado da arte e da técnica é o Bison. O Bison é um programa que baseado em uma gramática livre de contexto, constrói um programa

em C/C++ que analisa um arquivo de acordo com regras gramaticais. Bison foi desenvolvido por S. C. Johnson no laboratório da AT&T [GRUNE et al., 2012]. Neste trabalho, o Bison é utilizado como ferramenta base para geração do analisador sintático do compilador PON.

Um arquivo de entrada para o programa Bison é constituído da seguinte forma:

Algoritmo 10: Modelo do arquivo Bison

```
1   %{
2         #include <iostream>
3         using namespace std;
4
5         int yylex(void);
6         void yyerror(const char *s);
7
8         extern int line_num;
9   %}
10
11  %union
12  {
13        char *sval;
14  }
15
16  %start S
17
18  %token IF
19  %token THEN
20  %token ELSE
21  %token RETURN
22  %token END
23  %token EQ
24  %token NE
25  %token <sval> NUM
26  %token <sval> ID
27
28  %type <sval> code_block
29
30  %%
31
32  S           : if_clause { }
33                          ;
34
35  if_clause   : IF exp THEN code_block END { }
36              | IF exp THEN code_block ELSE code_block END { cout
37  << $4 <<""<< $6 << endl; }
38              ;
39
40  exp         : ID fator ID { cout << $1 <<""<< $3 << endl; }
41              ;
42
43  code_block  : RETURN NUM { $$ = $2; cout << $2 << endl; }
44
45  fator             : EQ { }
46              | NE { }
47              ;
```

```
48
49  %%
50
51  int main (int argc, char * argv[])
52  {
53        return yyparse();
54  }
55
56  void yyerror(const char *s) {
57        cout <<"Erro (parser) em/antes da linha "<< line_num <<"
58  ["<< s <<"]"<< endl;
59        exit(-1);
60  }
```

Conforme o Algoritmo 10, a primeira seção do arquivo Bison pode conter código em C e C++ além de declarações pertinentes ao processo de análise sintática. Como exemplo, na linha 2 é realizada o *include* da biblioteca *iostream*, o que permite ser possível invocar o método *cout.* Na linha 5 é declarado a função *yylex* que é implementado pelo analisador léxico. O analisador sintático realiza a chamada nessa função toda vez que é necessário encontrar o próximo *token* durante o processo de compilação. Já a linha 8 apresenta o trecho de código responsável por compartilhar a variável *line_number* que permite saber em qual linha de código o processo de compilação se encontra.

Por sua vez, a segunda seção do arquivo Bison consiste de uma lista de *tokens* (além de outros caracteres) que são esperados pelo analisador sintático como uma lista de símbolos utilizados pela gramática. Como exemplo, o trecho de código a seguir apresenta os *tokens* configurados para a expressão *IF*.

Algoritmo 11: Primeira seção do arquivo Bison

```
1   %start S
2
3   %token IF
4   %token THEN
5   %token ELSE
6   %token RETURN
7   %token END
8   %token EQ
9   %token NE
10  %token <sval> NUM
11  %token <sval> ID
12
13  %type <sval> code_block
```

Como é possível visualizar no Algoritmo 11, para o código fonte da expressão *IF* foram criados uma série de *tokens* que representam itens da linguagem. Esses *tokens*, quando compilados pelo Bison, são adicionados no

arquivo *if_clause.tab.h* que é utilizado no processo de análise sintática. O *token* S configurado na linha 1 do Algoritmo 11 é precedido pela palavra reservada *start*. Isto define que ele será o *token* de início do processo de análise sintática.

A terceira seção do arquivo Bison consiste na gramática livre de contexto para a linguagem. Nesta seção são adicionadas as regras de tradução. Cada regra consiste em uma ou mais produções da gramática e a sua ação semântica associada. O Algoritmo 12 apresenta a terceira seção do arquivo Bison criado para o programa *IF.*

Algoritmo 12: Segunda seção do arquivo Bison

```
1    S            : if_clause { }
2                 ;
3
4    if_clause    : IF exp THEN code_block END { }
5                 | IF exp THEN code_block ELSE code_block END { cout
6    << $4 <<""<< $6 << endl; }
7                 ;
8
9    exp          : ID fator ID { cout << $1 <<""<< $3 << endl; }
10         ;
11
12   code_block   : RETURN NUM { $$ = $2; cout << $2 << endl; }
13
14   fator        : EQ { }
15                | NE { }
16                ;
```

O algoritmo 12, mais precisamente na linha 4, apresenta o símbolo não terminal *if_clause* e suas produções. Para a regra do símbolo não terminal *if_clause*, duas produções foram criadas. Em uma produção para o Bison, cadeias de caracteres e dígitos, não declaradas como *tokens*, são consideradas símbolos não terminais. Geralmente os símbolos não terminais são escritos em caixa baixa. Cadeias de caracteres e dígitos declarados como *tokens* são considerados como símbolos terminais. Símbolos terminais geralmente são denotados em caixa alta. Lados direitos alternativos podem ser separados por uma barra em vertical e um ponto e vírgula vem após cada lado esquerdo com suas alternativas e suas ações semânticas. Vale ressaltar, que um símbolo não terminal pode possuir mais de uma produção associada, o que constitui de lados alternativos em um regra gramatical.

Quando uma produção é associada durante o processo de compilação sua ação semântica é executada. Uma ação semântica do Bison é uma sequência de instruções em C ou C++ e é desenvolvida dentro do bloco {}. No algoritmo 13, é

possível visualizar a ação semântica associada a produção do símbolo não terminal *code_block*. Essa produção é composta por dois símbolos terminais, sendo, *RETURN* e *NUM*. Dentro da ação semântica é possível capturar o valor associado ao *token*. Como exemplo, o símbolo *$1* refere-se ao valor do literal adicionado ao *token NUM.* A ação semântica é efetuada sempre que se reduz pela produção associada, de modo que normalmente a ação semântica calcula um valor $$ em termos dos $is. Neste exemplo, o símbolo $$ irá receber o valor de $2 que poderá ser utilizado durante o processo de construção ascendente da análise sintática, e ainda, armazenado na tabela de símbolos.

Algoritmo 13: Exemplos de produções

```
1    code_block        : RETURN NUM { $$ = $2; cout << $2 << endl; }
2    ...
```

A quarta seção do arquivo Bison consiste de código em C/C++. Nesta seção é incluso um método *main* que realiza a chamada na função *yyparse()*. A função *yyparse()* é o 'motor' do analisador sintático. Ainda, é necessário criar a função *yyerror()* que é utilizada para reportar os erros encontrados durante a compilação. O Algoritmo 14 apresenta as funções utilizadas no compilador PON na quarta seção do arquivo do Bison.

Algoritmo 14: Quarta seção do arquivo Bison

```
1    %%
2
3    int main (int argc, char * argv[])
4    {
5          return yyparse();
6    }
7
8    void yyerror(const char *s) {
9          cout <<"Erro (parser) em/antes da linha "<< line_num <<"
10   ["<< s <<"]"<< endl;
11         exit(-1);
12   }
```

O analisador sintático não apresenta uma saída (i.e. em termos de código), no entanto, uma árvore é implicitamente construída durante o processo de análise sintática, conforme apresentada na Figura 18. Durante este processo é criado uma representação interna da estrutura do programa. Essa representação interna é baseada no lado direito das regras. Quando um lado direito é encontrado sua

produção é reduzida até o seu lado esquerdo correspondente, de forma recursiva. A análise se faz completa quando todo o programa é reduzido para o símbolo de início da gramática (*start*).

Para geração do analisador sintático é necessário compilar o arquivo *if_clause.y,* representado pelo Algoritmo 10, com o comando `bison -d if_clause.y` na linha de comando. O resultado dessa compilação irá gerar dois arquivos, sendo o *if_clause.tab.h* e o *if_clause.tab.c*. O arquivo *if_clause.tab.h* contém a lista de *tokens* que deverá ser inserida no arquivo que define o analisador léxico, conforme mencionado anteriormente. Já o arquivo *if_clause.tab.c* define a função *yyparse()* responsável por iniciar o processo de compilação.

2.3.4.3 Geração de código

Durante a sub-etapas de análises léxica e sintática/semântica o compilador tem por objetivo interagir e armazenar informações pertinentes do programa fonte em uma estrutura de dados denominada tabela de símbolos. Essa interação ocorre quando uma produção, ou regra gramatical, é associada durante o processo de compilação. Uma vez que a produção é associada sua ação semântica é executada [GRUNE et al., 2012].

Uma ação semântica, por sua vez, consiste de um bloco de código onde é possível adicionar código em C ou C++. Este bloco de código permite capturar informações do programa fonte e incluí-los na tabela de símbolos conforme salientado. Deste modo, ao final das sub-etapas de análises léxica e sintática/semântica todas as produções associadas executarão suas ações semânticas e a tabela de símbolos, então, conterá todas as informações necessárias coletadas durante este processo [GRUNE et al., 2012].

Sendo assim, para geração de código, o compilador irá fazer uso das informações coletadas e armazenadas na tabela de símbolos. Essas informações serão utilizadas por rotinas específicas criadas com o intuito de gerar código alvo. Salienta-se que não existe uma ferramenta genérica para geração de código, ficando a cargo do criador do compilador a construção das rotinas de geração de código que se façam necessárias.

2.4 REFLEXÃO SOBRE A REVISÃO DA LITERATURA

Este capítulo apresentou uma revisão acerca dos conceitos relacionados com os paradigmas de programação. Ainda, uma revisão mais aprofundada sobre a contextualização do PON foi apresentada bem como as suas principais características. Neste capítulo foi possível compreender os conceitos relacionados aos compiladores bem como as ferramentas que são necessárias para a construção de compiladores.

Os conceitos de PON objetivam construir software mais facilmente. Por meio da programação inspirada na forma declarativa (e.g. fatos e regras). No tocante à codificação, o PON permite moldar uma solução em forma mais natural, encapsulando complexidades de execução mediante seu modelo de ciclo de notificações. Ademais, é caracterizado por uma pesquisa recente, extensa, sólida e emergente, incluindo pedidos de patente, artigos publicados em conferências e revistas, e um grupo de trabalho e pesquisa estabelecido. Como salientado neste capítulo, o PON foi materializado na forma de um *Framework* sendo que o mesmo se encontra em sua terceira versão. De fato, tal materialização permite compor *software* a luz dos conceitos estabelecidos no estado da arte do PON. No entanto, de acordo com o cálculo assintótico do PON os resultados de desempenho das aplicações geradas até o momento com tal *Framework* não apresentaram resultados de todo satisfatório, o que resulta em suposições de melhoria do estado da técnica do PON via utilização de um compilador específico.

Por sua vez, houve uma evolução no processo de criação de compiladores. Atualmente, é possível utilizar ferramentas para criação de módulos de *software* que contemplam as fases de compilação, como a fase de análise léxica e análise sintática. Dentre essas ferramentas, destacam-se a ferramenta Flex/Lex para criação de módulos de software de análise léxica, e a ferramenta Bison para criação de módulos de software de análise sintática. Ainda, para a concepção de um compilador específico se faz necessário entender as fases de compilação e gerar a gramática livre de contexto relacionado a linguagem a ser estabelecida. Essa gramática livre de contexto é utilizada pelas ferramentas de compilação e os módulos de software podem ser criados. Durante o processo de compilação, uma árvore sintática é gerada e informações pertinentes podem ser coletadas e armazenadas em uma estrutura de dados denominada tabela de símbolos. Tal

estrutura pode ser utilizada pelas fases seguintes do processo de compilação, como exemplo a fase de geração de código intermediário. Enfim, tal entendimento se faz necessário para compor um compilador específico.

No próximo capítulo serão abordadas a Linguagem e Compilador PON criados. É importante salientar que tanto a Linguagem quanto a versão inicial do Compilador PON foram resultados de uma disciplina ofertada pelo PPGCA da UTFPR e ministrada pelos Profs. João A. Fabro e Jean M. Simão.

3 LINGUAGEM E COMPILADOR PARA O PARADIGMA ORIENTADO A NOTIFICAÇÕES

Este capítulo apresenta as definições da linguagem (e do compilador) criados para o Paradigma Orientado a Notificações (PON). Primeiramente, a Seção 3.1 apresenta as definições sobre a linguagem PON, bem como os seus detalhes com a apresentação de exemplos. Ainda, este capítulo na sua Seção 3.2 apresenta detalhes do compilador PON com relação ao seu funcionamento e estrutura interna. Nesta Seção, ainda, é possível visualizar os resultados do compilador para as três possibilidades de geração de códigos-alvo desenvolvidas: 1. *Framework Otimizado do PON (versão 2.0) em C++;* 2. Código específico em C++ (“PON C++”); 3. Código específico em C (“PON C”). Finalmente, a Seção 3.3 apresenta uma reflexão acerca da linguagem e do compilador desenvolvidos[2].

3.1 LINGUAGEM PON

Uma linguagem de programação pode ser definida por meio de um formalismo de especificação denominado “gramática” [AHO et al., 2008]. Uma gramática define a estrutura geral de formação de sentenças válidas para uma linguagem. Para dar suporte à linguagem PON em um compilador próprio, foi especificada uma gramática segundo a *Backus-Naur Form* (BNF). Em tempo, toda a BNF definida para a Linguagem PON é apresentada no Apêndice B.

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

[2] Salienta-se que este trabalho evoluiu durante a apresentação de três seminários de qualificação de mestrado realizados no PPGCA. No primeiro seminário o objetivo foi realizar correções no compilador para que fosse possível criar mais aplicações no âmbito do mesmo [FERREIRA, 2014a]. No segundo seminário foram incluídas a linguagem e compilador as propriedades do PON descritas na Subseção 1.1.3 [FERREIRA, 201b]. Finalmente, no seminário três foram realizadas correções e validações do compilador, e ainda, foram coletados os relatos de utilização da linguagem e compilador PON por partes dos pesquisadores [FERREIRA, 2014c].

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*.

Nesse âmbito, o Algoritmo 15 apresenta o padrão de declarações da linguagem LingPON a começar pelo que concerne *FBEs*. É importante ressaltar que todos os algoritmos demonstrados nesta seção para fins didáticos foram obtidos com a criação da aplicação Mira ao Alvo, a qual será discutida posteriormente na Subseção 4.2.1.

Conforme apresentado no Algoritmo 15, o código PON é de fato dividido em cinco partes. A primeira parte representa a declaração dos *FBEs*. A segunda parte representa a definição das instâncias de tais *FBEs*. A terceira parte apresenta a estratégia de escalonamento a ser adotada. A quarta parte, por sua vez, representa a criação das *Rules*. Por fim, a quinta parte apresenta o bloco de código main.

Algoritmo 15: Padrão de declarações da linguagem PON

```
1   fbe Apple
2       . . .
3   end_fbe
4
5   fbe Archer
6       . . .
7   end_fbe
8
9   fbe Controller
10      . . .
11  end_fbe
12
13  ----------------
14
15  inst
16      Apple apple, apple1
17      Archer archer, archer1
18      Controller controller, controller1
19  end_inst
20
21  ----------------
22
23  strategy
24        . . .
25  end_strategy
26
27  ----------------
28
29  rule RlTurnOn1
30      . . .
31  end_rule
32
33  ----------------
34
35  main {
36        . . .
37  }
```

Nessa estrutura, a palavra reservada *FBE* anuncia o início da estrutura de um *FBE*. Por padrão, todos os *FBEs* do programa devem ser definidos na primeira parte do código. Na sequência, devem ser declaradas as instâncias dos *FBEs*. Para isso, a palavra reservada *inst* anuncia a abertura desse bloco de instanciações.

A próxima estrutura a ser declarada é a estratégia de escalonamento das *Rules* a qual faz uso da palavra reservada *strategy*. Ainda, as definições de *Rules* devem ser declaradas, fazendo uso da palavra reservada *Rule*. É pertinente ressaltar que tanto os *FBEs* quanto as *Rules* não possuem um número limitado de declarações.

Finalmente, a próxima estrutura a ser declarada é o bloco de código *main* o qual possibilita a inserção de código nativo na linguagem alvo escolhida durante o processo de compilação.

Conforme apresenta o Algoritmo 16, um *FBE* é composta por duas partes. A primeira parte representa a declaração dos *Attributes*, enquanto a segunda parte representa a definição dos *Methods*. Cada parte, assim como no código apresentado no algoritmo anterior, possui uma palavra reservada para a abertura (e.g. a*ttributes*) do bloco e outra para o fechamento (e.g. *end_attributes*).

Algoritmo 16: Exemplo de criação de *FBEs*

```
 1  fbe Archer
 2      attributes
 3          boolean atHasFired false
 4          integer atCount 0
 5      end_attributes
 6      methods
 7          method mtFire1(atHasFired = true)
 8          method mtFire2(atCount = atCount + 1)
 9          method mtFire3(atCount = atCount + atCount)
10          method mtFire4() begin_method //código específico em C/C++
11  end_method
12      end_methods
13  end_fbe
```

Os *Attributes* possuem uma estrutura comumente utilizada na programação atualmente, formada pelo tipo do dado, seguido do nome do atributo e seu respectivo valor inicial. Nesse âmbito, os tipos de dados disponíveis na linguagem são *boolean*, *integer*, *float*, *char* e *string*.

Os nomes de atributos, normalmente, seguem um padrão de nomenclatura já adotada em outros trabalhos anteriores do PON [Ronszcka, 2012; Valença, 2012], fazendo uso do prefixo *at*, seguido de um mnemônico que defina o propósito do

atributo em questão. Por fim, o valor inicial deve estar de acordo com o tipo de dado empregado na construção de tal atributo.

Os *Methods*, por sua vez, apresentam uma construção particular em sua estrutura. Esta se dá pelo anúncio da abertura de um *Method* através da palavra reservada method seguida do nome e sua respectiva funcionalidade. Ademais, assim como os *Attributes*, os nomes dos *Methods* seguem o padrão de nomenclaturas conhecido. Desta forma, no caso de *Methods* instanciados, faz-se uso do prefixo *mt*. Neste âmbito, o Algoritmo 16, entre as linhas 7 e 10, apresenta as construções válidas para os *Methods* na linguagem PON.

Na linha de código 7, a funcionalidade do *Method* mtFire1 está entre parênteses. Basicamente tal funcionalidade altera o valor do *Attribute atHasFired* para *true*. Já a funcionalidade do *Method* mtFire2 e mtFire3, linhas 8 e 9 respectivamente, apresentam a possibilidade de atribuição de uma operação (e.g. soma) em um *Attribute*. Finalmente, o *Method* mtFire4 possibilita ao desenvolvedor adicionar código nativo da linguagem alvo do compilador ao bloco de código do método definido entre as palavras reservadas *begin_method* e *end_method.*

A segunda parte do código consiste na instanciação dos *FBEs* definidos na primeira parte. Para isso, o Algoritmo 17 exemplifica o padrão de implementação sugerido.

Algoritmo 17: Exemplo de instanciações de *FBEs*

```
1    inst
2        Apple apple, apple1
3        Archer archer, archer1
4        Controller controller, controller1
5    end_inst
```

No Algoritmo 17, é possível observar entre as linhas 2 e 4, que três tipos distintos de *FBEs* são utilizados na criação de seis instâncias. É importante ressaltar que os fins-de-linha comumente aplicados em algumas linguagens (e.g. para C ou C++) não devem ser utilizados nessa versão da linguagem.

A terceira parte do código consiste na definição da estratégia de resolução de conflitos a ser utilizada. Conforme explicado na seção 2.2.3, a estratégia de resolução de conflitos do PON consiste em *NO_ONE*, *BREATH* e *DEPTH*. O Algoritmo 18 apresenta o padrão de implementação sugerido para adicionar a estratégia resolução de conflitos *NO_ONE* no código fonte em LingPON.

Algoritmo 18: Exemplo de definição de estratégia de escalonamento

```
1    strategy
2          no_one
3    end_strategy
```

A quarta parte do código consiste na criação do conhecimento lógico-causal da aplicação por meio da criação de regras, onde as *Rules* farão as conexões entre as entidades PON. O Algoritmo 19 demonstra o padrão de implementação para a criação de uma *Rule*.

Algoritmo 19: Exemplo de criação de Rules

```
1    rule RlTurnOn1
2        condition
3            subcondition A1
4                premise PrIsCrossed apple.atIsCrossed == false and
5                premise PrHasFired archer.atHasFired == false and
6                premise PrFire controller.atFire == true
7            end_subcondition
8            or
9            subcondition A2
10               premise PrIsCrossed archer.atCount == 0
11           end_subcondition
12       end_condition
13       action
14           instigation inFire archer.mtFire1();
15       end_action
16   end_rule
```

A definição de uma *Rule* é anunciada com a palavra-chave *rule* seguida de um nome para a mesma. É importante ressaltar que o nome da *Rule* nesse ponto é obrigatório, não podendo ser omitido. Ainda, a *Rule* pode ser considerada dependente de outra *Rule*. Esse é umas das propriedades do PON discutidas na seção 2.2.7. Para isso, após a declaração da *Rule* é necessário utilizar a palavra reservada *depends* seguida do identificador da outra *Rule* visada.

Basicamente, cada *Rule* é composta por três partes, que são as suas *Properties*, a *Condition* (expressão lógica) e *Action* (execução). No Algoritmo 19 não é possível visualizar o bloco *Properties* que pode ser utilizado para *Rule*. Deste modo, é possível perceber que esse bloco é opcional. O Algoritmo 20, como exemplo, apresenta o código utilizado para definição do bloco *Properties* de uma *Rule*.

Algoritmo 20: Propriedades das *Rules*

```
1  properties
2                priority 1
3                keeper true
4  end_properties
```

Como é possível visualizar no Algoritmo 20, o bloco *Properties* da *Rule* pode ser composto por duas propriedades, sendo a propriedade *Priority* e a propriedade *Keeper*.

A propriedade *Priority* define uma ordem de prioridade de execução de *Rules* quando duas ou mais *Rules* compartilham o mesmo *Exclusive Attribute* (*Atributo Exclusivo*, cf. seção 2.2.3) em algumas de suas *Premisses*. Sendo assim, a *Rule* aprovada que apresentar a maior prioridade vai ter sua execução priorizada com relação as demais *Rules*.

A propriedade *Keeper* define o comportamento de execução da *Rule*. A *Rule* com essa propriedade configurada como *true* somente executará suas *Instigations* com a chamada de um método específico, denominado "*execute"*. Ou seja, a *Rule* poderá estar ativa, no entanto, sua execução dependerá da chamada de um método de execução.

Nessa versão da linguagem, é obrigatória a utilização de *SubConditions*, seguidas de um identificador, mesmo para a composição de expressões simples. Como exemplo de tal construção, tem-se as linhas 3 a 9 do Algoritmo 19. Seguindo o estado da arte do PON, a construção de uma *SubCondition* necessita de, ao menos, uma *Premise*. No caso da utilização de mais de uma *Premise*, estas devem ser aninhadas com conjunções (*and*). Para utilização de "ou"- disjunções (*or*) - em uma *Rule* é necessário a criação de duas ou mais *SubConditions*. É possível observar tal possibilidade no Algoritmo 19 entre as *SubConditions* A1 e A2.

Para a definição de *Premises* na linguagem PON, a palavra reservada premise deve ser utilizada, seguida de um nome (opcional) e uma comparação. Ainda, pode ser aplicada à *Premise* uma propriedade inerente dos padrões de execução em PON, conhecida como *Attribute* Impertinente (cf. seção 2.2.6). Para tal, basta adicionar, após o identificador da *Premise,* a palavra reservada *imp*. Isto torna o *Attribute*, relacionado à *Premise*, impertinente a mesma. O Algoritmo 21 apresenta o trecho de código com a definição de uma *Premise* na linguagem PON.

Algoritmo 21: Definição de uma *Premise*

```
1    premise PrIsCrossed imp apple.atIsCrossed == false
```

A comparação é composta por três elementos: o valor de um *Attribute* vinculado a uma instância de um *FBE* (e.g. *apple*, instância de um *FBE* do tipo *Apple*), o valor de comparação (e.g. ==) e o valor a ser comparado (e.g. true). Este último pode ser tanto uma constante, quanto outro *Attribute*. Os operadores de comparação possíveis são: “==”, “<”, “>”, “<=”, “>=”, “!=”.

A segunda parte, que representa a execução da *Rule*, é anunciada a partir da palavra-chave *Action*. Este bloco consiste no vínculo de instigações a *Methods* definidos em *FBEs*. Conforme apresentado na linha 14 do Algoritmo 19, a estrutura das *Instigations* é composta pela palavra-chave *instigation* seguida de um nome (opcional) e um método de uma instância particular de um *FBE*.

O conjunto de padrões de nomenclatura prevê os seguintes prefixos para as entidades PON utilizadas nessa etapa: *rl* para *Rules*, *cd* para *Conditions*, *sc* para *SubConditions*, *pr* para *Premises*, *ac* para *Actions* e in para *Instigations*.

Finalmente, a quinta parte do código PON consiste na criação do bloco de código principal (*main*). Este trecho de código permite ao desenvolvedor adicionar código específico para a linguagem alvo definida. Conforme é apresentado no Algoritmo 22, entre as linhas 54 a 60, é possível observar tal estrutura. É pertinente ressaltar que código na linguagem alvo não é avaliado pelas regras de compilação da LingPON, sendo somente copiado ao código alvo gerado pelo compilador.

Como dito anteriormente, o exemplo utilizado nos testes foi baseado na aplicação Mira ao Alvo e o código PON elaborado é mostrado no Algoritmo 22. É possível observar em tal algoritmo nas linhas 3 e 4 a possibilidade de inserção de comentários seguindo o padrão comumente utilizado nas linguagens de programação usuais.

Algoritmo 22: Implementação da aplicação Mira ao Alvo na linguagem PON

```
1   fbe Apple
2       attributes
3           boolean atIsCrossed false //teste de comentario
4       end_attributes  /* teste de comentário */
5   end_fbe
6
7   fbe Archer
8       attributes
9           boolean atHasFired false
10          integer atCount 0
11      end_attributes
12      methods
13          method mtFire1(atHasFired = true)
14          method mtFire2(atCount = atCount + 1)
15          method mtFire3(atCount = atCount + atCount)
16          method mtFire4() begin_method //código específico em
17  C/C++ end_method
18      end_methods
19  end_fbe
20
21  fbe Controller
22      attributes
23          boolean atFire false
24      end_attributes
25  end_fbe
26
27  inst
28      Apple apple, apple1
29      Archer archer, archer1
30      Controller controller, controller1
31  end_inst
32
33  strategy
34        no_one
35  end_strategy
36
37  rule RlTurnOn1
38      condition
39          subcondition A1
40              premise PrIsCrossed apple.atIsCrossed == false and
41              premise PrHasFired archer.atHasFired == false and
42              premise PrFire controller.atFire == true
43          end_subcondition
44          or
45          subcondition A2
46              premise PrIsCrossed archer.atCount == 0
47          end_subcondition
48      end_condition
49      action
50          instigation inFire archer.mtFire1();
51      end_action
52  end_rule
53
54  main {
55        //exemplo de código fonte no main.
56        //isso não é validado pelo compilador
57        apple->setatIsCrossed(false);
58        archer->setatHasFired(false);
59        controller->setatFire(true);
60  }
```

3.2 COMPILADOR PARA O PON

Como salientado no capítulo 2, um compilador pode ser visto como um programa tradutor que transforma uma linguagem fonte escrita em alto nível em uma linguagem objeto geralmente/tradicionalmente de mais baixo nível, próximo a uma linguagem de máquina (mas não necessariamente). Para atingir este objetivo, o compilador é constituído de fases que operam em sequência, nomeadamente análise léxica, análise sintática, análise semântica e geração de código. Cada fase apresenta um objetivo específico e o resultado da fase anterior é utilizado pela fase posterior. Sendo assim, fez-se necessário o desenvolvimento do compilador específico para o PON, o qual foi concebido após a especificação da LingPON (vide apêndice B para sua descrição completa em BNF).

Deste modo, esta Seção tem por objetivo apresentar os detalhes do compilador desenvolvido, sendo a Subseção 3.2.1 responsável por discutir sobre os arquivos necessários para compor os módulos de análise léxica e sintática criados a partir das ferramentas de compilação utilizadas neste trabalho, no caso o Flex/Lex e Bison. Também são apresentados os componentes de compilação do LingPON criados para gerenciar as informações da tabela de símbolos e geração de código. A Subseção 3.2.3 apresenta detalhes da geração de código em *Framework* PON. Ao seu turno, a Subseção 3.2.4 apresenta detalhes da geração de código específico em linguagem C++, e a Subseção 3.2.5 apresenta detalhes da geração de código específico em linguagem C.

3.2.1 Configuração e componentes do Compilador PON

Conforme apresentado na Subseção 2.3.4, para desenvolver um compilador é possível fazer uso de ferramentas que criam módulos de análise léxica e análise sintática. Para o desenvolvimento do compilador PON foram utilizadas as ferramentas Flex/Lex para criar o módulo de análise léxica, e Bison para criação do módulo de análise sintática. O arquivo de configuração utilizado pelo Flex/Lex, para a especificação dos *tokens* da LingPON é apresentado em detalhes no Apêndice C.

Com o módulo de análise léxica configurado a próxima etapa contou com a criação do módulo de análise sintática. Para este módulo foi necessário criar o arquivo de configuração utilizado pela ferramenta Bison. Basicamente, no arquivo de configuração do Bison estão presentes os *tokens* definidos para a linguagem bem como a gramática livre de contexto. Deste modo, o apêndice D apresenta em detalhes o arquivo de configuração para o Bison.

Durante a fase de análise sintática/semântica o analisador somente realiza uma varredura entre os *tokens* encontrados no código fonte com a realização de chamadas recursivas à função *yylex()* do analisador léxico. Conforme este processo ocorre, as regras de produção definidas pela gramática livre de contexto são associadas e suas respectivas ações semânticas são executadas. Este processo é chamado de produção ascendente e permite capturar e armazenar informações úteis para a geração de código. Essas informações podem ser armazenadas em uma estrutura de dados, normalmente, conhecida como tabela de símbolos e que permite, ao final da fase de análise sintática e semântica, a geração do código esperado.

Deste modo, para o compilador PON foram criadas um conjunto de classes em C++ que comportam uma estrutura de dados para representação e armazenamento das informações pertinentes do processo de compilação, e posterior geração de código. A estas classes deu-se o nome de componentes do compilador PON. A Figura 22 apresenta como as fases de análise léxica, sintática e semântica se relacionam com os componentes do compilador PON.

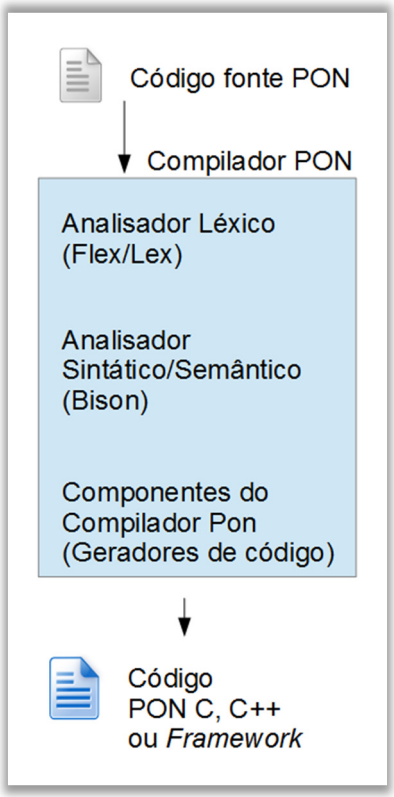


Figura 22 – Integração entre as fases do compilador e os componentes do compilador PON

Para abstrair os elementos PON durante o processo de compilação, algumas classes C++ foram criadas com o intuito de armazenar as informações pertinentes sobre cada elemento PON. A Figura 23 apresenta um diagrama de classes que representa as classes criadas e utilizadas durante o processo de montagem da tabela de símbolos.

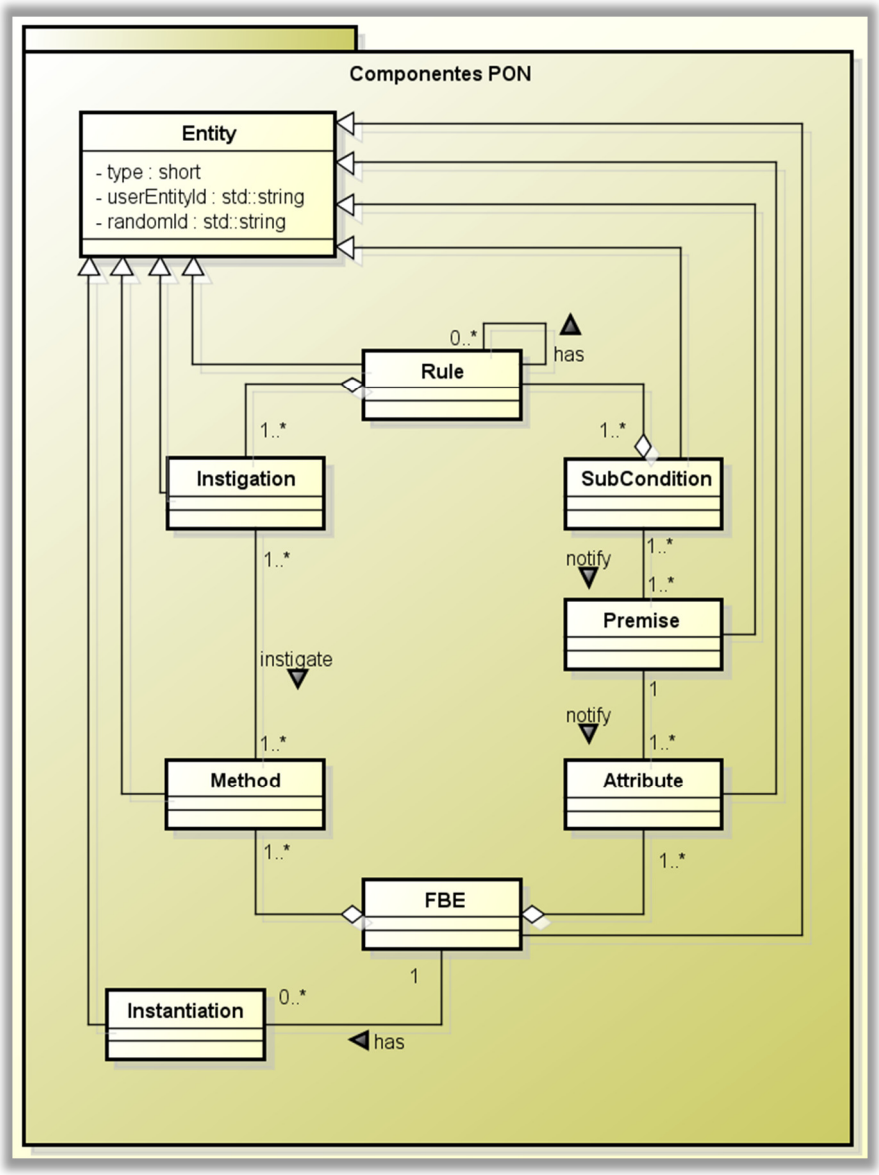


Figura 23 – Diagrama de classes dos componentes PON para a Tabela de Símbolos.

Conforme é possível visualizar na Figura 23, para o compilador PON foi criada uma classe C++ chamada de *Entity* a qual possui atributos para dados precisos cujas entidades derivadas reaproveitam. Entre essas informações estão o tipo de elemento, seu identificador, caso seja encontrado no código fonte, e um identificador randômico a ser utilizado caso o usuário opte por não adicionar um identificador a algum elemento.

Para compreender como o processo de análise sintática interage com instâncias dessas classes e realiza a inserção dessas informações nesta estrutura será utilizado como exemplo um código simplificado de construção de uma *FBE* inserida no código fonte. A Figura 24 demonstra o pseudocódigo do processo de

tradução de uma *FBE* e o seu relacionamento com a classe *Compiler* responsável por criar instâncias das entidades dos componentes PON e incluí-las na tabela de símbolos.

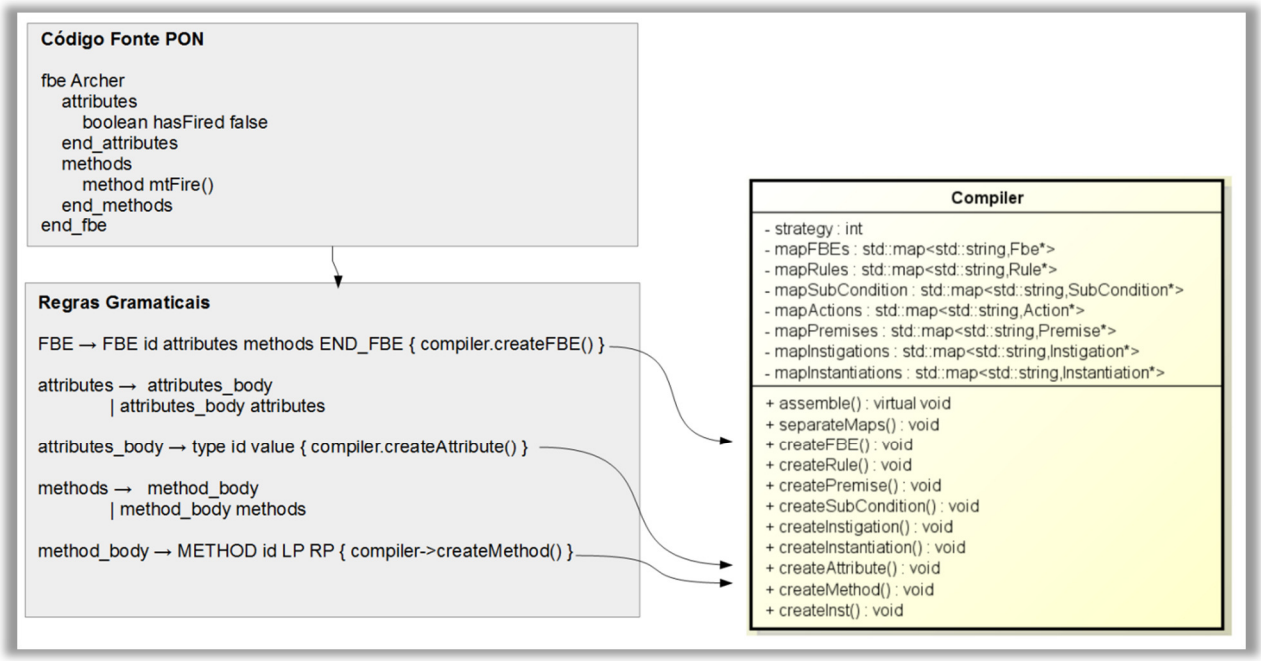


Figura 24 – Processo de tradução do compilador PON.

Como é possível visualizar na Figura 24, ao receber um código fonte e analisá-lo, o módulo de análise léxica/sintática interage com a classe *Compiler.* Neste exemplo, são realizadas três chamadas aos métodos *createMethod*, *createAttribute* e *createFBE* que são responsáveis pela criação de entidades que representam os conceitos de *Method*, *Attribute* e *FBE,* respectivamente.

Em tempo, a classe *Compiler* é uma classe utilitária que foi criada com o objetivo de gerar uma dada instância para manipular a criação dos objetos necessários pelo processo de compilação. Além disso, tal instância dela aloca cada objeto criado na tabela de símbolos do compilador. Deste modo, a imagem a seguir mostra detalhes dos métodos criados para a classe *Compiler* de modo a permitir alocar os componentes dinamicamente na tabela de símbolos.

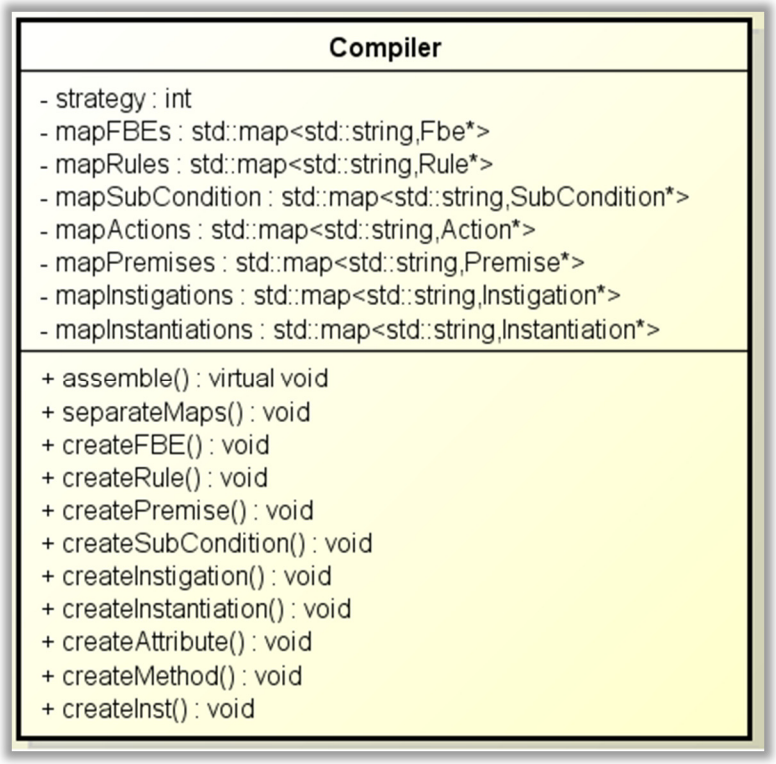


Figura 25 – Classe *Compiler*

Como é possível visualizar na Figura 25, a classe *Compiler* apresenta uma série de métodos utilitários que são chamados pelo processo de análise sintática para armazenar em estruturas de dados as informações pertinentes (e.g. *createInstigation*). Como exemplo, o Algoritmo 23 apresenta o código criado para o método *createInstigation* utilizado pelo compilador quando é necessária a inclusão de um elemento deste tipo.

Algoritmo 23: Método *createInstigation*

```
1   Entity * Compiler::createInstigation(std::string userEntityId,
2   std::string methodUsed) {
3
4         Entity *entityFound=
5   semanticAnalyser.getEntity(userEntityId);
6         Instigation *entity;
7
8         if (entityFound == 0) {
9
10              entity = new Instigation(userEntityId);
11              entity->Method = methodUsed;
12              semanticAnalyser.addEntity(entity);
13
14        } else {
15              semanticAnalyser.errorEntityExists(entityFound);
16        }
17        return entity;
18  }
```

O Algoritmo 23 apresenta o código utilizado pelo compilador para adicionar uma *Instigation* na tabela de símbolos do compilador PON. É possível perceber que na linha 5 do método *createInstigation* é realizada a chamada ao método *getEntity* da instância da classe *SemanticAnalyser*. Essa chamada é realizada para validar se o item a ser inserido na estrutura de dados do compilador já existe. Caso exista, uma mensagem de erro será exibida. Tal validação pode ser definida como uma forma de análise semântica realizada pelo compilador e justifica o nome da classe *SemanticAnalyser*. Ainda, na linha 12 é utilizado a chamada ao método *addEntity*. Esse método realiza a inclusão da Entidade no *Maps*[3] da classe *SemanticAnalyser*. Essa é uma estrutura única e genérica que tem por objetivo agrupar todos os itens capturados no processo de análise sintática. A Figura 26 apresenta detalhes de classe *SemanticAnalyser*.

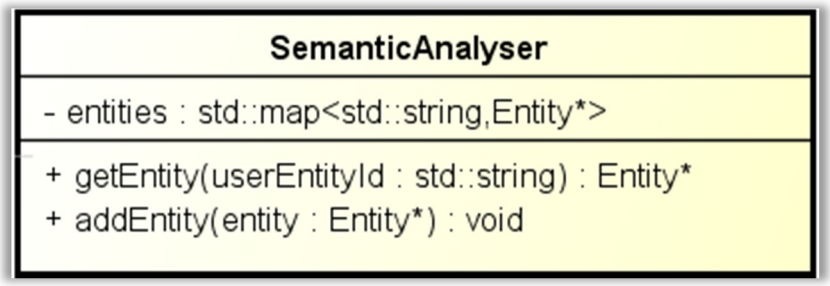


Figura 26 – Classe SemanticAnalyser

Os componentes PON capturados pelo processo de análise sintática são armazenados em um *Map* genérico na classe *SemanticAnalyser*. No entanto, para continuar com o processo de geração de código é necessário separar essa lista em itens específicos. Deste modo, é possível destacar o método *separateMaps* da classe *Compiler*. Este método é utilizado para que ao final do processo de análise sintática todos os elementos adicionados nas estruturas de dados sejam separados em *Maps* específicos (e.g *Maps* que armazenam *Entities* do tipo *Rule, Maps* que armazenam *Entities* do tipo *Condition* etc). Cada Entidade quando criada recebe um tipo específico. Ao todo no compilador existem 10 tipos específicos. Deste modo, o método *separateMaps* da classe *Compiler* realiza a separação das entidades

[3] Estrutura de dados do tipo mapa. Um mapa armazena pares (chave, valor) chamados de itens. Chaves e valores podem ser de qualquer tipo definido na linguagem utilizada. A chave é utilizada para achar um elemento rapidamente na estrutura.

baseado em seu tipo. O Algoritmo 24 apresenta os tipos definidos para o compilador PON.

Algoritmo 24: Tipos definidos para o compilador PON

```
1    static const short E_RULE = 1;
2    static const short E_CONDITION = 2;
3    static const short E_ACTION = 3;
4    static const short E_PREMISE = 4;
5    static const short E_INSTIGATION = 5;
6    static const short E_SUBCONDITION = 6;
7    static const short E_FBE = 7;
8    static const short E_INSTANTIATION = 8;
9    static const short E_METHOD = 9;
10   static const short E_ATRIBUTE = 10;
```

A fase de análise sintática finaliza quando o *token* referenciado como *START* é encontrado. Na ação semântica deste *token* é possível encontrar a chamada ao método *assemble* da classe *Compiler*. Este método foi concebido para ser um método abstrato, pois a classe *Compiler* não foi concebida para gerar código e somente ser uma classe utilitária para manipulação das informações na tabela de símbolos. O Algoritmo 25 apresenta a regra definida com *START* para o compilador PON. É importante salientar que o *token PROGRAM* foi definido como o símbolo inicial da gramática livre de contexto do PON. Sendo assim, quando a ação semântica desta regra é ativada, o método *assemble* da classe *Compiler* é acionado.

Algoritmo 25: Regra gramatical *START*

```
1    PROGRAM : FBEs inst strategyRules {compiler->assemble();}
2            ;
```

Para a geração de código, como parte do objetivo deste trabalho, foram criadas as classes *CCompiler*, *CPPCompiler* e *FrameworkCompiler* que geram respectivamente código alvo específico para: PON em linguagem C, PON em linguagem C++ e PON em *Framework* PON Otimizado (i.e. *Framework* PON versão Otimizado). A referência dessas classes foi adicionada no arquivo *bison_pon.y*. A Figura 27 apresenta o diagrama de classes contendo as classes de geração de código intermediário do compilador PON.

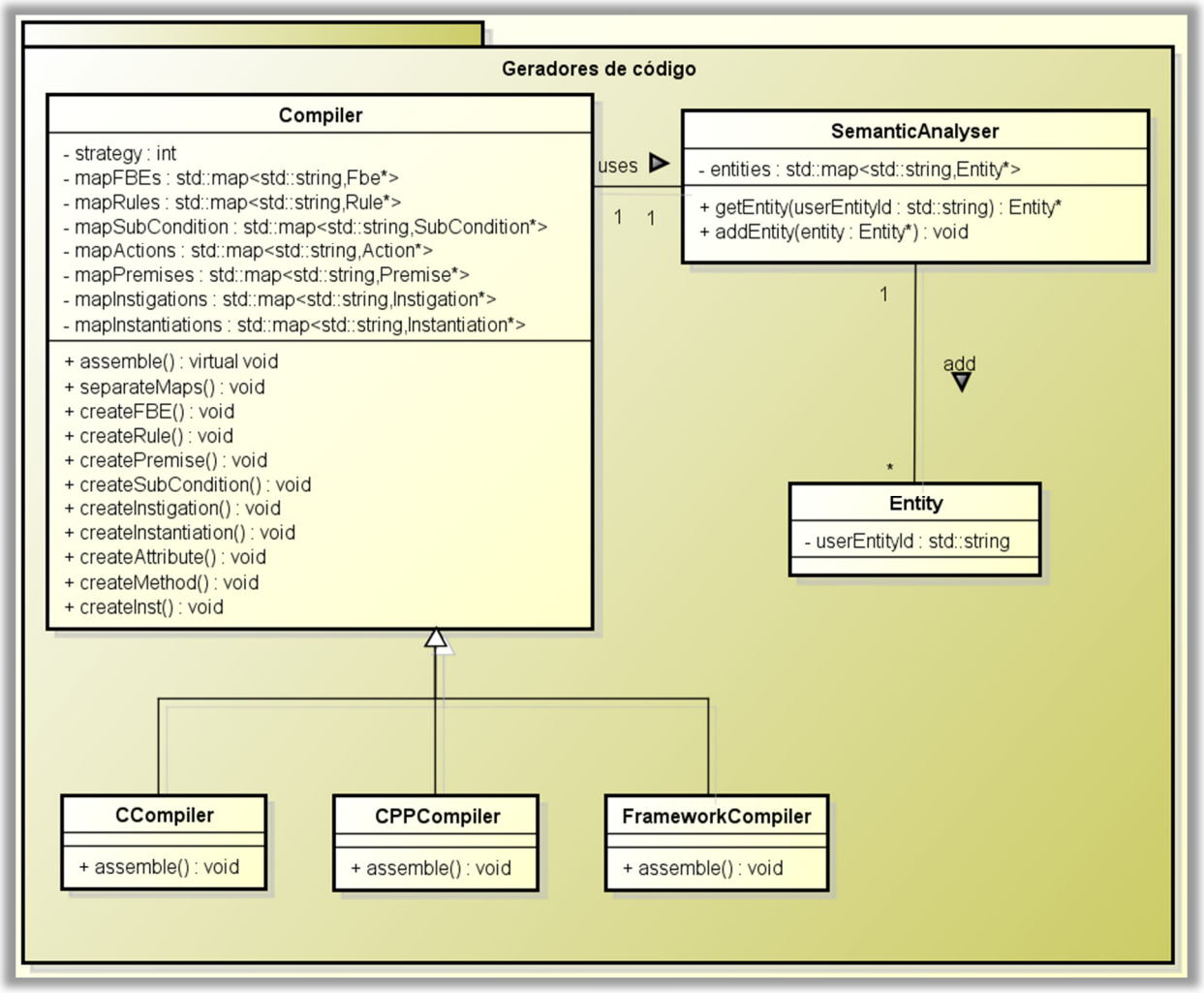


Figura 27 – Classes para geração de código intermediário

A chamada ao método *assemble* pela ação semântica do *token* START irá inicializar o processo de geração de código pelas classes correspondentes e, consequentemente, finalizar o processo de compilação.

Para realizar a criação do compilador PON é necessário compilar os arquivos gerados tanto pela ferramenta Flex/Lex em conjunto com os arquivos gerados pelo Bison. Estes arquivos foram adicionados em uma pasta específica em conjuntos com os componentes do compilador PON. O comando "`g++ lex.yy.c bison_pon.tab.c *.cpp -o pon`" irá compilar todos os componentes do compilador e resultar na geração do executável denominado pon.

Este executável poderá receber um arquivo fonte que contém um código PON e terá como saída o código esperado. O comando "`./pon 1 < exemplo.txt`" apresenta um exemplo de utilização do compilador PON. Nele o executável *pon* é chamado e são atribuídos dois parâmetros. O primeiro parâmetro

varia de 1 a 3 sendo esses valores correspondentes a geração de código intermediário em C, C++ e *Framework* respectivamente. Já o segundo parâmetro atribui ao compilador o código fonte que deverá ser analisado.

3.2.2 Implementação do Analisador Semântico em *Framework* Otimizado PON

A geração de código para o *Framework* Otimizado PON, basicamente, recria os componentes do *Framework* de acordo com o código PON criado. Cada elemento do código PON tem a sua correspondência no *Framework* conforme explicado na Subseção 2.2.9.

Para simular a criação de código *Framework* via compilador PON, um programa simples foi gerado, o Mira ao Alvo, o qual será discutido com mais detalhes no capítulo 4. Como é possível observar no Algoritmo 26 tem-se a criação de uma *FBE* chamada *Archer*.

Algoritmo 26: Código PON para *Archer*

```
1   fbe Archer
2       attributes
3           boolean atHasFired false
4           integer atCount 0
5       end_attributes
6       methods
7           method mtFire1(atHasFired = true)
8           method mtFire2(atCount = atCount + 1)
9           method mtFire3(atCount = atCount + atCount)
10          method mtFire4() begin_method //código
11  específico em C/C++ end_method
12      end_methods
13  end_fbe
```

No Algoritmo 27 é possível visualizar o código equivalente a *FBE Archer* para o *Framework* constituído na Linguagem PON. Note que o código gerado para o *Framework* já realiza as importações necessárias.

Algoritmo 27: Classe *Archer* em *Framework*

```
1   /*
2    * Archer.h
3    *
4    *  Created on: Jul 16, 2013
5    *  Author: add your name
6    */
7   #ifndef Archer_H_
8   #define Archer_H_
9   #include "framework/utils/SingleInclude.h"
10  class Archer : public FBE {
11  public:
12              Archer();
13              ~Archer();
14              //Atributes
15              Boolean * atHasFired;
16              Integer * atCount;
17              //methods
18              Method * mtFire1;
19              Method * mtFire2;
20              Method * mtFire3;
21              Method * mtFire4;
22  };
23  #endif /* Archer_H_ */
24
25  #include "Archer.h"
26  Archer::Archer(void) {
27        BOOLEAN(this,atHasFired,false);
28        INTEGER(this,atCount,0);
29        METHOD(this,mtFire1,atHasFired,true,0);
30        METHOD_OPERATION(this,mtFire2,atCount,new
31  Addition(atCount,1),0);
32        METHOD_OPERATION(this,mtFire3,atCount,new
33  Addition(atCount,atCount),0);
34        METHOD(this,mtFire4,,,0);
35  }
36  Archer::~Archer(void) {
37  }
```

No Algoritmo 28 é apresentado o trecho de código PON para criação de uma *Rule*.

Algoritmo 28: Código PON para criação de uma *Rule*

```
1   rule RlTurnOn1
2       condition
3           subcondition A1
4               premise PrIsCrossed apple.atIsCrossed == false and
5               premise PrHasFired archer.atHasFired == false and
6               premise PrFire controller.atFire == true
7           end_subcondition
8           or
9           subcondition A2
10              premise PrIsCrossed archer.atCount == 0
11          end_subcondition
12      end_condition
13      action
14          instigation inFire archer.mtFire1();
15      end_action
16  end_rule
```

O Algoritmo 28 apresenta o código PON referente a 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 29 apresenta o código *Framework* gerado que compõe o arquivo principal utilizado para executar o programa. 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 29: Classe Main.h *Framework*

```
1   #ifndef MAIN_H_
2   #define MAIN_H_
3   #include <iostream>
4   using std::cout;
5   using std::endl;
6   #include <cstdio>
7   #include "framework/NOPApplication.h"
8   #include "framework/utils/SingleInclude.h"
9   #include "Apple.h"
10  #include "Archer.h"
11  #include "Controller.h"
12
13  class Main : public NOPApplication {
14
15  public:
16        Main();
17        virtual ~Main();
18
19  public:
20        void initStartApplicationComponents();
21        void initFactBase();
22        void initRules();
23        void initSharedEntities();
24        void codeApplication();
25
26  public:
27        RuleObject * rlTurnOn1;
28        Premise* prFire;
29        Premise* prHasFired;
30        Premise* prIsCrossed;
31        Instigation* inFire;
32        Apple * apple;
33        Apple * apple1;
```

```
34          Archer * archer;
35          Archer * archer1;
36          Controller * controller;
37          Controller * controller1;
38  };
39
40  #endif
```

Finalmente, o Algoritmo 30 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 28. Como é possível observar o código alvo é capaz de associar cada elemento pertencente a *Rule* de forma correta.

Algoritmo 30: Método initRules *Framework*

```
1   void Main::initRules() {
2           Scheduler * scheduler = SingletonScheduler::getInstance();
3           RULE (rlTurnOn1,scheduler,Condition::CONJUNCTION);
4             rlTurnOn1->addPremise(prIsCrossed);
5             rlTurnOn1->addPremise(prFire);
6             rlTurnOn1->addPremise(prHasFired);
7             rlTurnOn1->addPremise(prIsCrossed);
8             rlTurnOn1->addInstigation(inFire);
9             rlTurnOn1->end();
10  }
```

3.2.3 Implementação PON C++

A implementação em C++ foi baseada nos princípios do PON. O código gerado pelo compilador apresenta classes C++ que representam os conceitos PON relativos a *FBE*, *Rule* e *Premise*. O conjunto de *Premises* associados a uma *Rule* específica no código C++ pode ser considerado como uma *Condition*.

Para analisar o código C++ gerado para a linguagem PON será utilizado como exemplo o código para o programa denominado Mira ao Alvo. O Algoritmo 26, descrito na Subseção anterior, apresenta o trecho de código referente a criação da *FBE* que representa o conceito do Arqueiro (*Archer*).

Já o Algoritmo 31 apresenta o código correspondente gerado pelo compilador PON para C++ referente ao FBE *Archer*. Como é possível observar uma classe C++ é gerada para representar este conceito, a qual segue o padrão C++ *builder* [GAMMA et al., 1995].

Algoritmo 31: Código em C++ gerado para a Classe Archer

```
1   #pragma once
2   #include "PrHasFired.h"
3   class PrHasFired;
4   class Archer
5   {
6         public:
7           Archer (void);
8           ~Archer (void);
9
10          bool atHasFired;
11          void setatHasFired(bool atHasFired);
12          bool getatHasFired();
13          int atCount;
14          void setatCount(int atCount);
15          int getatCount();
16          void mtFire1();
17          void mtFire2();
18          void mtFire3();
19          void mtFire4();
20
21          PrHasFired pr25;
22  };
23
24  #include “Archer.h”
25  Archer::Archer(void)
26  {
27        atHasFired = false;
28  }
29  Archer::~Archer(void){}
30
31  void Archer::setatHasFired(bool atHasFired) {
32        if(this->atHasFired != atHasFired) {
33              this->atHasFired = atHasFired;
34              pr25.setatHasFired(atHasFired);
35        }
36  }
37  bool Archer::getatHasFired() {
38        return this->atHasFired;
39  }
40  void Archer::setatCount(int atCount) {
41        if(this->atCount != atCount) {
42              this->atCount = atCount;
43        }
44  }
45  int Archer::getatCount() {
46        return this->atCount;
47  }
48  void Archer::mtFire1() {
49        this->sethasFired(true);
50  }
51  void Archer::mtFire2() {
52        this->setatCount(this->atCount = this->atCount+1);
53  }
54  void Archer::mtFire3() {
55        this->setatCount(this->atCount = this->atCount+this-
56  >atCount);
57  }
58  void Archer::mtFire4() {
59        //código específico em C/C++
60  }
```

Para satisfazer o conceito de *Rule*, *Condition*, *SubCondition*, *Premise*, *Action* e *Instigation* no código C++, algumas classes foram criadas. O Algoritmo 32 apresenta a classe C++ criada para atender o conceito de *Rule*, a qual é baseada no código PON apresentado pelo Algoritmo 28. Como é possível observar, entre as linhas 23 a 25, a classe C++ *Rule* apresenta uma série de referências para todas as *FBE,* que serão instigadas quando a *Rule* estiver ativa. Ainda, para cada *Premise* criada no código PON será gerada uma classe *Premise* em C++. Como é demonstrado no Algoritmo 32, entre as linhas 17 a 21, a classe *Rule* tem referência para cada *Premise* relacionada.

Algoritmo 32: Código C++ correspondente a implementação de uma *Rule*

```
1   #pragma once
2   #include <iostream>
3   using namespace std;
4   class PrFire;
5   class PrHasFired;
6   class PrIsCrossed;
7   class Archer;
8   class Apple;
9   class Controller;
10  class R1TurnOn {
11  public:
12        RlTurnOn(void);
13        ~RlTurnOn(void);
14
15        bool isApproved();
16        void setPrFire(PrFire * fire);
17        PrFire * a14;
18        void setPrHasFired(PrHasFired * hasFired);
19        PrHasFired * a13;
20        void setPrIsCrossed (PrIsCrossed * isCrossed);
21        PrIsCrossed * a12;
22
23        Archer * a5;
24        Apple * a2;
25        Controller * a8;
26  };
27
28  #include "RlTurnOn.h"
29  #include "PrFire.h"
30  #include "PrHasFired.h"
31  #include "PrIsCrossed.h"
32  #include "Archer.h"
33  #include "Apple.h"
34  #include "Controller.h"
35  RlTurnOn::RlTurnOn(){
36  }
37  RlTurnOn::~RlTurnOn(){
38  }
39  void RlTurnOn::setPrFire(PrFire * a14){
40        this->a14 = a14;
41        this->a14->condition = this;
```

```
42  }
43  void RlTurnOn::setPrHasFired(PrHasFired * a13){
44        this->a13 = a13;
45        this->a13->condition = this;
46  }
47  void RlTurnOn::setPrIsCrossed(PrIsCrossed * a12){
48        this->a12 = a12;
49        this->a12->condition = this;
50  }
51  bool RlTurnOn::isApproved(){
52        if(a14->isApproved & a13->isApproved & a12->isApproved){
53              a5->mtFire1();
54              return true;
55        } else {
56              return false;
57        }
58  }
```

Conforme apresentado pelo Algoritmo 32 (linha 51) toda *Rule* tem um método denominado *isApproved* este método é utilizado pelo processo de inferência para validar se a *Rule* foi satisfeita e deve acionar as *Instigations*.

O Algoritmo 33 apresenta o código C++ gerado para uma *Premise*. Neste código, entre as linhas 20 a 27, é possível observar que a *Premise* apresenta um método chamado *setatHasFired* que é correspondente ao atributo da classe *FBE* que a *Premise* faz referência. Existe um atributo chamado *isApproved*, linha 10, que é utilizado para armazenar o estado da *Premise* e uma referência para a *Rule*, linha 9, na qual a *Premise* está associada.

Algoritmo 33: Código C++ correspondente a uma *Premise*

```
1   #pragma once
2   #include "RlTurnOn.h"
3   class RlTurnOn;
4   class PrHasFired {
5         public:
6               PrHasFired(void);
7               ~PrHasFired(void);
8
9               RlTurnOn * condition;
10              bool isApproved;
11              void setatHasFired(bool atHasFired);
12  };
13
14  #include "PrHasFired.h"
15
16  PrHasFired::PrHasFired() {
17  }
18  PrHasFired::~PrHasFired() {
19  }
20  void PrHasFired::setatHasFired(bool atHasFired) {
21        if(atHasFired == false) {
22              this->isApproved=true;
```

```
23              this->condition->isApproved();
24          } else {
25              this->isApproved = false;
26          }
27    }
```

O código C++ gerado apresenta o mesmo conceito de execução do PON. Cada vez que um atributo de uma *FBE* é configurado com um valor específico, diferente do valor atribuído anteriormente, este vai acionar o processo de inferência. Este valor pode ser atribuído com a utilização do método *set* que é criado para cada atributo da *FBE*. Este método possui uma referência para cada *Premise* relacionada ao atributo e o novo valor é atribuído a *Premise*. Como exemplo, o Algoritmo 31 na linha 34 apresenta o trecho de código responsável por acionar a *Premise* relacionada ao atributo.

Por sua vez, cada *Premise,* ao receber o novo valor do atributo, verifica se este novo valor consegue satisfazer sua condição. Caso a condição da *Premise* seja verdadeira, esta vai acionar a *Rule* associada ao chamar o método *isApproved*. O Algoritmo 33 na linha 23 apresenta o trecho de código que contempla essa operação.

Na *Rule* o método *isApproved* verificará o estado de cada *Premise* associada, ao validar o valor do atributo *isApproved*, e caso todas as *Premise*s estejam ativas a *Rule* se tornará ativa. Com a *Rule* ativa, as *Instigations* relacionados a *Rule* são invocadas e o processo de inferência é finalizado.

Por fim, o Algoritmo 34 apresenta o trecho de código referente ao método *isApproved* da *Rule* RlTurnOn.

Algoritmo 34: Método *isApproved* da class *Rule*

```
1    bool RlTurnOn::isApproved(){
2          if(a14->isApproved & a13->isApproved & a12->isApproved){
3                a5->mtFire1();
4                return true;
5          } else {
6                return false;
7          }
8    }
```

3.2.4 Implementação PON C

A geração de código C é feita de forma a criar vários arquivos independentes com o objetivo de facilitar o entendimento de cada uma das partes

individualmente. Além disso, para cada elemento gerado existe sempre um Header (.h), que contém as definições, as estruturas e cabeçalhos de funções, e um arquivo (.c) que implementa os procedimentos e as funções pertinentes àquele módulo.

Existem alguns arquivos que são genéricos e em todos os casos, independente da aplicação, eles existem e possuem o mesmo nome. Representam as *Rules*, as *SubConditions* e *Premisses*. Todos esses arquivos seguem o mesmo padrão, têm uma estrutura (*struct*) com apenas um campo do tipo short (*isApproved*). Esse recebe um valor quando o elemento que está representando é aprovado (se torna verdadeiro). Todos os elementos do mesmo tipo (*Rule*, *SubCondition* ou *Premise*) são agrupados no seu respectivo arquivo e declarados como variáveis, independente das *FBEs* que fazem parte. Como as *Rules*, *SubConditions* ou *Premises* são semelhantes, apenas um exemplo ilustrativo, o da estrutura *Rule* e declaração das suas variáveis, são apresentados no Algoritmo 35.

Algoritmo 35: Código C correspondente a uma *Rule*

```
1    typedef struct RuleType {
2      short isApproved;
3      short isDependent;
4    } Rule;
5
6    Rule RlTurnOn1;
```

No tocante às *FBEs*, para cada uma delas são criados dois arquivos (um .h e outro .c). Eles implementam os atributos agrupando-os na forma de uma estrutura (*struct*). Neste arquivo é criada uma variável, do tipo daquela *FBE*, para cada instância existente no arquivo que possui o código relacionado as *FBE*'s. Dentro de cada arquivo *FBE*, foi criado um procedimento *setAttribute*() para cada um dos atributos existentes. Além disso, ao invés da sub-rotina receber como parâmetro a instância do atributo que deve ser modificada, as funções foram replicadas para cada instância individualmente.

Por exemplo, uma *FBE* chamada *Archer*, contém um atributo chamado *atHasFired* do tipo *bool* e tem duas instâncias: Archer e Archer1, assim o código gerado conterá duas funções que realizarão as mudanças nos atributos, são elas: *setarcheratHasFired(bool pValue)* e *setarcher1atHasFired(bool pValue)*.

O código em C gerado pelo compilador desenvolvido segue uma sequência de atos que é baseada nas ações esperadas pela execução de um programa escrito em PON. É importante destacar que o paralelismo entre as mesmas não foi

considerado nem tratado, assim tudo ocorre de forma sequencial. Uma mudança em um atributo é tratada em sua totalidade, ou seja, até finalizar a execução de todas as ações pertinentes a ela, e somente depois disso é que outra mudança será considerada.

Um exemplo ilustrativo do código gerado, baseado na aplicação (Mira ao Alvo), é mostrado no Algoritmo 36.

Algoritmo 36: Código C correspondente a estrutura *Archer*

```
1    void setarcheratHasFired( bool pValue){
2      if (pValue != archer.atHasFired) {
3        archer.atHasFired = pValue;
4       funcInstarcherAttatHasFiredNotifyPrPrHasFired();
5      }
6    }
7    void funcInstarcherAttatHasFiredNotifyPrPrHasFired(){
8      if (archer.atHasFired == false){
9          if (PrHasFired.isApproved == 0) {
10         setPrPrHasFiredisApproved(1);
11         funcInstarcherPrPrHasFiredNotifyScA1();
12         }
13     } else setPrPrHasFiredisApproved(0);
14
15   }
16   void funcInstarcherPrPrHasFiredNotifyScA1(){
17     if (PrFire.isApproved & PrHasFired.isApproved &
18   PrIsCrossed.isApproved){
19       setScA1isApproved(1);
20       funcInstarcherScA1NotifyRlRlTurnOn1();
21     }  else setScA1isApproved(0);
22
23   }
24   void funcInstarcherScA1NotifyRlRlTurnOn1(){
25     if (A2.isApproved | A1.isApproved){
26   setRlRlTurnOn1isApproved(1);
27       instarchermtFire1();
28    } else {
29       setRlRlTurnOn1isApproved(0);
30    }
31   }
```

O fluxo de execução do programa se dá da seguinte forma: a cada mudança de um *Attribute*, todas as *Premises* interessadas são avaliadas. No caso de serem aprovadas, configuram a variável (*isApproved*) de sua instância como verdadeira. Em seguida, a *SubCondition* que contém aquela premissa é notificada. Uma avaliação bit a bit das variáveis que indicam a aprovação das *Premises* existentes naquela *SubCondition* é realizada para verificar se ela é aprovada, linha 17 do código.

No caso dela ser aprovada, a *Rule* interessada é notificada. Em seguida, a função que avalia a *Rule* verifica se todas as *SubConditions* existentes nela estão válidas, linha 25. Se sim, a *Action* indicada pela *Rule* é executada finalizando assim uma etapa. Após essa execução, pode ocorrer outra mudança de *Attribute* que faz com que a cadeia seja disparada novamente.

3.3 REFLEXÕES SOBRE A LINGUAGEM E COMPILADOR PON

Este capítulo apresentou a definição da linguagem PON bem como seus detalhes com a utilização de exemplos baseados na construção de um aplicativo denominado Mira ao Alvo. Como foi possível observar, a linguagem PON foi pensada para expor de maneira simples os conceitos relativos ao PON e permitir a criação de código PON sem a necessidade do *Framework*.

Além da linguagem, este capítulo apresentou detalhes de como o compilador PON foi criado e quais ferramentas foram necessárias para criação do compilador. Vale ressaltar que, atualmente, as linguagens alvo do compilador são código em *Framework* Otimizado (i.e. Versão 2.0) para PON em C++, código específico em C++ para PON, código específico em C para PON. A diferença entre a geração de código para o *Framework* e o código específico em C++ é que, no segundo caso, geram-se diretamente as estruturas de notificação, sem utilizar necessariamente as estruturas de dados e classes do *Framework*. Já no caso da geração de código específico em C, optou-se por não utilizar nenhuma das facilidades da orientação a objetos, sendo que todas as notificações são implementadas diretamente através de chamadas de funções. No porvir, certamente que o conjunto de linguagens alvo pode evoluir, talvez alcançando Assembly para arquiteturas atuais baseadas nos conceitos de Von Neumann, Assembly para ArqPON e mesmo para hardware como o CO-PON e PONHD.

O objetivo principal de seguir os princípios do PON na geração de código alvo C++ foi garantir o ganho de desempenho que o PON preconiza ao evitar a avaliação de expressões causais desnecessárias, bem como evitar o alto acoplamento das entidades. No código gerado em C++ não foram evitadas a utilização de técnicas de passagens de parâmetros, ponteiros, alocação dinâmica de memória, classes etc. O objetivo da geração de código em C++, desta maneira, foi

fazer uso destas técnicas disponíveis no paradigma, pois facilitam a leitura do código alvo gerado. No entanto, por questões de desempenho o código gerado em C++ tende a exigir mais processamento quando comparado ao código em C, questões essas que serão apresentadas no próximo capítulo. Por sua vez, a implementação em C foi toda desenvolvida de forma a otimizar a velocidade de execução do código. Para que isso ocorresse, não foram utilizadas no código gerado as seguintes técnicas: passagens de parâmetros, ponteiros e alocação dinâmica de memória. De fato, isto constitui na otimização do tempo de processamento, no entanto, essas estratégias tendem a dificultar a leitura do código gerado. Conforme salientado na Subseção 3.2.4 o código gerado em C executa o processo de inferência do PON de forma sequencial executando todas suas ações pertinentes, uma vez, que um *Attribute* de uma *FBE* tem seu estado modificado. Essa técnica aumenta o tamanho do código gerado, mas antecipa algumas otimizações que seriam deixadas a cargo do compilador GCC. Isso é realizado a fim de se tentar chegar o mais próximo possível do código que seria gerado em Assembly.

Deste modo, conforme objetivos deste trabalho, este capítulo apresentou o resultado esperado com relação a criação de uma linguagem específica ao PON e ainda a evolução do estado da técnica do PON com a criação do seu compilador.

O próximo capítulo apresenta os estudos comparativos realizados com o código gerado pela linguagem e compilador PON. Foram criadas duas aplicações para serem utilizadas como base nos estudos comparativos, sendo elas a aplicação denominada Mira ao Alvo e a aplicação de Vendas. Deste modo, o estudo comparativo se guia pelos resultados, em termos de código, do compilador com cenários distintos de testes com o intuito de coletar amostras de resultados no tocante a tempo de processamento de cada aplicação.

4 ESTUDOS COMPARATIVOS

O objetivo deste capítulo é apresentar dois estudos comparativos práticos, em termos de eficiência de execução, com os resultados obtidos pela linguagem e compilador PON. Mais precisamente, a linguagem e compilador PON até o momento tem como resultado a geração de código intermediário para C, C++ e código *Framework* PON.

Isto dito, um intuito destes estudos comparativos é avaliar o resultado gerado para *Framework* PON com relação aos resultados gerados em código PON em C e em C++ no tocante a eficiência de execução (e.g. desempenho) a fim de validar se os resultados são satisfatórios. Outro intuito de tais estudos é comparar estes resultados de desempenho dos códigos PON para *Framework*, C++ e C para com resultados de código em Paradigma Orientado a Objetos (POO) - Paradigma Imperativo (PI).

O estudo será guiado pela codificação da aplicação Mira ao Alvo e da aplicação Sistema de Vendas. Estas aplicações são codificadas primeiramente em linguagem C++ POO/PI. Subsequentemente, essas aplicações são codificadas em linguagem PON com geração do código intermediário PON em C, C++ e *Framework*. Por fim, os programas gerados pelo compilador PON e em linguagem POO foram avaliados entre si.

Ainda, os tempos de execução em cada cenário de teste foram considerados apenas ao final da execução de todo programa (após todas as iterações). Para aferir os testes foram utilizados ambientes em Linux, Debian 7.1 (x86 e 64bit) em *hardware* contendo 4 GB RAM, Intel *Core* 2 Quad CPU Q6600 @ 2.40 GHz x 4. Todos os testes foram realizados em ambiente livre de preempção (Linux monousuário/modo de recuperação), os códigos foram compilados com GCC (linguagem C) e g++ (linguagem C++ e *Framework*PON C++) no modo de otimização máxima apresentado por cada compilador.

Ainda, um estudo comparativo adicional é realizado com o intuito de avaliar a facilidade de programação com a utilização da LingPON. Este estudo comparativo tem como um dos objetivos realizar a validação da complexidade de código (e.g. linhas de código) entre o código LingPON bem como seu equivalente em *Framework* e POO/PI. Vale ressaltar que tal estudo é guiado pelas mesmas aplicações desenvolvidas para os estudos comparativos anteriores. Ainda, são apresentados

relatos da utilização do compilador PON destacando suas vantagens e desvantagens de utilização.

Destaca-se que os códigos parciais utilizados durantes os estudos comparativos encontram-se no Apêndice E deste trabalho.

Isto considerado, a Seção 4.1 apresenta o estudo comparativo realizado com a construção da aplicação Mira ao Alvo em linguagem/compilador PON e linguagem POO. A seção 4.2 apresenta um estudo comparativo relacionado à aplicação de Vendas em liguagem/compilador PON e linguagem POO. Ainda, a Seção 4.3 apresenta um estudo comparativo relacionado a facilidade de programação ao utilizar a linguagem PON. Finalmente, a Seção 4.4 apresenta as considerações finais sobre os estudos comparativos realizados.

4.1 CASO DE ESTUDO – APLICAÇÃO MIRA AO ALVO

Esta seção apresenta um estudo comparativo quantitativo de uma implementação do sistema chamado Mira ao Alvo. Tal estudo comparativo tem por objetivo avaliar os resultados gerados pelo compilador PON no tocante a desempenho. Além da comparação entre os resultados gerados pelo compilador PON, será realizada uma comparação entre o resultado gerado pelo compilador PON com uma aplicação equivalente desenvolvida sob o viés do POO.

Deste modo, a Subseção 4.1.1 apresenta uma descrição do escopo da aplicação realizada para este estudo de caso. A Subseção 4.1.2 apresenta os resultados observados para execução das aplicações em C, C++ e *Framework*. A Subseção 4.1.3 apresenta os resultados da comparação do resultado do compilador PON versus a aplicação desenvolvida sob o viés do POO em C++. Finalmente, a Subseção 4.1.4 detalha as reflexões sobre o estudo comparativo realizado.

4.1.1 Escopo da Aplicação

A aplicação intitulada Mira ao Alvo consiste na implementação do tradicional jogo de mira ao alvo onde as entidades do tipo mira interagem com as entidades do tipo alvo. Neste ambiente, ambas as entidades são posicionadas a uma dada distância, sendo que a mira tenta atingir o alvo com o arremesso de um projétil

[BANASZEWSKI, 2009]. A Figura 28 apresenta o cenário que descreve os relacionamentos entre as entidades do tipo mira e entidades do tipo alvo para facilitar tal entendimento.

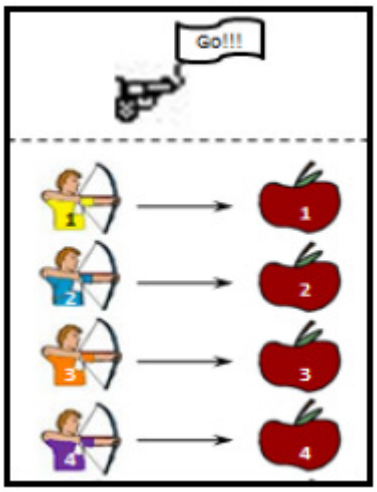


Figura 28 – Cenário do tradicional jogo Mira ao Alvo [BANASZEWSKI, 2009]

Conforme apresentado na Figura 28 as entidades do tipo mira, também denominados Arqueiros, e as entidades do tipo alvo, denominadas Maças, são identificados pelos números 1, 2, 3 e 4. Ainda, é possível visualizar uma entidade adicional denominada controladora, representada pelo símbolo de uma arma de fogo.

A presente aplicação apresenta algumas variações que tornam mais complexa a interação entre as miras e alvos do que no ambiente tradicional. Estas variações são relativas a quantidade de entidades mira e alvo, a definição de novos estados para estas entidades e mesmo a inserção de novas entidades ao ambiente. No âmbito deste trabalho, cada variação que altera a forma pela qual as miras e alvos interagem é chamada de cenário e cada entidade do cenário (e.g. miras e alvos) é chamada de personagem deste cenário.

De um modo geral, as entidades miras e as entidades alvos são representadas respectivamente por arqueiros e maçãs, sendo que há uma maçã para cada arqueiro. Em termos de implementação, cada par de arqueiro e maçã é representado por uma instância de entidade (i.e. um objeto em POO e um *FBE* em PON), os quais interagem de acordo com a validação de expressões causais pertinentes.

Estas expressões causais fazem menção aos atributos e estados destas instâncias. Neste contexto, cada arqueiro e cada maçã se relacionam entre si, sendo que um arqueiro somente pode flechar uma maçã na qual está relacionado. Ainda, os arqueiros também apresentam um atributo que permite saber se o mesmo já realizou o tiro (i.e. *hasFired*) para flechar a respectiva maçã. Em relação às maçãs, cada uma apresenta um atributo que explicita se a mesma já foi perfurada por uma flecha ou ainda não (i.e. *isCrossed*). Ainda, existe uma instância chamada Controladora (i.e. *Controller*) com o intuito de controlar o cenário de disparos. A controladora possui um atributo chamado 'atire' (i.e. *fire*) que é o ponto de partida para que todos os arqueiros, caso haja condição favorável, atinjam suas respectivas flechas.

Cada arqueiro somente pode interagir com a sua respectiva maçã após o atendimento de três premissas: (a) se a maça foi flechada, (b) se o arqueiro realizou um tiro, (c) se o controlador deu permissão para atirar. Se as três condições forem satisfeitas, o arqueiro está liberado para atingir a respectiva maçã com a projeção de sua flecha. Desta forma, percebe-se que para cada par de arqueiros e maçãs deve haver uma expressão causal para comparar os seus estados.

De um modo geral, os experimentos realizados sobre os cenários podem variar os estados dos personagens para que apenas uma porcentagem pré-definida de expressões causais sejam satisfeitas a fim de alterar o impacto de redundâncias nas instâncias imperativas e aumentar a quantidade de notificações nas instâncias do PON. Na maioria dos cenários, esta porcentagem cresce na escala de 10% até atingir a totalidade das expressões causais. Assim, um experimento pode ser seccionado em várias escalas, chamadas de fases de um experimento. Em cada fase de um experimento, o cenário é executado pela mesma quantidade de iterações para verificar as variações de comportamento entre as instâncias comparadas.

Em suma, esta aplicação oferece um escopo ideal para realizar os estudos comparativos entre os paradigmas em questão. Este permite demonstrar sucintamente como os paradigmas se comportam, permitindo discutir sobre as suas qualidades e deficiências[4].

[4] A definição da aplicação Mira ao Alvo foi baseada na descrição realizada na dissertação de mestrado de [BANASZEWSKI, 2009].

4.1.2 Resultados comparativos entre código específico PON em C, C++ e Framework

Primeiramente, comparar-se-á o desempenho entre as três versões de código intermediário gerados pelo compilador PON para criação da aplicação Mira ao Alvo. O intuito deste estudo comparativo é avaliar se o resultado gerado em código PON específico, tanto pela versão PON em C quanto a versão PON em C++, apresentaria os comportamentos determinados pelos conceitos do PON, além de comparar o desempenho (i.e. tempo de execução).

Para realizar o estudo comparativo entre as três versões de código intermediário, optou-se pela criação de dez cenários distintos. Cada cenário contou com a criação de cem arqueiros que interagem com a mesma quantidade de maçãs, sendo que cada arqueiro interage com uma respectiva maça. Ainda, cada arqueiro e cada maçã se relacionam com o controlador, o qual é responsável por indicar o status de fogo (*fire*) que indica que o arqueiro deve flechar a maçã.

Nestes experimentos, os arqueiros são representados pela classe *Archer* e as maçãs pela classe *Apple*, ainda o controlador pela classe *Controller*, ambas derivadas da classe *FBE*. Ainda, para cada instância de arqueiro, maçã e controlador foram criados uma *Rule* apropriada. O trecho de código a seguir apresenta a *Rule* criada.

Algoritmo 37: Código PON para a *Rule do* Mira ao Alvo

```
1    rule RlTurnOn1
2        condition
3            subcondition A1
4                premise PrIsCrossed apple.atIsCrossed == false and
5                premise PrHasFired archer.atHasFired == false and
6                premise PrFire controller.atFire == true
7            end_subcondition
8        end_condition
9        action
10           instigation inFire archer.mtFire1();
11       end_action
12   end_rule
```

Desta forma, para verificar o comportamento dos resultados gerados em C, C++ e *Framework* em várias situações, a quantidade de *Rules* aprovadas varia em cada fase do experimento na escala crescente de 10% pela variação da quantidade de *Apples* que são flechadas. Na medida em que a quantidade de *Rules* aprovadas

aumenta durante as fases do experimento, pode-se verificar o comportamento das versões em relação ao aumento das notificações, uma vez que as mudanças de estados se tornarão mais frequentes.

No primeiro experimento, ambas as versões foram comparadas sobre as mesmas condições por 10.000 iterações. Os resultados deste experimento estão ilustrados na Figura 29.

Exp. Verd.	PON C	PON C++	PON Framework
10%	10,787	28,848	46,400
20%	20,269	57,524	82,739
30%	32,242	88,218	124,505
40%	41,454	117,681	159,037
50%	54,132	144,574	202,963
60%	84,568	172,024	241,843
70%	97,816	200,493	280,103
80%	131,031	229,232	316,989
90%	167,817	260,474	358,955
100%	200,796	287,968	398,793

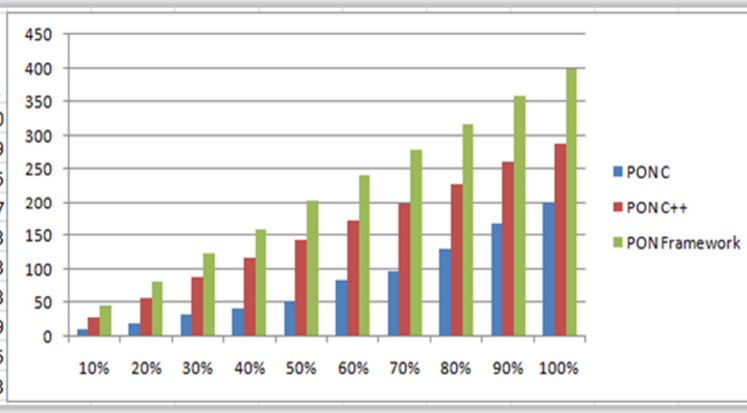


Figura 29 – Resultados para o primeiro experimento para aplicação Mira ao Alvo

Conforme explicita o gráfico de resultados, neste experimento, a versão gerada pelo compilador para código específico PON em C foi a que apresentou os melhores resultados. A versão para código específico PON em C++ teve um desempenho melhor quando comparado a versão para código em *Framework* PON C++, no entanto, seu desempenho foi inferior quando comparado a versão para código específico PON em C. Deste modo, é possível perceber uma maior eficiência dos códigos específicos PON em C e C++ quando comparado a código PON em *Framework*. Isto é devido ao fato dos códigos gerados em C e C++ serem específicos e não apresentaram componentes complexos como é o caso do código gerado para o *Framework*.

Conforme demonstrado no primeiro experimento, tanto o código específico PON gerado para linguagem C e para linguagem C++ obtiveram resultados melhores em termos de desempenho quando comparado ao código PON gerado para *Framework PON em C++*. Para confirmar este fato, outro experimento foi realizado com o intuito de aumentar a quantidade de *Rules* aprovadas seguindo os mesmos critérios do primeiro experimento. Neste novo experimento, os *Archers*

interagem com as *Apples* e *Controllers* por 100.000 iterações. Os resultados deste experimento estão expressos na Figura 30.

Exp. Verd.	PON C	PON C++	PON Framework
10%	92,650	283,101	408,483
20%	195,384	568,997	828,758
30%	312,497	855,995	1204,443
40%	403,485	1133,703	1584,553
50%	531,110	1426,517	2040,973
60%	835,082	1707,682	2478,301
70%	1144,622	2013,796	2842,067
80%	1313,646	2295,511	3227,591
90%	1670,688	2580,788	3663,963
100%	2001,401	2872,392	3938,623

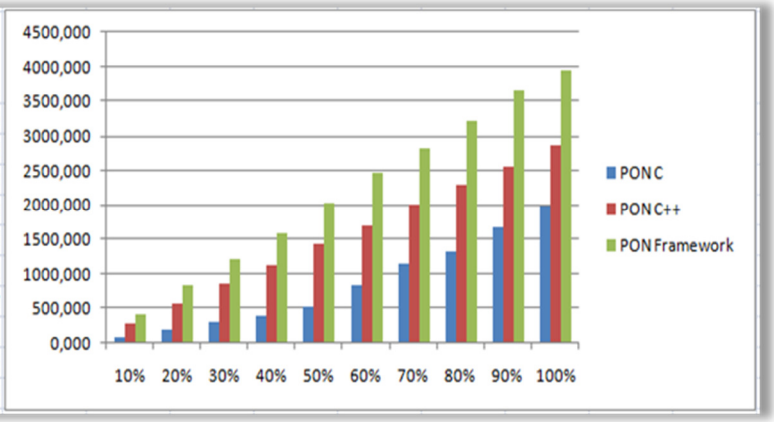


Figura 30 – Resultados para o segundo experimento para aplicação Mira ao Alvo

Conforme os resultados apresentados na Figura 30, assim como os resultados apresentados para o primeiro experimento, a versão de código específico PON em C foi a que obteve o melhor desempenho, seguido da versão de código específico PON em C++.

4.1.3 Resultados comparativos entre código específico PON em C e C++ versus PI/POO em C++

Esta Subseção tem por objetivo apresentar um estudo de caso a fim de tecer certa comparação do paradigma imperativo, mais precisamente de código POO em C++, com o resultado apresentado pelo código gerado para C e C++ pelo compilador PON. Os cenários de testes foram os mesmos executados na Subseção anterior e a figura a seguir apresenta os resultados obtidos para cem mil iterações.

Exp. Verd.	PON C	PON C++	PI C++
10%	92,650	283,101	510,352
20%	195,384	568,997	828,641
30%	312,497	855,995	1172,581
40%	403,485	1133,703	1476,716
50%	531,110	1426,517	1794,463
60%	835,082	1707,682	2028,070
70%	1144,622	2013,796	2389,739
80%	1313,646	2295,511	2619,019
90%	1670,688	2580,788	2910,018
100%	2001,401	2872,392	3210,018

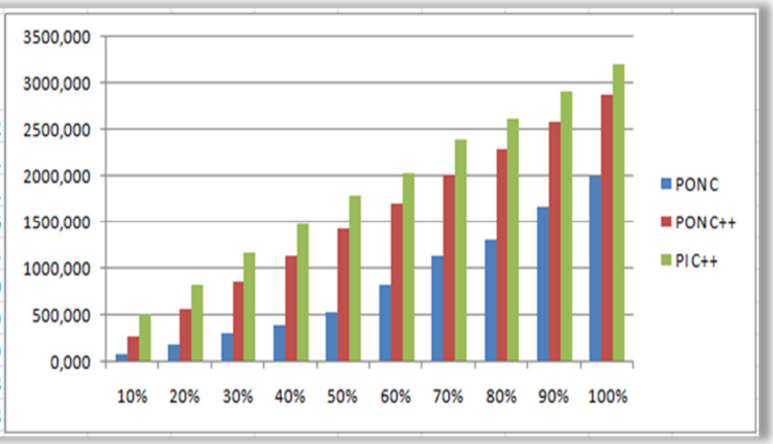


Figura 31 – Resultados em C/C++ versus PI para aplicação Mira ao Alvo

A escolha pelos cenários de cem mil iterações para avaliação entre o código gerado pelo compilador PON para C e C++ e sobre a versão do programa mira ao alvo escrito em POO se deu ao fato de contar com mais variações na avaliação de expressões causais. Como é possível ver no gráfico a versão gerada pelo compilador PON em C e C++ apresentaram resultados satisfatórios quando comparada com a versão em POO para este estudo de caso.

4.1.4 Comparativo do Código em Assembly Gerado

A fim de validar os resultados gerados em código intermediário específico PON para C e C++ versus o código gerado em PI/POO C++ para aplicação Mira ao Alvo, foram coletados o código gerado em *Assembly* para cada uma das abordagens. Este código em *Assembly* foi avaliado com a contagem de linhas de código, da compilação para *Assembly*, com o intuito de validar a complexidade do código gerado. A Figura 32 apresenta os resultados para tal comparativo.

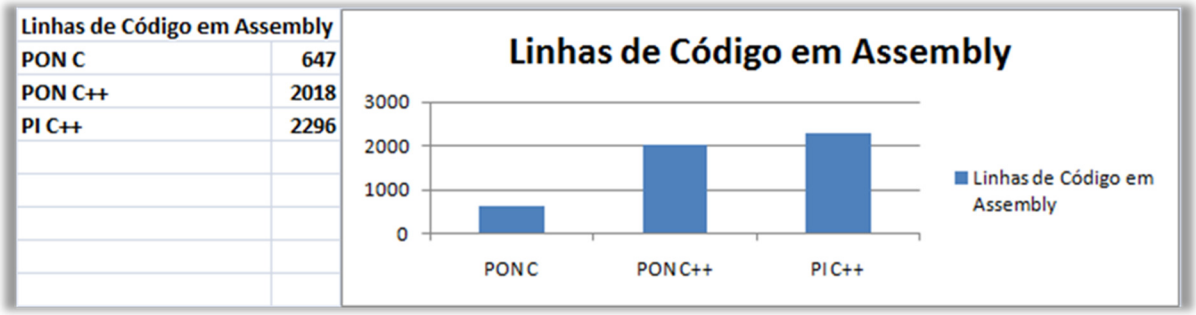


Figura 32 – Linhas de Código em *Assembly* para aplicação Mira ao Alvo

Como é possível observar na Figura 32, para este experimento realizado, o código em *Assembly* gerado para o código específico PON em C foi o que apresentou a menor contagem de linhas de código. Isto corrobora a validar o fato do código específico PON em C apresentar o melhor tempo de execução quando comparado ao código específico PON em C++ e ao código PI/POO C++. Já o código em *Assembly* gerado para o código específico PON em C++ apresentou uma contagem de linhas de código quase equivalente ao código gerado em PI/POO C++. Ainda assim, apresentou-se com uma quantidade menor de linhas de código o que corrobora para justificar o ganho de tempo de processamento do PON C++ quando comparado ao PI/POO C++.

4.1.5 Comparativo da Utilização de Memória

Para avaliação do consumo de memória nos cenários de testes, para cada teste, foi aferida a utilização de memória ao longo do teste e o maior consumo de memória foi capturado durante a execução e foi associado ao teste executado. Salienta-se que para coleta de utilização da memória foram utilizadas funções em C e em C++ para tal. Tais funções armazenam o consumo de memória durante a execução da aplicação e para todas as aplicações as mesmas estratégias foram utilizadas. Para avaliação, os cenários com 100 mil iterações foram os que apresentaram maior relevância de análise, devido ao consumo de memória e a diferença apresentada. A Figura a seguir ilustra o cenário de memória utilizado.

Exp. Verd.	PON C	PON C++	PON Framework	PI C++
10%	2012	3420	3660	3580
20%	2012	3420	3660	3580
30%	2012	3420	3660	3580
40%	2012	3420	3660	3580
50%	2012	3420	3660	3580
60%	2012	3420	3660	3580
70%	2012	3420	3660	3580
80%	2012	3420	3660	3580
90%	2012	3420	3660	3580
100%	2012	3420	3660	3580

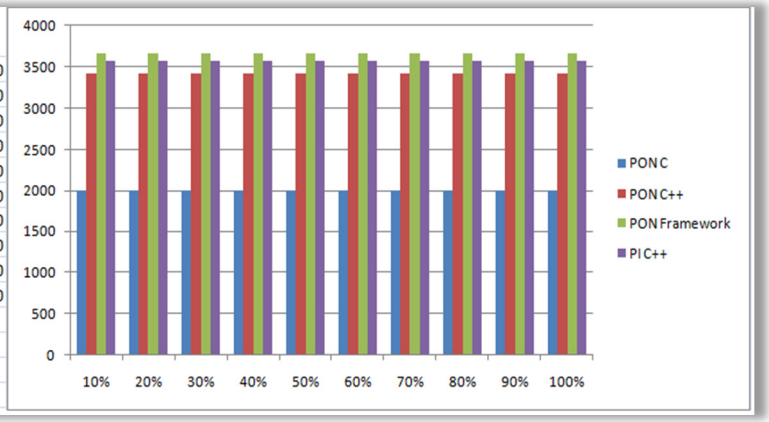


Figura 33 – Utilização de memória para aplicação Mira ao Alvo

Como é possível observar na Figura acima, a primeira coluna apresenta a quantidade de *Rules* aprovadas, a segunda coluna apresenta o uso de memória em kbytes para o programa compilado em código específico PON C, a terceira coluna o total de memória utilizada pelo programa compilado em código específico PON C++, a quarta coluna apresenta o consumo de memória utilizada pelo código PON em *Framework* C++ e, finalmente, a quinta coluna apresenta o total de memória utilizada pela aplicação desenvolvida sob o PI/POO em C++.

Os resultados demonstram que o programa compilado em código específico PON C foi o que atingiu o menor consumo de memória para executar os testes associados. Já o programa compilado em código específico PON C++ apresentou um resultado minimamente inferior quando comparado ao uso de memória pelo código em *Framework* PON *C++*. Ainda, é possível observar que a versão código específico PON C++ também apresentou um resultado minimamente inferior quando comparado ao uso de memória pelo código em PI/POO C++.

Deste modo, percebe-se que quando mais próximo da linguagem de máquina a linguagem PON é compilado, menor é o seu consumo de memória, pois estruturas complexas (e.g. classes) não são criadas, apenas o fluxo de execução, ocasionando assim, uma diminuição do consumo de memória.

4.1.6 Reflexões

Como foi possível visualizar no estudo comparativo realizado anteriormente a versão do código específico PON em C foi a que obteve os melhores resultados em termos de desempenho. Isto se deve ao fato dessa versão ser menos

dependentes de conceitos de orientação a objetos (i.e. classes e objetos e alocação dinâmica) como é o caso da versão em C++ e *Framework*.

A versão do código específico PON em C apresenta os conceitos de estruturas (i.e. *struct*) e funções que são chamadas conforme o processo de inferência do PON requer. Ainda, é possível observar que quanto mais o código alvo se aproxima do código de máquina (e.g. código gerado em C) mais baixo fica o consumo de processamento, o que aumenta a expectativa na geração de código alvo de máquina. Salienta-se que o código específico PON gerado em C, com a utilização do compilador gcc, geram código *Assembly* otimizado, sendo possível realizar essa opção no momento da compilação do código. Ainda, a versão em C contempla para cada elemento do tipo *FBE* a criação de um Header (.h) e um arquivo (.c) que implementa os procedimentos e as funções pertinentes àquele módulo. No entanto, conforme salientado na Subseção 3.2.4 independente da aplicação, alguns arquivos são genéricos, os quais representam as *Rules*, as *SubConditions* e *Premisses*. Deste modo, por mais complexo que seja a aplicação desenvolvida em PON o código gerado em PON C aumentará somente no número de elementos que representam as *FBE* o que concentra a chamada de funções pelo processo de inferência somente em determinados arquivos evitando assim o *overhead* em chamadas de funções[5], o que de fato, acrescenta um custo maior de processamento [LEE et al., 2000; TANG et al., 1999].

Os resultados apresentados para código específico PON C++ também se mostraram satisfatórios. É importante salientar que o código do compilador para código específico PON em C++ apresenta os conceitos de classes que representam os conceitos PON, relativos a *FBE*, *Rule* e *Premise*, bem como faz uso de alocação dinâmica de objetos. Desta forma é possível constatar que o código PON em C++ apresenta mais arquivos de código fonte que o código PON em C. Sendo assim, o processo de inferência do PON aplicado ao código PON em C++ realiza chamadas de funções em classes espalhadas em diferentes arquivos. Isto pode ocasionar o *overhead* em chamadas de métodos. Outra observação importante é relacionada ao fato do código em PON C++ fazer uso de alocação dinâmica de objetos. Isto

[5]***Overhead* em chamadas de métodos**: Um programa bem concebido é dividido em métodos curtos que realizam de forma coesa suas tarefas. No entanto, cada chamada a estes métodos requer a configuração da pilha, copiando seus parâmetros e endereços de retorno. Isso representa uma sobrecarga de CPU em comparação a um programa que faz tudo em uma única função monolítica.

contribui para aumentar o gasto de tempo de processamento. Deste modo, isto justifica o fato da versão em PON C++ apresentar um resultado menos satisfatório quando comparada a versão PON em C. Por outro lado, a versão em PON C++ obteve resultados de desempenho satisfatórios quando comparado ao resultado em *Framework.*

Neste experimento, como esperado, o *Framework* PON em C++ em termos de desempenho obteve resultados insatisfatórios quando comparado a versão de código específico PON em C e em C++. Isto é devido ao fato do *Framework* PON C++, conforme relatado em trabalhos anteriores, apresentar algumas desvantagens de implementação, como a sobrecarga de usar uma estrutura de dados computacionalmente cara, devido ao fato de ser genérica, e ser desenvolvido sob uma linguagem intermediária, no caso o C++. Isto motivou a criação do compilador PON que teve como objetivo otimizar as relações entre os objetos participantes do mecanismo de notificações. Estas otimizações incluem a eliminação de estruturas de dados para armazenar os objetos notificados (i.e. *Premises* e *Conditions*), uma vez que os objetos notificantes e notificados são conectados em tempo de compilação (i.e código gerado para C++). Além disso, o *Framework* se apresenta como uma estrutura genérica. Como exemplo, embora no desenvolvimento da aplicação Mira ao Alvo os conceitos PON como *Rules* dependentes e *Attributes* impertinentes não tenham sido utilizados, internamente, o *Framework* realiza validações nestes itens, pois, independente do uso dessas propriedades o *Framework* necessita verificar se estas propriedades foram adicionadas ou não no âmbito da aplicação. Isto não é observado no código gerado pelo compilador com relação a C e C++ e isto, de fato, fez com que os resultados das versões em C e C++ fossem satisfatórios.

Finalmente, a versão em PI/POO C++ se mostrou insatisfatória quando comparada a versão em PON C. Isto é devido, como salientado anteriormente, a versão em POO ser desenvolvida com as vantagens da orientação a objetos como a representação do conhecimento através de classes e a utilização de alocação dinâmica de objetos. Isto de fato contribui no aumento do tempo de processamento conforme discutido na versão em PON C++. Por outro lado, a versão em PI/POO C++ se mostrou equivalente a versão em PON C++ tendo seu resultado razoavelmente pior. Isto se deu ao fato das versões serem, de certa forma, equivalentes com relação à utilização de conceitos de OO. No entanto, a versão em

PON C++ pelo fato de ser concebida com a utilização dos preceitos do PON apresentou certas vantagens, isto relacionado ao fato de não gastar tempo de processamento com a validação de expressões causais de forma desnecessária.

4.2 CASO DE ESTUDO – APLICAÇÃO SISTEMA DE VENDAS

Esta seção apresenta um estudo comparativo quantitativo de uma implementação de um sistema de pedido de vendas. Tal estudo comparativo, assim como realizado na Seção 4.2, tem por objetivo avaliar os resultados gerados pelo compilador PON no tocante a desempenho. Além da comparação entre os três diferentes resultados gerados pelo compilador PON, será realizada uma comparação entre o resultado gerado pelo compilador PON para código específico PON em C e C++ com uma aplicação de pedidos de vendas desenvolvida sob o viés do POO.

Deste modo, a Subseção 4.2.1 apresenta uma descrição do escopo da aplicação realizada para este estudo de caso. A Subseção 4.2.2 apresenta os resultados observados para execução das aplicações com código específico PON em C, código específico PON em C++ e código *Framework PON em C++*. A Subseção 4.2.3 apresenta os resultados da comparação do resultado do compilador PON para C e C++ versus a aplicação desenvolvida sob o viés do POO. Finalmente, a Subseção 4.2.4 detalha as reflexões sobre o estudo comparativo realizado.

4.2.1 Escopo da aplicação

De modo geral, este caso de estudo consiste na implementação de um sistema de vendas usual. Essa aplicação consiste em uma tradicional aplicação *CRUD* (acrônimo inglês de *Create*, *Read*, *Update* e *Delete*). Em suma, o escopo dessa aplicação é composto pelo diagrama de casos de uso ilustrado na Figura 34 [FERREIRA et al., 2013; RONSZCKA, 2012].

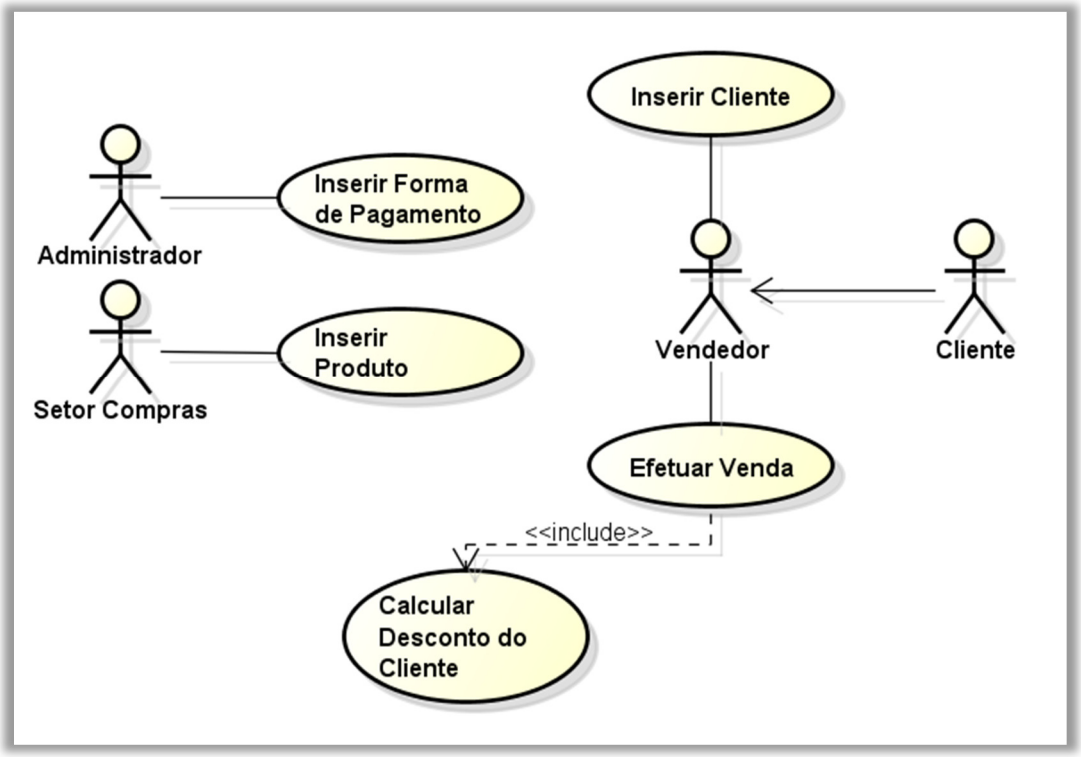


Figura 34 – Casos de uso para o Sistema de Vendas

Conforme ilustra a Figura 34, o ator Administrador é responsável por realizar o cadastro de formas de pagamentos. O ator Setor de Compras, por usa vez, é responsável por cadastrar as informações dos produtos. Ainda, o ator Cliente, solicita uma venda a um respectivo ator Vendedor. Este, por sua vez, cadastra o cliente e efetua a venda propriamente dita.

Para elucidar o comportamento e as responsabilidades de cada caso de uso da Figura 34, a Figura 35 apresenta um diagrama de atividades da execução de um pedido de venda.

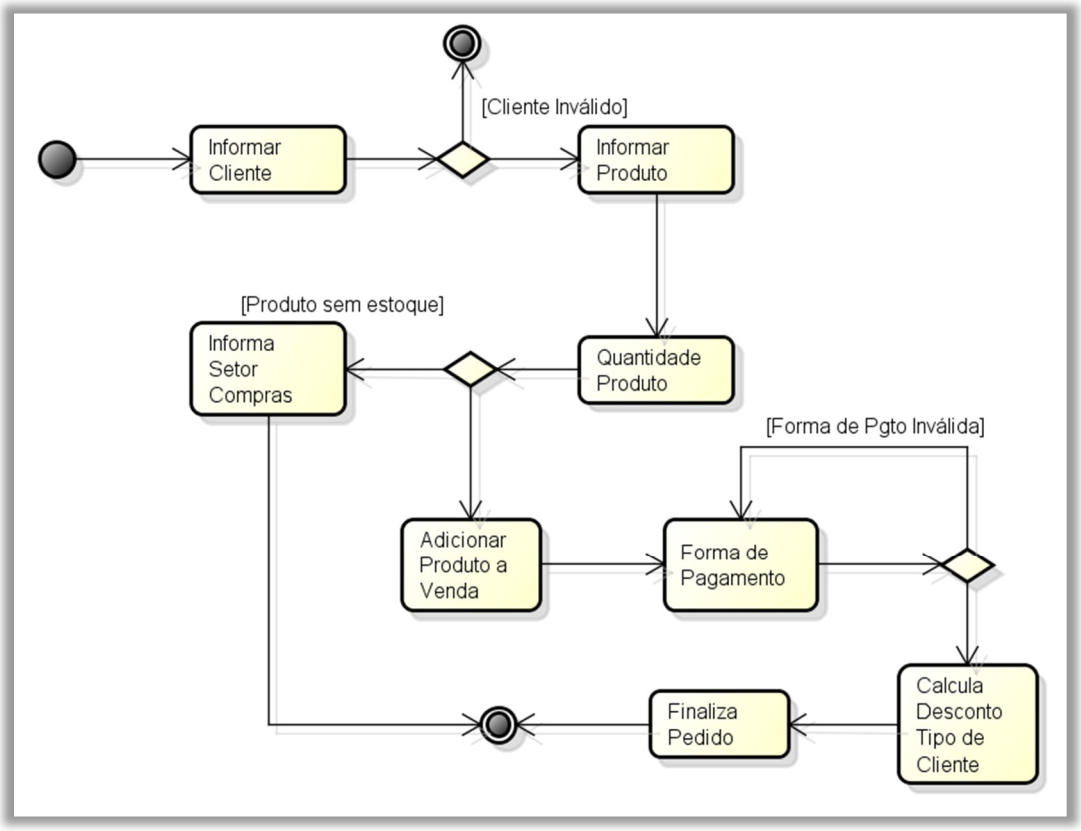


Figura 35 – Diagrama de atividades para o caso de uso efetuar venda

Conforme ilustrado na Figura 35, inicialmente o cliente (denotado pelo ator cliente) solicita a compra para um respectivo vendedor (denotado pelo ator vendedor). Assim, o vendedor informará o respectivo cliente que realizará o pedido. Uma vez escolhido e aprovado a venda para determinado cliente, deve ser informado o produto que irá compor o pedido. O sistema possui validações quanto à existência de produtos e clientes. Ademais, verifica-se o estoque disponível de tais produtos.

Após todo o ciclo de informe de produtos, a venda poderá ser finalizada após a inserção da forma de pagamento. Na implementação desse sistema, existem apenas duas formas de pagamento possíveis, a Vista ou a Prazo. O cliente, em seu cadastro, possui uma informação sobre seu limite de crédito. Caso a forma de pagamento escolhida tenha sido a Prazo, o sistema verifica se o cliente tem permissão para efetuar a compra, confrontando o valor total do pedido com seu limite de crédito.

Ademais, no cadastro do cliente há uma informação que lhe concede um tipo de classificação. Utiliza-se tal classificação para a concessão de descontos especiais durante a finalização da venda. Para tanto, existe um total de 20 tipos de

classificação de clientes que dispõem de descontos que variam de uma faixa de 0 as 95%.

De modo a facilitar o entendimento do escopo desse sistema, o diagrama de classes ilustrado na Figura 36 externa a essência da implementação do sistema de Vendas por meio de suas principais classes.

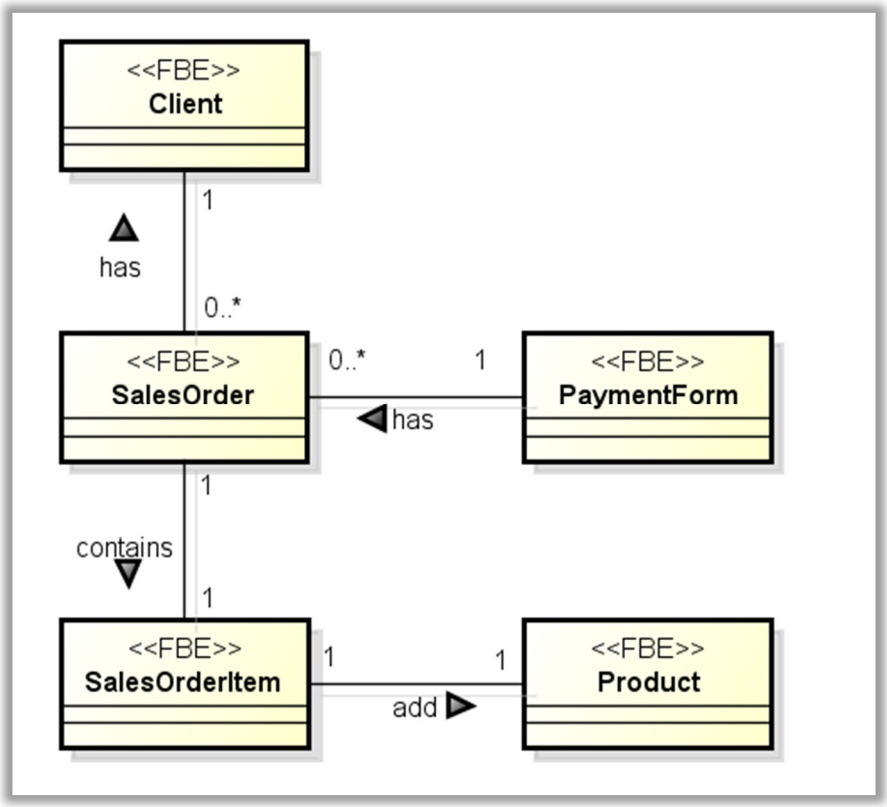
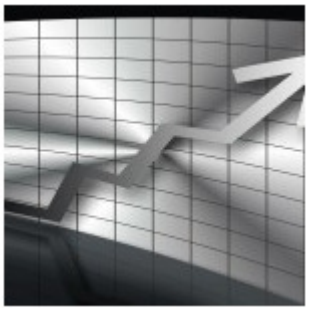


Figura 36 – Diagrama de classes para o Sistema de Vendas

Conforme ilustrado na Figura 36, a classe *SalesOrder* possui a responsabilidade de armazenar os dados de um pedido. Basicamente, um pedido é composto por um cliente, uma forma de pagamento e um item de pedido. No item de pedido é armazenado o produto e sua respectiva quantidade. As classes definidas neste diagrama de classes possuem o esteriótipo de *FBE*, pois no código PON elas são configuradas como tal.

A título de exemplificação, a Figura 37 demonstra a composição da *Rule* responsável por finalizar uma venda. Nela estão relacionadas às *Premises* que deverão ser satisfeitas para que a finalização da venda ocorra. Assim, a primeira *Premise* verificaria se a forma de pagamento selecionada foi a prazo. Neste caso é necessário validar o limite de crédito disponível para o cliente, o qual faria parte da

segunda *Premise* da Rule em questão. A terceira e última *Premise* validaria o tipo de desconto concedido para o cliente em questão (dentro os 20 possíveis tipos de descontos).



Se:

Forma Pagamento	**=**	À Prazo	**e**
Limite Crédito Cliente	**>=**	Total da Venda	**e**
Tipo Desconto Cliente	**=**	1	

Entao:

Conceder Desconto do tipo 1 (um)
Finalizar Venda

Figura 37 – *Rule* responsável por finalizar a venda

Conforme descrito no parágrafo anterior, existem 20 tipos de descontos que poderão ser concedidos aos clientes. Neste caso, para cada tipo de desconto será criado uma *Rule* correspondente, conforme apresentada na Figura 37. Desta forma, após a satisfação das *Premises* supracitadas, a *Rule* em questão concederia o desconto do pedido de vendas para o cliente e por fim finalizaria sua venda.

4.2.2 Desenvolvimento da aplicação

Esta Subseção tem por objetivo apresentar certos detalhes da codificação da aplicação de Vendas em LingPON. Salienta-se que o código completo da aplicação está descrito no Apêndice E. A título de exemplificação, o Algoritmo 38 demonstra a composição da *Rule* para verificar se o produto está aprovado para venda. Nesta *Rule* foram criadas duas *Premises*. A primeira *Premise* foi configurada para validar se o produto possui estoque suficiente. Já a segunda *Premise* é utilizada para verificar o status de inclusão do produto no pedido. Essa *Rule* quando aprovada realiza a chamada de três métodos distintos, que têm a responsabilidade de calcular o valor do item e informar a *Rule* de fechamento do pedido que o item já foi adicionado. Para compor o valor do item do pedido é levado em consideração o desconto concedido pelo tipo do pedido. O qual é configurado pelo Algoritmo 40 que será descrito a seguir. Ainda, é importante ressaltar que os algoritmos descritos nessa Subseção foram criados utilizando a linguagem PON.

Algoritmo 38: *Rule* para aprovação da inclusão de um item no Pedido

```
1    rule rlApproveProductItem
2    condition
3          subCondition a2
4                premise imp prApproveQuantity product.atCurrentStock >=
5    salesOrderItem.atQuantity and
6                premise                                prApproveProductItem
7    salesOrderItem.atAddSalesOrdermItem == true
8          end_subCondition
9    end_condition
10   action
11         instigation                                  inCalculateValueItem
12   salesOrderItem.mtCalculateValueItem();
13         instigation                                   inCalculateQuantity
14   salesOrderItem.mtCalculateQuantity();
15         instigation                              inMtApproveSalesOrderItem
16   salesOrderItem.mtApproveSalesOrderItem();
17   end_action
18   end_rule
```

O Algoritmo 39 demonstra a composição da *Rule* responsável por finalizar uma venda. Nela estão relacionadas as *Premises* que deverão ser satisfeitas para que a finalização da venda ocorra. Assim, a primeira *Premise* verifica a condição de fechamento do pedido. A segunda *Premise* valida se o limite de crédito do cliente é suficiente para fechar a venda. Essa *Rule* quando aprovada realiza a chamada para decrementar a quantidade no estoque do produto além de informar ao cliente que o pedido foi finalizado. Outra informação importante sobre essa *Rule* é o fato dela ser dependente da *Rule* que valida se o produto está aprovado para venda. Esta é uma propriedade que evita validações desnecessárias. Como exemplo, caso o limite de crédito do cliente seja alterado, isso não irá influenciar nessa *Rule* em questão, pois somente quando a *Rule* da qual ela é dependente for ativa é que ela será avaliada.

Algoritmo 39: *Rule* para execução de uma venda

```
1    rule rlExecuteSalesOrder depends rlApproveProductItem
2          condition
3                subCondition a4
4                      premise                          prSalesOrderClosed
5    salesOrder.atCloseSalesOrder == true and
6                      premise prCreditLimitOk client.atCreditLimit >=
7    salesOrder.atTotalSalesOrder
8                end_subCondition
9          end_condition
10         action
11               instigation                                  inLowerStock
12   salesOrderItem.mtLowerStock();
13               instigation inSaleApproved client.mtSaleApproved();
14         end_action
15   end_rule
```

Pertinente recordar que para compor o valor total do pedido o sistema conta com 20 tipos de descontos que poderão ser concebidos aos clientes. Neste caso, para cada tipo de desconto será criado uma *Rule* correspondente, conforme apresentada no Algoritmo 40. Sendo assim, cada *Rule* criada para validar o tipo de desconto conta com um *Premise* que avalia o tipo de desconto configurado para o cliente. Quando a *Rule* é aprovada o cliente recebe a percentagem de desconto que deverá ser utilizada para compor o total do pedido.

Algoritmo 40: *Rule* para tipo de desconto

```
1    rule rlDiscountType1
2    condition
3          subCondition a6
4                premise prTypeDiscount1 client.atTypeDiscount == 1
5          end_subCondition
6    end_condition
7    action
8          instigation inTypeDiscount1 client.mtTypeDiscount1();
9    end_action
10   end_Rule
```

4.2.3 Resultados comparativos entre código específico PON em C, C++ e *Framework*

De maneira a analisar a eficiência do código gerado pelo compilador dois experimentos foram realizados no sistema em questão. O primeiro e o segundo experimento são representados pelo seguinte cenário, sendo tais experimentos distintos pela configuração do tipo de desconto concedido ao Cliente, no caso 1 e 20 respectivamente.

- Número de clientes: 1
- Número de produtos: 1
- Quantidade no estoque: 10.000
- Número de pedidos: 10.000
- Tipo de desconto do cliente: 1 e 20.

A Figura a seguir apresenta os resultados obtidos após a execução dos cenários de testes. Neste experimento, o cenário de teste descrito acima foi executado por duas vezes. Em cada execução, o tipo de desconto do cliente foi configurado com o valor um e vinte respectivamente. Salienta-se que a escolha por

estes valores se dá pela diferença da quantidade de validações de expressões causais necessárias para avaliar o tipo de desconto do cliente.

Tipo de Desconto	PON C	PON C++	PON Framework
1	1059,440	1107,002	1181,998
20	1059,499	1107,736	1188,915

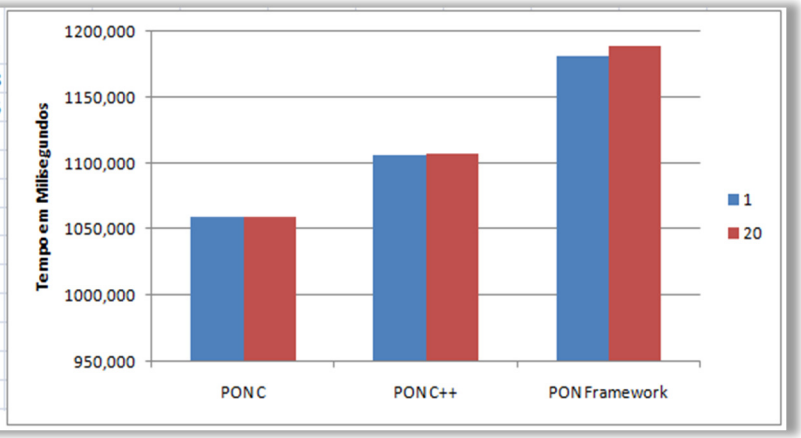


Figura 38 – Resultados para a execução do Sistema de Vendas

Conforme explicita o gráfico de resultados, neste experimento, a versão gerada pelo compilador PON em C foi a que apresentou os melhores resultados. Na sequência, a versão em PON C++ obteve um desempenho melhor quando comparada a versão PON *Framework*. Em contrapartida, a versão em PON C++, novamente, apresentou o desempenho insatisfatório quando comparado a versão em PON C, este mesmo comportamento já foi considerado pelos experimentos realizados para a aplicação Mira ao Alvo.

Outra observação importante a ser destacada neste gráfico se dá pela variação de tempo mínima em cada versão quando o tipo de desconto do cliente é modificado, ou seja, o tipo de desconto do cliente varia de 1 para 20 e os resultados para o cliente configurado com o valor 1 com relação aos resultados para o cliente configurado com o valor 20 se apresentam com o tempo de processamento semelhantes.

Neste experimento, é possível observar que as três versões se mantiveram com os valores constantes demonstrando que os resultados seguem os conceitos PON no que diz respeito à ausência de avaliação de expressões causais desnecessárias conforme explicado na subseção anterior.

4.2.4 Resultados comparativos entre código específico PON em C e C++ versus PI/POO em C++

Conforme realizado nos experimentos da aplicação do Mira ao Alvo, mais um experimento foi realizado neste estudo de caso. O propósito deste experimento foi avaliar a aplicação de vendas desenvolvida sob o viés do PI/POO C++ com o resultado apresentado pelo código gerado para PONC e C++. Este experimento adicional se faz necessário para avaliar o tempo de desempenho entre a versão gerada pelo compilador PON comparada à aplicação sobre o viés do PI/POO C++ no tocante a quantidade de avaliação de expressões causais. Como descrito na Subseção 4.3.2 a aplicação de vendas apresenta vinte tipos de clientes. Para validar esse comportamento no código desenvolvido na linguagem PON, uma regra específica foi criada e detalhada. O código abaixo apresenta o código em PI/POO C++ criada para simular esse comportamento.

Algoritmo 41: Código em C++ para definir tipo de desconto para um cliente

```
1    double SalesOrder::getCalculoDiscount() {
2          double discount = 0.0;
3
4          if (this->getCustomer()->getCustomerType() == 1) {
5                discount = 5;
6                return discount;
7          }
8
9
10         if (this->getCustomer()->getCustomerType() == 2) {
11               discount = 10;
12               return discount;
13         }
14
15
16         if (this->getCustomer()->getCustomerType() == 3) {
17               discount = 15;
18               return discount;
19         }
20   ...
```

Como é possível observar no código do POO para cada tipo de desconto do cliente uma avaliação causal é criada. Conforme o tipo de desconto do cliente aumenta é possível perceber que o número de avaliações causais crescerá. Pertinente ressaltar que o código acima poderia ser substituído por um código equivalente usando a cláusula *switch* ou mesmo evitar a redundância de avaliações causais, mas o objetivo deste experimento foi avaliar o descaso na concepção de

código em POO com o máximo de otimização possível. Deste modo este experimento teve por objetivo coletar a variação de desempenho na execução dos cenários de testes. Sendo assim, o gráfico a seguir apresenta os resultados coletados para este experimento.

Tipo de Desconto	PON C	PON C++	PI C++
1	1059,440	1107,002	1132,034
20	1059,499	1107,736	1206,555

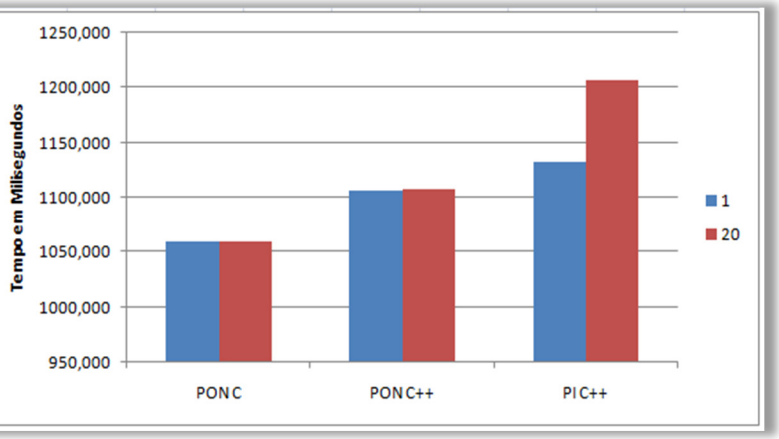


Figura 39 – Resultados para a execução do Sistema de Vendas em C++ vs PI/POO C++

Como é possível observar no gráfico acima, a versão em PON C e C++ geradas pelo compilador não apresentou variações no tempo de desempenho quando o tipo de cliente é modificado. Já na versão em PI/POO C++ o mesmo não acontece e é possível perceber de forma clara o aumento no tempo de processamento. Os resultados apresentados neste gráfico são para os cenários onde o tipo de desconto do cliente é igual a um e igual a vinte. A quantidade de expressões causais avaliadas no cenário do cliente de tipo um é muito inferior a quantidade de expressões causais avaliadas quando o tipo de cliente é igual a vinte. Deste modo, conforme esses números de avaliações aumentam o tempo de processamento também cresce para a versão em PI/POO C++. Sendo assim, para este estudo de caso é possível perceber que a aplicação desenvolvida sob o viés do PON apresenta resultados satisfatórios, tem termos de desempenho de processamento, quando comparada a uma aplicação equivalente desenvolvida em POO quando a quantidade de expressões causais avaliadas é ampla.

4.2.5 Comparativo do Código em Assembly Gerado

Como realizado no experimento anterior, da aplicação Mira ao Alvo, uma nova contagem de linhas de código em *Assembly* foi realizada, agora para a aplicação de Vendas. Este experimento visa validar os resultados gerados para os códigos em PON C e C++ e realizar um comparativo com a contagem de linhas de código em *Assembly* para o código gerado em POO/PI C++. Deste modo, o código em *Assembly* para cada uma das abordagens foi avaliado com a contagem de linhas de código com o intuito de validar a complexidade do código gerado. A Figura 40 apresenta os resultados para tal comparativo.

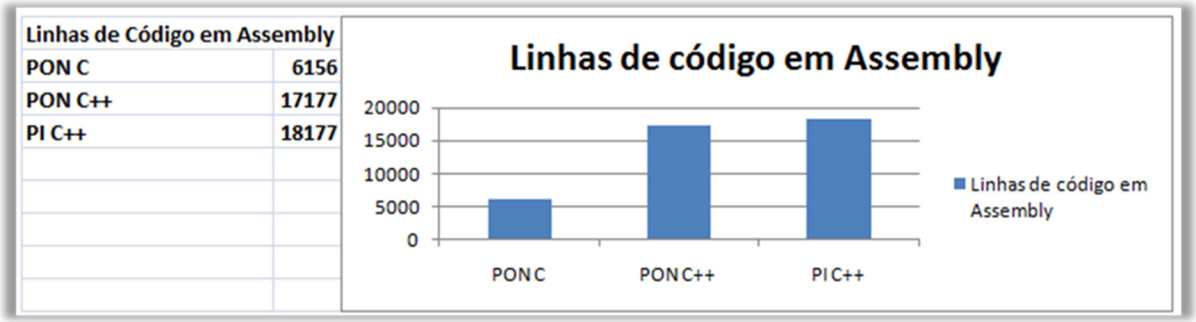


Figura 40 – Linhas de Código em *Assembly* para a aplicação de Vendas

Como é possível observar na Figura 40 o código em *Assembly* gerado para o código em PON C foi o que apresentou a menor contagem de linhas de código. Isto valida o fato do código em PON C apresentar o melhor tempo de execução quando comparado ao código em PON C++ e PI/POO C++. Já o código em *Assembly* gerado para PON C++ apresentou uma contagem de linhas de código equivalente ao código gerado em PI/POO C++.

4.2.6 Comparativo da Utilização de Memória

Assim como foi realizado o comparativo de utilização de memória na aplicação Mira ao Alvo, este mesmo comparativo foi realizado na aplicação de vendas. A avaliação foi realizada ao longo da execução dos casos de teste e o maior consumo de memória foi capturado durante a execução. A figura a seguir ilustra os resultados obtidos para consumo de memória coletado durante os testes realizados.

Tipo de Desconto	PON C	PON C++	PON Framework	PI C++
1	3024	5040	5320	5380
20	3024	5040	5320	5380

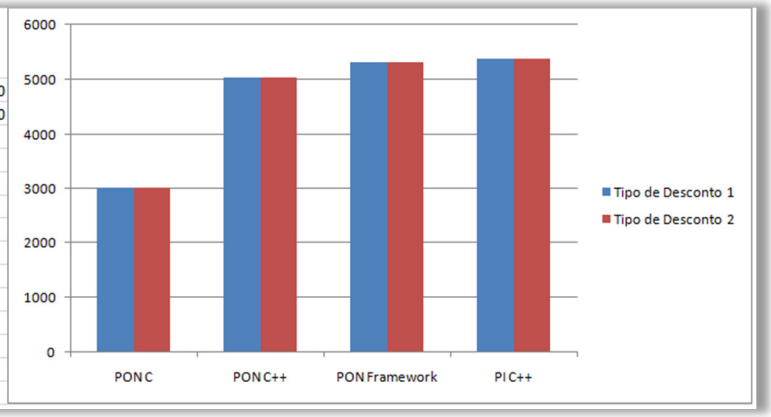


Figura 41 – Utilização de memória para aplicação de Vendas

Como é possível observar na Figura acima, a primeira coluna apresenta o tipo de desconto concedido ao cliente, a segunda coluna apresenta o uso de memória em kbytes para o programa compilado em PON C, a terceira coluna o total de memória utilizada pelo programa compilado em PON C++, a quarta coluna apresenta o consumo de memória utilizada pelo PON *Framework* e, finalmente, a quinta coluna apresenta o total de memória utilizada pela aplicação desenvolvida sob o PI/POO C++.

Os resultados demonstram, assim como para a aplicação Mira ao Alvo, que o programa compilado em C foi o que atingiu o menor consumo de memória para executar os testes associados. Já o programa compilado em C++ apresentou um resultado minimamente inferior quando comparado ao uso de memória pelo *Framework*. Ainda, é possível observar que a versão PON C++, também, apresentou um resultado minimamente inferior quando comparado ao uso de memória em PI/POO C++.

Deste modo, percebe-se, novamente, que quanto mais próximo da linguagem de máquina a linguagem PON é compilada, menor é o seu consumo de memória, pois estruturas complexas (e.g. classes) não são criadas, apenas o fluxo de execução, ocasionando assim, uma diminuição do consumo de memória.

4.2.7 Reflexões

Neste experimento, novamente os resultados para a versão PON C foram os mais satisfatórios em termos de desempenho. Na sequência vieram os resultados da

versão em PON C++ e, consequentemente, da versão em *Framework*. Este novo experimento confirmou os resultados relatados na Seção 4.2. Salienta-se que as mesmas considerações aplicadas na Subseção 4.1.6 podem ser aplicadas a este estudo comparativo.

Na comparação da versão C e C++ do compilador PON com a versão em PI/POO C++ foi possível observar as diferenças de comportamento entre os resultados gerados pelo compilador PON com o PI/POO C++, por meio da avaliação e expressões causais. A versão em PI/POO C++ demonstrou um aumento do tempo de execução no segundo cenário de teste. Isto é devido ao aumento das avaliações de expressões causais quando o tipo de cliente foi configurado com o valor vinte. Por sua vez, a versão C e C++ do compilador PON, apresentaram o mesmo tempo de execução em ambos os cenários. Isto demonstra que as versões do compilador PON não desperdiçam tempo de execução para avaliar expressões causais desnecessariamente.

4.3 CASO DE ESTUDO – FACILIDADE DE PROGRAMAÇÃO

Ao se inspirar em conceitos do paradigma declarativo, uma qualidade intrínseca aos conceitos do PON seria a oferta de facilidades na programação [BANASZEWSKI, 2009]. A programação em PON se tornaria mais fácil principalmente porque ela é intuitiva à forma cognitiva humana. Além de permitir a representação do conhecimento na forma natural ao ser humano, o PON também visa alcançar, até certo ponto, esforços menores na programação em termos de escrita de código [BANASZEWSKI, 2009].

Como exemplo deste compromisso, a criação dos objetos colaboradores e a conexão dos mesmos são realizadas de forma transparente ao programador, bastando que este apenas se concentre na concepção da *Rule* em si conforme exemplifica o código abaixo que traz a criação de objetos PON por meio da linguagem PON.

Algoritmo 42: Criação de objetos PON por meio da linguagem PON

```
1   rule RlTurnOn1
2       condition
3           subcondition A1
4               premise PrIsCrossed apple.atIsCrossed == false and
5               premise PrHasFired archer.atHasFired == false and
6               premise PrFire controller.atFire == true
7           end_subcondition
8           or
9           subcondition A2
10              premise PrIsCrossed archer.atCount == 0
11          end_subcondition
12      end_condition
13      action
14          instigation inFire archer.mtFire1();
15      end_action
16  end_rule
```

Neste âmbito, com o objetivo de melhorar a facilidade de programação em PON foi desenvolvida a linguagem PON que permite ao programador compor código diretamente sem a necessidade de uso do *Framework* PON que fora utilizado até o momento. Com a linguagem PON o desenvolvedor necessita somente estudar e entender os conceitos de tal linguagem e não precisa compreender uma série de componentes que eram necessários para compor o *software* com a utilização do *Framework*. Como exemplo disso, o Algoritmo 43 apresenta o trecho de código em *Framework* o qual contém o código necessário para criação de uma *FBE*.

Algoritmo 43: *FBE* escrita com código *Framework*

```
1   #ifndef Archer_H_
2   #define Archer_H_
3   #include "framework/utils/SingleInclude.h"
4   class Archer : public FBE {
5   public:
6               Archer();
7               ~Archer();
8               //Atributes
9               Boolean * atHasFired;
10              Integer * atCount;
11              //methods
12              Method * mtFire1;
13              Method * mtFire2;
14              Method * mtFire3;
15              Method * mtFire4;
16  };
17  #endif /* Archer_H_ */
18
19  #include "Archer.h"
20  Archer::Archer(void) {
21        BOOLEAN(this,atHasFired,false);
22        INTEGER(this,atCount,0);
23        METHOD(this,mtFire1,atHasFired,true,0);
24        METHOD_OPERATION(this,mtFire2,atCount,new
25  Addition(atCount,1),0);
```

```
26          METHOD_OPERATION(this,mtFire3,atCount,new
27    Addition(atCount,atCount),0);
28          METHOD(this,mtFire4,,,0);
29    }
30    Archer::~Archer(void) {
31    }
```

Como é possível observar no Algoritmo 43, para compor uma *FBE* em *Framework* além de entender os conceitos PON era necessário realizar a importação de componentes específicos do *Framework*, conforme a linha de código 3 demonstra. Além disso, para criar *Attributes* e *Methods* era necessário conhecer a sintaxe necessária bem como entender a assinatura dos métodos necessários para tal, conforme as linhas 9 e 12 demonstram. Em contrapartida, o Algoritmo 44 apresenta o trecho de código equivalente a criação da *FBE* na linguagem PON.

Algoritmo 44: *FBE* escrita com código *Framework*

```
1     fbe Archer
2         attributes
3             boolean atHasFired false
4                 integer atCount 0
5         end_attributes
6         methods
7             method mtFire1(atHasFired = true)
8                 method mtFire2(atCount = atCount + 1)
9                 method mtFire3(atCount = atCount + atCount)
10                method  mtFire4()  begin_method  //código  específico  em
11    C/C++ end_method
12        end_methods
13    end_fbe
```

Conforme apresentado no Algoritmo 44, a linguagem PON simplifica o processo de criação de componentes PON. Isto pode ser medido pela quantidade de linhas de código que foram necessários para criar a *FBE* em questão, sendo 13 linhas para a linguagem PON e 31 linhas para o código equivalente em *Framework*. Outra observação importante é o fato de não ser necessário importar componentes e ser somente necessário ter o conhecimento da linguagem PON para compor *software* com a linguagem.

Com a organização do conhecimento do programa em termos de *Rules*, o entendimento da semântica de um programa em PON se torna mais fácil do que no PI. As *Rules*, na linguagem PON, são definidas em um mesmo arquivo, apresentando uma visão geral do funcionamento do programa, diferentemente do PI onde o conhecimento pode ficar disperso pelos emaranhados de chamadas de métodos. Também, uma *Rule* é independente da sequência em que é definida,

podendo ser definida em qualquer ordem ou a qualquer momento em que venha ser necessária, sem afetar (fortemente) a execução das demais expressões. Assim, o programador pode se concentrar na concepção de uma única *Rule* por vez, de forma assaz independente das demais, uma vez que os possíveis conflitos serão resolvidos pelos modelos de resolução de conflito [RONSZCKA, 2012].

Deste modo, esta Seção apresenta comparativos relacionados a facilidade de programação com a utilização da Linguagem e Compilador PON. A Subseção 4.3.1 apresenta comparações sobre facilidade de programação entre a Linguagem e Compilador PON, *Framework* e PI avaliando a complexidade de código entre cada abordagem. A Subseção 4.3.2 apresenta relatos dos desenvolvedores que utilizaram a linguagem e compilador PON até o momento e, finalmente, a Subseção 4.3.3 apresenta as considerações finais sobre a facilidade de programação utilizando a linguagem e compilador PON.

4.3.1 Complexidade de Código

Uma forma de avaliar a facilidade de programação, de uma determinada linguagem se dá pela validação da sua complexidade de código-fonte. Deste modo, este trabalho se baseia na forma de comparação de complexidade de código apresentado pela dissertação de mestrado do Robson Xavier [XAVIER, 2014]. No trabalho citado o PON foi comparado ao POE (Paradigma Orientado a Eventos) utilizando como métricas a contagem de número de linhas de código (*LOC– lines of code*) e número de *tokens* na linguagem (i.e. medidas na Linguagem e Compilador PON e *Framework*).

Para tanto, foram utilizadas as ferramentas *cloc28* e *cloc –diff* para número de linhas de código-fonte e para número de *tokens* analisadores léxicos e sintáticos (implementados com as ferramentas flex e bison). Neste contexto, as medições seguiram as seguintes indicações: contagem de todo código-fonte (inclusive códigos acessórios como enumeradores e componente principal – classe *Main*) no tocante a linhas de código e contagem de todo código-fonte no tocante ao cálculo de número de *tokens* utilizados. Para medidas de número de linhas de código, no primeiro estudo de caso as mesmas se deram em maneira global (número de linhas de código-fonte absoluto.

Os estudos comparativos foram realizados comparando as aplicações Mira ao Alvo e aplicação de Vendas, detalhadas nas seções 4.1 e 4.2, respectivamente. A figura a seguir apresenta os resultados coletados para a aplicação de Mira ao Alvo.

Complexidade de Código	Linhas de Código	Tokens
PON	63	132
Framework	186	411
PI C++	160	278

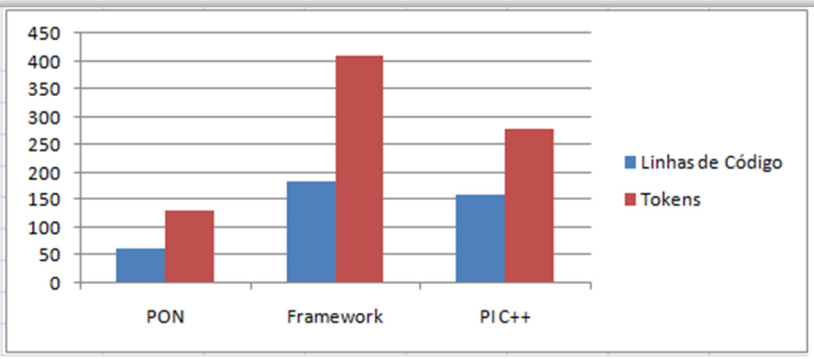


Figura 42 – Complexidade de código para aplicação Mira ao Alvo

A Figura 42 apresenta os dados relacionados a medição de número de linhas de código e número de *tokens* realizados para criar a aplicação Mira ao Alvo pela Linguagem e Compilador PON, *Framework* e PI/POO C++. Como é possível observar a linguagem e compilador PON apresentou os melhores resultados, tanto para o número de linhas de códigos e *tokens*. O *Framework* apresentou uma desvantagem quando comparado a linguagem e compilador PON e PI/POO C++, devido ao fato de ser necessária ao criar código PON com o *Framework* a utilização de várias importações de código e da necessidade da utilização de vários componentes que foram criados para o *Framework*, isto de fato, determinou a inclusão de mais linhas de código, assim como o conhecimento de mais *tokens* relacionados ao *Framework*.

Ao seu turno, o PI/POO C++ apresentou o resultado insatisfatório quando comparado a linguagem e compilador PON, no entanto, um resultado satisfatório quando comparado ao *Framework*. Isto demonstra que mesmo para aplicações simples o *Framework* necessita da utilização de vários componentes e importações, o que de certo modo, encarece a complexidade de código. Vale ressaltar que o código gerado para PI/POO C++ certamente poderia ser implementado de maneira mais otimizada. Entretanto, o experimento é representativo e visa avaliar uma realidade prática onde tal otimização seria inviável ou pelo menos custosa em termos de esforço de programação. Deste modo, o intuito deste estudo é demonstrar

que um código em PI/POO C++ desenvolvido a partir dos requisitos e sem observar questões de complexidade apresentou resultados piores quando comparados aos resultados apresentados tanto pela linguagem e compilador PON.

A próxima Figura apresenta os resultados coletados para a aplicação de Vendas.

Complexidade de Código	Linhas de Código	Tokens
PON	392	707
Framework	517	1089
PI C++	722	1262

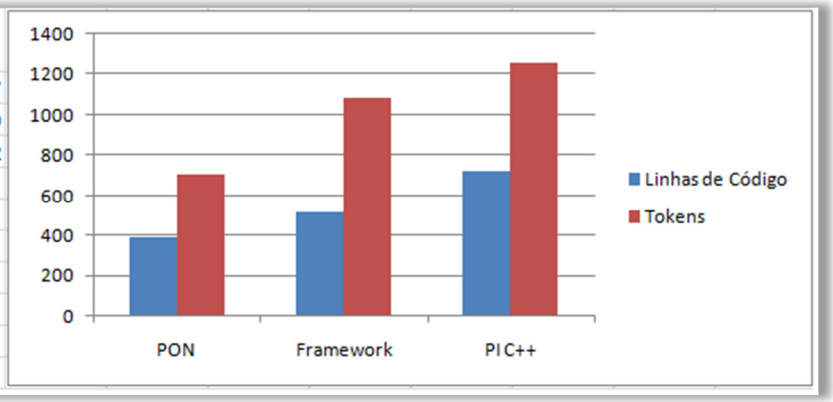


Figura 43 – Complexidade de código para a Aplicação de Vendas

A figura 43 apresenta os resultados relacionados a medição de linhas de código e contagem de *tokens* que foram utilizadas para o desenvolvimento da aplicação de Vendas. Como é possível observar neste cenário, a linguagem e compilador PON apresentou, novamente, os melhores resultados. Em comparação ao cenário anterior o número de linhas e *tokens* aumentaram isto, devido a complexidade maior que a aplicação de vendas apresenta. Mais uma vez, o resultado coletado para o *Framework* apresentou certa desvantagem quando comparado ao resultado da linguagem e compilador PON. Esta desvantagem se encontra no número de elementos que é necessário incluir em um código construído com o *Framework*. Já a aplicação desenvolvida em PI/POO C++ apresentou, neste cenário, o pior resultado. Como é possível observar tanto o número de linhas de código e *tokens* utilizados pela aplicação desenvolvida sob o viés do PI/POO C++ foram superiores aos utilizados tanto pelo código desenvolvido com a linguagem e compilador PON e *Framework*.

4.3.2 Relatos sobre a utilização da Linguagem e Compilador PON

A linguagem PON, conforme descrita neste trabalho se apresenta como uma alternativa para o desenvolvimento de aplicações sob o viés do PON. No entanto, tal linguagem não foi amplamente utilizada para o desenvolvimento de aplicações até o presente momento o que dificulta avaliar com precisão questões como facilidade de desenvolvimento e produtividade de código.

Sendo assim, como uma alternativa de avaliação sobre questões de facilidade de programação, foram coletados relatos de pesquisadores que fizeram uso da linguagem e compilador PON como testemunho das expectativas da utilização da mesma. Seguem os relatos repassados pelos pesquisadores:

"*A nova linguagem PON facilita a vida do desenvolvedor, uma vez que apresenta uma maneira simplificada para construção de programas PON sem a necessidade de um conhecimento prévio da linguagem C/C++. No Framework PON era necessário que o desenvolvedor conhecesse algumas particularidades de C/C++ como ponteiros, includes, namespaces, estruturas de dados, assim como o conhecimento de todas as classes das entidades PON para suas respectivas instanciações e amarrações através de estruturas de dados. A principal vantagem ao meu ver é que a linguagem se apresenta de forma clara e intuitiva possibilitando inclusive que novos programadores venham a conhecer e a compreender a teoria do PON de uma maneira didática, simplesmente pelo fato de construir as regras através do aninhamento das entidades notificantes do PON expressas explicitamente no código.*"

Adriano Francisco Ronszcka, doutorando UTFPR.

"*A linguagem PON é simples e interessante, mas ainda é muito prematura. Os problemas que eu acho que tem são: falta de algum mecanismo de controle de visibilidade (private, public) compreendo que o PON é sempre public, mas isso é potencial causador de problemas como sabemos. A implementação de um Method precisa ser mais bem trabalhada (objeto de trabalhos futuros). Pois alguns Methods precisarão fazer várias atividades complexas além de apenas alterar algum Attribute, poderá criar novas FBEs, Rules e por aí vai. Um fato que realmente chama a*

atenção é que o Framework (por ser implementada em OO) obviamente requer muita escrita de instruções para realizar pequenas coisas, nisso a linguagem PON ganha disparado (simplicidade)."

Clayton Kossoski, Mestrando UTFPR.

"*Na minha opinião é bem mais fácil utilizar a linguagem PON do que o Framework. Eu não tinha conhecimento prévio do PON e o Framework não tem manual de instruções, então foi difícil de entender, programar, enfim, usar. Já a linguagem PON é intuitiva e dá para entender como funciona.*"

Priscila Moraes Loris, doutoranda pela Unicamp.

De acordo com os relatos acima é possível perceber que a linguagem PON se mostra satisfatório em termos de facilidade de programação. Todos os pesquisadores relatam que uma das principais vantagens da linguagem é a sua simplicidade. Conforme salientado por Ronszcka com o advento da linguagem PON é possível a partir deste momento conhecer e compreender a teoria do PON de uma maneira didática olhando exemplos que comportam somente estruturas do PON e não elementos advindos de outros paradigmas. Ainda, a linguagem e compilador PON se apresentam como uma alternativa para ensinar o paradigma, pois a sua estrutura permite focar somente nos conceitos fundamentais do paradigma sem a necessidade de entender conceitos de paradigmas externos.

Ainda, vale ressaltar que a linguagem precisa evoluir. Conforme salientado pelo Kossoski a linguagem PON ainda precisa de evoluções relacionadas a integração de seus conceitos ao O.O e evoluir certos mecanismos internos como a maneira como seus *Methods* funcionam.

Não obstante, alguns pesquisadores da UTFPR em conjunto a uma disciplina ofertada pelo PGGCA, sob a supervisão do Prof. Dr. Jean M. Simão, fizeram uso da Linguagem e Compilador PON. Tal disciplina apresentou como resultado o envio para publicação de alguns artigos no qual tais pesquisadores relatam a utilização da LingPON. Tais artigos estão presentes no Anexo A deste trabalho e a seguir são apresentados relatos da utilização da LingPON por parte destes pesquisadores.

"*A princípio a implementacão em LingPON é interessante por se tratar de uma linguagem dedicada ao paradigma. A sintaxe e palavras reservadas são simples e intuitivas. No entanto, por se tratar de uma linguagem ainda em desenvolvimento, apresenta algumas limitações*."

Helio Henrique Lopes Costa Monte-Alto, Professor UFPR Palotina.

"*Quando desenvolvido em PON, utilizando sua linguagem nativa LingPON, as Rules que regem o funcionamento do sistema de controle são apresentadas de maneira mais clara e direta, uma vez que sua composição é feita através de entidades bem definidas e explicitamente declaradas. Entretanto, o LingPON ainda esta em fase de desenvolvimento. Portanto, surgiram algumas dificuldades para o desenvolvimento de uma aplicação complexa, tais como falta de suporte de declaração de um FBE como Attribute de outro FBE. Porém, mesmo com certas limitações, o LingPON ainda se apresenta como uma promissora ferramenta para o desenvolvimento de software em PON. Isso porque seu compilador poderá permitir a geração de código em diversas linguagens de programação."*

Leonardo Araujo Santos, Mestrando UTFPR

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

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

Leonardo F. Pordeus, Mestrando UTFPR

Ainda, a linguagem e o compilador PON deram suporte a estudos adicionais em outros trabalhos de pesquisa. Houve um esforço por parte do aluno de mestrado Robson Duarte Xavier, o qual comparou o PON a abordagens orientadas a eventos [XAVIER, 2014]. Neste trabalho ele realizou estudos comparativos com o intuito de avaliar a complexidade de código entre as abordagens. Segue o relato apresentado no trabalho do Xavier com relação a complexidade de código ao utilizar a linguagem PON.

"*Para comparar PON e POE em medidas de complexidade de código, utilizando como critérios de número de linhas de código (LOC), número de escopos (recipientes de escopo léxico - closures) e número de tokens necessários, os dados mostraram números menores de LOC para PON em vários casos, maior estabilidade de código-fonte entre cenários de software que foram construídos de modo complementar (manutenções evolutivas), números menores de escopos e números menores de tokens nas medidas dos softwares. Dessa maneira, indicam uma vantagem em concisão e expressividade de código fonte ao utilizar PON, e ainda maior em LingPON (que ainda está em desenvolvimento por outros pesquisadores do grupo de pesquisa PON).*"

Robson Duarte Xavier, Mestre UTFPR

Outro exemplo da utilização da linguagem e compilador PON se encontra na tese de doutorado do aluno Robson Ribeiro Linhares. Neste trabalho de doutorado o objetivo foi realizar o desenvolvimento de uma arquitetura de computação própria ao PON [LINHARES, 2015]. Em linhas gerais, neste trabalho foi possível observar que os estudos comparativos foram guiados com a utilização da linguagem e compilador PON e os resultados apresentados se mostraram satisfatórios, no tocante a desempenho, ao PON. Ainda, tal trabalho aponta a necessidade da evolução do compilador PON no tocante a geração de código *Assembly* específico, o que se constitui de um trabalho futuro relatado na Subseção 5.2.3 do presente trabalho.

Deste modo, conforme os relatos apresentados a linguagem PON se apresenta de forma satisfatória em termos de facilidade de programação, isto, devido a sua simplicidade. No entanto, o LingPON, conforme relatos, carece de

evoluções em seu estado da técnica, sendo tais evoluções destacadas como trabalhos futuros. Porém, tais deficiências não impediram a utilização da linguagem e os resultados da sua utilização, conforme salientado anteriormente, podem ser validados no Anexo A deste trabalho.

4.3.3 Considerações Sobre a Facilidade de Programação

Uma das características salientadas para o PON diz respeito a facilidade de programação. Esta facilidade surge dos conceitos preconizados pelo PON, ao bem da verdade, estes conceitos surgem para tornar a programação mais simples. Ainda, os conceitos do PON permitem estruturar o conhecimento de uma determinada aplicação de uma forma natural ao ser humano e proporciona a composição deste conhecimento de forma simplificada e transparente.

Como foi possível observar nos experimentos e relatos sobre facilidade de programação do PON, mais precisamente da sua linguagem e compilador, este apresentou resultados satisfatórios quando comparado ao *Framework* PON e a PI/POO C++ (mais precisamente a OO). Conforme estudos apresentados, a complexidade de código para compor *software* utilizado a linguagem e compilador PON é menor quando comparado ao *Framework* e PI/POO C++. Isto se deve ao fato do desenvolvedor somente se preocupar em entender os componentes da linguagem e compilador PON e não precisar incluir componentes, de certa forma, complexos como é o caso do *Framework*.

Conforme os relatos apresentados na Subseção anterior, a LingPON se destaca pela sua simplicidade e clareza. Os elementos PON são apresentados com um nível de abstração adequado. O desenvolvedor tem um maior controle na composição dos elementos e na maneira como estes elementos são conectados, no caso as *Rules* e *FBEs*.

Em contrapartida, a linguagem e compilador necessitam de evoluções. Conforme destacado nos relatos a linguagem e compilador apresentam certas deficiências relacionadas a sua análise semântica que dificultam, de certa maneira, a composição do software. Ainda, alguns conceitos do PON como regras de

formação[6] e estruturas de dados não foram adicionadas a linguagem o que sustenta a evolução da técnica da linguagem e compilador.

Sendo assim, conforme apresentado por tais estudos é possível vislumbrar que a linguagem e compilador PON converge como uma alternativa para o desenvolvimento de aplicações sob este viés e ainda com foco em uma das principais características do PON, no caso, a facilidade de programação.

4.4 CONSIDERAÇÕES FINAIS SOBRE OS ESTUDOS DE CASO

Conforme foi possível observar nos experimentos os resultados para a versão em PON C foram os mais satisfatórios quando comparados aos resultados das versões em PON C++ e *Framework*. Isto se deve ao fato da versão em C ser menos dependentes de estruturas de dados que dão suporte à orientação a objetos (i.e. classes que contemplam conceitos PON, tabelas internas que provêm o suporte à polimorfismo, e alocação dinâmica) como é o caso da versão em C++ e *Framework*. A versão em C apresenta os conceitos de *struct* e funções que são chamadas conforme o processo de inferência do PON requer.

É importante salientar, que este comportamento é possível de ser alcançado no código gerado em C++. Mas por motivos experimentais o código em C++, como dito anteriormente, apresenta classes que contemplam conceitos PON mencionados e essa estrutura representa a alocação dinâmica de memória mencionada. Ainda, o código gerado para linguagem PON C++ foi mais eficiente (utilizou menos tempo de processamento) que o código executado para PON *Framework*. Isto é devido ao fato do PON *Framework*, como relatado em trabalho anteriores, apresentar algumas desvantagens de implementação, como a sobrecarga de usar uma estrutura de dados computacionalmente cara, devido ao fato de ser genérica.

A versão em PI/POO C++ apresentou resultados insatisfatórios quando comparado a versão em PON C. Isto devido ao fato da versão em PI/POO C++ ser desenvolvida sem evitar o uso de classes e alocação dinâmica conforme mencionada na versão em PON C++. Quando comparada a versão em PI/POO C++

[6] Segundo Banazewski (2009) uso das Regras de Formação é bastante útil quando o conhecimento causal de uma Rule é comum para diferentes conjuntos de instâncias de FBEs, ou seja, quando várias regras ditas específicas representam a mesma estrutura e apenas se diferenciam nas instâncias referenciadas.

com a versão em PON C++ estas de certa forma se mostraram equivalentes. No entanto, no segundo estudo comparativo foi possível observar que a versão em PON C++ se mostrou estável com relação a tempo de processamento quando comparada a versão em PI/POO C++. Isto se deve ao fato da versão em PON C++ seguir as definições do PON e, consequentemente, não desperdiçar tempo de processamento em avaliações de expressões causais desnecessárias.

Ainda, os estudos relacionados a facilidade de programação sugerem que a linguagem PON se apresenta de fato como uma alternativa para o desenvolvimento de aplicações sob o domínio deste paradigma. De acordo com os estudos relacionados a complexidade de código a linguagem PON se destaca por sua simplicidade o que também foi salientado nos relatos sobre a utilização da linguagem.

De maneira geral, os resultados se mostraram bastante satisfatórios. Os esforços para a criação de uma linguagem própria para o PON e a implementação de versões mais otimizadas de geradores de código sob os princípios do PON, confirmam o potencial da compilação para que o PON atinja suas reais capacidades principalmente em termos de desempenho. Ainda, é possível observar que quanto mais o código alvo se aproxima do código de máquina (e.g. código gerado em C) mais baixo fica o consumo de processamento, o que aumenta a expectativa na geração de código alvo de máquina. Salienta-se que o código gerado em C, com a utilização do compilador *gcc*, gera código de máquina otimizado, sendo possível realizar essa opção no momento da compilação do código.

No próximo capítulo serão apresentadas as conclusões para esse trabalho bem como serão discutidos possíveis trabalhos futuros.

5 CONCLUSÃO E TRABALHOS FUTUROS

Este capítulo apresenta a conclusão final sobre o trabalho e aponta perspectivas para trabalhos futuros. Desta forma, a Seção 5.1 apresenta a conclusão deste trabalho. A Seção 5.2, por sua vez, apresenta os vislumbramentos para trabalhos futuros que possivelmente contribuirão ainda mais para a evolução do estado da técnica do PON no tocante a evolução da linguagem/compilador.

5.1 CONCLUSÃO

Este trabalho teve como objetivo principal a evolução do estado da técnica do PON no tocante a questões de desempenho e facilidade de programação por meio da criação de uma linguagem para o PON e da criação de um compilador baseado em tal linguagem. Para tanto, inicialmente tanto o PON quanto a sua materialização na forma de software (e.g *Framework*) e *hardware* foram detalhados. Posteriormente, o presente trabalho contou com a definição do conceito de compiladores e ferramentas de compilação necessárias para criação de tal linguagem e compilador. Ainda, fez-se necessária a apresentação da linguagem com seus detalhes e definições bem como as características definidas para o compilador.

A linguagem PON se apresenta ao desenvolvedor como uma alternativa para o desenvolvimento de aplicações sob o viés desse paradigma. Essa linguagem foi criada com o intuito de abstrair as características do PON em uma gramática apropriada. A partir deste momento, o desenvolvedor precisa somente conhecer a linguagem para poder criar aplicações e não se ater as características de componentes e associações como foi até então com a utilização do *Framework*.

Uma vez definida a linguagem fez-se necessário a criação de um compilador para a mesma. O compilador PON foi criado com o objetivo de evoluir o estado da técnica do PON com a criação de código a partir de uma linguagem PON previamente estabelecida. Para isso, tal compilador foi construído com o intuito de gerar código PON equivalente para três formatos. O primeiro formato tem como resultado o código para o atual *Framework* Otimizado do PON (versão 2.0). Ainda, os resultados do compilador contaram com a geração de código para C e C++. Estes, por sua vez, foram criados para validar se o compilador PON poderia ter uma

evolução com relação ao seu estado da técnica no tocante a desempenho (e.g. tempo de processamento).

Para validar a evolução do estado da técnica do PON, com o advento do seu compilador, estudos comparativos foram realizados. Duas aplicações foram criadas, sendo a aplicação denominada Mira ao Alvo e a aplicação de Vendas. Os estudos comparativos focaram em questões de desempenho e facilidade de programação.

No tocante a desempenho foi possível perceber que o código gerado pelo compilador para linguagem C apresentou os melhores resultados relacionado a tempo de processamento. Isto se deve ao fato do código em PON C ter uma complexidade menor quando comparado aos demais. Conforme foi possível visualizar na avaliação do código em *Assembly* gerado o código do compilador em C foi o que apresentou o menor número de linhas de código e consequentemente a menor complexidade o que confirma o seu baixo consumo de tempo de processamento. A versão do código específico PON em C apresenta os conceitos de estruturas (i.e. struct) e funções que são chamadas conforme o processo de inferência do PON requer. A versão em C contempla para cada elemento do tipo *FBE* a criação de um Header (.h) e um arquivo (.c) que implementa os procedimentos e as funções pertinentes àquele módulo. No entanto, independente da aplicação, alguns arquivos são genéricos, os quais representam as *Rules*, as *SubConditions* e *Premisses*. Deste modo, por mais complexo que seja a aplicação desenvolvida em PON o código gerado em PON C aumentará somente no número de elementos que representam as *FBEs* o que concentra a chamada de funções pelo processo de inferência somente em determinados arquivos evitando assim o *overhead* em chamadas de funções.

O código do compilador para C++ apresentou resultados insatisfatórios quando comparado ao código em C. É importante salientar que o código do compilador para código específico PON em C++ apresenta os conceitos de classes que representam os conceitos PON, relativos a *FBE*, *Rule* e *Premise*, bem como faz uso de alocação dinâmica de objetos. Ainda, é possível constatar que o código PON em C++ apresenta mais arquivos de código fonte que o código PON em C. Sendo assim, o processo de inferência do PON aplicado ao código PON em C++ realiza chamadas de funções em classes espalhadas em diferentes arquivos. Isto, de fato, pode ocasionar a sobrecarga em chamadas de métodos. Outra observação importante é relacionada ao fato do código em PON C++ fazer uso de alocação

dinâmica de objetos. Isto contribui para aumentar o gasto de tempo de processamento. Deste modo, isto justifica o fato da versão em PON C++ apresentar um resultado menos satisfatório quando comparada a versão PON em C. Por outro lado, a versão em PON C++ obteve resultados de desempenho satisfatórios quando comparado ao resultado em *Framework*.

Por sua vez, o Framework PON em C++ em termos de desempenho obteve resultados insatisfatórios quando comparado a versão de código específico PON em C e em C++. Isto é devido ao fato do *Framework* PON C++, conforme relatado em trabalhos anteriores, apresentar algumas desvantagens de implementação, como a sobrecarga de usar uma estrutura de dados computacionalmente cara, devido ao fato de ser genérica, e ser desenvolvido sob uma linguagem intermediária, no caso o C++.

Quando comparado os resultados do compilador PON com os resultados das aplicações equivalentes construídas em PI/POO C++, os resultados para o código do compilador em C ainda obteve os resultados mais satisfatórios sendo seguido dos resultados do código do compilador em C++. Isto é devido, como salientado anteriormente, a versão em PI/POO C++ ser desenvolvida com as características da orientação a objetos como a representação do conhecimento através de classes e a utilização de alocação dinâmica de objetos. Isto de fato contribui no aumento do tempo de processamento conforme discutido na versão em PON C++. Por outro lado, a versão em PI/POO C++ se mostrou, de certa forma, equivalente a versão em PON C++ tendo seu resultado razoavelmente pior. Isto se deu ao fato das versões serem, de certa forma equivalentes, com relação a utilização de conceitos de OO e pelo fato da PI/POO C++ não se utilizar dos conceitos do PON em sua concepção.

No tocante a facilidade de programação é possível perceber que a linguagem PON até o momento se apresenta de forma mais simplificada quando comparada ao *Framework* PON. Os estudos comparativos sobre complexidade de código demonstram tal simplicidade e os relatos dos pesquisadores do PON confirmam que a linguagem converge para ser uma alternativa para programar em PON que apresente mais produtividade.

Portanto, como foi possível notar com os resultados obtidos, pode-se concluir que a linguagem e compilador PON convergem para a melhoria do estado

da técnica do PON, no tocante a desempenho, e se apresenta como uma alternativa para o desenvolvimento de aplicações sob o viés desse paradigma.

5.2 TRABALHO FUTUROS

O presente trabalho abre perspectivas de pesquisa para ainda mais evoluir com a materialização da linguagem/compilador PON. Neste sentido, a Subseção 5.2.1 apresenta como trabalho futuro a evolução da linguagem e compilador PON sob o viés de facilidade de programação. A Subseção 5.2.2 apresenta como uma alternativa para trabalho futuro a criação de um analisador semântico. A Subseção 5.2.3, por sua vez, vislumbra a evolução do compilador para geração de código alvo em *Assembly*, a Subseção 5.2.4 vislumbra a otimização do código em PON C++. Finalmente, a Subseção 5.2.5 vislumbra a adição de conceitos de OO a linguagem.

5.2.1 Facilidade de Programação

A linguagem e compilador PON apresentados neste trabalho se mostram como uma alternativa na criação de aplicações sob o domínio desse paradigma. No entanto, a linguagem e compilador PON não foram submetidos a criação de aplicações robustas e com certo nível de dificuldade. Salienta-se como dificuldade, a forma de estruturar os programas utilizando tal linguagem.

Ademais, até o momento apenas algumas aplicações foram desenvolvidas e com escopos relativamente pequenos, o que não contribui efetivamente para validar questões de facilidade de programação de fato. Sendo assim, se torna muito pertinente a evolução da linguagem e compilador no tocante a facilidade de programação tendo em vista a criação de mais aplicações e a introdução de padrões para a linguagem.

Como descrito em trabalhos anteriores uma série de aplicações pertinentes ao PON podem ser aplicados no escopo da linguagem e compilador PON o que foca na facilidade de programação.

5.2.2 Analisador Semântico

O presente compilador não apresenta uma análise semântica completa. Até o presente momento o compilador somente realiza validações de tipos e repetição de identificadores. Sendo assim, para que a linguagem tenha maior aceitação e facilidade de codificação a análise semântica pode ser melhorada no intuito de emitir mensagens de erro de codificação mais amigáveis e outras validações pertinentes a linguagem PON (e.g. avisos ou *warnings* a respeito de atributos de *FBEs* não utilizados). Sendo assim, como trabalho futuro seria relevante a criação de um analisador semântico mais abrangente com relação as regras do PON.

5.2.3 Geração de Código-Alvo em Assembly

Ainda, a presente dissertação apresenta a evolução do PON com relação a criação de uma linguagem/compilador. Esse compilador teve como objetivo a geração de código em três formatos, sendo C, C++ e *Framework*. O intuito da geração do código nesses formatos foi a validação e avaliação do desempenho do código gerado em C, C++ quanto ao código gerado em *Framework* e POO.

No entanto, ainda se faz necessária a evolução da geração de código no tocante a linguagem de máquina. Pois, como foi possível observar, quanto mais próximo disto o código alvo fica, mais ganho em economia de tempo de processamento. Deste modo, este compilador permitiria compilar o código PON de maneira mais eficiente e eliminaria por completo o uso de estruturas de dados para armazenar referências a objetos a serem notificados, como foi o caso do código gerado em C++.

Sendo assim, uma boa alternativa seria a geração de código alvo em *Assembly* como trabalho futuro.

5.2.4 Otimização do código em PON C++

Conforme apresentado nos estudos comparativos deste trabalho, foi possível perceber que a versão PON C++ apresentou resultados insatisfatórios quando comparada a versão em PON C. Isto é devido ao fato da versão em PON

C++ fazer uso de recursos da orientação a objetos como é o caso de alocação dinâmica de objetos e a representação do conhecimento na forma de classes, o que de certa forma diminuiu o desempenho devido ao *overhead* nas chamadas de métodos.

Disto isto, vale ressaltar que a linguagem C++ é uma extensão da linguagem C que incorpora recursos de OO. Todas as características da linguagem C estão disponíveis na linguagem C++. Deste modo, é possível aperfeiçoar um código desenvolvido em C++ com certas características que aumentam o desempenho de aplicações desenvolvidas em C.

Sendo assim, um trabalho futuro almejado para este trabalho seria evoluir o código PON C++ no tocante a desempenho. Uma das possibilidades seria a inclusão de métodos *inline*. Tais métodos têm como objetivo tornar o código mais eficiente, tendo em vista que a versão em C++ apresenta mais classes e, consequentemente, mais métodos no sentido de representar os conceitos do PON. Outra possibilidade, seria diminuir a quantidade de classes para representar os conceitos de *Rules e Premises.* Na versão em PON C a representação de *Rules e Premises* é feita de forma genérica e isso aumenta o desempenho de tal versão devido ao fato de diminuir a alocação de chamadas de métodos na pilha de execução.

5.2.5 Adição de conceitos de OO à Linguagem PON

O POO apresenta uma séries de vantagens no tocante ao desenvolvimento de software sob o viés deste paradigma. Entre as vantagens destacam-se o agrupamento dos conceitos do mundo real na abstração em forma de classes em objetos. Outro benefício da POO está relacionado a reutilização de código, onde determinadas classes compartilham suas operações.

Deste modo, conforme salientado neste trabalho, o PON se inspira em tais vantagens da POO e até mesmo as revoluciona em certo ponto. No entanto, a linguagem e compilador criados para o PON até o momento não apresenta certas características da POO como herança e polimorfismo.

Sendo assim, como trabalho futuro vislumbra-se a inclusão destas características na linguagem e compilador do PON para que seja possível criar *FBEs* extensíveis assim como permitir criar métodos polimórficos para tais entidades.

REFERÊNCIAS

[AHO et al., 2008] Alfred V. Aho, Monica S. Lam, Ravi Sethi e Jeffrey D. Ullman. Compiladores: Princípios, técnicas e ferramentas. Addison Wesley, 2008.

[BANASZEWSKI et al., 2007] Roni Fábio Banaszewski, Jean Marcelo Simão, Cesar Augusto Tacla e Paulo Cézar Stadzisz. Notification Oriented Paradigm (NOP) - A Software Development Approach based on Artificial Intelligence Concepts. VI Congress of Logic Applied to Technology - LAPTEC 2007. Santos, 2007.

[BANASZEWSKI, 2009] Roni Fábio Banaszewski. Paradigma Orientado a Notificações: Avanços e Comparações. Dissertação de Mestrado, CPGEI/UTFPR. Curitiba, 2009. Disponível em: http://files.dirppg.ct.utfpr.edu.br/cpgei/Ano_2009/dissertacoes/Dissertacao_500_2009.pdf.

[BATISTA et al., 2011] Márcio Venâncio Batista, Roni Fábio Banaszewski, Adriano Francisco Ronszcka, Glauber Zárate Valença, Robson Ribeiro Linhares, Paulo Cézar Stadzisz, Cesar Augusto Tacla e Jean Marcelo Simão. Uma comparação entre o Paradigma Orientado a Notificações (PON) e o Paradigma Orientado a Objetos (POO) realizado por meio da implementação de um Sistema de Vendas. COMTEL 2011. Lima, Peru, 2011.

[BELMONTE et al., 2012] Danilo L. Belmonte, Jean Marcelo Simão e Paulo Cézar Stadzisz. Proposta de um Método para Distribuição da Carga de Trabalho usando o Paradigma Orientado a Notificações (PON). Revista SODEBRAS. 2012.

[BERDYCHOWSKI e ZABOLOTNY, 2010] Piotr P. Berdychowski and Wojciech M. Zabolotny. C to VHDL compiler. SPIE Conference. 2010.

[BROOKSHEAR, 2006] Glenn Brookshear. Computer Science: An Overview. Addison Wesley, 2006.

[CARDOSO e DINIZ, 2008] João M. P. Cardoso e Pedro C. Diniz. Compilation Techniques for Reconfigurable Architectures. Springer Publishing Company, Incorporated, 2008.

[GRUNE et al., 2012] Dick Grune, Ceriel J. H. Jacobs, Kees Van Reeuwijk, Koen Langendoen e Henri E. Bal. Modern Compiler Design. Springer, 2012.

[DONNELY e STALLMAN, 2013] Charles Donnelly and Richard Stallman. Bison: The Yacc-compatible parser generato. 2005. Disponível em: http://www.gnu.org/software/bison/manual/bison.pdf.

[FERREIRA, 2014a]. Cleverson Avelino Ferreira. Linguagem e Compilador para o Paradigma Orientado a Notificações (PON): Avanços e Comparações. Seminário de Acompanhamento I. Programa de Pós-Graduação em Computação Aplicada (PPGCA). Universidade Tecnológica Federal do Paraná, 2014.

[FERREIRA, 2014b]. Cleverson Avelino Ferreira. Linguagem e Compilador para o Paradigma Orientado a Notificações (PON): Avanços e Comparações. Seminário de Acompanhamento II. Programa de Pós-Graduação em Computação Aplicada (PPGCA). Universidade Tecnológica Federal do Paraná, 2014.

[FERREIRA, 2014c]. Cleverson Avelino Ferreira. Linguagem e Compilador para o Paradigma Orientado a Notificações (PON): Avanços e Comparações. Seminário de Acompanhamento III. Programa de Pós-Graduação em Computação Aplicada (PPGCA). Universidade Tecnológica Federal do Paraná, 2014.

[FERREIRA et al., 2013] Cleverson Avelino Ferreira, Jean Marcelo Simão, Paulo Cézar Stadzisz e Márcio Venâncio Batista. Notification Oriented Paradigm (NOP) and Object Oriented Paradigm (OOP): A Comparative Study by means of a Sale Order System. COMTEL 2013. Lima, Peru, 2013.

[FORGY, 1982] Charles L. Forgy. RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. Artificial Intelligence, vol. 19, pg 17-37 1982.

[FRIEDMAN-HILL, 2003] Ernest Friedman-Hill. Jess in *Action*: *Rule* Based System in Java. Greenwich, CT, USA: Manning Publications Co, 2003.

[GABBRIELLI e MARTINI, 2010] Maurizio Gabbrielli e Simone Martini. Programming Languages: Principles and Paradigms. Series: Undergraduate Topics in Computer Science. 1st Edition, XIX, 440 p., Softcover, 2010.

[GAMMA et al., 1995] Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[HUONG E KIM, 2011] Giang Nguyen Thi Huong and Seon Wook Kim. GCC2Verilog Compiler Toolset for Complete Translation of C Programming Language into Verilog HDL. ETRI Journal, 33(5). 2011.

[JASINSKI, 2012] Ricardo Pereira Jasinski. *Framework* para geração de *hardware* em vhdl a partir de modelos em PON (Paradigma Orientado a Notificações). Relatório da disciplina de lógica reconfigurável por *hardware*. CPGEI/UTFPR, Curitiba, 2012.

[LEE et al., 2000] Woo Hyong Lee, J. Morris Chang and Yusuf Hasan. A Dynamic Memory Measuring Tool for C++ Programs. Application-Specific Systems and Software Engineering Technology. 3rd IEEE Symposium, 2000.

[LINHARES, 2015] Robson Ribeiro Linhares. Contribuição para o desenvolvimento de uma arquitetura de computação própria ao Paradigma Orientado a Notificações (PON). Dissertação de doutorado. CPGEI/UTFPR, Curitiba, 2015.

[LINHARES et al., 2011] Robson Ribeiro Linhares, Adriano Francisco Ronszcka, Glauber Zárate Valença, Márcio Venâncio Batista, Fernando Augusto Witt, Carlos Raimundo Erig Lima, Jean Marcelo Simão e Paulo Cézar Stadzisz. Comparações entre o Paradigma Orientado a Objetos e o Paradigma Orientado a Notificações sob o contexto de um simulador de sistema telefônico. COMTEL 2011. Lima, Peru, 2011.

[LINHARES et al., 2014] Robson Ribeiro Linhares, Jean Marcelo Simão e Paulo Cézar Stadzisz. Arquitetura de Computador Orientada a Notificações – ARQPON. Pedido de Patente. INPI, 2014.

[LEVINE et al., 1992] Doug Brown, John Levine, Tony Mason. Lex & Yacc. O'Reilly, 1992.

[NEWELL e SIMON, 1972] Allen Newell and Herbert A. Simon. Human problem solving Englewood Cliffs. NJ: Prentice-Hall, 1972.

[ORACLE, 2013] Oracle. Java language and virtual machine specifications. 2013. Disponível em: http://docs.oracle.com/javase/specs/.

[PAXSON, 1995] Vern Paxson. Flex, version 2.5. Manual da Ferramenta Flex. 1995.

[PETERS, 2012] Eduardo Peters. Coprocessador para Aceleração de Aplicações Desenvolvidas Utilizando Paradigma Orientado a Notificações (PON). Dissertação de Mestrado, CPGEI/UTFPR. Curitiba, 2012. Disponível em: http://repositorio.utfpr.edu.br/jspui/bitstream/1/325/1/CT_CPGEI_M_%20Peters,%20Eduardo_2012.pdf.

[RICARTE, 2008] Ivan Ricarte. Introdução à compilação. Elsevier, 2008.

[RONSZCKA, 2012] Adriano Francisco Ronszcka. Contribuição para a concepção de aplicações no Paradigma Orientado a Notificações (PON) sob viés de padrões. Dissertação de Mestrado, CPGEI/UTFPR. Curitiba, 2012. Disponível em: http://repositorio.utfpr.edu.br/jspui/bitstream/1/327/1/CT_CPGEI_M_%20Ronszcka,%20Adriano%20Francisco_2012.pdf.

[RONSZCKA et al., 2011] Adriano Francisco Ronszcka, Danillo Leal Belmonte, Glauber Zárate Valença, Márcio Venâncio Batista, Robson Ribeiro Linhares, Cesar Augusto Tacla, Paulo Cézar Stadzisz e Jean Marcelo Simão. Comparações quantitativas e qualitativas entre o Paradigma Orientado a Objetos e o Paradigma Orientado a Notificações sobre um simulador de jogo. COMTEL 2011. Lima, Peru, 2011.

[ROY e HARIDI, 2004] Peter Van Roy e Seif Haridi. Concepts, Techniques, and Models of Computer Programming. MIT Press, 2004.

[SHEN e JUANG, 2008] Victor R. L. Shen e Tony Tong-Ying Juang. Verification of Knowledge-Based Systems Using Predicate/Transition Nets. IEEE Transaction on Systems, Man, and Cybernetics - Part A: Systems & Humans, V.38, N.1, 2008.

[SIMÃO, 2005] Jean Marcelo Simão. A Contribution to the Development of a HMS Simulation Tool and Proposition of a Meta-Model for Holonic Control. Tese de Doutorado, CPGEI/UTFPR. Curitiba, 2005. Disponível em: http://repositorio.utfpr.edu.br/jspui/bitstream/1/85/1/CT_CPGEI_D_Sim%C3%A3o%2c%20Jean%20Marcelo_2005.pdf.

[SIMÃO e STADZISZ, 2008] Jean Marcelo Simão e Paulo Cézar Stadzisz. Paradigma Orientado a Notificações (PON) – Uma Técnica de Composição e Execução de Software Orientado a Notificações. Pedido de Patente. INPI, 2008.

[SIMÃO e STADZISZ, 2009a] Jean Marcelo Simão e Paulo Cézar Stadzisz. Inference Process Based on Notifications: The Kernel of a Holonic Inference Meta-Model Applied to Control Issues. IEEE Transactions on Systems, Man and

Cybernetics. Part A, Systems and Humans, Vol. 39, Issue 1, 238-250, Digital Object Identifier 10.1109/TSMCA.2008.20066371, 2009.

[SIMÃO et al., 2012a] Jean Marcelo Simão, Roni Fábio Banaszewski, César Augusto Tacla e Paulo Cézar Stadzisz. Notification Oriented Paradigm (NOP) and Imperative Paradigm: A Comparative Study. Journal of Software Engineering and Applications (JSEA), 2012.

[SIMÃO et al., 2012b] Jean Marcelo Simão, Danillo Leal Belmonte, Adriano Francisco Ronszcka, Robson Ribeiro Linhares, Glauber Zárate Valença, Roni Fábio Banaszewski, João Alberto Fabro, César Augusto Tacla, Paulo Cézar Stadzisz e Márcio Venâncio Batista. Notification Oriented and Object Oriented Paradigm Comparison via Sale System. Journal of Software Engineering and Applications (JSEA), 2012.

[SIMÃO e STADZISZ, 2008] Jean Marcelo Simão, Paulo Cézar Stadzisz, Carlos Raimundo Erig Lima, Fernando Augusto Witt e Robson Ribeiro Linhares. Paradigma orientado a notificações em *hardware* digital. Pedido de Patente. INPI, 2012.

[TANG et al., 1999] Da-Chih David Tang, Ann Marie Grizzaffi Maynard, and Lizy Kurian John. Contrasting Branch Characteristics and Branch Predictor Performance of C++ and C Programs. Performance, Computing and Communications Conference, IEEE International, 1999.

[VALENÇA, 2012] Glauber Zárate Valença. Contribuição para a Materialização do Paradigma Orientado a Notificações (PON). Dissertação de Mestrado, PPGCA/UTFPR. Curitiba, 2012. Disponível em: http://repositorio.utfpr.edu.br/jspui/bitstream/1/393/1/CT_PPGCA_M_Valen%C3%A7a,%20Glauber%20Z%C3%A1rate_2012.pdf.

[VALENÇA et al., 2011] Glauber Zárate Valença, Roni Fábio Banaszewski, Adriano Francisco Ronszcka, Márcio Venâncio Batista, Robson Ribeiro Linhares, João Alberto Fabro, Paulo Cézar Stadzisz e Jean Marcelo Simão. *Framework* PON, Avanços e Comparações. Simpósio de Computação Aplicada. SCA 2011. Passo Fundo, Rio Grande do Sul, Brasil, 2011.

[WIECHETECK et al., 2011] Luciana Vilas Boas Wiecheteck, Paulo Cézar Stadzisz, e Jean Marcelo Simão. Um Perfil UML para o Paradigma Orientado a Notificações (PON). COMTEL 2011. Lima, Peru, 2011.

[WIECHETECK, 2011] Luciana Vilas Boas Wiecheteck. Um Método para Projetos de Software usando o Paradigma Orientado a Notificações. Dissertação de Mestrado, CPGEI/UTFPR. Curitiba, 2011. Disponível em: http://repositorio.utfpr.edu.br/jspui/bitstream/1/212/1/CT_CPGEI_M_Wiecheteck%2c%20Luciana%20Vilas%20Boas_2011.pdf.

[WITT et al., 2011] Fernando A. de Witt, Jean M. Simão, Robson R. Linhares, Paulo C. Stadzisz e Carlos Raimundo Erig Lima. Comparação entre o paradigma orientado a objetos (POO) e o paradigma orientado a notificações (PON) em um controle discreto em lógica reconfigurável. In Anais do XVI SICITE, Ponta Grossa, 2011.

[XAVIER, 2014] Robson Duarte Xavier. Paradigmas de desenvolvimento de software: comparação entre abordagens orientada a eventos e orientada a notificações. Dissertação de Mestrado, PPGCA/UTFPR. Curitiba, 2014.

APÊNDICE A - Trabalho sobre a Linguagem e Compilador PON

Neste apêndice é apresentado um trabalho entregue a disciplina de Linguagem e compiladores, a qual foi ofertada no segundo trimestre de 2014 e ministrada pelo Prof. Dr. João Alberto Fabro e pelo Prof. Dr. Jean Marcelo Simão. Salienta-se que o trabalho teve por objetivo criar uma versão prototipal da linguagem e compilador PON. Em suma, a versão prototipal demonstrou a viabilidade de se desenvolver uma linguagem e compilador para o presente paradigma e foi o ponto de início da evolução do presente trabalho. Este apêndice foi referenciado na Subseção 1.1.3.

Compilador para o Paradigma Orientado a Notificações

Adriano Francisco Ronszcka[1], Cleverson Avelino Ferreira[1],

Priscila Ap. de Moraes Ioris[1], Clayton Kossoski[1]

[1]Universidade Tecnológica Federal do Paraná –(UTFPR)

Curitiba – PR – Brasil

{ronszcka,cleversonavelino,priscila.moraes,claytonkssk}@gmail.com}

Trabalho realizado na disciplina Tópicos Avançados em Engenharia de Software (Linguagens e Compiladores) de ofertada pelo PPGCA e ministrada pelos Profs. Drs. João A. Fabro e Jean M. Simão.

1. INTRODUÇÃO

Ao longo dos últimos anos, o estado da técnica do Paradigma Orientado a Notificações (PON) evoluiu para a materialização de um *Framework* desenvolvido sob os princípios do paradigma orientado a objetos e baseado na linguagem de programação C++. Entretanto, observou-se em [Simão et. al., 2012] que tal materialização não proporciona um resultado satisfatório em termos de desempenho.

Apesar de esforços aplicados na otimização do *Framework* PON [Valença, 2009], observou-se que existe uma necessidade de melhorias no estado da técnica para que o PON atinja o que fora vislumbrado em seu estado da arte.

Nesse âmbito, uma proposta para a evolução do estado da técnica é o desenvolvimento de compiladores que traduzam o código PON para um código-alvo mais puro e menos dependente dos conceitos de outros paradigmas. Conforme

apontado em [Banaszewski, 2009 e Simão et. al. 2012] muitos desses conceitos levam ao forte acoplamento e a processamento redundante.

Tal necessidade alavancou o surgimento de uma linguagem própria para a programação em PON. Tal linguagem visaria principalmente facilitar a implementação de programas baseados no paradigma. Nesse sentido, uma linguagem verbosa e didática fortemente baseada nos conceitos do estado da arte foi proposta e implementada. Com o surgimento da linguagem houve a necessidade da construção de compiladores que tornem o código-alvo mais puro e menos dependente de estruturas de dados e *overheads* presentes na implementação genérica de *Framework* PON.

Nesse trabalho foram desenvolvidos três compiladores, com código-alvo para linguagens e paradigmas distintos. A primeira versão gera código-alvo para o atual *Framework* PON. A segunda versão, por sua vez, gera código-alvo para C++ baseado no paradigma orientado a objetos. Por fim, a terceira versão gera código-alvo para o ANSI C baseado no paradigma estruturado (imperativo).

Para fins comparativos, testes de tempos de processamento e uso de memória foram executados para as três versões do compilador, com a motivação de averiguar se os atuais esforços se aproximam do resultado esperado e previsto no estado da arte do PON (a luz de seu cálculo assintótico) [Simão, 2005].

Os resultados obtidos se mostraram bastante positivos, nos quais as novas versões do compilador apresentaram, em geral, melhores resultados em tempo de processamento e uso de memória com relação ao estado da técnica atual do Framework PON.

Este artigo está organizado como segue: a Seção 2 sucintamente reflete o estado da arte em paradigmas de programação. A Seção 3 apresenta o PON. A Seção 4 descreve sucintamente sobre a função de um compilador. A Seção 5 apresenta a Linguagem PON criada para dar suporte à implementação de três compiladores para o paradigma. A Seção 6 apresenta a implementação do compilador PON-C++. A Seção 7 apresenta a implementação do compilador em PON-*Framework*. A Seção 8 apresenta a implementação de compilador para linguagem PON-C. A Seção 9 apresenta os testes de tempos de processamento e uso de memória para os códigos gerados dos três compiladores. A Seção 10, por fim, apresenta as conclusões e perspectivas de trabalhos.

2. PARADIGMAS

A utilização de técnicas de PI, particularmente POO, costuma atrair os desenvolvedores devido a questões como inércia cultural, riqueza de abstração e flexibilidades algorítmicas. Em PI, em suma, concebem-se programas como sequências de instruções utilizando-se para tanto de buscas sobre entidades passivas (dados e comandos) organizadas segundo uma lógica de execução que envolve, inclusive, a avaliação de expressões causais (*e.g.* se-então) [Banaszewski et al. 2007, Brookshear 2006, Gabrielli and Martini 2010].

Particularmente, estas expressões são frequentemente avaliadas desnecessariamente, degradando desempenho. Isto pode ser exemplificado considerando um conjunto de expressões *se-então* que avaliam os estados de objetos dentro de um laço de repetição dito 'infinito'. Cada expressão condicional avalia certos estados dos atributos de objetos e, se aprovada, chama alguns métodos dos objetos que podem mudar os estados dos atributos. Isto é apresentado no Algoritmo[Brookshear 2006, Gabrielli and Martini 2010, Simão e Stadzisz 2008, Simão e Stadzisz 2009].

Algoritmo 1: Pseudocódigo para o Paradigma Imperativo

```
while (true) do
  if((objeto1.atributo1 = 1) AND (objeto2.atributo1 = 1)) then
        objeto1.método1();
        objeto2.método2();
  end if
  if((objeto1.atributoN = N) AND (objeto2.atributoN = N)) then
        objeto1.métodoN();
        objeto2.métodoN();
  endif
end while
```

Neste exemplo, é observado que o laço de repetição força a avaliação (ou inferência) de todas as condições de maneira sequencial. Entretanto, muitas delas são desnecessárias porque somente alguns objetos têm o valor de seus atributos modificado. Isto até pode ser considerado não importante neste exemplo simples e pedagógico, sobretudo se o número N de expressões causais for pequeno. Entretanto, se for considerado um sistema complexo, integrando muitas partes como aquela, pode-se ter uma grande diferença de desempenho [Forgy 1982, Hill 2003, Simão e Stadzisz 2008, Simão e Stadzisz 2009].

Por sua vez, em PD, salientando SBR, existe a vantagem da programação em alto nível. Primeiramente, define-se uma Base de Fatos composta por entidades como objetos com atributos/métodos. Subsequentemente define-se a Base de Regras com relações causais relativas aos elementos da Base de Fatos. Estas duas bases são processadas por meio de uma Máquina de Inferência. Esta automaticamente compara regras e fatos (e.g. estados de atributos) gerando novos fatos e, portanto, um ciclo de inferência. Não obstante a organização e mesmo eficientes algoritmos de inferência, a programação em PD normalmente é computacionalmente cara em termos de estruturas de dados a serem processadas [Scott 2000, Simão e Stadzisz 2008, Simão e Stadzisz 2009].

Independentemente, em uma análise mais profunda, PI e PD são similares no tocante à inferência, que normalmente se dá por entidades monolíticas baseadas em pesquisas sobre entidades passivas ou voltadas à passividade (i.e. fracamente reativas) que conduzem a programas com passos de execução interdependentes. Estas características contribuem para a existência de sobreprocessamento e forte acoplamento entre expressões causais e estrutura de fatos/dados, o que dificulta a execução dos programas de maneira otimizada, bem como paralela ou distribuída [Gabrielli e Martini 2010, Simão e Stadzisz 2008, Simão e Stadzisz 2009]. Ainda que haja alternativas outras de programação, como orientações a eventos e mesmo orientação a dados, elas apenas atenuam ou fatoram o problema, não o resolvendo conforme discutido em [Banaszewski 2009, Simão e Stadzisz 2008, Simão e Stadzisz 2009].

3. PARADIGMA ORIENTADO A NOTIFICAÇÕES (PON)

O PON encontra inspirações no PI, tais como a flexibilidade algorítmica e a abstração em forma de classes/objetos da POO. O PON também aproveita conceitos próprios do PD, como facilidade de programação em alto nível e a representação do conhecimento em regras dos SBR. Assim, o PON provê a possibilidade de uso (de parte) de ambos os estilos de programação em seu modelo, ainda que os evolua e mesmo os revolucione (de certa maneira) no tocante ao processo de inferência ou cálculo lógico-causal [Simão e Stadzisz 2008, Simão e Stadzisz 2009, Banaszewski 2009].

De fato, o PON apresenta resposta aos problemas destes paradigmas, como repetição de expressões lógicas e reavaliações desnecessárias delas (i.e. redundâncias estruturais e temporais) e, particularmente, o acoplamento forte de entidades no tocante às avaliações ou cálculo lógico-causal. Justamente, o PON apresenta outra maneira de realizar estas avaliações ou inferências por meio de entidades computacionais de pequeno porte, ativas e desacopladas que colaboram por meio de notificações pontuais e são criadas a partir do 'conhecimento' de regras.

Mais precisamente, o PON possui objetos que tratam dos elementos da base de fatos, que são genericamente modelados pela classe *FBE* (*Fact Base Element*), conforme a Figura 1. Cada *FBE* trata de seus atributos por meio de objetos da classe *Attribute* e seus serviços por meio de objetos da classe *Method*. Os objetos *FBE*, por meio de seus Attributes e *Method*s, são passíveis de correlação causal por meio de *Rule*s, as quais se constituem em elementos fundamentais do PON.

A Figura 2apresenta um exemplo de *Rule*, na forma de uma regra causal. Mesmo se passível de representação nesta forma, cada *Rule* é uma entidade computacional composta por outras entidades, conforme Figura 1, que podem ser vislumbradas do ponto de vista de objetos/classes. Por exemplo, a *Rule* apresentada é composta por um objeto *Condition* e um objeto *Action*. A *Condition* trata da decisão da *Rule*, enquanto a *Action* tratada execução das ações da *Rule*. Assim sendo, *Condition* e *Action* trabalham juntas para realizar o conhecimento lógico e causal da *Rule*.

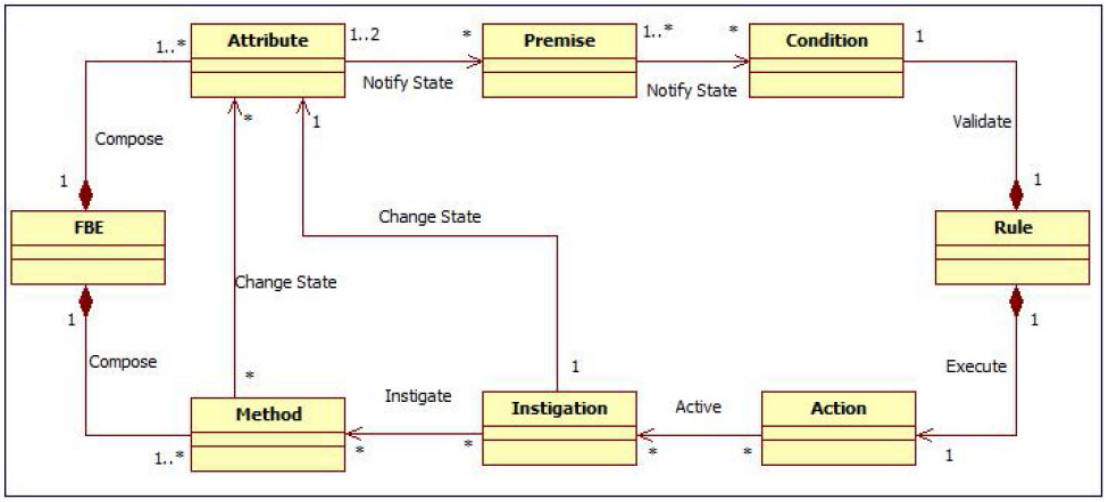


Figura 1: Principais entidades do PON e seus relacionamentos por notificação

Esta *Rule* apresentada na Figura 2 faria parte de um controle de manufatura inteligente, onde equipamentos são tratados a partir de objetos inteligentes (smart-drivers em verdade) que são subtipos de *FBE*. A *Condition* desta *Rule* lida com a decisão de transporte de peça a partir de uma 'Mesa' (Smart-Table) para um 'Torno' (Smart-Lathe) utilizando um 'Robô' (Smart-Robot). Na verdade, esta *Condition* é composta por três *Premise*s que se constituem em outro tipo de objeto inteligente. Estas *Premise*s em questão fazem as seguintes verificações sobre as *FBE*s: a) o Torno está livre? b) o Robô está livre? c) há peça na posição 2 da Mesa? Assim, conclui-se (em geral) que os estados dos atributos das *FBE*s compõem os fatos a serem avaliados pelas *Premise*s.

Na verdade, cada *Premise* avalia o estado de um (ou mesmo dois) Attributes de *FBE*. De fato, para cada mudança de estado de um *Attribute* da *FBE*, ocorrem automaticamente avaliações (lógicas) somente nas *Premise*s relacionadas com eventuais mudanças nos seus estados. Igualmente, a partir da mudança de estado das *Premise*s, ocorrem automaticamente avaliações (causais) somente nas *Condition*s relacionadas com eventuais mudanças de seus estados. Isto tudo se dá por meio de uma cadeia de notificações entre objetos inteligentes, o que se constitui no fundamento do PON conforme modelado na Figura 1 e esboçado na Figura 3.

Em suma, cada *Attribute* notifica as *Premise*s relevantes sobre seus estados somente quando se fizer efetivamente necessário. Cada *Premise*, por sua vez, notifica as *Condition*s relevantes dos seus estados usando o mesmo princípio.

Baseado nestes estados notificados é que a *Condition* pode ser aprovada ou não. Se a *Condition* é aprovada, a respectiva *Rule* pode ser ativada executando sua *Action*. Outrossim, o conhecimento de quem se deve notificar se dá na composição das *Rule*s, em um dado ambiente de desenvolvimento PON [Banaszewski 2009].

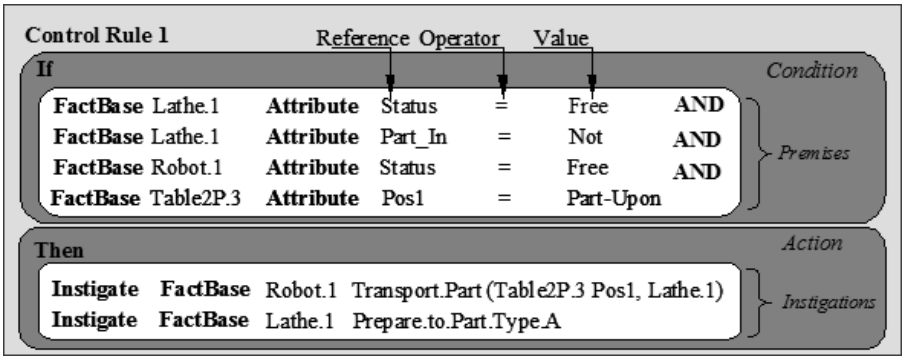


Figura 2: Exemplo de uma *Rule* em PON [Banazewski 2009]

Por sua vez, uma *Action* também é um objeto inteligente que se conecta a objetos inteligentes de outro tipo, as *Instigations*. Neste exemplo, a *Action* contém duas *Instigations* para: a) ativar o Robô para transportar peças da Mesa (posição 2) para o Torno; e b) preparar o Torno para receber a peça. Efetivamente, o que uma Instigation faz é instigar um ou mais métodos responsáveis por realizar serviços ou habilidades de uma *FBE*. Cada método da uma *FBE* é também tratado por um objeto inteligente, chamado de *Method*. A execução de um *Method* geralmente muda o estado de um ou mais *Attributes*. Na verdade, os conceitos de *Attribute* e *Method* representam uma evolução dos conceitos de atributos e métodos de classe da OO. A diferença é o desacoplamento explícito da classe proprietária e a "inteligência" colaborativa pontual para com *Premises* e *Instigations* [Simão e Stadzisz 2008].

Com isto considerado, nota-se que a essência da computação no PON está organizada e distribuída entre entidades autônomas e reativas que colaboram por meio de notificações pontuais. Este arranjo forma o mecanismo de notificações, o qual determina o fluxo de execução das aplicações. Por meio deste mecanismo, as responsabilidades de um programa são divididas entre os objetos do modelo, o que permite execução otimizada e “desacoplada” (i.e. minimamente acoplada) útil para o aproveitamento correto de mono-processamento, bem como para o processamento distribuído.

A natureza do PON leva a uma nova maneira de compor software, onde os fluxos de execução são distribuídos e colaborativos nas entidades. Muito embora o PON permita compor software em alto nível na forma de regras, sem o conhecimento de sua essência, este conhecimento é deveras importante. Por exemplo, é importante saber dos impactos de desempenho e das estratégias de distribuição, como o agrupamento de elementos de maior fluxo de notificações juntos. Assim sendo, o PON permite uma nova maneira de estruturar, executar e pensar os artefatos de software.

Outrossim, o PON atualmente está materializado na forma de um framework/wizard em C++ enquanto a linguagem e compilador PON foram desenvolvidos nesse trabalho. Muito embora um paradigma possa se materializar em outro (como um programa POO materializado em uma linguagem procedimental) e isto seja natural em paradigmas emergentes, seria mais confortável e apropriado um ambiente efetivamente voltado para o PON [Banaszewski 2009].

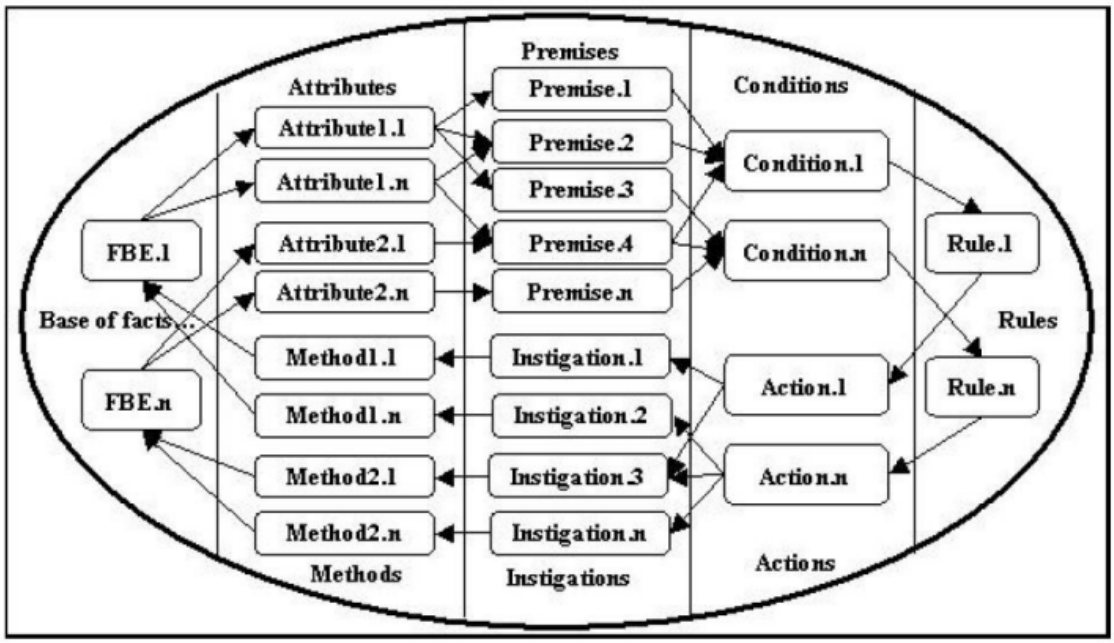


Figura 3: Inferência por Notificações [Simão e Stadzisz 2008, Simão e Stadzisz 2009]

4. COMPILADORES

O objetivo maior desse trabalho é o desenvolvimento de uma linguagem e um respectivo compilador que materialize o PON de maneira a melhor respeitar suas propriedades elementares, sendo: minimização do processamento lógico-causal, desacoplamento entre si dos elementos envolvimento no processamento lógico-

causal e a facilidade de desenvolvimento de software com conhecimento lógico causal. Nesse contexto, faz-se necessário uma breve explicação sobre os compiladores de uma forma geral.

A definição de compiladores segundo Aho [Aho et al. 2006] é:

> "*Um compilador é um programa que lê um programa escrito em uma linguagem (linguagem fonte) e o traduz para um programa equivalente em outra linguagem - a linguagem alvo*".

Independente da linguagem fonte ou alvo, em geral, o compilador é dividido em fases. Cada uma das quais opera no sentido de transformar o código fonte de uma representação para outra. Os principais componentes de um compilador são citados brevemente a seguir [Aho et al. 2006]:

1. **Análise Léxica:** É também chamada de análise linear. Consiste na separação dos elementos da linguagem em *tokens*.
2. **Análise Sintática:** É também chamada de análise hierárquica ou análise gramatical. Consistem no agrupamento de *tokens* do programa fonte em frases gramaticais que são usadas pelo compilador a fim de sintetizar a saída. Usualmente as frases gramaticais são representadas por uma árvore gramatical.
3. **Análise Semântica:** Analisa as frases formadas na árvore gramatical da fase anterior para detectar possíveis erros semânticos para a posterior geração de código.
4. **Geração de Código Intermediário:** Gera-se um código em uma representação intermediária que será utilizada para a geração do código em linguagem de máquina. Geralmente esse código intermediário é um tipo de linguagem de montagem.
5. **Otimizações:** O código intermediário é analisado no sentido de verificar se existem possíveis otimizações a serem realizadas no sentido de melhorar o desempenho do código de máquina a ser gerado.
6. **Geração de Código:** Gera-se o código binário para a tecnologia alvo, já com as otimizações realizadas na fase anterior.

7. **Tabela de Símbolos:** Armazena todos os campos dos atributos envolvidos na compilação. Existe um registro único para cada atributo.

Ainda, a tradução do compilador deve reportar erros no caso de encontrá-los. Subsequentemente, após não haver mais erros “lingüístico-gramaticais” (i.e. erros léxicos, sintáticos e mesmo semi-semânticos), passa-se a geração de código. Dentro do conceito mais tradicional de compilador deve ser gerado um código em linguagem de máquina (ou *Assembly*) ou os *bytecodes*, que serão executados na máquina virtual Java [*Oracle, 2013*]. No entanto, a linguagem fonte e a linguagem alvo são determinadas pelo desenvolvedor do compilador. Isso significa que ele pode traduzir um código escrito em uma linguagem hipotética A para a própria linguagem C, C++ ou Java. Em todo caso, o objetivo final de todo programa é que ele possa ser executado no *hardware*, que pode ser específico para a aplicação ou um Processador de Propósito Geral.

Quando o código gerado é uma linguagem já conhecida, a tradução para o “hardware” é deixada a cargo do compilador padrão para àquela linguagem. No entanto, quando a tradução para o “hardware” não é conhecida deve ser gerado um tradutor para o *hardware* que seja específico a determinada aplicação[Huong e Kim 2011, Cardoso e Diniz 2008, Berdychowski e Zabolotny 2010].

5. A LINGUAGEM PON

Uma gramática define a estrutura geral de formação de uma sentença válida para uma linguagem. Para dar suporte a linguagem PON em um compilador próprio, foi definida a *BNF (Backus-Naur Form)*. Na notação *BNF* [Lewis e Papadimitrion, 2000], as palavras entre os símbolos < e > são chamadas variáveis. O símbolo ::= indica que a variável à sua esquerda pode ser substituída pelos valores à direita.

De modo geral, o código PON segue um padrão de declarações, no qual o desenvolvedor precisa primeiramente definir os *FBEs* de seu programa, em seguida declarar as instâncias de tais *FBEs* e, por fim, definir as *Rules* que farão a conexão das notificações entre as entidades PON. Nesse âmbito, o Algoritmo 2 apresenta o padrão de declarações da linguagem PON.

Conforme apresenta o Algoritmo 2, o código PON é dividido em três partes. A primeira parte representa a declaração dos *FBEs*, a segunda parte, por sua vez, representa a definição das instâncias de tais *FBEs* e, por fim, a terceira parte representa a criação das *Rules*.

Nessa estrutura, a palavra reservada fbe anuncia o início da estrutura de um FBE. Por padrão, todos os FBEs do programa devem ser definidos na primeira parte do código. Na sequência, devem ser declaradas as instâncias dos FBEs. Para isso, a palavra reservada inst anuncia a abertura desse bloco de instanciações. Por fim, as definições de Rules devem ser declaradas, fazendo uso da palavra reservada *rule*. É importante ressaltar que tanto os FBEs quanto as Rules não apresentam um número limitado de declarações.

Algoritmo 2: Padrão de declarações da linguagem PON

```
1       fbe Apple
2       . . .
3       end_fbe
4
5       fbe Archer
6       . . .
7       end_fbe
8
9       ----------------
10
11      inst
12      Apple apple1, apple2
13      Archer archer1, archer2
14      end_inst
15
16      ----------------
17
18      Rule RlTurnOn1
19      . . .
20      end_rule
21
22      Rule RlTurnOn2
23      . . .
24      end_rule
```

Essa primeira versão da linguagem do PON apresenta uma estrutura mais verbosa e particularmente didática, podendo posteriormente ser estendida para uma estrutura mais polida para fins de desenvolvimento avançados. De modo a apresentar melhor a utilização da linguagem em questão, os algoritmos a seguir exemplificam os padrões de implementação adotados nessa versão da linguagem PON.

Algoritmo 3: Exemplo de criação de FBEs

```
1  fbe Archer
2          attributes
3                  boolean atHasFired false
4          end_attributes
5          methods
6                  method mtFire(atHasFired = true)
7          end_methods
8  end_fbe
```

Conforme apresenta o Algoritmo 3 um *FBE* é composto por duas partes. A primeira parte representa a declaração dos *attributes*, enquanto a segunda parte representa a definição dos *methods*. Cada parte, assim como apresentado anteriormente, possui uma palavra reservada para a abertura (*e.g. attributes*) do bloco e outra para o fechamento (*e.g. end_attributes*) do mesmo.

Os *attributes* possuem uma estrutura comumente utilizada na programação atualmente, formada pelo tipo do dado, seguido do nome do atributo e seu respectivo valor inicial.

Nesse âmbito, os tipos de dados disponíveis na linguagem são *boolean, integer, float* e *string*. Os nomes de atributos normalmente seguem um padrão de nomenclatura já adotada em outros trabalhos anteriores do PON [Ronszcka, 2012; Valença, 2012], fazendo uso do prefixo *at*, seguido de um bom nome que defina o propósito do atributo em questão. Por fim, o valor deve estar de acordo com o tipo de dado empregado na construção de tal atributo.

Os *methods*, por sua vez, apresentam uma construção particular em sua estrutura. Esta se dá pelo anuncio da abertura de um *Method* através da palavra reservada *method* seguida do nome e sua respectiva funcionalidade. Ademais, assim como os *Attributes,* os nomes dos *methods* poderiam seguir o padrão de nomenclaturas conhecido, desta forma, fazendo uso do prefixo *mt*. Por fim, conforme apresentado no Algoritmo 3, mais precisamente na linha 7, a funcionalidade do *Method mtFire* está entre parênteses, onde basicamente tal funcionalidade altera o valor de um determinado *Attibute* para *true*.

A segunda parte do código consiste na instanciação dos *FBEs* definidos na primeira parte. Para isso, o Algoritmo 4 exemplifica o padrão de implementação sugerido para tal.

Algoritmo 4: Exemplo de instanciações de FBEs

```
1    inst
2           Apple apple1, apple2
3           Archer archer1, archer2
4    end_inst
```

Conforme apresenta o Algoritmo 4, o bloco de código é iniciado com a palavra reservada **inst** e finalizado com a palavra reservada **end_inst**.Basicamente, a estrutura das instâncias é composta pelo nome do *FBE* seguido do nome da instância do mesmo, podendo essa última, apresentar mais de uma ocorrência, representando a criação de múltiplas instâncias distintas do mesmo *FBE*.

Ainda, no Algoritmo 4, é possível observar nas linhas 2 e 3, que dois tipos distintos de *FBEs* são utilizados na criação de quatro instâncias. É importante ressaltar que os fins-de-linha comumente aplicados em algumas linguagens (;) não devem ser utilizados nessa versão da linguagem.

Por fim, a última etapa consiste na arquitetura da aplicação, onde as *Rules* farão as conexões entre as entidades PON, proporcionando desta forma, o fluxo de execução de uma aplicação. O Algoritmo 5 demonstra o padrão de implementação para a criação de uma *Rule.*

Algoritmo 5: Exemplo de criação de Rules

```
1    rule rlTurnOn
2           condition
3                  subcondition
4                         premise prIsCrossed apple1.atIsCrossed == false
5                         and
6                         premise prHasFired archer1.atHasFired == false
7                         and
8                         premise prReadyToFire controller1.atHasFired == true
9                  end_subcondition
10          end_condition
11          action
12                 instigation inArcherFire1 archer1.mtFire
13                 instigation inAppleCrossed1 apple1.mtExplode
14          end_action
15   end_rule
```

Conforme apresenta o Algoritmo 6, a etapa final consiste na definição das *Rules*. A definição de uma *Rule* é anunciada com a palavra chave ***Rule*** seguida de um nome para a mesma. É importante ressaltar que o nome da *Rule* nesse ponto é opcional, podendo ser omitido.

Basicamente, uma *Rule* é composta por duas partes, que são a *Condition* (expressão lógica) e *Action* (execução). Nessa versão da linguagem, é obrigatória a

utilização de *Subconditions*, mesmo para a composição de expressões simples. Sendo assim, a construção dessa primeira parte é expressa nas linhas 3 a 11. Seguindo o estado da arte do PON, a construção de uma *Condition* necessita de pelo menos uma *Premise*. No caso da utilização de mais de uma *Premise*, estas devem ser aninhadas com conjunções (*and*) ou disjunções (*or*).

Para a definição de *Premises* na linguagem PON, a palavra reservada **Premise** deve ser utilizada, seguida de um nome (opcional) e uma comparação. A comparação é composta por três elementos, o *Attribute* vinculado a uma instância de um *FBE* (*e.g.* apple1, instância de um *FBE* do tipo Apple), o valor de comparação (*e.g.* ==) e o valor a ser comparado (*e.g. true*). Este último pode ser tanto uma constante, quanto um outro *Attribute*.

A segunda parte, que representa a execução da *Rule* é anunciada a partir da palavra chave **Action**. Este bloco consiste no vínculo de instigações a *Methods* definidos em *FBEs*. Conforme apresenta as linhas 12 e 13 do Algoritmo 5, a estrutura de *Instigation*s são compostas pela palavra chave **Instigation** seguida de um nome (opcional) e um método de uma instância particular de um FBE.

O conjunto de padrões de nomenclatura prevê os seguintes prefixos para as entidades PON utilizadas nessa etapa: rl para *Rules*, cd para *Conditions*, sc para *Subconditions*, pr para *Premises*, ac para *Actions* e in para *Instigations*.

O exemplo utilizado para os testes foi baseado na aplicação Mira ao Alvo e o código PON com apenas uma regra é mostrado no Algoritmo 6.

Algoritmo 6: Implementação da aplicação Mira ao Alvo na linguagem PON

```
1   fbe Apple
2         attributes
3               boolean isCrossed false
4         end_attributes
5         methods
6               method mtFire(isCrossed = true)
7         end_methods
8   end_fbe
9   fbe Archer
10        attributes
11              boolean hasFired false
12        end_attributes
13        methods
14              method mtFire(hasFired = true)
15        end_methods
16  end_fbe
17  fbe Controller
18        attributes
19              boolean fire false
```

```
20        end_attributes
21  end_fbe
22  inst
23        Apple apple1
24        Archer archer1
25        Controller controller1
26  end_inst
27  Rule RlTurnOn1
28        condition
29              subcondition A1
30                    premise PrIsCrossed1 apple1.isCrossed == false
31  and
32                    premise PrHasFired1 archer1.hasFired == false
33  and
34                    premise PrFire1 controller1.fire == true
35              end_subcondition
36        end_condition
37        action
38              instigation inFire11 apple1.mtFire();
39              instigation inFire12 archer1.mtFire();
40        end_action
41  end_rule
```

6. IMPLEMENTAÇÃO *FRAMEWORK*

A geração de código *Framework* pelo compilador, basicamente recria os componentes do *framework* de acordo com o código PON criado. Cada elemento do código PON tem a sua correspondência no *Framework.* Neste modelo o objetivo do compilador foi automatizar a criação de código Framework, pois a geração de código apenas com o *Framework* exige certo grau de conhecimento. De fato, para criar um projeto básico em PON apenas com a utilização do *Framework* o desenvolvedor precisa saber a estrutura de arquivos disponíveis (i.e. estrutura do pacote core e conhecimento das camadas da arquitetura do Framework PON) [Banaszewski, 2009]. Com a utilização do código PON e do compilador a geração desse código é feita de forma automática e o desenvolvedor apenas precisa conhecer o código PON.

Para simular a criação de código *Framework* via compilador PON, um programa simples foi gerado, o Mira ao Alvo. Como é possível observar no Algoritmo 3 tem-se a criação de uma *FBE* chamada *Archer*. No Algoritmo 7 temos o código equivalente a *FBE Archer* para o *Framework*. Note que o código gerado já realiza as importações necessárias..

Algoritmo 7: Classe Archer Framework

```
1    #ifndef Archer_H_
2    #define Archer_H_
3    #include “framework/utils/SingleInclude.h”
4    class Archer : public FBE {
```

```
5           public:
6                   Archer();
7                   ~Archer();
8                   //attributes
9                   Boolean * hasFired;
10                  //methods
11                  Method * mtFire;
12  };
13  #endif /*Archer_H_*/
```

No Algoritmo 8 é apresentado o trecho de código PON para criação de uma *Rule*.

Algoritmo 8: Código PON para criação de uma Rule

```
1   Rule RlTurnOn1
2           condition
3                   subcondition A1
4                           premise PrIsCrossed1 apple1.isCrossed == false and
5                           premise PrHasFired1 archer1.hasFired == false and
6                           premise PrFire1 controller.fire == true
7                   end_subcondition
8           end_condition
9           action
10                  instigation inFire11 apple1.mtFire();
11                  instigation inFire12 archer1.mtFire();
12          end_action
13  end_rule
```

Note que a *Rule* é composta por uma série de elementos. Cada elemento da *Rule* é mapeado pelo *Framework* e o código alvo gerado irá criar as instâncias necessárias para compor o programa. O Algoritmo 9 apresenta o código *Framework* gerado que compõe o arquivo principal utilizado para executar o programa. Neste arquivo é possível visualizar as referências para os elementos PON como a *Rule*, as *Premises* e as *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 9: Classe Main.h Framework

```
1   #ifndef MAIN_H_
2   #define MAIN_H_
3   #include <iostream>
4   using std::cout;
5   using std::endl;
6   #include <cstdio>
7   #include “framework/NOPApplication.h”
8   #include “framework/utils/SingleInclude.h”
9   #include “Apple.h”
10  #include “Archer.h”
11  #include “Controller.h”
12  class Main : public NOPApplication {
13  public:
14          Main();
15          virtual ~Main();
16  public:
```

```
17          void initStartApplicationComponents();
18          void initFactBase();
19          void initRules();
20          void initSharedEntities();
21          void codeApplication();
22    public:
23          RuleObject * rlRlTurnOn1;
24          Premise* prPrFire1;
25          Premise* prPrHasFired1;
26          Premise* prPrIsCrossed1;
27          Instigation* ininFire11;
28          Instigation* ininFire12;
29          Apple * apple1;
30          Archer * archer1;
31          Controller * controller1;
32    };
33    #endif
```

O Algoritmo 10 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 8. Como é possível observar o código alvo é capaz de associar cada elemento pertencente à *Rule* de forma correta.

Algoritmo 10: Método initRules Framework

```
1     void Main::initRules() {
2           Scheduler * scheduler = SingletonScheduler::getInstance();
3           Rule(rlRlTurnOn1,scheduler,Condition::CONJUNCTION);
4           rlRlTurnOn1->addPremise(prPrFire1);
5           rlRlTurnOn1->addPremise(prPrHasFired1);
6           rlRlTurnOn1->addPremise(prPrIsCrossed1);
7           rlRlTurnOn1->addInstigation(ininFire11);
8           rlRlTurnOn1->addInstigation(ininFire12);
9           rlRlTurnOn1->end();
10    }
```

7. IMPLEMENTAÇÃO C++

A implementação em C++ foi baseada nos princípios do PON. O código gerado pelo compilador apresenta classes C++ que representam os conceitos PON relativos a *FBE*, *Rule* e *Premise*. O conjunto de *Premises* associados a uma *Rule* específica no código C++ pode ser considerado como uma *Condition*. O objetivo principal de seguir os princípios do PON na geração de código alvo C++ foi garantir o ganho de desempenho que o PON preconiza ao evitar a avaliação de expressões causais desnecessárias, bem como, evitar o alto acoplamento das entidades.

Para analisar o código C++ gerado para a linguagem PON será utilizado como exemplo o código para o programa denominado Mira ao Alvo. O Algoritmo 11 apresenta o trecho de código referente à criação da *FBE* que representa o conceito

do Arqueiro (Archer). Já o Algoritmo 12 apresenta o código correspondente gerado pelo compilador PON para C++ referente ao Arqueiro. Como é possível observar uma classe C++ é gerada para representar este conceito a qual segue o padrão C++ *builder*.

Algoritmo 11: Código PON para Archer

```
1   fbe Archer
2          attributes
3                 boolean hasFired false
4          end_attributes
5          methods
6                 method mtFire (hasFired = true)
7          end_methods
8   end_fbe
```

Algoritmo 12: Código em C++ gerado para a Classe Archer

```
1    #pragma once
2    #include “PrHasFired1.h”
3    class PrHasFired;
4    class Archer
5    {
6           public:
7                  Archer (void);
8                  ~Archer (void);
9                  bool hasFired;
10                 void sethasFired(bool hasFired);
11                 bool gethasFired();
12                 void mtFire();
13                 PrHasFired1 pr25;
14   };
15
16   #include “Archer.h”
17   Archer::Archer(void)
18   {
19          hasFired = NULL;
20   }
21   Archer::~Archer(void){}
22
23   void Archer::sethasFired(bool hasFired) {
24          if(this->hasFired == 0 || this->hasFired != hasFired) {
25                 this->hasFired = hasFired;
26                 pr25.sethasFired(hasFired);
27          }
28   }
29   bool Archer::gethasFired() {
30          Return this->hasFired;
31   }
32   void Archer::mtFire() {
33          This->sethasFired(true);
34   }
```

O Algoritmo 6 nas linhas 27 a 41 apresentam 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 *Premise*s. 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. Para satisfazer o conceito de *Rule*, *Condition*, *SubCondition*, *Premise*, *Action* e Instigation no código C++ algumas classes foram criados. O Algoritmo 13 apresenta a classe C++ criada para atender o conceito de *Rule*. Como é possível observar a classe C++ *Rule* apresenta uma série de referências para todas as *FBE* que serão instigadas quando a *Rule* estiver ativa. Ainda, para cada *Premise* criada no código PON será gerada uma classe *Premise* em C++. Como é demonstrado no Algoritmo 13 a classe *Rule* tem referência para cada *Premise* relacionada.

Toda *Rule* tem um método denominado *isApproved* este método é utilizado pelo processo de inferência para validar se a *Rule* foi satisfeita e deve acionar as *Instigations*. O Algoritmo 14 apresenta o código C++ gerado para uma *Premise*. Neste código é possível observar que a *Premise* apresenta um método chamado *setfire* que é correspondente ao atributo da classe *FBE* que a *Premise* faz referência. Existe um atributo chamado isApproved que é utilizado para armazenar o estado da *Premise* e uma referência para a *Rule* na qual a *Premise* está associada.

Algoritmo 13: Código C++ correspondente a implementação de uma Rule

```
1    #pragma once
2    #include <iostream>
3    using namespace std;
4    class PrFire;
5    class PrHasFired;
6    class PrIsCrossed;
7    class Archer;
8    class Apple;
9    class Controller;
10   class R1TurnOn {
11         public:
12               R1TurnOn(void);
13               ~R1TurnOn(void);
14               bool isApproved();
15               void setPrFire(PrFire * fire);
16               PrFire * a14;
17               void setPrHasFired(PrHasFired * hasFired);
18               PrHasFired * a13;
19               void setPrIsCrossed (PrIsCrossed * isCrossed);
20               PrHasFired * a12;
21               Archer * a5;
22               Apple * a2;
23               Controller * a8;
24   };
25
26   #include “R1TurnOn.h”
27   #include “PrFire.h”
28   #include “PrHasFired.h”
29   #include “PrIsCrossed.h”
30   #include “Archer.h”
```

```
31  #include “Apple.h”
32  #include “Controller.h”
33  RlTurnOn::RlTurnOn(){
34  }
35  RlTurnOn::~RlTurnOn(){
36  }
37  void RlTurnOn::setPrFire(PrFire * a14){
38        this->a14 = a14;
39        this->a14->condition = this;
40  }
41  void RlTurnOn::setPrFire(PrFire * a13){
42        this->a13 = a13;
43        this->a13->condition = this;
44  }
45  void RlTurnOn::setPrFire(PrFire * a12){
46        this->a12 = a12;
47        this->a12->condition = this;
48  }
49  boolRlTurnOn::isApproved(){
50        if(a14->isApproved && a13->isApproved && a12->isApproved){
51              a5->meFire();
52              a2->meFire();
53              a8->meFire();
54              return true;
55        } else {
56              return false;
57        }
58  }
```

Algoritmo 14: Código C++ correspondente a uma Premise

```
1   #pragma once
2   #include “RlTurnOn.h”
3   #include <iostream>
4   using namespace std;
5   class RlTurnOn;
6   class PrHasFired{
7         public:
8               PrHasFired(void);
9               ~PrHasFired(void);
10              RlTurnOn * condition;
11              bool isApproved;
12              void sethasFired(bool hasFired);
13  };
14  #include “PrHasFired.h”
15
16  PrHasFired::PrHasFired() {
17  }
18  PrHasFired::~PrHasFired() {
19  }
20  void PrHasFired::sethasFired(bool hasFired) {
21        if(hasFired == false) {
22              this->isApproved=true;
23              this->condition->isApproved();
24        } else {
25              this->isApproved = false;
26        }
27  }
```

O código C++ gerado apresenta o mesmo conceito de execução do PON. Cada vez que um atributo de uma *FBE* é configurado com um valor específico, diferente do valor atribuído anteriormente, este vai acionar o processo de inferência. Este

valor pode ser atribuído com a utilização do método set que é criado para cada atributo da *FBE*. Este método possui uma referência para cada *Premise* relacionada ao atributo e o novo valor é atribuído a *Premise*. O Algoritmo 12 nas linhas 23 a 28 apresentam o trecho de código responsável por acionar cada *Premise* relacionada ao atributo.

Por sua vez, cada *Premise* ao receber o novo valor do atributo vai verificar se este novo valor consegue satisfazer se sua condição foi aprovada. Caso a condição da *Premise* seja verdadeira, esta vai acionar a *Rule* associada ao chamar o método isApproved. O Algoritmo 13 nas linhas 49 a 55 apresenta o trecho de código que contempla essa operação.

Na *Rule* o método isApproved vai verificar o estado de cada *Premise* associada e caso todas as *Premise*s estejam ativas a *Rule* se tornará ativa. Com a *Rule* ativa as *Instigations* relacionados a *Rule* serão invocadas e o processo de inferência será finalizado. O Algoritmo 15 apresenta o trecho de código referente ao método isApproved da *Rule* RlTurnOn.

Algoritmo 15: Método isApproved da class Rule

```
1    bool RlTurnOn1::isApproved() {
2          if(pr27->isApproved & pr25->isApproved & pr23->isApproved) {
3                apple1->mtFire();
4                archer1->mtFire();
5                return true;
6          } else {
7                return false;
8          }
9    }
```

8. IMPLEMENTAÇÃO C

A implementação em C foi toda desenvolvida de forma a otimizar a execução do código gerado em detrimento do tamanho do mesmo. A geração de código foi pensada com o intuito de chegar mais próximo possível do que seria gerado em *Assembly*, assim as otimizações do GCC foram mínimas ou nenhuma. Para que isso ocorresse, não foram utilizadas no código gerado as seguintes técnicas: passagens de parâmetros, ponteiros e alocação dinâmica de memória. Dessa forma, a tradução do C para o código de máquina (que foi deixada a cargo do Compilador GCC) pôde ser mais direta.

A geração de código é feita de forma a criar vários arquivos independentes com o objetivo de facilitar o entendimento de cada uma das partes individualmente.

Além disso, para cada elemento gerado existe sempre um Header (.h), que contém as definições as estruturas e cabeçalhos de funções e um arquivo (.c) que implementa os procedimentos e as funções pertinentes àquele módulo.

Existem alguns arquivos que são genéricos e em todos os casos, independente da aplicação, eles existem e possuem o mesmo nome. Representam as regras (*Rule*s), as subcondições (*SubConditions*) e premissas (*Premisses*). Todos esses arquivos seguem o mesmo padrão, têm uma estrutura (*struct*) com apenas um campo do tipo short (*isapproved*). Esse recebe um valor quando o elemento que está representando é aprovado (se torna verdadeiro). Todos os elementos do mesmo tipo (Regra, SubCondição ou Premissa) são agrupados no seu respectivo arquivo e declarados como variáveis, independente das *FBE*s que fazem parte. Como as Regras, Subcondições e Premissas são semelhantes, apenas um exemplo ilustrativo, o da estrutura *Rule* e declaração das suas variáveis, são apresentados no Algoritmo 16.

Algoritmo 16: Código C correspondente a uma Rule

```
1    typedef struct RuleType {
2    short isApproved;
3    } Rule;
4    Rule RlTurnOn1, RlTurnOn2, RlTurnOn3;
```

No tocante às *FBE*s, para cada uma delas são criados dois arquivos (um .h e outro .c), eles implementam os atributos agrupando-os na forma de uma estrutura (struct).

Nele é criada uma variável, do tipo daquela *FBE*, para cada instância existente no arquivo original. Dentro de cada arquivo (*FBE*), foi criado um procedimento `setAttribute()` para cada um dos atributos existentes. Além disso, ao invés da sub-rotina receber como parâmetro a instância do atributo que deve ser modificada, as funções foram replicadas para cada instância individualmente.

Por exemplo, uma *FBE* chamada Archer, contém um atributo chamado `hasFired` do tipo `bool` e tem duas instâncias: `Archer1` e `Archer2`, assim o código gerado conterá duas funções que realizarão as mudanças nos atributos, são elas: `setArcher1hasFired(bool pValue)` e `setArcher2hasFired(bool pValue)`. Essa técnica aumenta o tamanho do código gerado, mas antecipa algumas otimizações

que seriam deixadas a cargo do compilador GCC. Isso é realizado a fim de se tentar chegar o mais próximo possível do código que seria gerado em *Assembly*.

O código em C gerado pelo compilador desenvolvido segue uma sequência de atos que é baseada nas ações esperadas pela execução de um programa escrito em PON. É importante destacar que o paralelismo entre as mesmas não foi considerado nem tratado, assim tudo ocorre de forma sequencial. Uma mudança em um atributo é tratada em sua totalidade, ou seja, até finalizar a execução de todas as ações pertinentes a ela, e somente depois disso é que outra mudança será considerada.

Um exemplo ilustrativo do código gerado, baseado na aplicação (Mira ao Alvo), é mostrado no Algoritmo 17.

Algoritmo 17: Código C correspondente a estrutura Apple

```
1    void setapple1isCrossed (bool pValue){
2          if(pValue != apple1.isCrossed){
3                apple1.isCrossed = pValue;
4                funcInstapple1AttisCrossedNotifyPrPrIsCrossed1(){
5          }
6    }
7    void funcInstapple1AttisCrossedNotifyPrPrIsCrossed1(){
8          if(apple1.isCrossed == false){
9                setPrPrIsCrossedisApproved(1);
10               funcInstapple1PrPrIsCrossed1NotifyScA1(){
11         } else setPrPrIsCrossedisApproved(0);
12   }
13   void funcInstapple1PrPrIsCrossed1NotifyScA1(){
14         if(PrFire1.isApproved & PrHasFired1.isApproved &
15               PrIsCrossed1.isApproved){
16                     setScA1isApproved(1);
17                     funcInstapple1ScA1NotifyRlRlTurnOn1();
18         } else setScA1isApproved(0);
19   }
20   void funcInstapple1ScA1NotifyRlRlTurnOn1(){
21         if(A1.isApproved){
22               setRlRlTurnOnisApproved(1);
23               instapple1meFire();
24               instarcher1meFire();
25               instcontroller1meFire();
26         } else setRlRlTurnOn1isApproved(0);
27   }
28   void instapple1meFire(){
29         Setapple1isCrossed (true);
30   }
```

O fluxo de execução do programa se dá da seguinte forma: A cada mudança de um *Attribute*, todas as premissas interessadas são avaliadas. No caso de serem aprovadas, configuram a variável (`isApproved`) de sua instância como verdadeira. Em seguida, a subcondição que contém aquela premissa é notificada. Uma avaliação da *bit* a *bit* das variáveis que indicam a aprovação das premissas

existentes naquela subcondição é realizada para verificar se ela é aprovada (linha 14 do código). No caso dela ser aprovada, a regra interessada é notificada. Em seguida, a função que avalia a regra verifica se todas as subcondições existentes nela estão válidas (linha 20), se sim, as ações indicadas pela regra são executadas finalizando assim uma etapa. Após essa execução, pode ocorrer outra mudança de atributo que faz com que a cadeia seja disparada novamente.

9. TESTES DE TEMPOS DE PROCESSAMENTO E USO DE MEMÓRIA

A fim de fazer comparações de eficiência entre os três implementações supracitados, um estudo comparativo de desempenho foi realizado para três cenários: 1, 10 e 100 *Rules*,sendo que para cada um destes cenários foram testadas 1, 10, 100, 1.000, 10.000 e 100.000 iterações. Os testes foram realizados em duas arquiteturas de sistemas operacionais, x86 e x64 bit.

De um modo geral, as entidades miras e as entidades alvos são representadas respectivamente por arqueiros e maçãs, sendo que há uma maçã para cada arqueiro. Em termos de implementação, cada arqueiro e maçã é representado por um objeto (i.e. um elemento da base de fatos), os quais interagem de acordo com a validação de expressões causais pertinentes. Estas expressões causais fazem menção aos atributos/estados destes objetos [Banazewski, 2010].

Neste contexto, cada arqueiro e cada maçã recebem um identificador numérico, sendo que um arqueiro somente pode flechar uma maçã que apresente o identificador numérico correspondente ao seu. Ainda, os arqueiros também apresentam um atributo que denota os seus estados de pronto (i.e. status) para agir sobre o cenário (i.e. flechar a respectiva maçã).

Em relação às maçãs, estas também apresentam um atributo que denota os seus estados de pronto (i.e. status). Ainda, estas apresentam um atributo que explicita se a mesma já foi perfurada por uma flecha ou ainda não (i.e. isCrossed). Também, as maçãs apresentam um atributo que se refere a sua coloração (i.e. color), uma vez que as maçãs podem se apresentar em duas diferentes cores: vermelha ou verde. Com esta fartura de atributos, as maçãs representam um papel importante na concepção de cenários devido à possibilidade de variar os seus estados na composição de diferentes expressões causais.

Em linguagem C a estrutura *Archer* representa os arqueiros, a estrutura Apple representa as maçãs, a estrutura *Controller* representa o controlador, que define quando o arqueiro deverá disparar contra a maçã. De maneira semelhante, em C++ as classes *Archer*, *Apple* e *Controller* representam as *FBE*s do sistema a serem testadas. O código gerado para o paradigma framework PON, utilizou-se a estrutura atual do framework, [Valença et. al.] com todas suas especificidades e classes requeridas para seu funcionamento (*Archer*, *Apple* e *Controller*).

Os tempos foram considerados apenas ao final da execução de todo programa (após todas iterações). Para aferir os testes foram utilizados ambientes em Linux, Debian 7.1 (x86 e 64bit) em hardware contendo 4 GB RAM, Intel Core 2 Quad CPU Q6600 @ 2.40 GHz x 4. Todos os testes foram realizados em ambiente livre de preempção (Linux monousuário/modo de recuperação), os códigos foram compilados com GCC (linguagem C) e g++ (linguagem C++ e framework). Ao total são 144 casos de testes.

9.1 USO DE MEMÓRIA

Observou-se que a execução do conjunto de testesno sistema em 64 bits utilizou aproximadamente quatro vezes mais memória que os executados em 32 bits. Também foi constatado que para cada cenário de testes (com 1, 10 ou 100 *Rules*) os resultados aferidos não alteravam com relação ao número de iterações, ou seja, sempre eram constantes os usos de memória independentemente do número de iterações testado.

9.2 PRIMEIRO CENÁRIO: 1 *RULE* EM 32 E 64 BITS

Para o conjunto de testes em 32 bits, os resultados da linguagem C em comparação com os resultados do framework foram em média 1,84% dos resultados aferidos em framework, ou seja, em linguagem C executou-se em aproximadamente 50 vezes menos tempo. Quando framework comparado com linguagem C++, os resultados em linguagem C++ obtiveram uma média de 1,31% do tempo de execução do framework, ou seja, framework expendeu 76 vezes mais tempo de processamento que C++. Foi constatado um desempenho superior (menor tempo de processamento) nos resultados obtidos para linguagem C++ em comparação com

linguagem C. Tanto C quando C++ conseguiram executar com muito menos tempo de processamento que framework.

Para o conjunto de testes em 64 bits foi constatado um equilíbrio da média dos tempos de execução entre C e C++, sendo que C++ consumiu ligeiramente menos processamento que C. Quando comparados tempos da linguagem C com framework, constatou-se que linguagem C processou com 1,89% do tempo de framework, aproximadamente 50 vezes mais eficiente. Quando C++ foi comparado com framework obteve uma média de 1,7% do tempo de processamento que framework (C 58% mais eficiente que framework).

De modo geral, em 32 e 64 bits, com este cenário, a execução em linguagem C expendeu mais tempo de execução (menor desempenho) em comparação com C++. Dos 12 casos deste conjunto de teste, em sete (58,33%) a linguagem C apresentou menor desempenho, sendo que cinco casos que mais expenderam tempo foram conferidos em 32 bits, houve dois empates de tempo de execução em 64 bits.

Se pôde observar que os resultados de tempos de execução para framework foram menos eficientes aos relacionados a C e C++. A linguagem C++ teve melhor desempenho que C e framework.

9.3 SEGUNDO CENÁRIO: 10 *RULES* EM 32 E 64 BITS

Para o conjunto de testes em 32 bits, os resultados da linguagem C em comparação com os resultados do framework foram em média 2,13% dos resultados aferidos em framework, ou seja, em linguagem C computou-se em aproximadamente 45 vezes menos tempo. Quando framework comparado com linguagem C++, os resultados em linguagem C++ obtiveram uma média de 4,49% do tempo de execução do framework, ou seja, framework expendeu mais de 20 vezes tempo de processamento que C++. Tanto C quando C++ conseguiram executar com muito menos tempo de processamento que framework, sendo que C conseguiu em média metade do tempo de C++.

Para o conjunto de testes em 64 bits foi constatada uma disparidade maior dos tempos de execução entre C e C++, sendo que C++ consumiu três vezes mais processamento que C. Quando a linguagem C foi comparada com framework, constatou-se que esta processou com 2,15% do tempo de framework, aproximadamente 45 vezes mais eficiente. Quando C++ foi comparado com

framework obteve uma média de 5,99% do tempo de processamento que framework, aproximadamente 15 vezes mais eficiente de framework.

Em todos os casos de testes para este cenário entre C e C++, a linguagem C exigiu menos tempo de processamento que a linguagem C++. Sendo a linguagem C mais eficiente para todos esses conjuntos de testes.

Os tempos de execução dos conjuntos de testes para framework foram sempre superiores aos obtidos em C e C++. Portanto C e C++ são muito superiores em ganho de desempenho para o cenário com 10 *Rules*, notando-se um ganho superior em tempo de processamento quando todos os casos executados com linguagem C.

9.4 TERCEIRO CENÁRIO: 100 *RULES* EM 32 E 64 BITS

Para o conjunto de testes em 32 bits, os resultados da linguagem C em comparação com os resultados do framework foram em média 7,22% dos resultados aferidos em framework, ou seja, em linguagem C computou-se em aproximadamente 13 vezes menos tempo que framework. Quando framework comparado com linguagem C++, os resultados em linguagem C++ obtiveram uma média de 47% do tempo de execução do framework, ou seja, framework expendeu duas vezes a mais tempo de processamento que C++. A linguagem C conseguiu executar com muito menos tempo que C++ e framework.

Para o conjunto de testes em 64 bits a linguagem C++ aferiu tempos próximos a *Framework* e até houve um caso com pior resultado. Para o caso com 10 iterações C++ expendeu 14% a mais de tempo de execução do que *Framework* (Tabela 1), para casos a partir de 100 iterações constatou-se um empate técnico em tempos de execução entre os paradigmas C++ e *Framework* (Figura 4). Os tempos de execução dos conjuntos de testes para framework e C++ se apresentaram muito próximos, principalmente em 64 bits.

A linguagem C conseguiu tempos muito superiores aos relativos a C++ e *framework* (Figura 4), sendo, portanto a linguagem mais eficiente dentre as três.

Tabela 1 - Comparação de tempos de processamento (ms) entre C++ e Framework em 64 bits

Número de iterações	C++	Framework	% tempo de processamento de C++ em relação a Framework
1	1,795166	2,310791	77,68%
10	9,075928	7,952881	114,12%
100	63,03003	63,177	99,76%
1.000	581,9861	598,553	97,23%
10.000	5805,216	5961,658	97,37%
100.000	58024,92	59018,43	98,31%

9.5 RESULTADOS GERAIS EM LINGUAGEM C

Os códigos executados para a linguagem C tiveram desempenho superior relacionado a tempos de processamento comparados aos executados em C++ e Framework na grande maioria dos testes (91,66% dos 144 casos) nas duas plataformas (32 e 64 bits), e em 100% dos casos comparados com framework, tanto em 32 quanto em 64 bits. O tempo de execução para linguagem C comparada com C++ apenas apresentou menor desempenho para alguns casos contendo uma *Rule*.

Em todos os casos, os resultados em linguagem C utilizaram menos memória quando comparados com C++ e framework. Constatou-se que o consumo de memória em linguagem C é aproximadamente três vezes menor em relação a C++ e Framework principalmente por não haver alocação dinâmica de memória, sendo utilizadas apenas instruções simples.

9.6 RESULTADOS GERAIS EM LINGUAGEM C++

O código gerado para linguagem C++ foi mais eficiente (utilizou menos processamento) que o código executado para framework na grande maioria dos casos (41,66%). Houve quatro casos (33%) em que os resultados para C++ empataram com resultados do framework (64 bits com 100 *Rules* e de 100 a 100.000 iterações) e um caso (8,3%) que C++ perde para framework (64 bits, 100 *Rules* com 10 iterações). Em todos os casos, os resultados obtidos em linguagem C++ utilizaram pouco menos de memória quando comparados com framework (média de aproximadamente 2,4% menos RAM utilizada).

9.7 RESULTADOS GERAIS EM LINGUAGEM PARA O *FRAMEWORK*

Em todos os conjuntos de testes, os resultados com *framework* expendiam muito mais processamento em comparação com código executado para linguagem C, e na maioria dos casos quando comparados com código executado para C++. Constatou-se que com mais regras (i. e. 100 *Rules*) os resultados de tempo de processamento para framework tendem a estabilizar no sistema em 64 bits, principalmente quando comparados com C++, onde houve casos de empates e a até um caso melhor (Figura 4).

De maneira geral, esses resultados comprovam que os códigos gerados para C e C++ executam com menos exigência de processamento e memória que *framework* para casos de até 100 *Rules* em 64 bits.

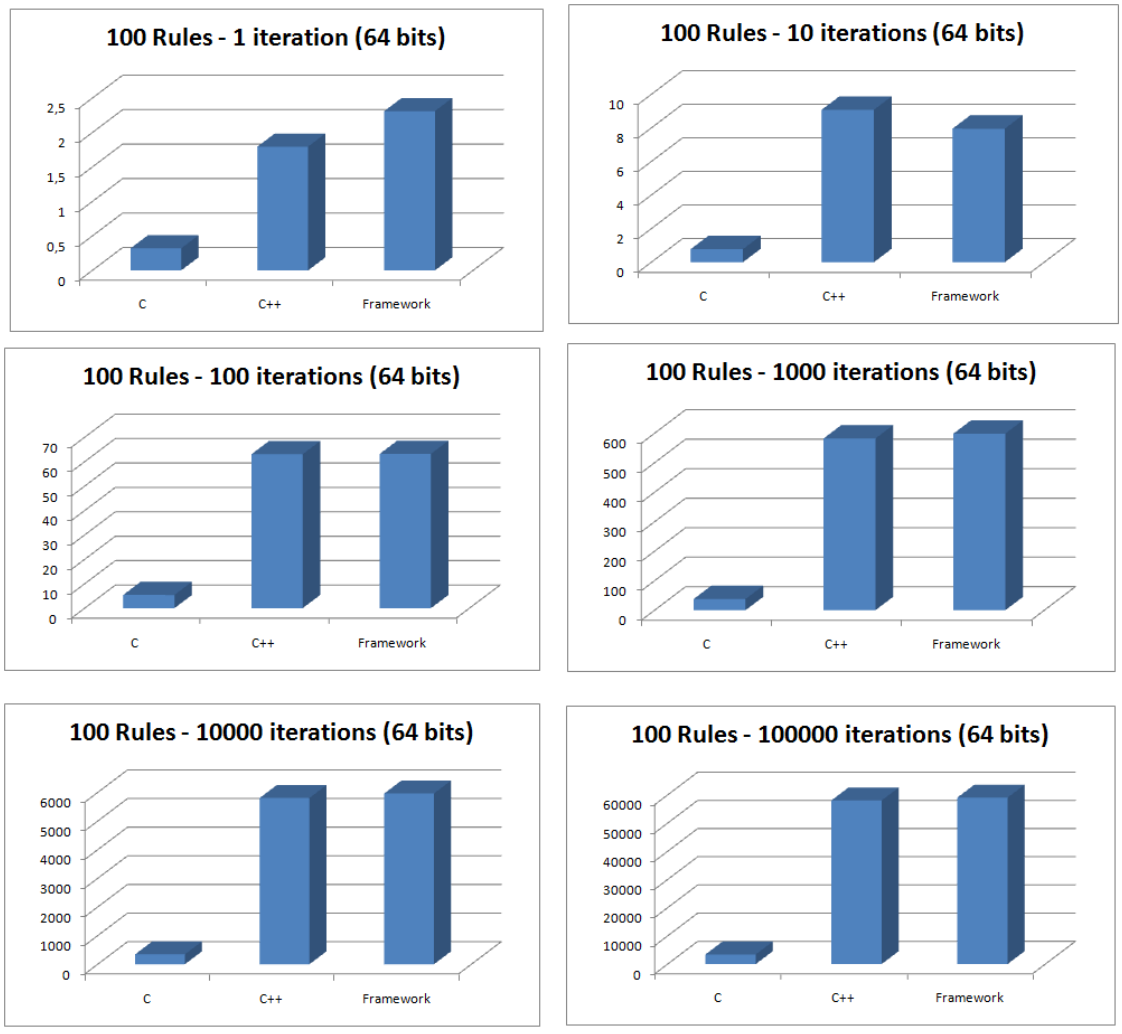


Figura 4: Comparação de tempos de execução C, C++e Framework para 64 bits

A Figura 4apresenta único o cenário mais desfavorável para C++ em relação a *framework*, que foi com 100 *Rules* em 64 bits.

9.8 COMPARAÇÕES GERAIS PARA TODOS OS CASOS DE TESTES

Com base em todos os casos de testes, a Tabela 1 apresenta a comparações gerais de tempos de processamento entre os três paradigmas.

Tabela 1: Comparação de tempo de processamento entre os paradigmas para 32 e 64 bits

Primeiro comparado com segundo	Ganho % dos casos	Empate % dos casos	Perda % dos casos
C e C++	91,66	1,041	7,291
C e *Framework*	100	0	0
C++ e C	7,291	1,041	91,66
C++ e *Framework*	94,79	4,166	1,041
Framework e C	0	0	100
Framework e C++	1,041	4,166	94,79

10. CONCLUSÕES

De maneira geral, os resultados se mostraram bastante satisfatórios. Os esforços para a criação de uma linguagem própria para o PON e a implementação de versões mais otimizadas de geradores de código final sob os princípios do PON, confirmam o potencial da compilação para que o PON atinja suas reais capacidades principalmente em termos de desempenho.

É importante salientar que este trabalho foi resultado de um esforço realizado durante a disciplina de Linguagens e Compiladores ministrada pelo Prof. Dr. Jean Marcelo Simão e o Prof. Dr. João Alberto Fabro. Ainda, este trabalho envolveu a participação de quatro alunos sendo: Adriano Francisco Ronszcka, Clayton Kossoski, Cleverson Avelino Ferreira e Priscila Ap. de Moraes Ioris. Todos os envolvidos foram responsáveis pela definição da linguagem e criação dos módulos de análise léxica e análise sintática/semântica. O aluno Adriano Francisco Ronszcka ficou responsável pela criação dos componentes PON responsáveis pela representação da tabela de símbolos. O aluno Cleverson Avelino Ferreira ficou responsável pela criação do módulo de geração de código em C++ sob o viés do PON e código *Framework* Otimizado. A aluna Priscila Ap. de Moraes Ioris ficou responsável pela criação do módulo de geração de código em C sob o viés do PON, e finalmente, o aluno Clayton Kossoski ficou responsável pelos testes dos resultados alcançados com o compilador.

De acordo com os resultados apresentados, em todos os conjuntos de testes os resultados obtidos com framework consumiram mais memória quando comparados com C e C++. Com base em todos os testes de cada paradigma, em 32 e 64 bits, foi extraído o percentual geral em que cada paradigma ganhou, perdeu ou empatou em tempo de execução com outro paradigma comparado.

A versão do compilador em C apresentou melhores resultados nos termos comparados. Basicamente, nessa versão foram adaptados os conceitos do PON para uma implementação menos onerosa (eliminação de classes e estruturas de dados) baseando a implementação no paradigma estruturado.

Ainda, vislumbra-se a continuação desse trabalho em dissertações e teses, no âmbito de prosseguir com as otimizações dos compiladores em C e C++ no tocante a otimização de recursos computacionais, utilizando-se estruturas menos onerosas de processamento (e.g.estruturas *go to*, passagem de parâmetro por referência e otimização de fluxo),viabilizando a existência de outras versões mais eficientes, com códigos otimizados em *Assembly* e linguagens de mais baixo nível.

Referências do Apêndice A

Aho, A., Sethi, R., e Ullman, J. (2006). Compilers: Principles, Techniques e Tools. Addison Wesley, *second edition*.

Auerbach, J., Bacon, D., Burcea, I., Cheng, P., Fink, S., Rabbah, R., e Shukla, S. (2012). *A compiler e runtime for heterogeneous computing. In Design Automation Conference* (DAC), 2012 49th ACM/EDAC/IEEE, *pages* 271–276.

Banaszewski, R. F. (2009). Paradigma orientado a notificações: Avanços e comparações. Dissertação de mestrado, Universidade Tecnológica Federal do Paraná.

Banaszewski, R. F., Stadzisz, P. C., Tacla, C. A., e Simão, J. M. (2007). *Notification oriented paradigm (nop): A software development approach based on artificial intelligence concepts. Proceedings of the VI Congress of LAPTEC.*

Banazewski, R. F. Paradigma Orientado a Notificações: Avanços e Comparações. Dissertação de Mestrado, CPGEI/UTFPR. Curitiba, 2009.

Barve, A. e Joshi, B. (2012). *A parallel lexical analyzer for multi-core machines. In Software Engineering (CONSEG)*, 2012 *CSI Sixth International Conference on, pages* 1–3.

Brookshear, J. G. (2006). Coputer Science: An Overview. Addison Wesley, 9 ed edition. Cardoso, J. M. P. e Diniz, P. C. (2008). *Compilation Techniques for Reconfigurable Architectures. Springer Publishing Company, Incorporated.*

Cardoso, J. M. P., Teixeira, J., Alves, J. C., Nobre, R., Diniz, P. C., Coutinho, J. G. F., e Luk, W. (2012). *Specifying compiler strategies for fpga-based systems. In FieldProgrammable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on, pages* 192–199.

Forgy, C. L. (1982). *Rete: A fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence,* 19(1):17 – 37.

Gabrielli, M. e Martini, S. (2010). *Programming Languages: Principles e Paradigms.*

Undergraduate Topics in Computer Science. Springer, 1st edition edition.

Gligoric, M., Mehlitz, P., e Marinov, D. (2012). *X10x: Model checking a new programming language with an "old"model checker. In Software Testing, Verification e Validation (ICST), 2012 IEEE Fifth International Conference on, pages* 11–20.

Hill, E. F. (2003). *Jess in Action: Java Rule-Based Systems. Manning Publications Co., Greenwich, CT, USA.*

Huong, G. N. T. e Kim, S. W. (2011). *Gcc2verilog compiler toolset for complete translation of c programming language into verilog hdl. ETRI Journal*, 33(5).

LEWIS, H. R. PAPADIMITRION, C. H. Elementos de Teoria da Computação. 2.ed. Porto Alegre, Bookman, 2000.

Oracle (2013). Java language e virtual machine specifications.

Pharr, M. e Mark, W. (2012*). ispc: A spmd compiler for high-performance cpu programming. In Innovative Parallel Computing (InPar),* 2012, pages 1–13.

Pornsoongsong, W. e Chongstitvatana, P. (2012*). A parallel compiler for multi-core microcontrollers. In Digital Information e Communication Technology e it's Applications (DICTAP), 2012 Second International Conference on, pages* 373–377.

Ronszcka, A. F. (2012). Aprendizado com o paradigma orientado a notificações. Dissertação de Mestrado, Universidade Tecnológica Federal do Paraná.

Ronszcka, A. F., Belmonte, D. L., Valença, G. Z., Batista, M. V., Linhares, R. R., Tacla, Stadzisz, P. C., Simão, J. M. (2011). Comparações quantitativas e qualitativas entre o paradigma orientado a objetos e o paradigma orientado a notificações sobre um simulador de jogo. In III Congreso Internacional de Computación y Telecomunicaciones - COMTEL 2011, Lima - Peru.

Scott, M. L. (2000). *Programming Language Pragmatics. Morgan Kaufmann Publishers Inc, San Francisco, CA, USA.*

SIMÃO, J. M. A *Contribution to the Development of a HMS Simulation Tool and Proposition of a Meta-Model for Holonic Control.* Tese de Doutorado, UTFPR, CPGEI, Curitiba, Brazil, 2005.

SIMÃO, J. M. ; Belmonte, D. L. ; Ronszcka, A. F. ; Linhares, R. R. ; VALENÇA, G. Z. ; BANASZEWSKI, R. F. ; Fabro J. A. ; TACLA, C. A. ; STADZISZ, P. C. ; Batista, M. V. . *Notification Oriented and Object Oriented Paradigms comparison via Sale System. Journal of Software Engineering and Applications (print),* v. 5, p. 695-710, 2012.

Simão, J. M. e Stadzisz, P. C. (2008). 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 e a Agência de Inovação/UTFPR em 2007. No. INPI Provisório 015080004262. No INPI Efetivo PI0805518-1. Patente submetida ao INPI.

Simão, J. M. e Stadzisz, P. C. (2009). *Inference process based on notifications: The kernel of a holonic inference meta-model applied to control issues. IEEE Transactions on Systems Man e Cybernetics. Part A, Systems and Humans,* (39):238–250. *Digital Object Identifier* 10.1109/TSMCA.2008.20066371.

Simão, J. M., Banaszewski, R., Tacla, C., e Stadzisz, P. (2012). *Notification oriented paradigm (nop) e imperative paradigm: A comparative study. Journal of Software Engineering e Applications,* 5(6):402–416.

Valença, G. Z. Contribuição Para A Materialização Do Paradigma Orientado A Notificações (PON). Dissertação (Mestrado em Programa de Pós-graduação em Computação Aplicada) - Universidade Tecnológica Federal do Paraná. PPGCA/UTFPR, Curitiba, 2012.

Vanlença, G. Z., Banaszewski, R. F., Ronszcka, A. F., Batista, M. V., Linhares, R. R., A., F. J., Stadzisz, P. C., e Simão, J. M. (2011). Framework PON, avanços e comparações. In III Simpósio de Computação Aplicada, Passo Fundo – RS.

APÊNDICE B - BNF da linguagem PON

Neste apêndice se encontra a especificação da linguagem PON segundo a Backus-Naur Form (BNF). Uma linguagem de programação é definida por meio de um formalismo denominado gramática ou gramática livre de contexto. Essa gramática define a estrutura das regras gramaticais definidas para a linguagem e pode ser especificada segundo uma BNF. Salienta-se que a BNF evoluiu da versão prototipal apresentada no apêndice A com a inclusão das propriedades do PON descritas na Subseção 1.1.3. Este apêndice foi referenciado na Seção 3.1 do presente trabalho.

```
<PROGRAM>           ::= <fbes><inst><strategy><rules><main>
                    | <fbes><inst><strategy><rules>

<inst>              ::= <INST><declarations><END_INST>

<strategy>          ::= <STRATEGY><estrategy_declaration><END_STRATEGY>

<estrategy_declaration>    ::= <NO_ONE>
                           | <BREADTH>
                           | <DEPTH>

<declarations>      ::= <declaration>
                    | <declaration><declarations>

<declaration>::= <type><ids>

<ids>               ::= <id>
                    | <id><COMMA><ids>

<rules>             ::= <rule>
                    | <rule><rules>

<rule>              ::= <RULE><rule_body><END_RULE>
                    | <RULE><id><rule_body><END_RULE>
                    | <RULE><depends><id><rule_body><END_RULE>
                    | <RULE><id><depends><id><rule_body><END_RULE>

<depends>           ::= <DEPENDS>

<rule_body>         ::= <decl_condition><decl_action>
                    | <decl_properties><decl_condition><decl_action>

<decl_properties>   ::= <PROPERTIES><properties_body><END_PROPERTIES>


<properties_body>   ::= <properties_type><value>
```

```
                    | <properties_body><properties_type><value>

<properties_type>   ::= <PRIORITY>
                    | <KEEPER>

<decl_condition>    ::= <CONDITION><condition_body><END_CONDITION>
                    | <CONDITION><id <condition_body><END_CONDITION>

<condition_body>    ::= <subcondition><operator><condition_body>
                    | <subcondition>

<operator>          ::= <AND>
                    | <OR>

<subcondition>      ::= <SUBCONDITION><id><subcondition_body><END_SUBCONDITION>


<subcondition_body> ::= <premise><AND><subcondition_body>
                          | <premise>

<premise>           ::= <PREMISE><exp>
                    | <PREMISE><id><exp>
                    | <PREMISE><IMP><exp>
                    | <PREMISE><IMP><id><exp>

<exp>               ::= <fator><comp><fator>

<comp>              ::= <EQ>
                    | <NE>
                    | <LT>
                    | <GT>
                    | <LE>
                    | <GE>

<fator>             ::= <id>
                    | <NUMBER>
                    | <boolean>

<boolean>           ::= <TRUE>
                    | <FALSE>

<decl_action>       ::= <ACTION><action_body><END_ACTION>
                    | <ACTION><id><action_body><END_ACTION>

<action_body>       ::= <action_elements><action_body>
                    | <action_elements>

<action_elements>   ::= <instigation>
                    | <method_use>
                    | <exp><SEMICOLON>

<instigation>       ::= <INSTIGATION><method_use>
                    | <INSTIGATION><id><method_use>
```

```
<method_use>        ::= <id><LP><RP><SEMICOLON>

<id>                ::= <ID>
                    | <ID><POINT><ID>

<fbes>              ::= <fbe>
                    | <fbe><fbes>

<fbe>               ::= <FBE><fbe_body><END_FBE>
                    | <FBE><id><fbe_body><END_FBE>

<fbe_body>          ::= <decl_attributes><decl_methods>
                    | <decl_attributes>

<decl_attributes>   ::= <ATTRIBUTES><attributes><END_ATTRIBUTES>

<attributes>        ::= <attributes_body>
                    | <attributes_body><attributes>

<attributes_body>   ::= <type><id><value>
                    | <type><id>

<type>              ::= <BOOLEAN>
                    | <INTEGER>
                    | <PFLOAT>
                    | <STRING>
                    | <id>

<value>             ::= <NUMBER>
                    | <boolean>
                    | <id>
                    | <FLOATVALUE>
                    | <STRINGVALUE>

<decl_methods>      ::= <METHODS><methods><END_METHODS>

<methods>           ::= <method_body>
                    | <method_body><methods>

<method_body>       ::= <METHOD><id><LP><id><ASSIGN><id><method_operator><value><RP>
                    | <METHOD><id><LP><id><ASSIGN><id><method_operator><id><RP>
                    | <METHOD><id><LP><id><ASSIGN><value><RP>
                    | <METHOD><id><LP><id><ASSIGN><id><RP>
                    |   <METHOD><id><LP><id><ASSIGN><id><method_operator><value><RP>
<INNER_CODE>
                    |
<METHOD><id><LP><id><ASSIGN><id><method_operator><id><RP><INNER_CODE>
                    | <METHOD><id><LP><id><ASSIGN><value><RP><INNER_CODE>
                    | <METHOD><id><LP><id><ASSIGN><id><RP><INNER_CODE>


<method_operator>   ::= <PLUS>
                    | <MINUS>
                    | <MULT>
                    | <DIV>
```

```
<main>              ::= <MAIN><INNER_CODE>
```

APÊNDICE C - Arquivo de configuração do Flex/Lex para o PON

Neste apêndice se encontra o arquivo de configuração da ferramenta Flex/Lex criado para gerar o módulo de análise léxica do compilador PON. Basicamente, no arquivo de configuração da ferramenta Flex/Lex estão presentes os *tokens* definidos para a linguagem. Tais *tokens* são capturados durante a análise léxica com a utilização de expressões regulares. Este arquivo foi referenciado na Subseção 3.2.1 do presente trabalho.

```
%{
     #include <stdio.h>
     #include "bison_pon.tab.h"
     using namespace std;

     int line_num = 1;
%}

%%

rule                              return RULE;
condition                         return CONDITION;
action                            return ACTION;
premise                           return PREMISE;
instigation                       return INSTIGATION;
subcondition                      return SUBCONDITION;
fbe                               return FBE;
attributes                        return ATTRIBUTES;
methods                           return METHODS;
method                            return METHOD;
inst                              return INST;
strategy                          return STRATEGY;
properties                        return PROPERTIES;
imp                               return IMP;
depends                           return DEPENDS;
main                              return MAIN;

end_rule                          return END_RULE;
end_condition                     return END_CONDITION;
end_action                        return END_ACTION;
```

```
end_subcondition                return END_SUBCONDITION;
end_fbe                         return END_FBE;
end_attributes                  return END_ATTRIBUTES;
end_methods                     return END_METHODS;
end_inst                        return END_INST;
end_strategy                    return END_STRATEGY;
end_properties                  return END_PROPERTIES;

integer                         return INTEGER;
boolean                         return BOOLEAN;
pfloat                          return PFLOAT;
string                          return STRING;
char                            return CHAR;
priority                        return PRIORITY;
keeper                          return KEEPER;

no_one                          return NO_ONE;
breadth                         return BREADTH;
depth                           return DEPTH;

and                             return AND;
or                              return OR;
true                            return TRUE;
false                           return FALSE;

"("                             return LP;
")"                             return RP;
"{"                             return LB;
"}"                             return RB;
"["                             return LC;
"]"                             return RC;

"="                             return ASSIGN;

"=="                            return EQ;
"!="                            return NE;
"<"                             return LT;
">"                             return GT;
"<="                            return LE;
">="                            return GE;
```

```
";"                                 return SEMICOLON;
","                                 return COMMA;

"+"                                 return PLUS;
"-"                                 return MINUS;
"*"                                 return MULT;
"/"                                 return DIV;

"."                                 return POINT;

[0-9]+                yylval.sval=strdup(yytext); return NUMBER;

[-+]?[0-9]*\.?[0-9]+  yylval.sval=strdup(yytext); return FLOATVALUE;

\".*\"          yylval.sval=strdup(yytext); return STRINGVALUE;

"//".*          ;


"/*"(.|\n)*"*/"  ;

[a-zA-Z\_]([a-zA-Z0-9\_])*  {
                                    yylval.sval = strdup(yytext);
                                    return ID;
                            }

[\t\f ""]                   ;

\n                          line_num++;

.    {
     fprintf (stderr, "'%c' (0%o): Caractere ilegal na linha %d\n",
     yytext[0], yytext[0], line_num);
}

"{"(.|\n)*"}"    yylval.sval=strdup(yytext); return INNER_CODE;

%%

int yywrap()
{
```

```
    return 1;
}
```

APÊNDICE D - Arquivo de configuração do Bison para o PON

Neste apêndice se encontra o arquivo de configuração do Bison criado para gerar o módulo de análise sintática do compilador PON. Basicamente, no arquivo de configuração do Bison estão presentes os *tokens* definidos para a linguagem bem como a gramática livre de contexto. Este arquivo foi referenciado na Subseção 3.2.1 do presente trabalho.

```
%{
	#include <string.h>
	#include "Entity.h"
	#include "CCompiler.h"
	#include "FrameworkCompiler.h"
	#include "CPPCompiler.h"

	using namespace std;

	int yylex(void);
	void yyerror(const char *s);

	class token_id {

		public:
			const char *s;


	};

	std::vector<token_id *> ids;
	std::vector<Entity *> conditions;
	std::vector<Entity *> instigations;
	std::vector<Method *> methods;
	std::vector<Attribute *> attributes;
	std::vector<std::string> idList;

	extern int line_num;

	Compiler *compiler = NULL;

	typedef struct {
		char leftId[100];
		int op;
```

```
                char rightId[100];
        } PremiseCompType;

        PremiseCompType premiseComp, method;

        typedef struct {
                char leftId[100], rightId[100];
        } MethodType;

        MethodType methodDeclaration, instantiationDeclaration;

        typedef struct {
                char id[100]; std::vector<Entity *> *conditions;
        } SubConditionType;

        SubConditionType    subConditionType[10000];//TODO    DEIXAR    DINAMICO
CONFORME OUTROS ELEMENTOS
        int subCount = 0;
        int priority = 0;
        int keeper = 0;

%}

%union
{
int ival;
float fval;
char *sval;
void *pval;
}

%start PROGRAM

%token RULE
%token CONDITION
%token ACTION
%token PREMISE
%token INSTIGATION
%token SUBCONDITION
%token FBE
%token ATTRIBUTES
%token METHODS
%token METHOD
%token INST
%token STRATEGY
%token PROPERTIES
```

```
%token IMP
%token DEPENDS

%token END_RULE
%token END_CONDITION
%token END_ACTION
%token END_SUBCONDITION
%token END_FBE
%token END_ATTRIBUTES
%token END_METHODS
%token END_INST
%token END_STRATEGY
%token END_PROPERTIES

%token AND <sval>
%token OR <sval>

%token LP
%token RP
%token LB
%token RB
%token LC
%token RC

%token ASSIGN

%token EQ
%token NE
%token LT
%token GT
%token LE
%token GE

%token SEMICOLON
%token COMMA

%token <sval> PLUS
%token <sval> MINUS
%token <sval> MULT
%token <sval> DIV

%token POINT

%token INTEGER     // criar esses dois
%token PFLOAT      // pra separar melhor
%token STRING
```

```
%token BOOLEAN
%token CHAR

%token PRIORITY
%token KEEPER

%token NO_ONE
%token BREADTH
%token DEPTH

%token <sval> TRUE
%token <sval> FALSE

%token <sval> NUMBER
%token <sval> FLOATVALUE
%token <sval> STRINGVALUE
%token <sval> ID

%token COMMENT

%token INNER_CODE

%token MAIN

%type <sval> id
%type <sval> boolean

%type <pval> exp

%type <ival> comp

%type <sval> fator

%type <sval> method_use

%type <sval> operator

%type <pval> subcondition

%type <pval> premise

%type <pval> subcondition_body

%type <pval> type

%type <sval> value
```

```
%type <sval> method_operator

%type <ival> BOOLEAN

%type <ival> INTEGER

%type <ival> PFLOAT

%type <ival> STRING

%type <ival> properties_type

%type <ival> PRIORITY

%type <ival> KEEPER

%type <sval> INNER_CODE

%%

PROGRAM            : fbes inst strategy rules main {compiler->assemble();}
                   | fbes inst strategy rules {compiler->assemble();}
                   ;

inst               : INST declarations END_INST
                   ;

strategy           : STRATEGY estrategy_declaration END_STRATEGY
                   ;

estrategy_declaration    : NO_ONE { compiler->strategy = 0; }
                         | BREADTH { compiler->strategy = 1; }
                         | DEPTH { compiler->strategy = 2; }
                         ;


declarations             : declaration
                         | declaration declarations
                         ;

declaration              :           type           ids           {compiler-
>createInstantiation((char*)$1, idList); idList.clear();}
                         ;
```

```
ids                     : id {idList.push_back($1);}
                        | id COMMA ids {idList.push_back($1);}
                        ;

rules                   : rule
                        | rule rules
                        ;

rule                    : RULE rule_body END_RULE { compiler->createRule("",conditions, instigations, priority, keeper, false, ""); conditions.clear(); instigations.clear(); priority = 0; keeper = 0;}
                        | RULE id rule_body END_RULE { compiler->createRule($2,conditions, instigations, priority, keeper, false, ""); conditions.clear(); instigations.clear(); priority = 0; keeper = 0;}
                        | RULE depends id rule_body END_RULE { compiler->createRule("",conditions, instigations, priority, keeper, true, string($3)); conditions.clear(); instigations.clear(); priority = 0; keeper = 0;}
                        | RULE id depends id rule_body END_RULE { compiler->createRule($2,conditions, instigations, priority, keeper, true, string($4)); conditions.clear(); instigations.clear(); priority = 0; keeper = 0;}
                        ;

depends                 : DEPENDS {}
                        ;

rule_body               : decl_condition decl_action
                        | decl_properties decl_condition decl_action
                        ;

decl_properties         : PROPERTIES properties_body END_PROPERTIES
                        ;

properties_body         : properties_type value {
                              if ($1 == 1) {
                                    priority = atoi($2);
                              }
                              if ($1 == 2) {
                                    keeper = 1;
                              }
                        }
                        | properties_body properties_type value {
                              if ($2 == 1) {
                                    priority = atoi($3);
                              }
                              if ($2 == 2) {
                                    keeper = 1;
```

```
                                        }
                                    }
                                    ;

        properties_type             : PRIORITY { $$ = 1; }
                                    | KEEPER { $$ = 2;  }

        decl_condition              : CONDITION condition_body END_CONDITION
                                    | CONDITION id condition_body END_CONDITION
                                    ;

        condition_body              :   subcondition   operator   condition_body   {
conditions.push_back(compiler->createSubCondition(string(((SubConditionType*)$1)-
>id),string($2), *((SubConditionType*)$1)->conditions)); }
                                    |  subcondition  {  conditions.push_back(compiler-
>createSubCondition(string(((SubConditionType*)$1)-
>id),"",*((SubConditionType*)$1)->conditions)); }
                                    ;

        operator                    : AND { $$ = (char *)"and";}
                                    | OR { $$ = (char *)"or"; }
                                    ;

        subcondition                :      SUBCONDITION      id      subcondition_body
END_SUBCONDITION        {         strcpy(subConditionType[subCount].id,         $2);
subConditionType[subCount].conditions = new std::vector<Entity *>(conditions); $$ =
&subConditionType[subCount]; conditions.clear(); subCount++; }

                                    ;

        subcondition_body           :     premise     AND     subcondition_body     {
conditions.push_back((Entity*)$1); }
                                    | premise { conditions.push_back((Entity*)$1); }
                                    ;

        premise                     :  PREMISE  exp  {$$  =  compiler->createPremise("",
((PremiseCompType*)$2)->leftId,  ((PremiseCompType*)$2)->op,  ((PremiseCompType*)$2)-
>rightId,0);}
                                    | PREMISE id exp  {$$ = compiler->createPremise($2,
((PremiseCompType*)$3)->leftId,  ((PremiseCompType*)$3)->op,  ((PremiseCompType*)$3)-
>rightId,0);}
                                    |   PREMISE   IMP   exp      {$$   =   compiler-
>createPremise("",    ((PremiseCompType*)$3)->leftId,    ((PremiseCompType*)$3)->op,
((PremiseCompType*)$3)->rightId,1);}
```

```
                                | PREMISE IMP id exp     {$$ = compiler-
>createPremise($3,    ((PremiseCompType*)$4)->leftId,    ((PremiseCompType*)$4)->op,
((PremiseCompType*)$4)->rightId,1);}
                                ;

        exp                     : fator comp fator { strcpy(premiseComp.leftId,
$1); premiseComp.op = $2; strcpy(premiseComp.rightId, $3); $$ = &premiseComp; }

        comp                    : EQ {$$ = 1;}
                                | NE {$$ = 2;}
                                | LT {$$ = 3;}
                                | GT {$$ = 4;}
                                | LE {$$ = 5;}
                                | GE {$$ = 6;}
                                ;

        fator                   : id { $$ = $1;}
                                | NUMBER {$$ = $1;}
                                | boolean {$$ = $1;}
                                ;

        boolean                         : TRUE {$$ = (char *)"true";}
                                | FALSE {$$ = (char *)"false";}
                                ;

        decl_action             : ACTION action_body END_ACTION
                                | ACTION id action_body END_ACTION
                                ;

        action_body             : action_elements action_body
                                | action_elements
                                ;

        action_elements                 : instigation
                                | method_use
                                | exp SEMICOLON
                                ;

        instigation             :   INSTIGATION   method_use     {string   str($2);
instigations.push_back(compiler->createInstigation("",str)); }
                                |  INSTIGATION  id  method_use    {string  str3($3);
string str2($2); instigations.push_back(compiler->createInstigation(str2,str3)); }
                                ;

        method_use              : id LP RP SEMICOLON {$$ = $1; }
                                ;
```

```
        id                          : ID         {$$ = $1;}
                                    | ID POINT ID      { char * buf = strcat($1,".");
buf = strcat(buf,$3); $$=buf; }
                                    ;

        fbes                        : fbe
                                    | fbe fbes
                                    ;

        fbe                         :  FBE  fbe_body  END_FBE  {compiler->createFBE("",
attributes, methods); attributes.clear(); methods.clear(); }
                                    | FBE id fbe_body END_FBE {compiler->createFBE($2,
attributes, methods); attributes.clear(); methods.clear();  }
                                    ;

        fbe_body                    : decl_attributes decl_methods
                                    | decl_attributes
                                    ;

        decl_attributes                 : ATTRIBUTES attributes END_ATTRIBUTES
                                    ;

        attributes                  : attributes_body
                                    | attributes_body attributes
                                    ;

        attributes_body                 :       type        id        value       {
attributes.push_back(compiler->createAttribute(string($2), *(int*)$1, string($3)));
}
                                    |   type   id   {   attributes.push_back(compiler-
>createAttribute(string($2), Attribute::A_ID, string((char*)$1)));}
                                    ;

                                    //Attribute * Compiler::createAttribute(std::string
userEntityId, int type, std::string value) {

        type                        : BOOLEAN { short i = 1; $$ = &i; }
                                    | INTEGER { short i = 2; $$ = &i; }
                                    | PFLOAT { short i = 3; $$ = &i; }
                                    | STRING { short i = 4; $$ = &i; }
                                    | id { $$ = $1; }         //Esse   id   é   para   os
nomes das classes - deve ser tratado ainda a analise semantica
                                    ;

        value                       : NUMBER
```

```
                                | boolean
                                | id          // quando é string
                                | FLOATVALUE
                                | STRINGVALUE
                                ;

        decl_methods                    : METHODS methods END_METHODS
                                ;

        methods                         : method_body
                                | method_body methods
                                ;

        method_body             : METHOD id LP id ASSIGN id method_operator value
RP  {  methods.push_back(compiler->createMethod(string($2),  string($4),  string($6),
string($7), false, string($8),"")); } //mtTempoAtual(tempoAtual = tempoAtual + 1)
                                | METHOD id LP id ASSIGN id method_operator id RP {
methods.push_back(compiler->createMethod(string($2),     string($4),     string($6),
string($7),  true,  string($8),""));  }  //mtTempoAtual(tempoAtual  =  tempoAtual  +
novoValor)
                                |   METHOD   id   LP   id   ASSIGN   value   RP   {
methods.push_back(compiler->createMethod(string($2),    string($4),    string("-1"),
string("-1"), false, string($6),""));  } // mtFire(isCrossed = true)
                                |   METHOD   id   LP   id   ASSIGN   id   RP   {
methods.push_back(compiler->createMethod(string($2),    string($4),    string("-1"),
string("-1"), true, string($6),""));  } // mtFire(isCrossed = novoValor)


                                | METHOD id LP id ASSIGN id method_operator value
RP  INNER_CODE  {  methods.push_back(compiler->createMethod(string($2),  string($4),
string($6),      string($7),      false,      string($8),      string($10)));      }
//mtTempoAtual(tempoAtual = tempoAtual + 1)
                                | METHOD id LP id ASSIGN id method_operator id RP
INNER_CODE   {   methods.push_back(compiler->createMethod(string($2),   string($4),
string($6),      string($7),      true,      string($8),      string($10)));      }
//mtTempoAtual(tempoAtual = tempoAtual + novoValor)
                                | METHOD id LP id ASSIGN value RP INNER_CODE {
methods.push_back(compiler->createMethod(string($2),    string($4),    string("-1"),
string("-1"), false, string($6), string($8)));  } // mtFire(isCrossed = true)
                                |  METHOD  id  LP  id  ASSIGN  id  RP  INNER_CODE  {
methods.push_back(compiler->createMethod(string($2),    string($4),    string("-1"),
string("-1"), true, string($6), string($8)));  } // mtFire(isCrossed = novoValor)
                                ;

        method_operator         : PLUS {$$ = (char *)"+";}
                                | MINUS {$$ = (char *)"-";}
```

```
                                | MULT {$$ = (char *)"*";}
                                | DIV {$$ = (char *)"/";}
                                ;

        main                    :   MAIN   INNER_CODE   {   compiler->mainBlock   =
string($2); }

        %%

        extern char *yytext;

        main (int argc, char * argv[])
        {
             if (atoi(argv[1]) == 1) {
                   cout  <<"Compilado  para  C,  veja  os  resultados  na  pasta
compiladosc"<< endl;
                   compiler = new CCompiler();
             } else if (atoi(argv[1]) == 2) {
                   cout  <<"Compilado  para  C++,  veja  os  resultados  na  pasta
cppcompilados"<< endl;
                   compiler = new CPPCompiler();
             } else if (atoi(argv[1]) == 3) {
                   cout <<"Compilado para Framework, veja os resultados na pasta
compilados"<< endl;
                   compiler = new FrameworkCompiler();
             } else {
                   cout <<"Escolha uma oção correta: 1 para C, 2 para C++ ou 3 para
framework"<< endl;
                   exit(0);
             }
             return yyparse();
        }

        void yyerror(const char *s) {
             cout  <<"Erro  (parser)  em/antes  da  linha  "<<  line_num  <<"  ["<<  s
<<"]"<< endl;
             exit(-1);
}
```

APÊNDICE E - Códigos-fonte relativos aos estudos

Este apêndice apresenta os códigos-fonte parciais dos experimentos comparativos realizados com a criação das duas aplicações denominadas respectivamente de: Mira ao Alvo e Aplicação de Vendas. Estas aplicações foram criadas sob o viés do Paradigma Orientado a Notificações (PON) com a utilização da sua linguagem e compilador. Ainda, as mesmas aplicações, de forma equivalente, foram criadas sob o viés do Paradigma Imperativo (PI).

A seção 1 apresenta o código fonte da aplicação Mira ao Alvo com a utilização da Linguagem PON. Já a seção 2 apresenta o código equivalente da aplicação Mira ao Alvo sob o viés do PI/C++. A seção 3 apresenta o código fonte da aplicação de vendas sob o viés do PON com a utilização da linguagem e do compilador. Finalmente, a seção 4 apresenta o código PI/C++ equivalente criado para a aplicação de vendas.

1. Código fonte da aplicação Mira ao Alvo em PON

```
fbe Apple
    attributes
        boolean atIsCrossed false //teste de comentario
    end_attributes  /* teste de comentário */
end_fbe

fbe Archer
    attributes
        boolean atHasFired false
        integer atCount 0
    end_attributes
    methods
        method mtFire1(atHasFired = true)
        method mtFire2(atCount = atCount + 1)
        method mtFire3(atCount = atCount + atCount)
        method mtFire4() begin_method //código específico em C/C++
end_method
    end_methods
end_fbe

fbe Controller
    attributes
        boolean atFire false
    end_attributes
end_fbe

inst
    Apple apple, apple1
    Archer archer, archer1
    Controller controller, controller1
end_inst

strategy
       no_one
end_strategy
```

```
36
37  rule RlTurnOn1
38      condition
39         subcondition A1
40              premise PrIsCrossed apple.atIsCrossed == false and
41              premise PrHasFired archer.atHasFired == false and
42              premise PrFire controller.atFire == true
43         end_subcondition
44         or
45         subcondition A2
46              premise PrIsCrossed archer.atCount == 0
47          end_subcondition
48      end_condition
49      action
50          instigation inFire archer.mtFire1();
51      end_action
52  end_rule
53
54  main {
55         //exemplo de código fonte no main.
56         //isso não é validado pelo compilador
57
58         apple->setatIsCrossed(false);
59         archer->setatHasFired(false);
60         controller->setatFire(true);
61
63  }
```

2. Código fonte da aplicação Mira ao Alvo em POO

2.1 Arquivo principal MiraAoAlvo.cpp

```
 1  //============================================================
 2  // Name        : MiraAoAlvo.cpp
 3  // Author      : Cleverson
 4  // Version     :
 5  // Copyright   :
 6  // Description :
 7  //============================================================
 8  #include <iostream>
 9  using namespace std;
10  #include <sys/time.h>
11  #include <vector>
12  #include "Archer.h"
13  #include "Apple.h"
14  #include "Controller.h"
15  #include <stdlib.h>
16  #include <stdio.h>
17  #define MICRO_PER_SECOND 1000000
18
19  int main() {
20
21         std::vector<Controller *> controllers;
22         std::vector<Apple *> apples;
23         std::vector<Archer *> archers;
24
25         for (int i = 0; i < 100; i++) {
26                Apple * apple = new Apple();
27                Archer * archer = new Archer();
28                Controller * controller = new Controller();
29
30                apple->setIsCrossed(false);
31                archer->setHasFired(false);
32                if (i < 10) {
```

```
33                      controller->setFire(true);
34              } else {
35                      controller->setFire(false);
36              }
37
38              controllers.push_back(controller);
39              apples.push_back(apple);
40              archers.push_back(archer);
41      }
42
43      struct timeval start_time;
44      struct timeval stop_time;
45
46      float time_diff;
47
48      gettimeofday (&start_time, NULL);
49
50      for (int j = 0; j < 100; j++) {
51              if   (apples[j]->getIsCrossed()   ==   false   &&   archers[j]-
52  >getHasFired() == false && controllers[j]->getFire() == true) {
53                      archers[j]->fire();
54              }
55      }
56
57      gettimeofday (&stop_time, NULL);
58
59      time_diff = (float)(stop_time.tv_sec - start_time.tv_sec);
60      time_diff    +=    (stop_time.tv_usec    -    start_time.tv_usec)    /
61  (float)MICRO_PER_SECOND;
63
64      cout << Archer::count << endl;
65      printf("\n%f\n",time_diff);
66
67      return 0;
68  }
```

2.2 Arquivo Archer.h e Archer.cpp

```
1   #pragma once
2
3   class Archer
4   {
5   public:
6       Archer(void);
7       ~Archer(void);
8
9       bool hasFired;
10      static int count;
11
12      void setHasFired(bool hasFired);
13      bool getHasFired();
14
15      void fire();
16  };
17
18
19  #include "Archer.h"
20
21  int Archer::count = 0;
22
23  Archer::Archer(void)
24  {
25  }
26
27
28  Archer::~Archer(void)
```

```
29  {
30  }
31
32  void Archer::setHasFired(bool hasFired) {
33        this->hasFired = hasFired;
34  }
35
36
37  bool Archer::getHasFired() {
38        return this->hasFired;
39  }
40
41  void Archer::fire() {
42        this->hasFired=true;
43        count++;
44  }
```

3. Código-fonte da aplicação de Vendas em PON

```
 1  fbe Client
 2      attributes
 3                string name "ClienteName"
 4                pfloat atCreditLimit 1000.0
 5                integer countSales 0
 6                integer atTypeDiscount 0
 7                pfloat atPercDescount 0.0
 8      end_attributes
 9      methods
10                method mtSaleApproved(countSales = countSales + 1)
11                method mtSaleCanceled(countSales = countSales - 1)
12                method mtTypeDiscount1(atPercDescount = 0.95)
13                method mtTypeDiscount2(atPercDescount = 0.90)
14                method mtTypeDiscount3(atPercDescount = 0.85)
15                method mtTypeDiscount4(atPercDescount = 0.80)
16                method mtTypeDiscount5(atPercDescount = 0.75)
17                method mtTypeDiscount6(atPercDescount = 0.70)
18                method mtTypeDiscount7(atPercDescount = 0.65)
19                method mtTypeDiscount8(atPercDescount = 0.60)
20                method mtTypeDiscount9(atPercDescount = 0.55)
21                method mtTypeDiscount10(atPercDescount = 0.50)
22                method mtTypeDiscount11(atPercDescount = 0.45)
23                method mtTypeDiscount12(atPercDescount = 0.40)
24                method mtTypeDiscount13(atPercDescount = 0.35)
25                method mtTypeDiscount14(atPercDescount = 0.30)
26                method mtTypeDiscount15(atPercDescount = 0.25)
27                method mtTypeDiscount16(atPercDescount = 0.20)
28                method mtTypeDiscount17(atPercDescount = 0.15)
29                method mtTypeDiscount18(atPercDescount = 0.10)
30                method mtTypeDiscount19(atPercDescount = 0.05)
31                method mtTypeDiscount20(atPercDescount = 0.00)
32       end_methods
33  end_fbe
34
35  fbe Product
36      attributes
37                string name "productName"
38                pfloat atPrice 10.0
39                integer atMinimumStock 1
40                integer atCurrentStock 0
41                boolean atRequestPurchase false
42      end_attributes
43      methods
44                method mtRequestPucharse(atRequestPurchase = true)
45      end_methods
46  end_fbe
47
```

```
fbe SalesOrder
    attributes
              SalesOrderItem atSalesOrderItem
              boolean atCloseSalesOrder false
              pfloat atTotalSalesOrder 0.0
              Client atClient
    end_attributes
    methods
              method mtCalculateTotalValue(atTotalSalesOrder =
atSalesOrderItem.atTotalValue * atClient.atPercDescount)
    end_methods
end_fbe

fbe SalesOrderItem
    attributes
              Product atProduct
              boolean atStatusSalesOrderItem false
              pfloat atTotalValue 0.0
              integer atQuantity 0
              boolean atAddSalesOrdermItem false
    end_attributes
    methods
              method mtLowerStock(atProduct.atCurrentStock =
atProduct.atCurrentStock - atQuantity)
              method mtCalculateValueItem(atTotalValue = atProduct.atPrice)

              method mtCalculateQuantity(atTotalValue = atTotalValue *
atQuantity)
              method mtApproveSalesOrderItem(atStatusSalesOrderItem = true)
    end_methods
end_fbe

inst
   Client client
   Product product
   SalesOrderItem salesOrderItem
   SalesOrder salesOrder
end_inst

strategy
       no_one
end_strategy

rule rlApproveProductItem
    condition
       subcondition a2
            premise imp prApproveQuantity product.atCurrentStock >=
salesOrderItem.atQuantity and
            premise prApproveProductItem salesOrderItem.atAddSalesOrdermItem
== true
       end_subcondition
    end_condition
    action
              instigation inCalculateValueItem
salesOrderItem.mtCalculateValueItem();
              instigation inCalculateQuantity
salesOrderItem.mtCalculateQuantity();
              instigation inMtApproveSalesOrderItem
salesOrderItem.mtApproveSalesOrderItem();
    end_action
end_rule

rule rlCheckInventory
    condition
       subcondition a1
            premise prMinimumStock product.atCurrentStock <=
product.atMinimumStock
        end_subcondition
```

```
116      end_condition
117      action
118          instigation inMinimumStock product.mtRequestPucharse();
119      end_action
120  end_rule
121
122  rule rlApproveAndAddSOI
123      condition
124         subcondition a3
125              premise prApproveAndAddSOI salesOrderItem.atStatusSalesOrderItem
126  == true
127          end_subcondition
128      end_condition
129      action
130          instigation inCalculateTotalValue
131  salesOrder.mtCalculateTotalValue();
132      end_action
133  end_rule
134
135  rule rlExecuteSalesOrder depends rlApproveProductItem
136      condition
137         subcondition a4
138              premise prSalesOrderClosed salesOrder.atCloseSalesOrder == true
139              and
140              premise prCreditLimitOk client.atCreditLimit >=
141  salesOrder.atTotalSalesOrder
142          end_subcondition
143      end_condition
144      action
145         instigation inLowerStock salesOrderItem.mtLowerStock();
146          instigation inSaleApproved client.mtSaleApproved();
147      end_action
148  end_rule
149
150  rule rlCancelSalesOrder depends rlApproveProductItem
151      condition
152         subcondition a5
153              premise prSalesOrderCancel salesOrder.atCloseSalesOrder == true
154  and
155              premise prCreditLimitNok client.atCreditLimit <
156  salesOrder.atTotalSalesOrder
157           end_subcondition
158      end_condition
159      action
160          instigation inSaleCanceled client.mtSaleCanceled();
161      end_action
162  end_rule
163
164  rule rlDiscountType1
165      condition
166         subcondition a6
167              premise prTypeDiscount1 client.atTypeDiscount == 1
168          end_subcondition
169      end_condition
170      action
171          instigation inTypeDiscount1 client.mtTypeDiscount1();
172      end_action
173  end_rule
174
175  rule rlDiscountType2
176      condition
177         subcondition a7
178              premise prTypeDiscount2 client.atTypeDiscount == 2
179          end_subcondition
180      end_condition
181      action
182          instigation inTypeDiscount2 client.mtTypeDiscount2();
183      end_action
```

```
184  end_rule
185
186  rule rlDiscountType3
187      condition
188         subcondition a8
189              premise prTypeDiscount3 client.atTypeDiscount == 3
190          end_subcondition
191      end_condition
192      action
193          instigation inTypeDiscount3 client.mtTypeDiscount3();
194      end_action
195  end_rule
196
197  rule rlDiscountType4
198      condition
199         subcondition a9
200              premise prTypeDiscount4 client.atTypeDiscount == 4
201          end_subcondition
202      end_condition
203      action
204          instigation inTypeDiscount4 client.mtTypeDiscount4();
205      end_action
206  end_rule
207
208  rule rlDiscountType5
209      condition
210         subcondition a10
211              premise prTypeDiscount5 client.atTypeDiscount == 5
212          end_subcondition
213      end_condition
214      action
215          instigation inTypeDiscount5 client.mtTypeDiscount5();
216      end_action
217  end_rule
218
219  rule rlDiscountType6
220      condition
221         subcondition a11
222              premise prTypeDiscount6 client.atTypeDiscount == 6
223          end_subcondition
224      end_condition
225      action
226          instigation inTypeDiscount6 client.mtTypeDiscount6();
227      end_action
228  end_rule
229
230  rule rlDiscountType7
231      condition
232         subcondition a12
233              premise prTypeDiscount7 client.atTypeDiscount == 7
234          end_subcondition
235      end_condition
236      action
237          instigation inTypeDiscount7 client.mtTypeDiscount7();
238      end_action
239  end_rule
240
241  rule rlDiscountType8
242      condition
243         subcondition a13
244              premise prTypeDiscount8 client.atTypeDiscount == 8
245          end_subcondition
246      end_condition
247      action
248          instigation inTypeDiscount8 client.mtTypeDiscount8();
249      end_action
250  end_rule
251
```

```
252  rule rlDiscountType9
253      condition
254         subcondition a14
255              premise prTypeDiscount9 client.atTypeDiscount == 9
256          end_subcondition
257      end_condition
258      action
259          instigation inTypeDiscount9 client.mtTypeDiscount9();
260      end_action
261  end_rule
262
263  rule rlDiscountType10
264      condition
265         subcondition a15
266              premise prTypeDiscount10 client.atTypeDiscount == 10
267          end_subcondition
268      end_condition
269      action
270          instigation inTypeDiscount10 client.mtTypeDiscount10();
271      end_action
272  end_rule
273
274  rule rlDiscountType11
275      condition
276         subcondition a16
277              premise prTypeDiscount11 client.atTypeDiscount == 11
278          end_subcondition
279      end_condition
280      action
281          instigation inTypeDiscount11 client.mtTypeDiscount11();
282      end_action
283  end_rule
284
285  rule rlDiscountType12
286      condition
287         subcondition a17
288              premise prTypeDiscount12 client.atTypeDiscount == 12
289          end_subcondition
290      end_condition
291      action
292          instigation inTypeDiscount12 client.mtTypeDiscount12();
293      end_action
294  end_rule
295
296  rule rlDiscountType13
297      condition
298         subcondition a18
299              premise prTypeDiscount13 client.atTypeDiscount == 13
300          end_subcondition
301      end_condition
302      action
303          instigation inTypeDiscount13 client.mtTypeDiscount13();
304      end_action
305  end_rule
306
307  rule rlDiscountType14
308      condition
309         subcondition a19
310              premise prTypeDiscount14 client.atTypeDiscount == 14
311          end_subcondition
312      end_condition
313      action
314          instigation inTypeDiscount14 client.mtTypeDiscount14();
315      end_action
316  end_rule
317
318  rule rlDiscountType15
319      condition
```

```
320         subcondition a20
321              premise prTypeDiscount15 client.atTypeDiscount == 15
322          end_subcondition
323      end_condition
324      action
325          instigation inTypeDiscount15 client.mtTypeDiscount15();
326      end_action
327  end_rule
328
329  rule rlDiscountType16
330      condition
331         subcondition a21
332              premise prTypeDiscount16 client.atTypeDiscount == 16
333          end_subcondition
334      end_condition
335      action
336          instigation inTypeDiscount16 client.mtTypeDiscount16();
337      end_action
338  end_rule
339
340  rule rlDiscountType17
341      condition
342         subcondition a22
343              premise prTypeDiscount17 client.atTypeDiscount == 17
344          end_subcondition
345      end_condition
346      action
347          instigation inTypeDiscount17 client.mtTypeDiscount17();
348      end_action
349  end_rule
350
351  rule rlDiscountType18
352      condition
353         subcondition a23
354              premise prTypeDiscount18 client.atTypeDiscount == 18
355          end_subcondition
356      end_condition
357      action
358          instigation inTypeDiscount18 client.mtTypeDiscount18();
359      end_action
360  end_rule
361
362  rule rlDiscountType19
363      condition
364         subcondition a24
365              premise prTypeDiscount19 client.atTypeDiscount == 19
366          end_subcondition
367      end_condition
368      action
369          instigation inTypeDiscount19 client.mtTypeDiscount19();
370      end_action
371  end_rule
372
373  rule rlDiscountType20
374      condition
375         subcondition a25
376              premise prTypeDiscount20 client.atTypeDiscount == 20
377          end_subcondition
378      end_condition
379      action
380          instigation inTypeDiscount20 client.mtTypeDiscount20();
381      end_action
382  end_rule
383
384  main {
385
386         product->setatCurrentStock(1);
387         client->setatTypeDiscount(1);
```

```
388         salesOrderItem->setatQuantity(1);
389         salesOrderItem->setatAddSalesOrdermItem(true);
390         salesOrder->setatCloseSalesOrder(true);
391
392  }
```

4. Código-fonte da aplicação de Vendas em POO

4.1 Arquivo Principal.cpp

```
  1  #include <iostream>
  2  using namespace std;
  3
  4  #include <vector>
  5
  6  #include <stdio.h>
  7
  8  #include "Client.h"
  9  #include "Product.h"
 10  #include "SalesOrder.h"
 11
 12  #include <sys/time.h>
 13  #include <stdio.h>
 14
 15  #define MICRO_PER_SECOND 1000000
 16
 17  int main() {
 18
 19         struct timeval start_time;
 20         struct timeval stop_time;
 21
 22
 23         Client * client = new Client();
 24         client->setName("Client 1");
 25         client->setClientType(1);
 26         client->setCreditLimit(100000.0);
 27
 28         Product * product = new Product();
 29         product->setName("Produto 1");
 30         product->setPrice(10.0);
 31         product->setMinimumStock(10000);
 32
 33         SalesOrderItem * salesOrderItem = new SalesOrderItem();
 34         salesOrderItem->setProduct(product);
 35         salesOrderItem->setQuantity(1);
 36
 37         PaymentForm * paymentForm = new PaymentForm();
 38         paymentForm->setName("Dinheiro");
 39         paymentForm->setPaymentType(1);
 40
 41         SalesOrder * salesOrder = new SalesOrder();
 42         salesOrder->setClient(client);
 43         salesOrder->setItem(salesOrderItem);
 44         salesOrder->setPaymentForm(paymentForm);
 45
 46  float time_diff;
 47
 48         gettimeofday (&start_time, NULL);
 49
 50         for (int i = 0; i < 10000; i++) {
 51
```

```
52              if       (salesOrderItem->getQuantity()       <=       product-
53  >getMinimumStock()) {
54 
55                      salesOrderItem->calcTotalValue();
56 }
57                      salesOrderItem->setStatus(true);
58 
59              } else {
60 
61                      product->requestPurchase();
62 .
63                      salesOrderItem->setStatus(false);
64 !
65              }
66 }
67              if (salesOrderItem->getStatus()) {
68 |
69                      salesOrder->calculateTotalValue();
70 }
71                      if      (client->getCreditLimit()      >      salesOrder-
72  >getTotalValue()) {
73 }
74                              product->lowerStock(salesOrderItem-
75  >getQuantity());
76                              client->saleApproved();
77 
78                      } else {
79 }
80                              client->saleCanceled();
81 
82                      }
83 )
84              }
85 .
86      }
87 !
88      gettimeofday (&stop_time, NULL);
89 }
90      time_diff = (float)(stop_time.tv_sec - start_time.tv_sec);
91      time_diff    +=    (stop_time.tv_usec    -    start_time.tv_usec)    /
92  (float)MICRO_PER_SECOND;
93 |
94      printf("\n%f\n",time_diff);
95 }
96      std::cout << "Number of sales: " << client->getTotalSales() << "!" <<
97  std::endl;
98 
99 
100     return 0;
101 }
```

4.2 Arquivo de vendas SalesOrder.h e SalesOrder.cpp

```
#pragma once

#include "SalesOrderItem.h"
#include "PaymentForm.h"
#include "Client.h"

#include <string>
using namespace std;

#include <iostream>
using namespace std;

#include <list>

class SalesOrder
{
private:
	int id;
	double totalValue;
	Client * client;
	PaymentForm *paymentForm;

	list<SalesOrderItem *> * items;
	SalesOrderItem * item;

public:

	SalesOrder(int id, Client * client, PaymentForm *
paymentForm);

	SalesOrder();
	~SalesOrder();

	void setId(int id);
	int getId();

	void setClient(Client * client);
	Client * getClient();

	void setPaymentForm(PaymentForm * paymenteForm);
	PaymentForm * getPaymentForm();

	void setItems(list<SalesOrderItem *> * items);
	list<SalesOrderItem *> * getItems();

	void setItem(SalesOrderItem * item);
	SalesOrderItem * getItem();

	double getDiscount();
	void calculateTotalValue();
	double getTotalValue();

};


#include "SalesOrder.h"

SalesOrder::SalesOrder(int id, Client * client, PaymentForm *
```

```
 59  paymentForm) {
 60        this->id = id;
 61        this->client = client;
 62        this->paymentForm = paymentForm;
 63        this->totalValue = 0.0;
 64        this->items = new list<SalesOrderItem *>();
 65
 66
 67  }
 68
 69  SalesOrder::SalesOrder()
 70  {
 71        this->id = 0;
 72        this->totalValue = 0;
 73        this->items = 0;
 74
 75        this->client = 0;
 76        this->paymentForm = 0;
 77
 78  }
 79
 80  SalesOrder::~SalesOrder()
 81  {
 82  }
 83
 84  void SalesOrder::setId(int id) {
 85        this->id = id;
 86  }
 87
 88  int SalesOrder::getId() {
 89        return this->id;
 90  }
 91
 92  double SalesOrder::getTotalValue() {
 93        return this->totalValue;
 94  }
 95
 96  void SalesOrder::setItems(list<SalesOrderItem *> * items) {
 97        this->items = items;
 98  }
 99
100  SalesOrderItem * SalesOrder::getItem() {
101        return this->item;
102  }
103
104  void SalesOrder::setItem(SalesOrderItem * item) {
105        this->item = item;
106  }
107
108  list<SalesOrderItem *> * SalesOrder::getItems() {
109        return this->items;
110  }
111
112  void SalesOrder::setPaymentForm(PaymentForm * paymentForm) {
113        this->paymentForm = paymentForm;
114  }
115
116  PaymentForm * SalesOrder::getPaymentForm() {
117        return this->paymentForm;
118  }
119
```

```
120  void SalesOrder::setClient(Client * client) {
121        this->client = client;
122  }
123
124  Client * SalesOrder::getClient() {
125        return this->client;
126  }
127
128  void SalesOrder::calculateTotalValue() {
129
130        totalValue = item->getTotalValue() * getDiscount();
131
132  }
133
134  double SalesOrder::getDiscount() {
135
136        double discount = 0.0;
137
138        if (this->getClient()->getClientType() == 1) {
139              discount = 0.95;
140              return discount;
141        }
142
143        if (this->getClient()->getClientType() == 2) {
144              discount = 0.90;
145              return discount;
146        }
147
148        if (this->getClient()->getClientType() == 3) {
149              discount = 0.85;
150              return discount;
151        }
152
153        if (this->getClient()->getClientType() == 4) {
154              discount = 0.80;
155              return discount;
156        }
157
158        if (this->getClient()->getClientType() == 5) {
159              discount = 0.75;
160              return discount;
161        }
162
163        if (this->getClient()->getClientType() == 6) {
164              discount = 0.70;
165              return discount;
166        }
167
168        if (this->getClient()->getClientType() == 7) {
169              discount = 0.65;
170              return discount;
171        }
172
173        if (this->getClient()->getClientType() == 8) {
174              discount = 0.60;
175              return discount;
176        }
177
178        if (this->getClient()->getClientType() == 9) {
179              discount = 0.55;
180              return discount;
```

```
181         }
182
183         if (this->getClient()->getClientType() == 10) {
184               discount = 0.50;
185               return discount;
186         }
187
188         if (this->getClient()->getClientType() == 11) {
189               discount = 0.45;
190               return discount;
191         }
192
193         if (this->getClient()->getClientType() == 12) {
194               discount = 0.40;
195               return discount;
196         }
197
198         if (this->getClient()->getClientType() == 13) {
199               discount = 0.35;
200               return discount;
201         }
202
203         if (this->getClient()->getClientType() == 14) {
204               discount = 0.30;
205               return discount;
206         }
207
208         if (this->getClient()->getClientType() == 15) {
209               discount = 0.25;
210               return discount;
211         }
212
213         if (this->getClient()->getClientType() == 16) {
214               discount = 0.20;
215               return discount;
216         }
217
218         if (this->getClient()->getClientType() == 17) {
219               discount = 0.15;
220               return discount;
221         }
222
223         if (this->getClient()->getClientType() == 18) {
224               discount = 0.10;
225               return discount;
226         }
227
228         if (this->getClient()->getClientType() == 19) {
229               discount = 0.05;
230               return discount;
231         }
232
233         if (this->getClient()->getClientType() == 20) {
234               discount = 0.00;
235               return discount;
236         }
237
238         return discount;
239
240   }
```

ANEXO A - Artigos relacionados a LingPON

Neste anexo se encontram três relatórios no formato de artigos. Em tais relatórios os autores relatam a utilização da linguagem e do compilador PON. Esses relatórios foram o resultado de uma disciplina ofertado pelo Programa de Pós-Graduação em Computação Aplicada (PPGCA/UTFPR), sob a supervisão do Prof. Dr. Jean M. Simão, e tiveram como objetivo destacar as vantagens e desvantagens da utilização da LingPON na criação de aplicações diversas. Esses relatórios foram referenciados na Subseção 4.3.2 deste trabalho. Por fim, os relatórios anexados tiveram o acordo dos autores e do professor da disciplina.

Desenvolvendo o jogo Pac-Man com o Paradigma Orientado a Notificações

Helio Henrique Lopes Costa Monte-Alto
Universidade Federal do Paraná
Palotina, Paraná
Email: heliohenrique@ufpr.br

***Resumo*—Este artigo apresenta o desenvolvimento, resultados e sugestões derivadas da modelagem e implementação de um simulador do jogo Pac-Man utilizando o Paradigma Orientado a Notificações (PON). O simulador foi implementado com a linguagem LingPON, uma materialização do PON que conta com um compilador dedicado à programação neste paradigma. O trabalho tem como propósito experimentar o paradigma e suas atuais materializações, assim como colaborar com sugestões para seu desenvolvimento por meio de um estudo de caso razoavelmente complexo em termos de quantidade de elementos e regras causais.**

***Keywords*—Paradigma Orientado a Notificações (PON), simulador de jogo**

I. Introdução

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

Devido aos avanços tecnológicos, com relação ao aumento da quantidade de transistores, aumento de frequência do *clock*, paralelismo e distribuição, muitas vezes não há uma preocupação com o avanços na área de desenvolvimento de software. Como consequência, continua-se a não utilização dos recursos dos processadores de forma eficiente. Por outro lado, a necessidade de softwares cada vez mais complexos é maior e demanda mais recursos por parte dos processadores.

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

O PON propõe uma solução para os problemas destes paradigmas, que apresentam deficiências com relação a redundâncias estruturais, temporais e forte acoplamento entre suas entidades, diminuindo o desempenho e gerando maior dificuldade de paralelização e distribuição [2]. Para o desenvolvimento de software fazendo uso do PON, foram realizadas pesquisas com *framework* C++ [3] e uma segunda versão otimizada do framework C++ [4], permitindo a criação de software PON sob abordagem de POO. Ademais, pesquisas recentes visam o desenvolvimento de um compilador e de uma linguagem específica para o PON, denominada LingPON [5] [6].

Neste artigo será apresentado um estudo de caso implementado com o LingPON. O estudo de caso utilizado é um simulador do jogo Pac-Man [7]. Por ser uma aplicação com uma quantidade razoável de elementos e regras causais, acreditamos que seja um estudo de caso interessante para o PON.

As próximas seções estão organizadas da seguinte maneira: a Seção II apresenta os conceitos sobre o PON. A Seção III apresenta algumas materializações do PON, que incluem o *framework* PON e a linguagem LingPON. A Seção IV apresenta um trabalho relacionado: um exemplo do jogo Pac-Man desenvolvido em PON híbrido com POO. A Seção V apresenta a modelagem da aplicação de exemplo. A Seção VI apresenta os detalhes de implementação e as principais dificuldades de desenvolvimento. A Seção VII apresenta as sugestões de melhorias na materialização utilizada, assim como trabalhos futuros. Enfim, a Seção VIII apresenta as conclusões e considerações sobre o paradigma.

II. Paradigma Orientado a Notificações (PON)

O PON foi proposto como um novo paradigma de desenvolvimento de software, que apresenta algumas vantagens quando comparado aos paradigmas tradicionais (mais especificamente, o PI – Paradigma Imperativo - e o PD – Paradigma Declarativo) no que diz respeito ao seu modelo lógico. Tais vantagens são constituídas por uma maior facilidade na concepção de sistemas que apresentem paralelismo ou distribuição, além da redução ou eliminação de alguns dos problemas clássicos de software PI e PD, tais como redundâncias de execução e acoplamento excessivo entre entidades computacionais [8].

Estruturalmente, o software PON é representado na forma de Base de Fatos (FBE – *Facts Base Element*) e Regras (*Rules*). Os elementos FBE são utilizados para representar objetos do mundo real em um sistema computacional, por meio de estados (atributos) e serviços (métodos). Os elementos *Rules*,

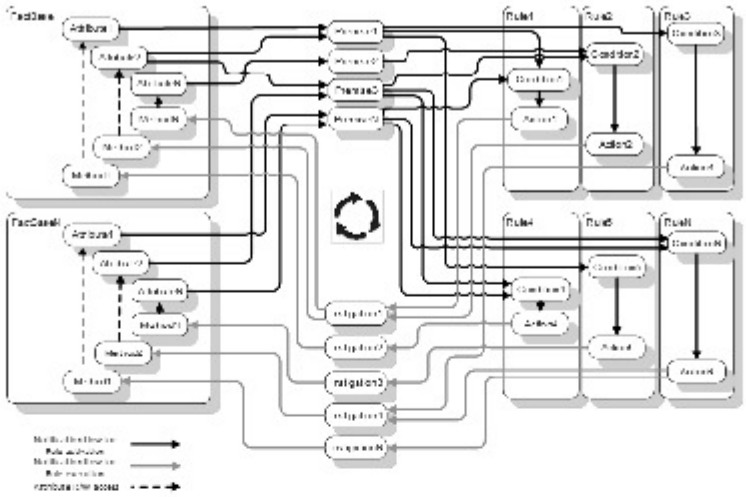


Figura 1. Colaboração por notificações das entidades do PON [5]

por sua vez, definem o cálculo lógico-causal a ser efetuado sobre os estados dos FBEs, controlando a execução dos seus serviços. A colaboração entre estes elementos ocorre por meio de notificações diretas, que é um processo de inferência essencialmente distinto dos processos utilizados em software PI e em Sistemas Baseados em Regras (SBR) do PD [3].

Nos FBEs os estados (atributos) são representados por meio de objetos da classe *Attribute*, enquanto os serviços dessas entidades (métodos) são representados pela classe *Method*. O elemento que representa uma *Rule* é composto por uma condição, representada pelo objeto da classe *Condition* e uma Ação, pelo objeto da classe *Action*. O objeto da classe *Condition* efetua um cálculo lógico sobre o valor de uma ou mais premissas, representadas por objetos da classe *Premise*. Estas são responsáveis por efetuar cálculo relacional sobre os valores dos atributos de um FBE, encapsulados em objetos da classe *Attribute*. O objeto da classe *Action*, por sua vez referencia um ou mais objetos da classe *Insigation*, por meio dos quais é capaz de disparar métodos dos FBEs, encapsulados em objetos da classe *Method*. Os métodos, por sua vez, modificam o valor em um *Attribute* quando chamados. Quando isto acontece, o próprio *Attribute* notifica as entidades *Premise* relacionadas a ele. Assim novas premissas são avaliadas - ou reavaliadas - de acordo com o valor atual dos atributos. Uma vez que todas as entidades *Premise* vinculadas a uma *Condition* apresentam seus valores lógicos verdadeiros, esta é finalmente satisfeita. Com isso, a entidade *Action* agregada a esta Rule é executada, invocando os *Methods* necessários através das entidades *Insigation* [3].

A Figura 1 apresenta um exemplo genérico de instanciação das entidades que compõem um software PON, bem como as relações de envio e recebimento de notificação. As 4 linhas em negrito representam o fluxo de notificações para a ativação das *Rules* (cálculo lógico-causal), ao passo que as linhas mais claras representam o fluxo para a execução das *Rules* (cálculo factual).

III. MATERIALIZAÇÕES DO PON

A primeira materialização feita para o PON foi um *framework* escrito na linguagem C++. Tal materialização possibilitou, de fato, a criação de aplicações sob o domínio desse novo paradigma para ambientes computacionais usuais baseados na arquitetura de Von Neumann. Apesar destas contribuições, o desenvolvimento de aplicações no PON ainda não apresenta resultados satisfatórios em termos de desempenho tal qual deveria à luz do seu cálculo assintótico de crescimento linear. Tal desempenho insatisfatório está relacionado, inclusive, ao uso de estruturas de dados caras no seio do *framework* [3]. Enfim, chegou-se à conclusão de que existe a necessidade de melhorias no estado da técnica para que o PON atinja o que fora vislumbrado em seu estado da arte [6].

Assim, [6] propõe o desenvolvimento de linguagens e compiladores que traduzam o código PON em um código alvo mais puro e menos dependente de conceitos de outros paradigmas, buscando também evitar o uso de estruturas de dados computacionalmente caras. Além disso, uma linguagem dedicada ao PON facilita a implementação de programas neste paradigma. Com base nessas justificativas, foi proposta a linguagem LingPON e seu compilador.

A atual versão do compilador é capaz de traduzir o código LingPON em três linguagens alvo diferentes: *framework* PON em C++, C++ puro e C puro.

A geração do código no *framework* basicamente recria os componentes do *framework* de acordo com o código PON

criado. Cada elemento do código PON tem a sua correspondência no *framework*. Isto permite primariamente facilitar o desenvolvimento de aplicações PON no *framework* de modo mais natural por meio do LingPON.

Na implementação em C++, o código gerado pelo compilador apresenta classes C++ que representam os conceitos PON relativos a FBE, *Rule* e *Premise*. O conjunto de *Premises* associados a uma *Rule* específica no código C++ pode ser considerado como uma *Condition*. O objetivo principal de seguir os princípios do PON na geração de código alvo C++ foi garantir o ganho de desempenho que o PON preconiza ao evitar a avaliação de expressões causais desnecessárias, bem como evitar o alto acoplamento das entidades [6].

A implementação em C foi toda desenvolvida de forma a otimizar a execução do código gerado em detrimento do tamanho do mesmo. A geração de código foi pensada com o intuito de chegar mais próximo possível do que seria gerado em Assembly, assim as otimizações do GCC foram mínimas ou nenhuma. Para que isso ocorresse, não foram utilizadas no código gerado as seguintes técnicas: passagens de parâmetros, ponteiros e alocação dinâmica de memória. Dessa forma, a tradução do C para o código de máquina (que foi deixada a cargo do Compilador GCC) pôde ser mais direta [6].

IV. Trabalhos Relacionados

Uma tentativa de implementar a aplicação do Pac-Man em PON foi realizada anteriormente por [9]. Trata-se de uma versão híbrida do simulador em que parte foi implementada utilizando o POO e parte em PON, utilizando o *framework* para C++.

As regras do jogo são, em geral, as mesmas utilizadas no simulador apresentado neste trabalho. Provavelmente a principal diferença é a modelagem das decisões do Pac-Man sobre qual direção tomar. Dentre as principais particularidades dessa modelagem, tem-se a classificação de esquinas em categorias. As esquinas representam o encontro ou cruzamento de dois ou mais corredores que compõem o labirinto, cada qual com seu formato distinto.

O labirinto é formado por 9 diferentes formatos de esquinas, com o intuito de minimizar a quantidade de regras, visto que o tratamento das ações dos personagens se baseia nessa classificação e não em todas as partes do labirinto. Conforme ilustra a Figura 2, cada formato particular de esquina é representado por um valor em uma escala de 1 a 9.

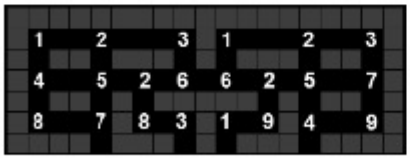


Figura 2. Labirinto dividido em 9 diferentes tipos de esquinas [9]

Os personagens só alteram sua direção quando estiverem sobre uma esquina, podendo retornar ao caminho pelo qual vieram ou escolher outro corredor para seguir. A Figura 3 ilustra

Figura 3. Exemplos de regras de movimentação do Pac-Man [9]

exemplos de regras de movimentação para o personagem Pac-Man.

O simulador apresentado neste trabalho não segue a mesma modelagem, como será apresentado em mais detalhes na Seção V. Tal decisão foi tomada de modo a tornar a implementação do labirinto e da movimentação dos agentes mais flexível - sem depender, por exemplo, da classificação de esquinas. Embora esta modelagem de regras de movimentação baseada em esquinas tenha sido interessante no escopo de um simulador híbrido - no qual a parte implementada em PON se resume às regras de movimentação - não seria tão interessante no escopo de um simulador em PON puro implementado em LingPON, principalmente devido às limitações atuais deste, como será apresentado na Seção VI-A.

V. Modelagem

De modo geral, o simulador desenvolvido neste trabalho possui características do clássico jogo Pac-Man. Tal qual o jogo de inspiração, o ambiente possui corredores que formam um labirinto, limitando as ações de movimento dos indivíduos do cenário. Os indivíduos ativos no cenário são o Pac-Man e seus inimigos Fantasmas. Por se tratar de um simulador, cada um dos indivíduos apresentam comportamento autônomo (i.e. não são controlados por um usuário). A Figura 4 apresenta uma imagem de um cenário completo do jogo Pac-Man.

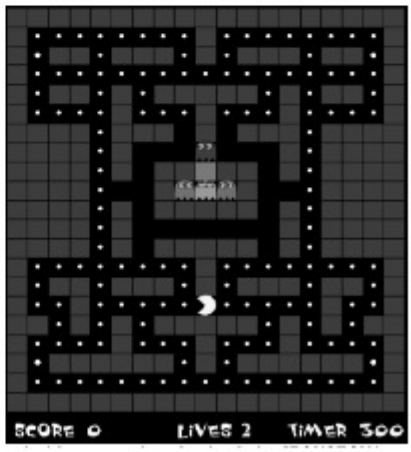


Figura 4. Ambiente do jogo Pac-Man [9]

O primeiro problema encontrado relaciona-se a algumas limitações do atual estado do LingPON. Ainda não existe uma maneira de criar múltiplas instâncias de uma mesma FBE de forma automatizada. Cada nova instância deve ser declarada e referenciada individualmente. Isto leva a outro problema: cada regra ou subcondição que envolva todas as instâncias de uma FBE devem ser replicadas para cada instância. Portanto, o retrabalho no desenvolvimento - e consequentemente a dificuldade de implementação - aumenta em função da quantidade de elementos da aplicação. No caso do jogo Pac-Man implementado, cada unidade de parede seria uma instância da FBE *Wall*. Em um esquema de labirinto razoavelmente grande, como o apresentado na Figura 4, será necessária a declaração de mais de 100 instâncias da FBE *Wall*. Para cada regra envolvendo percepção e colisão de paredes, será necessária a replicação para cada uma das mais de 100 paredes. O mesmo se aplica ao inserir os pontos no labirinto. Como veremos mais adiante, mesmo com um labirinto muito menor existe um retrabalho excessivo e um código resultante consideravelmente grande.

Para simplificar a implementação no estado atual do LingPON, foi desenhado o labirinto apresentado na Figura 5. O labirinto possui 10 paredes (sem contar os limites laterais), 16 pontos (sendo 2 pontos energizadores), um Pac-Man e um Fantasma. O Pac-Man é posicionado na posição (3,3) do *grid* e o Fantasma na posição (4,1). No início da simulação, o Pac-Man está livre para absorver os pontos que se encontram nos corredores e acumular pontos (*score*), procurando evitar contato com os Fantasmas que tentam colidir com ele. O objetivo principal do Pac-Man é comer os pontos e assim maximizar os pontos ganhos. Já o Fantasma também é livre para andar pelo labirinto e tem como objetivo capturar o Pac-Man.

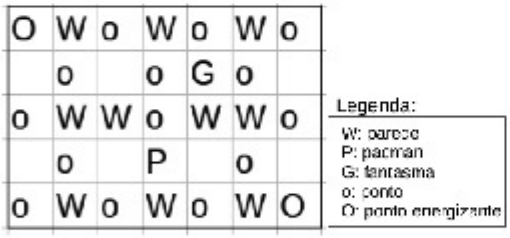


Figura 5. Exemplo de labirinto pequeno.

Além dos pontos normais, existem ainda pontos energizadores, que, quando comidos pelo Pac-Man, fazem o Fantasma ficar "assustado", invertendo-se os papéis do jogo: o Pac-Man passa de presa a predador, podendo comer o Fantasma, e este passa a ter por objetivo fugir do Pac-Man. O efeito do ponto energizador dura por um tempo determinado ou até que o Pac-Man coma o Fantasma. Quando isto acontece, o Fantasma volta à sua posição de origem em seu estado normal, como predador. Ao comer o Fantasma, o Pac-Man também ganha mais pontos do que ganharia ao comer os pontos espalhados pelo labirinto.

Dados os objetivos do jogo e dos agentes envolvidos, pode-se atribuir a seguinte lista de prioridades no que diz respeito às ações do Pac-Man:

1) Evitar paredes;
2) Evitar Fantasma;
3) Perseguir Fantasma assustado;
4) Perseguir ponto energizante;
5) Perseguir ponto comum.

Para o Fantasma em seu estado normal (predador) a ordem de prioridades é um pouco mais simples: (1) Evitar paredes; (2) Perseguir Pac-Man. Sob o efeito do ponto energizador, a ordem de prioridades do Fantasma assustado torna-se: (1) Evitar paredes; (2) Evitar Pac-Man.

O algoritmo utilizado para ambos os agentes é um tipo de algoritmo guloso que decide a direção a tomar com base nessas ordens de prioridade. Em caso de empate - e.g., na direção sul e na norte o Pac-Man só enxerga um ponto - a decisão é tomada com base na primeira regra disparada pela aplicação em PON.

A percepção dos agentes é baseada em um campo de visão estabelecido. Tanto o Pac-Man quanto o Fantasma possuem um campo de visão de três unidades do *grid*. Isto é exemplificado na Figura 6.

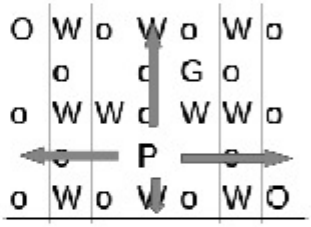


Figura 6. Ilustração do campo de visão do Pac-Man.

Tendo sido estabelecidos a configuração do labirinto, os objetivos dos agentes, e a forma de percepção e tomada de decisões dos agentes, serão apresentados as FBEs e regras associadas a cada um dos elementos do jogo. Os elementos identificados que foram projetados como FBE são apresentados brevemente a seguir:

- *Pacman*: representa o Pac-Man e mantém os atributos e métodos próprios a ele;
- *Ghost*: representa o Fantasma e mantém os atributos e métodos próprios a ele;
- *MazeLimits*: mantém as dimensões do labirinto;
- *Wall*: representa uma parede, tendo como atributos apenas a posição (x,y) no labirinto;
- *Dot*: representa um ponto, tendo como atributos a posição (x,y) no labirinto e uma *flag* que indica sua existência, isto é, se ainda não foi comido;
- *EnergizerDot*: representa um ponto energizante, tendo atributos idênticos à FBE *Dot*;
- *Score*: representa a pontuação do jogo, com apenas um atributo para mantê-la e atualizá-la;
- *Ticker*: mantém *flags* para gerenciamento do *clock* do jogo.

As próximas subseções apresentam mais pormenorizadamente cada uma dessas FBEs e as principais regras vinculadas.

A. FBE Pacman e principais regras

O comportamento do Pac-Man é o principal item a ser modelado neste simulador. O Pac-Man é um agente (*softbot*) situado em um ambiente virtual (o labirinto). Portanto, tem características inerentes a agentes de software. Ele está situado em um ambiente, é capaz de percebê-lo, é capaz de tomar suas próprias decisões com base em suas percepções, e é capaz de reagir, isto é, tomar ações com base em estímulos do ambiente [10]. A questão da localidade já foi definida, sendo resolvida pela configuração do labirinto. Já a percepção deverá ser tratada com base na localidade do Pac-Man e dos elementos à sua volta. Portanto deverão ser criadas regras específicas para a percepção, que incluam em suas premissas condições envolvendo a posição do Pac-Man no labirinto assim como a posição dos demais objetos, como as paredes, os pontos e o Fantasma. A estas regras damos o nome de Regras de Percepção.

O resultado da avaliação de uma regra de percepção - o objeto *Condition* - deve ser uma instigação que acione um método que permita modificar alguns atributos do tipo *flag* na FBE *Pacman*, aos quais denominamos *atributos de percepção*. Um exemplo de um pedaço de regra de percepção - apenas uma subcondição - é mostrado na Figura 7, na qual observamos a regra que permite verificar se existe parede à direita - isto é, na direção leste. A regra ilustrada é composta de duas premissas que testam a posição da parede em relação ao Pac-Man, e uma instigação que faz a chamada ao método *perceiveWallRight* da FBE *Pacman*. Tal método consiste apenas na modificação do atributo *wallAtRight* para o valor 1, indicando que foi detectada uma parede à direita.

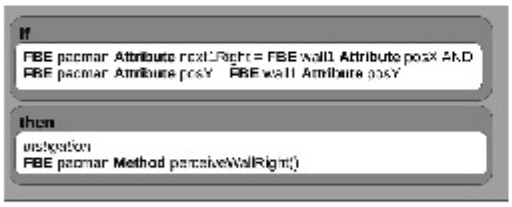


Figura 7. Exemplo de regra de percepção.

Pelo Paradigma Orientado a Notificações, a mudança do valor de um atributo pode notificar e ativar premissas de outras regras de modo que o agente possa utilizar essas informações de percepção para ponderar sobre as decisões a tomar. Tais regras serão chamadas de Regras de Decisão, que deverão ter premissas que testem os atributos de percepção modificados anteriormente pelas regras de percepção. A Figura 8 ilustra uma regra de decisão que possui duas premissas - uma para verificar o atributo *wallAtRight* e uma para verificar o *ticker*, como será explicado mais adiante - e uma instigação que chama o método *dontGoToRight*, que por sua vez modifica um atributo chamado *rightDecision* de modo que o Pac-Man não tenha a preferência de seguir para a direção leste. Estes *atributos de decisão* foram modelados de acordo com a ordem de prioridade apresentada anteriormente. Por exemplo, o método *dontGoToRight* subtrai o valor 20 do atributo *rightDecision*, de modo que a decisão por tomar essa direção seja desfavorável.

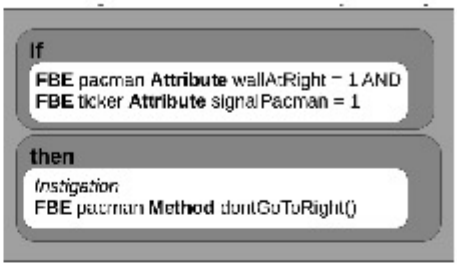


Figura 8. Exemplo de regra de decisão.

Portanto, os resultados da ativação das premissas dessas regras de decisão são instigações que fazem a chamada a métodos que atualizam atributos com pesos para cada situação. Pesos negativos são somados aos atributos de decisão para casos desfavoráveis ao Pac-Man - como é o caso da detecção de parede ou de Fantasma - e pesos positivos para casos favoráveis - como a detecção de pontos. Tais mudanças de atributos geram notificações que podem ativar premissas para as Regras de Ação.

As regras de ação são regras simples que apenas comparam os atributos de decisão de modo a escolher a direção mais favorável para o agente se mover. A ativação de suas premissas causa a instigação que chama os métodos de movimento, que efetivamente farão o Pac-Man se mover no labirinto. A Figura 9 ilustra uma regra de ação.

Figura 9. Exemplo de regra de ação.

Além desse ciclo que envolve percepção, decisão e ação, é necessário que haja regras que permitam verificar a ocorrência de colisões. Por exemplo, se o Pac-Man colide com um ponto, aquele ponto deve deixar de existir e o Pac-Man deve ganhar pontos (*score*). Se o Pac-Man colide com o Fantasma, o jogo deve terminar ou ser reiniciado. Tais colisões podem ser verificadas logo após o movimento do Pac-Man. A Figura 10 demonstra uma Regra de Colisão.

Por fim, a cada ciclo de *clock* as regras de percepção, decisão, ação e colisão devem ser repetidas. Para que isso possa ocorrer, é necessário que todas os atributos modificados para descrever o atual estado do mundo de acordo com as

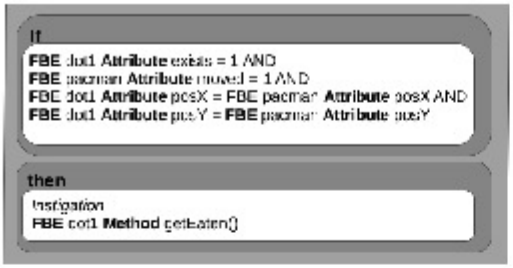


Figura 10. Exemplo de regra de colisão.

percepções dos agentes, assim como os atributos que descrevem as decisões, sejam reinicializados para seus valores *default*. Também é o momento em que a instância da FBE *Ticker* deve ser modificada para que sinalize o fim do ciclo. A regra que permite que ocorra essa transição foi chamada de Regra de Passo, e parte dela é apresentada na Figura 11.

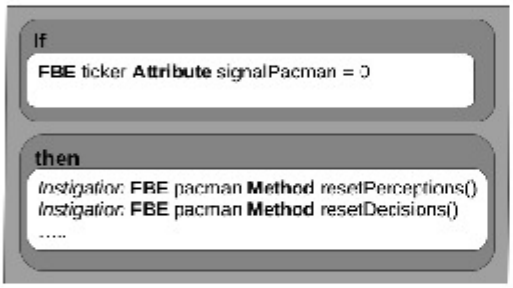


Figura 11. Exemplo de regra de passo.

A avaliação dessas camadas de regras cria, portanto, um ciclo de regras que é ilustrado na Figura 12. Cada passo que compõe este ciclo é implementado como um conjunto de regras, de modo que a execução das regras subsequentes depende do resultado das regras do passo anterior. Mais adiante serão apresentados artifícios adicionais que foram implementados para que este esquema funcionasse corretamente.

B. FBE Ghost e principais regras

A simulação do Fantasma é bem parecida com a simulação do Pac-Man. O que mais diferencia o comportamento de ambos os agentes são suas prioridades. O Fantasma não se importa com os pontos, tendo como fatores de decisão somente a presença de paredes e do Pac-Man. O ciclo de regras é o mesmo, mudando-se apenas as regras específicas de percepção e decisão. Outra diferença é o fato de que, ao ser comido pelo Pac-Man sob o efeito do ponto energizador, o Fantasma deve voltar à sua posição inicial.

C. FBE MazeLimits

Serve para manter uma instância única que define as dimensões x e y do labirinto. O labirinto pode ser aumentado

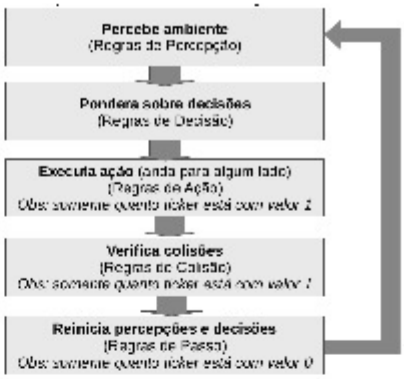


Figura 12. Ciclo de execução da simulação do Pac-Man.

ao se atribuir valores mais altos a seus atributos e ao criar novas instâncias de paredes e pontos.

D. FBE Wall, Dot e EnergizerDot

Todas essas FBEs têm em comum o fato de serem estáticas - no sentido de não se movimentarem pelo labirinto. A única diferença entre o *Wall* e o *Dot* (e também o *EnergizerDot*) é o atributo *exists* nos últimos, que permite sinalizar se o ponto está presente ou não, pois pode já ter sido comido pelo Pac-Man.

E. FBE Ticker

O Ticker foi implementado de modo a permitir as transições de estado e para facilitar a coordenação dos disparos de regras do ciclo. Esta FBE possui dois atributos: o *pacmanSignal* e o *ghostSignal*. Esta separação foi feita para permitir que futuramente a velocidade do Pac-Man e do Fantasma possam ser distintas. A ideia é que, quando *pacmanSignal* for igual a zero (valor padrão), será ativada a premissa que executará a regra de passo, que reinicia os atributos de percepção e decisão, conforme apresentado na Figura 11. Tal reinicialização é necessária para assegurar o disparo das notificações, que ocorre somente quando há mudança de valores de atributos. Isto também serve para evitar confusão envolvendo os atributos, principalmente os de decisão, cujos valores devem ser gerados especificamente para o atual estado do mundo.

A cada período de tempo definido para o *clock* de transição do jogo, o valor de *pacmanSignal* é modificado para 1. Tal mudança de estado do atributo de Ticker causa notificações para premissas que estão nas regras de decisão e ação. Todas essas regras contam com a premissa adicional de *ticker.pacmanSignal* ser igual a 1. Isto serve para "segurar"essas regras de modo que não sejam avaliadas precipitadamente, o que pode causar confusão nas decisões e ações. O atributo do *ticker* volta a valer 0 como resultado da execução de uma regra de ação.

VI. Implementação e Dificuldades

A implementação deste simulador foi totalmente realizada em LingPON, com exceção da parte visual - que consiste em imprimir o estado atual do labirinto na tela a cada mudança de estado - e do temporizador responsável por efetuar a mudança do estado dos atributos do *ticker* a cada período de tempo definido. Este é o principal diferencial deste trabalho em relação ao trabalho de [9], que apresentou uma versão híbrida de um simulador do jogo Pac-Man na qual parte doi implementada em C++ puro usando o POO e parte foi implementada usando o *framework* PON.

A princípio a implementação em LingPON é interessante por se tratar de uma linguagem dedicada ao paradigma. A sintaxe e palavras reservadas são simples e intuitivas. No entanto, por se tratar de uma linguagem ainda em desenvolvimento, apresenta algumas limitações.

Alguns artifícios adicionais tiveram que ser implementados para melhorar a coordenação dos disparos das regras e, assim, corrigir alguns *bugs* e facilitar os testes e depuração. Um novo atributo foi adicionado, tanto na FBE *Pacman* quanto na FBE *Ghost*, chamado *moved*, que sinaliza se naquele ciclo de *clock* o agente já se moveu ou não. Caso ele ainda não tenha se movido ($moved = 0$), a ativação de uma nova premissa que verifica esse atributo permite a execução do objeto *Acrion* das regras de decisão e ação. As instigações das regras de ação também ganharam uma nova chamada ao método *serMoved* que muda o atributo *moved* para 1. Quando isto acontece, as regras de ação são impedidas de serem executadas por meio da nova premissa, e as regras de colisão podem ser executadas com uma nova premissa que é ativada quando $moved = 1$. Finalmente, na regra de passo foi adicionada mais uma instigação para reiniciar este atributo para 0. Isto evita o *bug* dos agentes andarem mais de uma vez no mesmo ciclo, e facilita a depuração da aplicação ao coordenar melhor o disparo das regras.

Um dos principais problemas encontrados já foi mencionado brevemente na Seção V. A linguagem LingPON não fornece apoio à criação de múltiplas instâncias de forma compacta, pois também carece de estruturas de dados como listas ou conjuntos. Além disso, toda premissa envolve referências a instâncias específicas, não havendo a possibilidade de premissas genéricas para conjuntos de instâncias. Como ficará evidente na análise feita a seguir, isto dificulta o desenvolvimento pelo fato da linguagem não prover mecanismos que promovam o reuso de código, o que acarreta em um nível elevado de retrabalho.

A. Dificuldades de Implemenuação das Regras de Percepção

As regras de percepção devem ter subcondições ligadas a premissas que verifiquem as posições no campo de visão do Pac-Man - o mesmo se aplica às regras de percepção do Fantasma. Cada uma das posições no *grid* do labirinto que estejam no campo de visão devem ser comparadas com cada um dos objetos presentes no labirinto. Por exemplo, cada uma das 3 posições ao norte do Pac-Man devem ser comparadas com cada ponto no labirinto e com o Fantasma. A fórmula encontrada para calcular a quantidade de subcondições para as regras de percepção relacionadas às instâncias de uma FBE é dada a seguir:

$$QuantSubConds = QuantInstanctas \times AlcanceVtsao \times 4$$

Onde $QuantInstanctas$ é a quantidade de instâncias da FBE - e.g. 16 para as instâncias de pontos - $AlcanceVtsao$ é a distância em que o Pac-Man poderá enxergar as instâncias, e o número 4 refere-se às direções possíveis de se locomover no labirinto: norte, sul, leste e oeste.

Considerando o labirinto já apresentado na Figura 5, temos a seguinte relação das quantidade de subcondições para as regras de percepção:

- $(10 + 1) \times 1 \times 4 = 44$ subcondições para detecção de paredes. Isto se justifica pois temos 10 paredes no meio do labirinto e mais a checagem de limite. Isto deve ser feito para as quatro direções. O Pac-Man verifica apenas a existência de paredes adjacentes a ele próprio, portanto para este caso o alcance da visão é 1.
- $16 \times 3 \times 4 = 192$ subcondições para detecção de pontos. Isto se justifica pois temos 16 pontos (incluindo os energizantes). Deve-se verificar, para cada um dos pontos, se ele está no campo de visão do agente, isto é, em qualquer das 4 direções e num alcance de 3 unidades do labirinto.
- $1 \times 3 \times 4 = 12$ subcondições para detecção do Fantasma. Deve-se verificar se o Fantasma está no campo de visão do Pac-Man, de modo similar à detecção dos pontos.
- $1 \times 3 \times 4 = 12$ subcondições para detecção de Fantasma assustado. Similar à detecção de Fantasma normal.

No total, foram geradas 260 subcondições espalhadas em 16 regras. É um número elevado que revela o grau de retrabalho para implementar a aplicação utilizando o LingPON. Isto se deve principalmente à limitação do LingPON de não fornecer algum recurso que permita a criação de regras aplicáveis a uma classe - ou conjunto - de indivíduos. Caso isto fosse implementado no compilador, a quantidade de subcondições nesta aplicação do Pac-Man poderia ser reduzida de 260 para $44 = 8 + 12 + 12 + 12$, pois não seria necessário replicar subcondições para cada uma das 10 paredes e para cada um dos 16 pontos. Isto também permitiria adicionar uma maior quantidade de elementos, como mais paredes - permitindo a criação de labirintos maiores - mais pontos e mais Fantasmas sem causar grande impacto na carga de trabalho do desenvolvedor. De fato, a atual implementação do LingPON não provê meios adequados de reusabilidade do código e escalabilidade de desenvolvimento devido à falta de algum recurso da linguagem que apoie a criação de múltiplas instâncias e de regras que se apliquem a conjuntos de instâncias.

B. Dificuldades de Compilação

A primeira dificuldade com a compilação decorre de dois fatores: (i) a falta de maturidade do compilador para a compilação de aplicações maiores e mais complexas, e (ii) o tamanho do código decorrente das limitações apresentadas na Seção VI-A. O código resultante da implementação do simulador em LingPON chegou a ultrapassar 5000 linhas

devido ao alto número de regras e subcondições. À medida que o tamanho do código aumentava, notou-se que o compilador falhava cada vez mais. É interessante também observar que, até certo ponto do desenvolvimento, era possível compilar o código para todas as três linguagens alvo: *framework*, C++ e C. À medida que o código aumentava, falhas na compilação passaram a se tornar mais comuns até o ponto em que não era mais possível compilar. Esta causa das falhas de compilação - o tamanho do código - pôde ser constatada ao separar partes do código em diferentes arquivos e compilá-los separadamente, o que foi em geral bem sucedido. Portanto, o compilador ainda carece de robustez para ser capaz de compilar códigos grandes.

Como mencionado, a solução encontrada para superar esse problema foi dividir o código em diferentes arquivos e compilá-los separadamente. Isto levou a outra dificuldade: mesclar os códigos gerados pelas diferentes compilações. Tal mesclagem é razoavelmente fácil de se realizar se o código for compilado para o *framework* PON. Neste caso, em geral, é necessário modificar apenas dois arquivos do código gerado: o *Main.h* e o *Main.cpp*, nos quais são declaradas e definidas as regras, premissas e instigações. As diferenças entre as diferentes partes do código dividido são apenas novas regras (com suas condições, premissas e instigações), pois as definições das FBEs permanecem as mesmas em todos os arquivos. Portanto, torna-se relativamente fácil realizar esta mesclagem que consiste apenas em copiar o código referente à criação das regras no formato do *framework* PON.

Apesar dessa divisão no código em LingPON ser algo negativo decorrente das limitações da linguagem, há uma observação interessante a se destacar a partir disso. A facilidade em dividir o código em várias partes mantendo-se apenas as FBEs em comum é um indício do baixo acoplamento inerente ao PON. É interessante como, embora deva haver uma coordenação entre as diferentes camadas de regras da aplicação, elas mantêm uma certa independência.

A solução de se compilar o código em partes, no entanto, não funciona tão bem para a compilação em C e C++. De modo a traduzir o código em PON para o Paradigma Procedimental e para o POO, é necessário que várias adaptações sejam feitas. Por exemplo, em C++, para cada FBE uma classe é criada, e os métodos da FBE são traduzidos para métodos da classe. No entanto, como no PON o método é chamado para atualizar atributos, e a atualização de atributos resulta em notificações de premissas que envolvem estes atributos, é necessário que essas notificações fiquem acopladas ao código do método no C++. Portanto, cada método tem em seu corpo, além de seu próprio código, também chamadas a procedimentos que simulam as notificações no código gerado em C++. Logo, a mesclagem dos códigos compilados separadamente em C++ e em C não é simples como a mesclagem dos códigos em *framework*, pois envolve a modificação de várias partes do código em que é feita essa relação entre os atributos e métodos das FBEs e as regras.

Infelizmente esta última dificuldade impediu a compilação do simulador completo em C++ para fins de comparação com o código gerado em *framework*, o que seria algo de interesse para os pesquisadores do PON.

Outra dificuldade encontrada, em particular à compilação em *framework* PON, é o fato de regras com múltiplas subcondições não estarem compilando corretamente. As múltiplas subcondições em PON envolvem a utilização do operador lógico OR entre cada subcondição. Ao compilar para o *framework*, tais operadores lógicos se perdem, gerando um código inconsistente com o que foi escrito em LingPON. Para resolver o problema, foi necessário desmembrar as regras com mais de uma subcondição em várias regras separadas, o que aumentou mais ainda o tamanho do código e o retrabalho.

VII. Sugestões de melhorias

Como foi constatado, ainda existem melhorias a serem feitas para que o LingPON possa amadurecer como um compilador confiável para o PON. Isto se deve ao fato dele ainda estar em fase de desenvolvimento, portanto sugestões de correções e melhorias são válidas.

A primeira sugestão a ser feita é a correção de *bugs* ainda presentes no compilador, como as falhas de compilação com códigos muito grandes. Isto facilitaria a implementação e os testes para as novas aplicações, e também permitiria a compilação em C e C++ para que se possa comparar com o código gerado em *framework*.

Dentre as melhorias, acredito que uma das prioridades deveria ser implementar alguma forma de criação mais facilitada de instâncias e de regras genéricas para classes - ou conjuntos - de instâncias. Isto por si só, como já mencionado na Seção VI-A, já reduziria em grande proporção o tamanho do código, a dificuldade de desenvolvimento e o retrabalho. Possivelmente seja necessária a implementação de estruturas de dados, como listas e/ou conjuntos, para tornar isto possível.

Outra sugestão de melhoria é um detalhe que possibilitaria o desenvolvimento de operações mais encapsuladas e, consequentemente, código mais coeso. Atualmente só é possível que os métodos sejam compostos de uma única linha de código. Portanto, quando uma única operação deve implicar na modificação de múltiplos atributos da FBE, é necessário que sejam criados vários métodos distintos. Cada vez que esta operação deve ser chamada numa instigação, todos os métodos que compõem esta única operação devem ser chamados um a um. Isto foi necessário, por exemplo, nas regras de passo implementadas no simulador do Pac-Man, nas quais é necessário que todos os atributos de percepção e decisão sejam reiniciados. Para isto, foi necessária a criação de vários métodos para reinicializar cada atributo de percepção. Se houvesse a possibilidade de escrever métodos com múltiplas linhas de código, um único método que reinicializasse todos os atributos seria suficiente e muito mais interessante para a criação de um código mais coeso e com maior reusabilidade.

Os trabalhos futuros envolvem a necessidade de se aplicar as melhorias sugeridas para o compilador LingPON, assim como melhorias na própria aplicação desenvolvida como estudo de caso. Tais evoluções desses trabalhos permitirão fazer comparativos entre as linguagens geradas pelo compilador. A aplicação do Pac-Man, após maiores evoluções, também pode vir a se tornar um modelo de como projetar e implementar aplicações em PON, em especial quando se trata de jogos e simuladores de jogos.

VIII. Conclusões e opinião sobre o paradigma

O Paradigma Orientado a Notificações apresenta uma forma muito diferente de se pensar sobre a resolução de problemas por meio de recursos computacionais. No início houve uma certa dificuldade em conceber o modo como a aplicação de exemplo funcionaria, pois a programação em PON diferencia-se muito do modo como os programadores estão acostumados a programar. Mesmo assim, após estudá-lo melhor, percebe-se que, na realidade, a programação em PON é intuitiva, talvez até mesmo mais intuitiva que os paradigmas vigentes sob as lentes de uma análise imparcial. Tal intuitividade também se reflete na implementação em LingPON, por mais que a linguagem esteja ainda em fase de desenvolvimento. As principais dificuldades ao se programar com o LingPON foram técnicas, decorrentes do estágio ainda pouco amadurecido da linguagem e do compilador.

Outro ponto que se pôde notar é o baixo acoplamento da aplicação. Mesmo que haja dependências entre as regras, o que acontece em qualquer software, elas mantêm uma certa independência entre si. Por exemplo, em nenhum momento é necessário que uma regra faça uma referência direta a outra. Toda a coordenação é feita por meio das notificações causadas pelas mudanças de atributos no decorrer do fluxo da aplicação. Isto ficou visível também quando foi necessário dividir o código em arquivos separados e compilá-los separadamente. O LingPON não provê ainda mecanismos de importação de código, e mesmo assim tal divisão foi possível sem grandes problemas, graças a esse evidente baixo acoplamento entre as regras, que constituem parte fundamental de uma aplicação em PON. De fato, caso houvesse um mecanismo que fizesse a mesclagem automática dos códigos compilados separadamente, seria possível escrever cada regra em um arquivo separado, mantendo-se apenas as definições das FBEs. Isto pode tornar bem flexíveis as refatorações e modularizações do código.

Enfim, é interessante notar uma das principais vantagens propostas pelo PON: a evitação de redundâncias temporais e espaciais [3]. Ao executar a aplicação gerada no *framework*, por exemplo, as regras que vão sendo disparadas são escritas no *console*. É interessante observar como as premissas das regras não são exaustivamente avaliadas graças à ideia das notificações, na qual as premissas são avaliadas à medida que ocorrem mudanças nos valores dos atributos que elas verificam. Isto, se de fato comprovado por meio de mais experimentos e melhorias nas materializações, tem um grande potencial de contribuição para a comunidade científica de computação e áreas afins.

IX. Bibliografia Consultada

A bibliografia consultada para o desenvolvimento deste trabalho está listada na Seção de referências.

Referências

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, April 1965.

[2] J. M. Simão and P. C. Stadzisz, "Inference based on notifications: a holonic metamodel applied to control issues," *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, vol. 39, no. 1, pp. 238–250, 2009.

[3] R. F. Banaszewski, "Paradigma Orientado a Notificações: Avanços e Comparações," Master's thesis, Universidade Tecnológica Federal do Paraná, 2009.

[4] 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.

[5] R. Xavier, "Paradigmas de desenvolvimento de software: comparação entre abordagens orientada a eventos e orientada a notificações," Master's thesis, Universidade Tecnológica Federal do Paraná, 2014.

[6] C. A. Ferreira, "Linguagem e Compilador para o Paradigma Orientado a Notificações (PON): Avanços e Comparações," Universidade Tecnológica Federal do Paraná, Curitiba, PR, Tech. Rep., 2014.

[7] J. Pittman, "The Pac-Man Dossier," 2015, disponível em http://home.comcast.net/ jpittman2/pacman/pacmandossier.html. Acesso em 27/06/2015.

[8] J. G. Brookshear, *Computer science - an overview (4. ed.)*. Benjamin/Cummings, 1994.

[9] A. F. Ronszcka, D. L. Belmonte, G. Z. Valença, M. V. Batista, R. R. Linhares, C. A. Tacla, P. C. Stadzisz, and J. M. Simão, "Comparações quantitativas e qualitativas entre o paradigma orientado a objetos e o paradigma orientado a notificações sobre um simulador de jogo," in *III Congresso Intern. de Computación y Telecom.-COMTEL, Lima, Peru*, 2011.

[10] M. Wooldridge, *An Introduction to Multiagent Systems*, 2nd ed. Chichester, UK: Wiley, 2009.

Paradigma Orientado a Notificações aplicado sobre simulador da Robocup

Leonardo Araujo Santos
Pós-Graduação em Engenharia Elétrica e Informática Industrial
Universidade Tecnológica Federal do Paraná
Curitiba, Paraná
Email: leonardo.2013@alunos.utfpr.edu.br

***Resumo*—Esse artigo é resultado do trabalho prático apresentado na disciplina "Paradigma Orientado a Notificações"do CPGEI na primeira fase do ano de 2015. É apresentado uma comparação entre o PI e o PON no desenvolvimento do código de controle dos robôs da *Robocup Small Size League*. Também é apresentado um possivel avanço para o PON no que diz respeito a forma em que as FBEs se relacionam com as *Rules*.**

I. Introdução

Ao longo dos últimos anos, o Paradigma Orientado a Notificações vem sendo estudado e discutido como uma possível alternativa para problemas observados nos principais paradigmas de programação vigentes (Paradigma Imperativo e Paradigma Declarativo)[4]. O PON visa unificar as principais vantagens do PD e do PI para tentar solucionar várias de suas deficiências, como demonstrado em [4][8]. À luz de sua teoria, o PON é naturalmente composto de entidades distribuídas de processamento, uma vez que suas entidades básicas são completamente autônomas[15] . De forma a verificar essa propriedade do PON, um exemplo de código de controle de robôs da *Robocup* foi remodelado e re-codificado utilizando o LingPON.

II. Paradigma Orientado a Notificaçoes

O Paradigma Orientado a Notificações (PON) foi concebido a partir do conceitos apresentados por Jean Marcelo Simão em sua dissertação de mestrado e tese de doutorado [1][2]. Esse paradigma emergente visa solucionar certos problemas existentes nos paradigmas usuais de programação, nomeadamente o Paradigma Declarativo (PD) e o Paradigma Imperativo (PI)[3], principalmente em relação às redundâncias estruturais, redundâncias temporais e forte acoplamento de entidades que compões expressões causais[15].

O PON é um paradigma que se inspira e evolui particularmente de dois sub-paradigmas do PI e PD, são eles o Paradigma Orientado a Objetos (POO) e o Paradigma Lógico com ênfase em Sistemas Baseados em Regras (SBR). Quando comparado com o POO, observa-se que o PON reaproveita seus principais conceitos, tais como abstração em classes/objetos e flexibilidade na programação. O PON também reaproveita certos conceitos dos SBR, como a representação de conhecimento em termos de regras e facilidades de programação declarativa [8].

O PON apresenta uma nova forma de realizar avaliações de expressões lógicas através de entidades computacionais de pequeno porte, ativas e desacopladas que são criadas a partir de uma base de regras e colaboram por meio de notificações pontuais.[7][4][5][6]

Em PON, um elemento da base de fatos é genericamente modelado pela classe *FBE* (*Fact Base Element*). Cada *FBE* trata de seus atributos através de objetos da classe *Attribute* e seus serviços por meio de objetos da classe *Method*. Por meio de seus *Attributes* e *Methods*, os objetos *FBE* são passíveis de relações causais por meio de *Rules*, as quais se constituem em elementos fundamentais do PON [15] e gerenciam o conhecimento sobre qualquer comportamento causal no sistema.

Por sua vez, cada *Rule* é composta por objetos da classe *Condition* e *Action*. A *Condition* trata da decisão da *Rule*, enquanto a *Action* trata da execução das ações da *Rule*. Sendo assim, *Condition* e *Action* trabalham juntas para realizar o conhecimento lógico e causal da *Rule* [14][7][4][5].

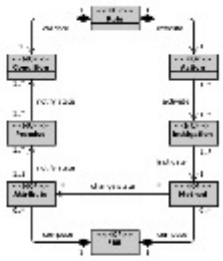


Figura 1. Entidades PON

Cada *Condition* é composta por uma ou mais *Premises*, que avalia o estado de um ou dois *Attributes* de *FBE*. Quando o estado de um *Attribute* se altera, esse *Attribute* notifica somente as *Premises* relevantes. Cada *Premisse*, de forma similar, notifica somente as *Conditions* relevantes dos seus estados. Baseado nos estados notificados, uma *Condition* pode ser aprovada ou não. Em caso positivo, a respectiva *Rule* ativa sua *Action*.

Por sua vez, a *Action* também se conecta a outro objetos, as *Instigations*. As *Instigations* estão conectadas a um ou mais *Methods* de um *FBE* e são responsáveis por realizar seus serviços. Por fim, a execução de um *Method* geralmente altera o valor dos *Attributes* de um *FBE*, iniciando assim um novo ciclo de notificações [14][7][4][5].

A essência da computação no PON está organizada e distribuída em entidades autônomas e reativas que colaboram por meio de notificações pontuais [15]. Isso faz com que o PON apresente uma nova maneira de estruturar e conceber artefatos de *software*, permitindo sua execução otimizada e desacoplada[14][7][4][5].

De forma a comparar o desempenho do PON frente aos paradigmas de programação vigentes, o PON foi primeiramente materializado na forma de um *Framework* implementado na linguagem de programação C++. Utilizando essa primeira materialização, o PON foi comparado em termos de desempenho com implementações PI/POO e PD/SBR. Essas comparações apresentaram resultados favoráveis ao PON [8].

Entretanto, observou-se em [4] e [10] que tais materializações não proporcionam um resultado condizente com os cálculos assintóticos de desempenho do PON. Portanto, chegou-se a conclusão de que uma linguagem e um compilador deveriam ser desenvolvidos de forma a traduzir o código PON para código alvo, de forma a reduzir a dependência de estruturas de dados complexas que poderiam comprometer o desempenho.

Dessa forma, uma linguagem nativa para o PON - Ling-PON - e um compilador foram apresentados em [11]. Essa linguagem foi idealizada de forma a expor de maneira simples os conceitos relativos ao PON e permitir a criação de aplicações sem a utilização de um framework[11].

III. Descrição do Ambiente Simulador *Robocup*

A *Robocup* busca promover pesquisas nas áreas de robótica e inteligência artificial através da proposição de plataformas estimulantes fundamentadas em problemas do mundo real que sejam capazes de atrair o grande público[12].

Uma das categorias dessa competição é a *RoboCup Small Size League*. Nessa categoria, cada time é composto por seis robôs autônomos, ou seja, executam ações sem intervenção humana. Eles são identificados e rastreados por duas câmeras centrais, as quais enviam essas informações para um computador externo, o qual as processa e envia comando para os robôs através de comunicação *wireless*.

Visando permitir o desenvolvimento do *software* sem depender de robôs reais, foi utilizado um ambiente simulado da *Robocup*. Esse ambiente é composto por duas aplicações: *grSim Simulator* e *Referee Box*.

O *grSim Simulator* é um simulador funcional do ambiente de jogo utilizado na *Robocup*. Os robôs simulados possuem características muito próximas às reais, tais como dimensões e velocidades. Além disso, essa aplicação é responsável por simular a leitura óptica da posição dos robôs e da bola e disponibilizá-las através de uma interface de rede à aplicação de controle dos robôs. A aplicação de controle irá processar esses dados e enviar comandos referente à cada um dos robôs ao *grSim Simulator*, que irá os executar e atualizar as novas percepções do ambiente. Portanto, o *grSim Simulator* não possui lógica alguma de controle, apenas executa comandos recebidos e disponibiliza o estado atual do ambiente de jogo à quem interessa.

Outra aplicação que compõe o ambiente simulado é o *Referee Box*. Essa aplicação visa simular o árbitro da partida, sendo

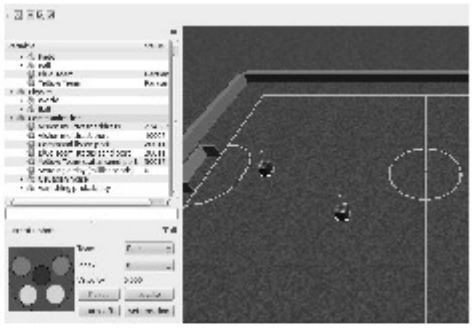


Figura 2. Visualização da interface gráfica do *grSim Simulator*.

Figura 3. Visualização da interface gráfica do *Referee box*.

responsável por determinar as condições de jogo, por exemplo: início de partida, determinação de a qual time pertence a posse de bola, marcação de falta, validação de gol e etc. Na atual versão do *Referee Box*, isto é feito manualmente. Quando o usuário escolhe um comando, a aplicação o disponibiliza através da interface de rede (porta 10001), representado-o por uma letra. Por exemplo, sempre que o usuário escolhe o comando 'Stop Game', o *Referee Box* envia a letra 'S'.

Baseando-se nos comandos recebidos de ambas aplicações, o *software* de controle deve deliberar sobre qual será a próxima ação a ser executada por cada um dos robôs que está sob seu controle e enviar tais comandos ao *grSim Simulator*, conforme mostrado na imagem 4. Tal processamento é baseado em regras pré-definidas que avaliam as atuais condições de jogo para calcular alguns parâmetros, tais como: direção tangencial e normal que o robô deve prosseguir, a sua velocidade angular e se o robô deve conduzir a bola ou chutá-la. Tais parâmetros devem ser enviados ao *grSim Simulator* para que seja executado.

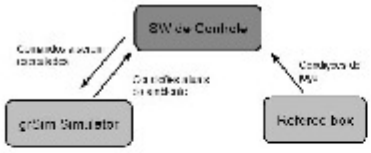


Figura 4. Arquitetura funcional do ambiente da *Robocup*.

IV. Comparação PI x PON

De forma a realizar uma comparação qualitativa entre o PON e o PI, o código de controle dos jogadores do *Robocup*, que atualmente está escrito em C++, foi reescrito utilizando o LingPON.

A atual implementação do algoritmo de controle dos jogadores apresenta resultados satisfatórios quanto a sua performance e eficiência. Entretanto, foram observados alguns problemas tais como forte acoplamento entre classes e expressões causais e controle centralizado, impossibilitando a distribuição de processamento.

Na atual implementação (C++), o controle dos robôs é feito através de uma única classe. Conforme visto no fragmento de código abaixo, a classe *Strategy* centraliza a tomada de decisão, determinando qual comando cada um dos robôs irá executar. No caso do Código 1, quando *Referee Box* envia o comando *"IndirectKickBlue"*, a classe *Strategy* escolhe um robô para executar tal comando através da função *getKicker()* e o posiciona próximo a bola, como mostrado no fragmento de código abaixo.

Código 1. Regra de 'Indirect Kick' em C++.

```
switch ( cmd_Juiz )
{
    case IndirectKickBlue:
    {
        if (getTeamColor() == BLUE)
        {
            Robot *robot = getKicker();
            goToKickPosition(robot);
        }
    }
}
```

Nesse cenário, o robô é um mero executor de comandos, sem influência nenhuma sobre a decisão de que comando irá executar. Sendo assim, utilizando essa abordagem não é possível distribuir o processamento da lógica que controla as ações dos robôs.

Remodelando o sistema para adequá-lo ao LingPON, cada robô, o juiz e a bola foram considerados como sendo FBEs, tendo atributos e métodos inerentes. Em Código 2 é aprsentado, de forma simplificada, os atributos e métodos que compõe a FBE *Robot*. Portanto, não existe mais uma entidade central que decide quando o robô irá executar suas ações e sim *Rules* que, utilizando notificações recebidas de cada uma das FBEs que compõe o sistema, disparam a execução dos métodos de cada robô através das *insrigarions*.

Código 2. FBE Robot em LingPON

```
fbe Robot
    attributes
        string playPosition "Indefinida"
        integer PosX -1
        integer PosY -1
        integer PosToGoX -1
        integer PosToGoY -1
    end_attributes
    methods
        method execMove()
        method position_indirect_kick_right()
        method position_indirect_kick_left()
        method position_receive_ball()
    end_methods
end_fbe
```

A *Rule* apresentada em Código 1 foi reescrita em LingPON e suas *Premises* são apresentadas no Código 3 e sua *insrigarion* no Código 4. Suas *Premises* avaliam se o comando recebido pelo juiz é de "Indirect Kick"e se a bola está no lado do campo em que o robô está jogando. Caso essas condições sejam satisfeitas, o robô deverá se mover para a posição que o possibilite chutar a bola.

Código 3. Premisses da *Rule* de 'Indirect Kick' em LingPON.

```
premise gameController.atCmdReferee == 'I' and
premise robot.playPosition == "LeftRobot" and
premise ball.posY >= 0
```

Código 4. Instigation da *Rule* de 'Indirect Kick' em LingPON.

```
instigation robot.position_indirect_kick_left();
```

Essa abordagem fez com que cada robô se comportasse como uma entidade autônoma de processamento, tornando trivial a tarefa de executar esse processamento de forma distribuída. Para tal, seria necessário apenas que cada uma das FBEs do sistema soubesse, à priori, o endereço das *Rules* às quais colabora. Da mesma maneira, *insrigarions* das *Rules* deveriam saber o endereço das FBEs, de forma a notificá-las quando necessário.

Durante o desenvolvimento das *Rules* que regem o controle de comportamento dos jogadores, notou-se que para cada nova instância criada do tipo *Robot* uma séria de *Rules* haviam de ser duplicadas. Isso ocorre por uma limitação do PON, o qual determina que cada *Rule* esteja diretamente associado à uma instância específica da FBE. Com esse cenário, chegou-se a conclusão de que algumas *Rules* são inerentes à existência da FBE, isto é, toda nova instância de uma dada FBE que for criada deve possuir, por padrão, essas *Rules*. Isso se torna possível quando as *conditions* dessas *Rules* dependem única e exclusivamente dos atributos da própria FBE e suas *insrigarions* estão relacionadas aos métodos da própria FBE.

Para exemplificar tal ideia, analisaremos a *Rule* que controla o posicionamento de um dado robô. A figura 5 apresenta o fluxo de processamento para tal *Rule*. Primeiramente, a posição atual do robô é lida através de sensores. Dado as características do PON, a *Rule* somente será ativada caso a posição do robô tenha se alterado desde sua última avaliação. Nesse caso, a *Rule* verifica se a posição atual do robô é igual à posição para a qual ele deseja ir.Caso seja, o robô já atingiu sua posição de destino e pode permanecer parado em tal posição. Entretanto, em caso negativo, o robô deve recalcular a velocidade de seus motores de forma a continuar tentando atingir a posição de destino.

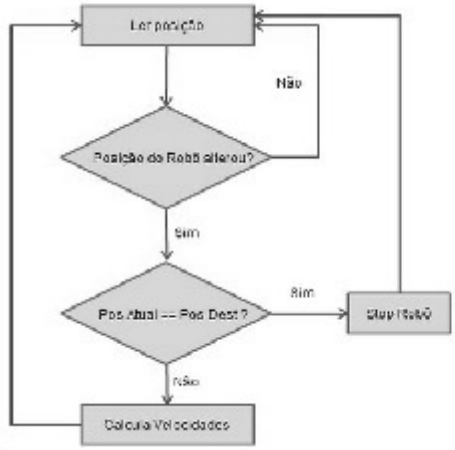


Figura 5. Fluxograma que representa a *Rule* de controle de posição do robô.

Portanto, faz-se necessário um estudo para entender quais vantagens a criação de *Rules* dentro de um escopo de FBE trariam para sistemas desenvolvidos utilizando o Paradigma Orientado a Notificações e como tais *Rules* poderiam ser modeladas, de forma a não ir de encontro com os princípios teóricos já estudados e consolidados do PON.

V. Conclusão

Esse trabalho apresentou uma comparação qualitativa entre o PON e PI/POO em relação à facilidade de desenvolvimento de um *software* de controle para os robôs da *Robocup*. Por suas características naturais, o PON apresenta maior facilidade de desenvolver código sem redundâncias estruturais ou temporais e forte acoplamento, facilitando assim o desenvolvimento de software paralelo ou distribuídos.

Quando desenvolvido em PON, utilizando sua linguagem nativa LingPON, as *Rules* que regem o funcionamento do sistema de controle são apresentadas de maneira mais clara e direta, uma vez que sua composição é feita através de entidades bem definidas e explicitamente declaradas. Entretanto, o LingPON ainda está em fase de desenvolvimento. Portanto, surgiram algumas dificuldades para o desenvolvimento de uma aplicação complexa, tais como falta de suporte de declaração de um *FBE* como *Attribute* de outro *FBE*. Porém, mesmo com certas limitações, o LingPON ainda se apresenta como uma promissora ferramente para o desenvolvimento de *software* em PON. Isso porque seu compilador poderá permitir a geração de código em diversas linguagens de programação.

Durante o desenvolvimento do *software* de controle, observou-se um possível campo de pesquisa sobre o PON. Atualmente, as *Rules* são compreendidas como entidades de escopo global no sistema. Observou-se que, para certos casos, as *Rules* poderiam ser definidas internamente ao escopo de um *FBE*. Utilizando essa abordagem, seria possível que cada objeto novo criado a partir de uma definição de *FBE* possuiria, internamente, esse conjunto de *Rules*. Isso reduziria a necessidade de duplicação de *Rules* no sistema.

Referências

[1] Simão, Jean Marcelo. Proposta de uma Arquitetura de Controle para Sistemas Flexíveis de Manufatura Baseada em Regras e Agentes. Dissertação de Mestrado, CPGEI/UTFPR, 2001.

[2] Simão, Jean Marcelo. A Contribution to the Development of a HMS Simulation Tool and Proposition of a Meta-Model for Holonic Control. Teste de Doutorado, CPGEI/UTFPR, 2005.

[3] Ronszcka, Adriano Francisco. Contribuição para a concepção de aplicações no paradigma orientado a notificações (PON) sob o viés de padrões. Dissertação de Mestrado, CPGEI/UTFPR, 2012.

[4] Simão, Jean M, Tacla, Cesar A, Stadzisz, Paulo C Banaszewski, Roni F (2012). Notification Oriented Paradigm (NOP) and Imperative Paradigm: A Comparative Study. Journal of Software Engineering and Applications, 5, 402.

[5] Belmonte, D. L., Ronszcka, A. F., Linhares, R. R., Banaszewski, R. F., Tacla, C. A., Stadzisz, P. C., Batista, M. V. (2012). Notification Oriented and Object Oriented Paradigms Comparison via Sale System. Journal of Software Engineering and Applications, 5(09), 695.

[6] Linhares, R. R., Belmonte, D. L., Banaszewski, R. F. Comparações entre duas materializações do Paradigma Orientado a Notificações (PON): Framework PON Prototipal versus Framework PON Primário.

[7] Linhares, R. R., Ronszcka, A. F., Valença, G. Z., Batista, M. V., Witt, F. A., Lima, C. R. E., ... Stadzisz, P. C. (2011, October). 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 Intern. de Computación y Telecom.-COMTEL, Lima, Peru.

[8] Roni Fábio Banaszewski. Paradigma Orientado a Notificações: Avanços e Comparações. Dissertação de Mestrado, CPGEI/UTFPR. Curitiba, 2009.

[9] Glauber Zárate Valença. Contribuição para a Materialização do Paradigma Orientado a Notificações (PON). Dissertação de Mestrado, PPGCA/UTFPR. Curitiba, 2012.

[10] Cleverson Avelino Ferreira, Jean Marcelo Simão, Paulo Cézar Stadzisz e Márcio Venâncio Batista. Notification Oriented Paradigm (NOP) and Object Oriented Paradigm (OOP): A Comparative Study by means of a Sale Order System. COMTEL 2013. Lima, Peru, 2013.

[11] Cleverson Avelino Ferreira. Linguagem e Compilador para o Paradigma Orientado a Notificações (PON): avanços e comparações. Seminário de acompanhamento II, PPGCA/UTFPR. Curitiba 2014.

[12] "RoboCup Brasil Site Oficial", http://www.robocup.org.br/ (Acessado em 03/06/2015).

[13] Simão, J. M.; Stadzisz, P. C. "Paradigma Orientado a Notificações (PON) – Uma Técnica de Composição e Execução de Software Orientado a Notificações". Pedido de Patente junto ao INPI/Brazil em 2008 e a Agência de Inovação/UTFPR em 2007. No INPI Efetivo PI0805518-1

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

[15] Robson Duarte Xavier. Paradigmas de Desenvolviment de Software: Comparação entre abordagens Orietada a Eventos e Orientada a Notificações. Dissertação de Mestrado, PPGCA/UTFPR. Curitiba, 2014.

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

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

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

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

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

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

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

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

I. INTRODUÇÃO

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

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

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

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

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

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

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

II. PARADIGMA ORIENTADO A NOTIFICAÇÕES (PON)

O PON foi proposto como um novo paradigma de desenvolvimento de software, que apresenta algumas vantagens

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

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

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

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

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

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

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

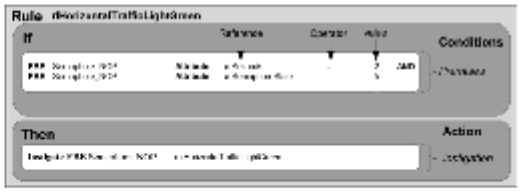


Figura 1. Exemplo de *Rule* para controle independente.

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

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

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

III. DESCRIÇÃO DO CTA SIMULATOR

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

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

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

A. Simulador

O simulador representa elementos do mundo real, como veículos, ruas, pistas, quadras, sinaleiros, sensores e cruzamentos em uma região de simulação matricial composta por dez ruas verticais e 10 ruas horizontais de mão única, formando uma matriz 10x10 com um total de 100 interseções [14].

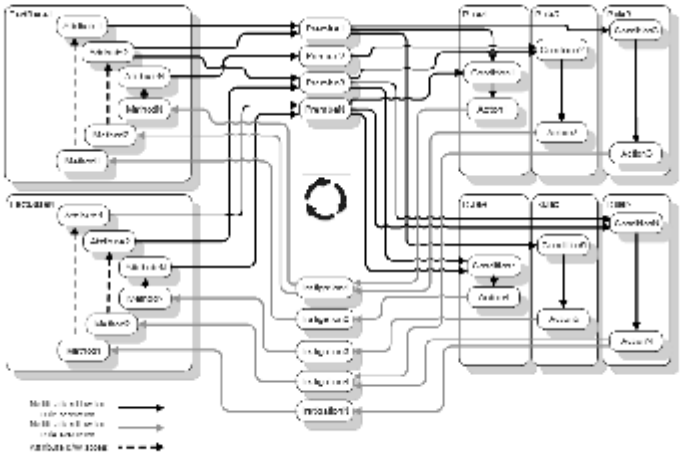


Figura 2. Colaboração por notificações das entidades do PON [12]

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

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

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

B. Estratégias de controle

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

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

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

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

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

C. Desenvolvimento do CTA

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

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

Figura 3. Diagrama de estados para controle independente.

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

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

Figura 5. Exemplo de FBE para controle independente.

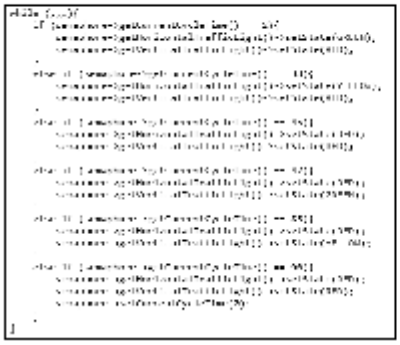


Figura 6. Exemplo de código em PI.

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

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

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

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

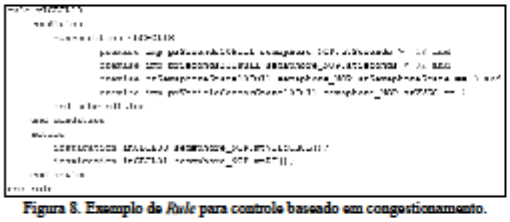


Figura 8. Exemplo de *Rule* para controle baseado em congestionamento.

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

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

Para implementação da estratégia baseada em tráfego facilitado foi desenvolvido um novo diagrama de estados baseado no diagrama da Figura 3 e na descrição da estratégia na Seção III B. Para esta estratégia, foram implementadas 23 *Rules*, que avaliam o tempo, o estado atual, sensor de congestionamento e a *flag* de tráfego facilitado de cada semáforo. Porém, durante a implementação desta estratégia de controle, foram encontradas dificuldades no desenvolvimento devido à estrutura da implementação do código equivalente em C++.

IV. RESULTADOS

Os experimentos realizados tiveram como objetivo analisar o desempenho das estratégias de controle implementadas em PON, comparadas as implementações em PI. Para isso, foram desenvolvidas as estratégias de controle independente e baseado em congestionamento descrito anteriormente, em ambos os paradigmas. Para executar os experimentos foi utilizado um notebook com processador core i7 – 4500U 1.8GHz – 2.4GHz e 8GB de memória RAM, com sistema operacional Windows 8.1. O sistema operacional Windows 8.1 foi utilizado devido ao ambiente de desenvolvimento escolhido.

Para análise dos experimentos de desempenho, a aplicação foi executada alterando o número de repetições: 90, 200, 500, 1000, 1500 e 2000, na qual cada repetição representa 1 segundo do semáforo. A Tabela I e a Figura 9 apresentam os resultados dos experimentos executados, que correspondem ao tempo de execução em milissegundos das duas estratégias implementadas, alterando o número de repetições.

TABELA I. RESULTADOS RELATIVOS AO EXPERIMENTO 1.

Strategy	90	200	500	1000	1500	2000
Independent_Strategy	17.243	38.380	104.793	201.136	301.194	399.688
Independent_Strategy_NOP	20.854	50.355	132.532	270.337	408.743	550.617
Congestion_Level_Strategy	24.413	55.197	154.076	306.681	459.726	610.568
Congestion_Level_Strategy_NOP	32.467	74.002	199.991	406.324	605.500	805.553

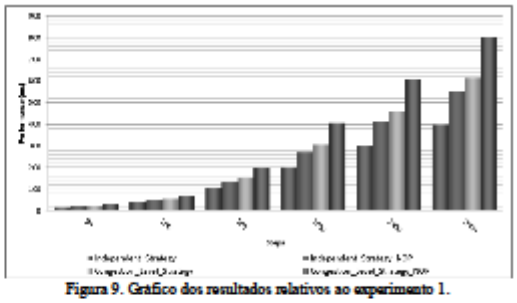


Figura 9. Grafico dos resultados relativos ao experimento 1.

A partir dos dados da Tabela I e da Figura 9, observou-se que a implementação em PON teve desempenho inferior quando comparado ao da implementação em PI. Porém, isso ocorreu devido ao fato de que a implementação da pertinência não estava sendo feita da forma correta, além do código equivalente gerado em C++ ainda apresentar redundâncias estruturais. Desta forma, o código gerado foi alterado manualmente, seguindo os princípios do PON para eliminar redundâncias estruturais e implementar o conceito de *Premises* impertinentes de maneira correta. Após as alterações, os experimentos foram executados novamente, e gerados os dados da Tabela II e da Figura 10.

TABELA II. RESULTADOS RELATIVOS AO EXPERIMENTO 2.

Strategy	90	200	500	1000	1500	2000
Independent_Strategy	17.243	38.380	104.793	201.136	301.194	399.688
Independent_Strategy_NOP	20.789	48.355	117.233	237.906	385.907	477.710
Congestion_Level_Strategy	24.413	55.197	154.076	306.681	459.726	610.568
Congestion_Level_Strategy_NOP	22.490	50.112	128.372	271.875	388.382	513.299

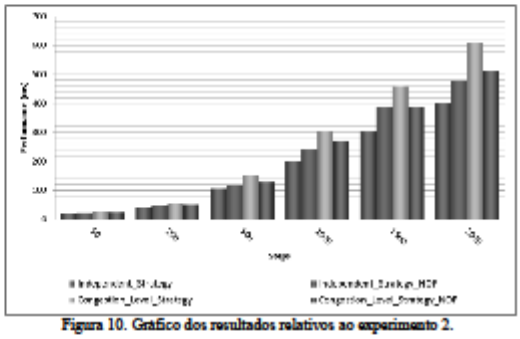


Figura 10. Grafico dos resultados relativos ao experimento 2.

Após os devidos ajustes a partir do código equivalente gerado em LingPon, os resultados da estratégia baseada em congestionamento obtiveram melhor desempenho em relação à implementação da mesma estratégia desenvolvida em PI. Esse resultado se deu pela eliminação das redundâncias estruturais que ainda estão presentes nas entidades *Premisas* e na avaliação dessas entidades quando envolvem *Attributos* de tempo ou sensores de congestionamento, que não são pertinentes.

V. CONCLUSÕES E TRABALHOS FUTUROS

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

Este trabalho apresentou o desenvolvimento de uma aplicação para a comparação de desempenho entre software desenvolvido em PON e software desenvolvido em PI (POO). A avaliação dos resultados permite elaborar conclusões não somente relativas à comparação de desempenho, mas também a respeito da complexidade de desenvolvimento sob este novo paradigma.

Quando realizados os primeiros experimentos comparando a implementação da aplicação CTA Simulator em ambos os paradigmas, a implementação em PI se demonstrou mais eficiente do que a implementação equivalente em PON. Porém, a partir das alterações realizadas no código gerado a partir do LingPon, foi possível verificar melhora no desempenho do PON, com relação a mesma implementação em PI, devido a eliminação de redundâncias temporais presentes no código em PI. Além da melhora de desempenho, percebeu-se uma facilidade para incluir e alterar o código gerado em PON em um software que já havia sido desenvolvido utilizando outro paradigma. No caso deste artigo, foi inserido no CTA Simulator, sob paradigma orientado a objetos.

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

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

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

REFERÊNCIAS

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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