LUIZ CARLOS VIANA MELO

ADAPTAÇÃO DO PARADIGMA ORIENTADO A NOTIFICAÇÕES PARA DESENVOLVIMENTO DE SISTEMAS *Fuzzy*

Dissertação submetida ao Programa de Pós-Graduação em Computação Aplicada da Universidade Tecnológica Federal do Paraná como requisito parcial para a obtenção do título de Mestre em Computação Aplicada.

Curitiba PR
Agosto de 2016

LUIZ CARLOS VIANA MELO

ADAPTAÇÃO DO PARADIGMA ORIENTADO A NOTIFICAÇÕES PARA DESENVOLVIMENTO DE SISTEMAS *Fuzzy*

Dissertação submetida ao Programa de Pós-Graduação em Computação Aplicada da Universidade Tecnológica Federal do Paraná como requisito parcial para a obtenção do título de Mestre em Computação Aplicada.

Área de concentração: *Engenharia de Sistemas Computacionais*

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

Curitiba PR
Agosto de 2016

Dados Internacionais de Catalogação na Publicação

M528a Melo, Luiz Carlos Viana
2016 Adaptação do paradigma orientado a notificações para desenvolvimento de sistemas fuzzy / Luiz Carlos Viana Melo .-- 2016.
xviii, 124 p.: il.; 30 cm.

Disponível também via World Wide Web.
Texto em português, com resumo em inglês.
Dissertação (Mestrado) - Universidade Tecnológica Federal do Paraná. Programa de Pós-Graduação em Computação Aplicada. Área de Concentração: Engenharia de Sistemas Computacionais, Curitiba, 2016.
Bibliografia: p. 83-86.

1. Paradigma orientado a notificações. 2. Sistemas difusos. 3. Programação baseada em regras. 4. Inferência (Lógica). 5. Linguagem de programação (Computadores). 6. Framework (Programa de computador). 7. C++ (Linguagem de programação de computador). 8. Métodos de simulação. 9. Computação - Dissertações. I. Fabro, João Alberto, orient. II. Simão, Jean Marcelo, coorient. III. Universidade Tecnológica Federal do Paraná. Programa de Pós-graduação em Computação Aplicada. IV. Título.

CDD: Ed. 22 -- 621.39

Biblioteca Central da UTFPR, Câmpus Curitiba

ATA DE DEFESA DE DISSERTAÇÃO DE MESTRADO Nº 47

Aos 26 dias do mês de agosto de 2016 realizou-se na sala B-204 a sessão pública de Defesa da Dissertação de Mestrado intitulada "Adaptação do Paradigma Orientado a Notificações para Desenvolvimento de Sistemas Fuzzy", apresentado pelo aluno **Luiz Carlos Viana Melo** como requisito parcial para a obtenção do título de Mestre em Computação Aplicada, na área de concentração "Engenharia de Sistemas Computacionais", linha de pesquisa "Sistemas Embarcados".

Constituição da Banca Examinadora:

Prof. Dr. João Alberto Fabro UTFPR - CT (Presidente)_____

Prof. Dr. Paulo Cezar Stadzisz UTFPR - CT _____

Prof. Dr. Fabiano Silva UFPR - CT _____

Profª. Drª. Myriam Regattieri de Biase Delgado 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".

A todos que me apoiaram...

Resumo

Este trabalho trata do tema de Paradigma Orientado a Notificações (PON) e sua adequação para prover suporte a conceitos *fuzzy*. O PON se inspira em elementos dos paradigmas imperativo e declarativo, buscando resolver inconvenientes de ambos. Ao decompor uma aplicação em uma rede de entidades computacionais menores que são executadas apenas quando necessário, o PON elimina a necessidade de realizar computações desnecessárias e alcança melhor desacoplamento lógico-causal facilitando o reaproveitamento e distribuição. Ademais, o PON permite expressar o seu conhecimento lógico-causal em alto nível, por meio de regras no formato SE-ENTÃO. Os sistemas *fuzzy*, por sua vez, realizam inferências em bases de conhecimento lógico-causal (regras SE-ENTÃO) que lidam com problemas que envolvem imprecisão.

Uma vez que o PON utiliza regras SE-ENTÃO de uma forma alternativa, reduzindo avaliações redundantes e acoplamento, este trabalho foi realizado para identificar, propor e avaliar as mudanças necessárias a serem realizadas sobre o PON para que este possa ser utilizado no desenvolvimento de sistemas *fuzzy*. Após a realização da proposta, foram criadas materializações na forma de um *framework* em linguagem C++, e uma linguagem de programação própria (LingPONFuzzy) com suporte a inferência *fuzzy*. A partir delas foram criados casos de estudo e realizados diversos testes para validar a solução proposta.

Os resultados dos testes mostram uma redução significativa no número de regras avaliadas em relação a um sistema *fuzzy* desenvolvido utilizando ferramentas convencionais (*frameworks*), o que poderia representar uma melhoria no desempenho das aplicações.

Palavras-chave: Paradigma orientado a notificações, Sistemas *Fuzzy*, Sistemas Baseados em Regras.

Abstract

This work proposes to adjust the Notification Oriented Paradigm (NOP) so that it provides support to fuzzy concepts. NOP is inspired by elements of imperative and declarative paradigms, seeking to solve some of the drawbacks of both. By decomposing an application into a network of smaller computational entities that are executed only when necessary, NOP eliminates the need to perform unnecessary computations and helps to achieve better logical-causal uncoupling, facilitating code reuse and application distribution over multiple processors or machines. In addition, NOP allows to express the logical-causal knowledge at a high level of abstraction, through rules in IF-THEN format. Fuzzy systems, in turn, perform logical inferences on causal knowledge bases (IF-THEN rules) that can deal with problems involving uncertainty.

Since PON uses IF-THEN rules in an alternative way, reducing redundant evaluations and providing better decoupling, this research has been carried out to identify, propose and evaluate the necessary changes to be made on NOP allowing to be used in the development of fuzzy systems. After that, two fully usable materializations were created: a C++ framework, and a complete programming language (LingPONFuzzy) that provide support to fuzzy inference systems. From there study cases have been created and several tests cases were conducted, in order to validate the proposed solution.

The test results have shown a significant reduction in the number of rules evaluated in comparison to a fuzzy system developed using conventional tools (frameworks), which could represent an improvement in performance of the applications.

Keywords: Notification Oriented Paradigm, Fuzzy Systems, Rule Based Systems.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviações

PON	Paradigma Orientado a Notificações
PI	Paradigma Imperativo
PD	Paradigma Declarativo
POO	Paradigma Orientado a Objetos
PL	Paradigma Lógico
SBR	Sistemas Baseados em Regras
FBE	*Fact Base Element* - Elemento da Base de Fatos
BNF	*Backus Normal Form* - Forma Normal de Backus
LingPON	Linguagem PON
PID	Proporcional Integral Derivativo
SE	Sistemas Especialistas
CBRA	Controle Baseado em Regras e Agentes
SIF	Sistema de Inferência *fuzzy*

Capítulo 1

Introdução

No mundo real existem diversas informações que normalmente não são ou não podem ser representadas através de informações precisas. Um exemplo desta afirmação está presente na seguinte definição: "esta pessoa é um pouco alta". A partir dela o avaliador está classificando a pessoa como uma pessoa alta mas não na sua totalidade. Este tipo de classificação não se encaixa nas definições matemáticas de classes ou conjuntos clássicos, mas desempenham um papel importante no pensamento humano [Zadeh, 1965].

Para lidar com este tipo de informação, foram definidos por Lofti A. Zadeh [Zadeh, 1965] os conjuntos nebulosos ou *fuzzy*, os quais podem ter um papel importante em diversas áreas do processamento de informação, como o controle de processos industriais, processamento de linguagem natural, entre outros. Os conjuntos *fuzzy* representam uma ponte de ligação entre processamento simbólico e numérico. A teoria de sistemas nebulosos utiliza símbolos (termos linguísticos) com os quais estão associadas semânticas bem definidas que, após serem convertidas em funções de pertinência, possibilitam o processamento numérico destes símbolos ou conceitos [Pedrycz and Gomide, 1998].

Baseada na teoria dos conjuntos *fuzzy*, a lógica *fuzzy* proporciona os mecanismos para realizar inferências lógicas baseadas em informações imprecisas. Assim como uma função algébrica realiza o mapeamento de uma variável de entrada para uma variável de saída, a lógica *fuzzy* realiza o mapeamento de um grupo de entrada em um grupo de saída, sendo que estes grupos podem ser proposições linguísticas ou outra forma de informação imprecisa. Este mapeamento é feito através de implicações entre premissas-condições e conclusões-ações (antecedentes e consequentes) [Ross, 2010].

Os sistemas que utilizam a lógica *fuzzy* para a realização de inferências sobre o conhecimento representado pelas implicações entre premissas-condições e conclusões-ações, descritas anteriormente, são conhecidos como sistemas *fuzzy*. A vantagem destes é que o conhecimento pode ser descrito na forma de regras SE-ENTÃO similares àquelas empregadas em linguagem natural [Sánchez-Solano et al., 2007].

Ao seu turno, o Paradigma Orientado a Notificações (PON) foi proposto por Simão [Simão, 2005] como uma solução de controle, que posteriormente evoluiu para um paradigma de programação. PON buscou alguma inspiração nos paradigmas imperativo e declarativo, mais precisamente nos subparadigmas imperativos orientado a objetos e orientado a eventos, e no subparadigma lógico-declarativo. Entretanto, ao mesmo tempo, ele busca solucionar algumas das deficiências destes (sub) paradigmas, tais como redundâncias estruturais e temporais, deficiências estas que levam a repetições desnecessárias e acoplamento [Simão et al., 2012c].

O PON utiliza as principais vantagens do paradigma declarativo, mais precisamente a expressividade das regras lógico-causais dos sistemas baseados em regras, que oferecem uma abstração e linguagem mais próxima da cognição humana. Ele também usa as vantagens do paradigma imperativo, mais precisamente a reusabilidade do código, flexibilidade e abstração através de classes e objetos do subparadigma orientado a objetos [Simão et al., 2012c]. Porém, o PON apresenta uma nova forma de inferência, o que difere dos paradigmas atuais por fazer cada avaliação lógico-causal se orientar a eventos factuais.

A principal ideia por trás do PON é a forma como o *software* detecta mudanças em variáveis e realiza inferências sobre ela. Nos paradigmas imperativos atuais existem duas formas para detectar mudanças nos valores de uma variável: através de *polling* e notificações de eventos. No *polling*, o laço ou *loop* de um programa realiza uma leitura contínua (em uma dada frequência) do valor das variáveis do sistema, realiza computação sobre elas e dispara ações caso certas condições sejam atendidas. Esta abordagem é considerada sequencial já que apenas uma condição é analisada por vez. Devido ao fato de que o *loop* é executado mesmo quando não há modificação nos valores das variáveis, esta abordagem desperdiça recursos como tempo do processador e energia elétrica [Peters et al., 2012].

Uma alternativa ao *polling* é a programação orientada a eventos, onde a computação é realizada apenas quando ocorre um evento. Em algumas abordagens, isto elimina a necessidade de se ter um *loop* que monitora o estado de uma variável, o que reduz a quantidade de computação desnecessária. Porém esta alternativa torna o desenvolvimento de aplicações complexo, normalmente resultando em programas maiores [Xavier, 2012]. Além disso, dadas as restrições de *hardware*, o sistema de controle de eventos pode realizar um *polling* para fazer o despacho dos mesmos [Peters et al., 2012, Xavier, 2012].

No caso da programação declarativa, o programador deve focar no que o programa deve realizar em vez de como será realizado. Isto libera o programador de lidar com muitos detalhes não importantes. Porém, programas desenvolvidos sob o paradigma declarativo são normalmente mais lentos para executar e menos flexíveis que os feitos com a programação imperativa [Banaszewski, 2009, Peters et al., 2012]. Isto ocorre pois as estruturas de dados utilizadas nos motores de inferência ainda implicam em muito consumo de capacidade de processamento. Além disto, a programação declarativa também apresenta acoplamento em seu código, de maneira similar à programação imperativa, dado que as máquinas de inferência usam algum tipo de *polling*, isto é, pesquisa em laço associada a estruturas de dados com o seu custo computacional [Simão et al., 2012c, Ronszcka, 2012].

Tendo estes problemas como motivação, o PON combina e evolui a programação baseada em eventos e a programação declarativa a fim de solucioná-los. De fato, o PON elimina a necessidade de realizar computação desnecessária e melhora o desacoplamento dos módulos. Desse modo, por exemplo, ele facilita a reusabilidade do código. Isto é alcançado através da decomposição da aplicação em uma rede de elementos notificadores menores (factuais-execucionais e lógico-causais) que são executados apenas quando necessário [Xavier, 2012]. Estes constituintes do PON serão explicados no decorrer deste trabalho.

Isto dito, este trabalho tem como objetivo propor a integração entre os conceitos de PON e sistemas *fuzzy* na forma de uma extensão ao PON que permita obter o suporte a conceitos e inferências *Fuzzy*. O presente trabalho contém o resultado do estudo sobre as mudanças necessárias a serem realizadas sobre o PON para que este possa ser utilizado no desenvolvimento de sistemas *fuzzy*. Neste capítulo introdutório, a seção 1.1 contém a descrição das motivações que levaram ao desenvolvimento deste trabalho. A seção 1.2 contém a justificativa para sua re-

alização, a seção 1.3 descreve os objetivos do trabalho, a seção 1.4 descreve o método utilizado para a elaboração deste trabalho e, por fim, a seção 1.5 descreve a estrutura do restante deste documento.

1.1 Motivação

A lógica *fuzzy* tem sido aplicada em diversas áreas de pesquisa que vão desde teoria de controle até inteligência artificial [Ross, 2010]. A facilidade do desenvolvimento de controladores usando lógica *fuzzy* na captura do conhecimento de especialistas e na tradução deste conhecimento em estratégias de controle robustas, sem a necessidade de um modelo matemático do sistema controlado, levou a um aumento no número de aplicações de controle utilizando técnicas de inferência *fuzzy* nas últimas décadas [Sánchez-Solano et al., 2007].

Estes controladores normalmente são implementados através da utilização de sistemas embarcados, que, por serem projetados para executar uma ou algumas poucas funcionalidades específicas, utilizam a menor quantidade de *hardware* possível para desempenhar suas tarefas. Isto garante um custo reduzido, menor consumo de energia e, devido à sua especialização, maior eficiência [Peters et al., 2012]. Porém, por se tratarem de sistemas com recursos reduzidos, dispõem de grandes limitações de processamento, memória e consumo de energia, tornando complexo o desenvolvimento de *software*. Além disto, os paradigmas de programação convencionais não contribuem com a construção de programas eficientes. Nestes, as expressões ou regras causais (SE-ENTÃO) e os elementos factuais (atributos ou variáveis) são tratados como entidades passivas, onde seus relacionamentos são estabelecidos por meio de pesquisas definidas pelo fluxo de execução dos programas (frequentemente algum tipo de *polling*), causando um considerável desperdício de processamento [Banaszewski, 2009].

O PON trata estes problemas oferecendo uma forma de programação mais fácil, mais próxima à forma cognitiva humana, além de visar alcançar esforços mínimos em termos de escrita de código. Graças a sua estrutura, o PON evita realizar buscas sobre elementos passivos, uma vez que os elementos factuais apresentam sensibilidade na detecção de mudanças de estados e apresentam capacidades para notificar as entidades afetadas por esta mudança, particularmente as regras lógico-causais [Banaszewski, 2009]. Isto evita que as demais entidades sejam avaliadas desnecessariamente. Além disso, isto permite o desacoplamento das entidades, permitindo inclusive que as entidades sejam reaproveitadas, reduzindo assim a quantidade de recursos utilizados pelas aplicações [Banaszewski, 2009].

Devido as similaridades na forma de representação do conhecimento, no passado foram realizados estudos preliminares para verificar se é possível adaptar o PON para ser utilizado no desenvolvimento de sistemas *fuzzy* [Simão et al., 2003a, de Souza et al., 2009]. Os resultados apresentados nos estudos preliminares mostraram que uma adaptação é possível e poderia apresentar ganho de desempenho, além das vantagens já citadas do PON. Este ganho é alcançado através da redução significativa do número de regras avaliadas em relação a um sistema *fuzzy* equivalente, como foi o caso citado no protótipo descrito em [de Souza et al., 2009], onde a redução foi de cerca de 80%.

1.2 Justificativa

Conforme citado, os resultados apresentados nos estudos preliminares [Simão et al., 2003a, de Souza et al., 2009] foram promissores. Porém, por serem preliminares, estes estudos não entram em detalhes sobre as adaptações realizadas sobre o paradigma, o que tornou difícil a replicação destas para outros trabalhos relacionados ao PON. Consequentemente, isto acabou ocasionando a falta de suporte ao desenvolvimento de sistemas *fuzzy* em geral pelas materializações (implementações) atuais do PON, tendo aquelas adaptações ficado ultrapassadas. Ademais, as adaptações foram singelas, não permitindo diversos tipos de dados, e os testes foram insuficientes para conclusões mais efetivas.

De fato, as atuais materializações do PON em *software*, tais como o *framework* definido em [Valença, 2012, Ronszcka, 2012] e a linguagem de programação PON definida em [Ferreira, 2015], não disponibilizam mecanismos para o desenvolvimento de sistemas *fuzzy*. É verdade que tais mecanismos, ainda que limitados, foram disponibilizadas em materializações anteriores mas não alcançaram de fato as materializações atuais de *software* [de Souza et al., 2009] e tão pouco de *hardware*.

Sendo assim, a fim de solucionar os problemas citados, neste trabalho é realizada a definição explícita das alterações que devem ser realizadas sobre o PON para que este possa ser utilizado no desenvolvimento de sistemas *fuzzy*. Tais definições advêm do que foi apresentado e pesquisado nos trabalhos precedentes, de ajuste destes e também no avanço de lacunas como a expansão de tipagens. Ademais, para validar a definição, são propostas e desenvolvidas extensões do *framework* e linguagem PON para tal propósito, havendo testes em sistema realísticos. Tais testes permitiram avaliar se certas propriedades do PON (como eliminação de redundâncias estruturais e temporais) se mantêm quando estendido para o âmbito dos sistemas *fuzzy*.

1.3 Objetivos

Nesta seção é apresentado o objetivo geral deste trabalho, bem como os objetivos específicos a serem alcançados, derivados do objetivo geral.

1.3.1 Objetivo geral

O objetivo do projeto consiste em realizar a integração no PON dos conceitos de lógica *fuzzy* para possibilitar a realização de inferências *fuzzy*, bem como verificar se é possível manter as propriedades do PON ao integrar tais conceitos.

1.3.2 Objetivos específicos

Os objetivos específicos deste trabalho são:

- Definição efetiva e explícita das modificações realizadas sobre o PON, para prover suporte à sistemas *fuzzy*;
- Validação das modificações através do desenvolvimento de um protótipo de um novo *framework* para o PON com suporte à lógica *fuzzy*;

- Modificação da linguagem de programação do PON, e respectivo compilador, para suportar as novas estruturas do sistema de inferência;
- Realização de teste comparativos de desempenho com outras materializações a fim de verificar as propriedades de eliminação de redundâncias temporais e estruturais (PON e *fuzzy*).

1.4 Metodologia

O desenvolvimento do trabalho foi realizado em diversas etapas ao longo do tempo. Algumas delas foram sendo identificadas à medida que o desenvolvimento era realizado. As etapas realizadas do trabalho, em ordem cronológica, foram as seguintes:

1. Estudo bibliográfico sobre os assuntos relacionados ao tema;
2. Definição das modificações a serem realizadas sobre o paradigma;
3. Implementação das modificações levantadas na forma de um *framework* em linguagem de programação C++;
4. Realização de testes comparativos com outros sistemas;
5. Definição das modificações a serem realizadas sobre a linguagem de programação do PON;
6. Implementação das modificações levantadas na linguagem e no compilador;
7. Re-execução dos testes comparativos com outros sistemas usando a linguagem PON com suporte a lógica *fuzzy*.

A primeira etapa do trabalho consistiu em realizar um estudo bibliográfico sobre o tema, o que envolve a busca e leitura de trabalhos que são relacionados ao PON e a sistemas *fuzzy*. Esta etapa foi importante para melhor entendimento dos conceitos envolvidos e também para buscar estudos anteriores realizados sobre o tema.

A segunda etapa envolveu o aprofundamento dos estudos sobre os conceitos empregados para a definição das modificações a serem realizadas para permitir a integração dos conceitos *fuzzy* ao PON. Estas modificações foram:

- Adicionar as estruturas de conjuntos *fuzzy* e variáveis linguísticas ao PON;
- Modificar o mecanismo de notificações do PON para propagar os graus de pertinência/ativação das entidades;
- Modificar as *Regras* para calcular o seu grau de ativação a partir dos operadores *fuzzy*;
- Criar estruturas para a realização do processo de “defuzzyficação”.

Na terceira etapa são feitas modificações no *framework* do PON definido em [Valença, 2012, Ronszcka, 2012] para adicionar os conceitos *fuzzy*. Após a realização destas modificações, na etapa quatro são feitos testes comparativos sobre as materializações modificadas. Estes testes são divididos da seguinte forma:

- Testes comparativos com sistemas desenvolvidos utilizando as versões originais das materializações;
- Testes comparativos com sistemas *fuzzy* desenvolvidos com *frameworks* convencionais (não PON).

Para os testes comparativos com as materializações originais do paradigma são utilizadas duas versões de sistemas: a primeira desenvolvida com as materializações originais e a segunda com as materializações modificadas com conceitos *fuzzy*. Nesta etapa são utilizados exemplos já elaborados em outros trabalhos. Nos testes comparativos com sistemas *fuzzy* também são utilizadas duas versões de sistema: a primeira desenvolvida utilizando um *framework fuzzy* convencional e a segunda utilizando as materializações modificadas. Neste caso são definidas algumas aplicações *fuzzy* para a realização dos testes.

O objetivo da realização dos testes é avaliar o quanto as modificações realizadas no mecanismo de inferência impactam no desempenho das aplicações. Em ambos os casos, são comparados o tempo de execução dos sistemas, quantidade de regras ativadas, e o resultado obtido através da execução deles.

Na quinta etapa é realizado o estudo para realização das modificações propostas no paradigma sobre a linguagem de programação PON apresentada por [Ferreira, 2015]. Como resultado deste estudo é definida uma nova linguagem com suporte às estruturas necessárias para a realização da inferência *fuzzy* como, por exemplo, os conjuntos *fuzzy*. A partir desta linguagem, na etapa seis são realizadas as modificações necessárias para que o compilador suporte a nova versão da linguagem.

Por fim, na etapa sete é realizada a re-execução dos testes comparativos realizados na etapa quatro, porém as aplicações de teste foram reescritas utilizando a linguagem PON *fuzzy*. A metodologia dos testes utilizada nesta etapa é a mesma utilizada na etapa anterior.

Durante o desenvolvimento deste trabalho, foram feitas submissões de artigos científicos em eventos e revistas da área. Os artigos publicados podem ser visualizados nos anexos. A composição desta dissertação foi realizada a medida que novos resultados dos estudos eram gerados.

1.5 Estrutura do documento

Este primeiro capítulo apresentou a descrição dos motivos que levaram à elaboração do presente trabalho, assim como as suas justificativas, objetivos gerais e específicos.

No capítulo 2 são apresentados os conceitos relacionados a sistema *fuzzy* e ao PON. O capítulo detalha o funcionamento dos sistemas de inferência *fuzzy* e do PON, mostrando os passos realizados em cada uma das etapas dos respectivos processos de inferência.

No capítulo 3 são descritas as modificações realizadas sobre o PON para adição do suporte ao desenvolvimento *fuzzy*. Também são descritas as mudanças realizadas sobre a linguagem PON e seu compilador.

No capítulo 4 são descritos os casos de estudo realizados para validação do trabalho realizado. Nele estão descritos os casos de testes e os resultados obtidos nas duas baterias de testes executadas.

Por último, o capítulo 5 apresenta as considerações finais sobre o desenvolvimento e os resultados obtidos. Também são descritos possíveis trabalhos futuros para dar continuidade ao desenvolvimento de tema.

Capítulo 2

Revisão bibliográfica

Para iniciar a revisão do estado da arte do tema selecionado, foi feita uma pesquisa para contextualização dos conceitos de sistemas baseados em lógica *fuzzy* (seção 2.1) e do Paradigma Orientado a Notificações (seção 2.2).

2.1 Sistemas *Fuzzy*

Os estudos sobre sistemas *fuzzy*, também conhecidos como sistemas nebulosos, iniciaram a partir de 1965 quando Lofti A. Zadeh desenvolveu a teoria dos conjuntos *fuzzy*. Esta teoria generaliza a teoria clássica dos conjuntos, permitindo representar conceitos que não podem ser bem representados usando limites claramente definidos [Pedrycz and Gomide, 1998].

2.1.1 Conjuntos *fuzzy*

Um conjunto *fuzzy* é um conjunto contendo os elementos do universo de discurso que possuem graus variáveis de adesão neste conjunto. Esta ideia difere dos conjuntos clássicos já que nestes o elemento só fará parte de um conjunto caso o seu grau de adesão seja completo. Assim como acontece com os conjuntos clássicos, elementos em um conjunto *fuzzy* poderão também ser membros de outros conjuntos no mesmo universo, sendo que os graus de adesão em cada conjunto poderão ser diferentes [Ross, 2010].

A relação entre um elemento e os conjuntos aos quais ele pertence é representada pelo seu grau de pertinência àquele conjunto. A equação 2.1 contém a notação matemática do grau de pertinência μ_A de um elemento x em um conjunto A. Um elemento é dito membro de um conjunto *fuzzy* quando o seu grau de pertinência μ_A é maior que zero para o conjunto A.

$$\mu_A(x) \in [0,1] \tag{2.1}$$

O grau de pertinência de um elemento em um conjunto é calculado utilizando a função de pertinência que define o conjunto *fuzzy*. Alguns dos tipos de funções de pertinências que são utilizados são os seguintes:

- Trapezoidal: onde $a \leq m \leq n \leq b$

$$\mu_A(x) = \begin{cases} 0 & \text{se } x \leq a \\ \frac{x-a}{m-a} & \text{se } a < x \leq m \\ 1 & \text{se } m < x \leq n \\ \frac{b-x}{b-n} & \text{se } n < x \leq b \\ 0 & \text{se } x \geq b \end{cases}$$

- Triangular: onde $a \leq m \leq b$

$$\mu_A(x) = \begin{cases} 0 & \text{se } x \leq a \\ \frac{x-a}{m-a} & \text{se } a < x \leq m \\ \frac{b-x}{b-m} & \text{se } m < x \leq b \\ 0 & \text{se } x \geq b \end{cases}$$

- Gaussiana: onde $\sigma_k > 0$

$$\mu_A(x) = exp^{-\sigma_k(x-m)^2}$$

- Singleton:

$$\mu_A(x) = \begin{cases} 1 & \text{se } x = m \\ 0 & \text{caso contrário} \end{cases}$$

A figura 2.1 ilustra os gráficos dos tipos de função de pertinência citados.

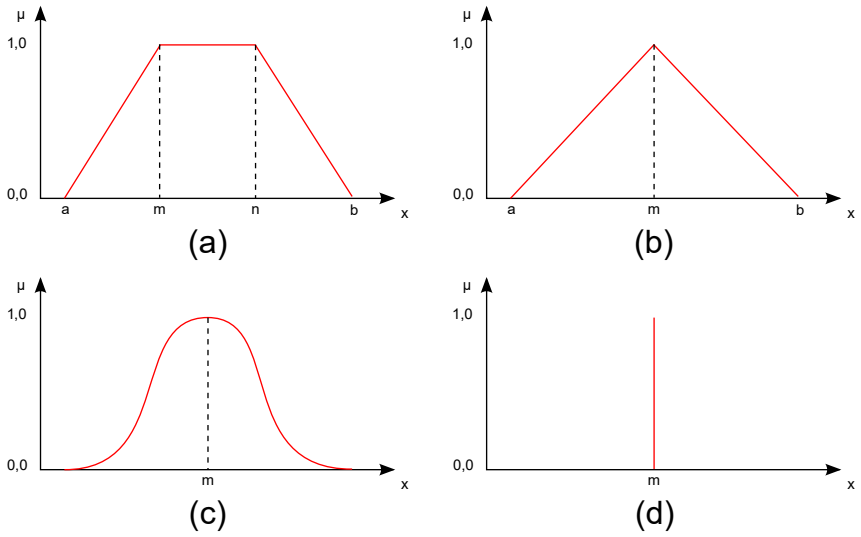


Figura 2.1: Exemplos de funções de pertinência: (a) trapezoidal, (b) triangular, (c) gaussiana e (d) singleton.

Um conceito relacionado com conjuntos *fuzzy* é o de variável linguística que representa um identificador que pode assumir um dentre vários valores [Mamdani, 1974]. Desta forma, uma variável linguística pode assumir um valor linguístico que representa um conjunto *fuzzy*. Na figura 2.2 é possível visualizar um exemplo de variável linguística que representa a altura de uma pessoa com os conjuntos *fuzzy* que representam seus possíveis valores (Baixo, Médio e Alto). No gráfico é possível determinar o grau de pertinência do elemento para os

conjuntos *fuzzy* com base no valor da altura. Para um valor de altura de 160 cm, pode-se afirmar que a pessoa "é 0,75 baixa e 0,25 média". Estes valores são calculados a partir das funções de pertinência que definem os conjuntos.

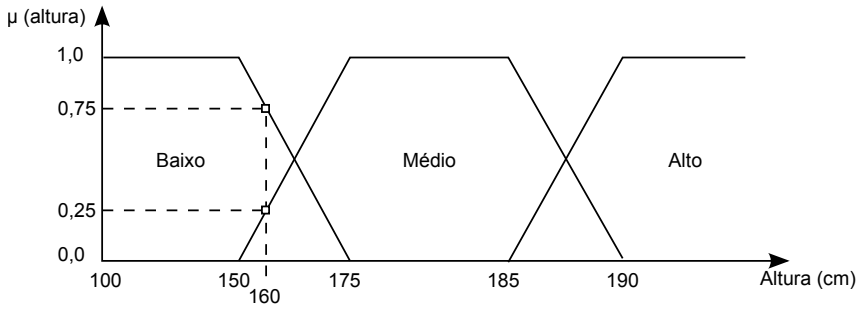


Figura 2.2: Variável linguística que representa a altura de uma pessoa - Adaptado de [Fabro, 1996].

Assim como os conjuntos clássicos, os conjuntos *fuzzy* também podem ser manipulados através da utilização de operações sobre conjuntos. Os operadores sobre os conjuntos *fuzzy* definidos em [Zadeh, 1965] a partir do grau de pertinência são os seguintes [da Silva Delgado, 2002]:

- Interseção : $\mu_{(A\cap B)}(x) = min[\mu_A(x), \mu_B(x)]$;
- União : $\mu_{(A\cup B)}(x) = max[\mu_A(x), \mu_B(x)]$;
- Complemento : $\mu_{(\bar{A})}(x) = 1 - \mu_A(x)$.

Baseada na teoria dos conjuntos *fuzzy*, a lógica *fuzzy* proporciona os mecanismos para realizar inferências lógicas baseadas em informações imprecisas. Na seção a seguir será apresentado o mecanismo de inferência através da realização de operações sobre os conjuntos *fuzzy*.

2.1.2 Lógica *fuzzy*

Analogamente a teoria dos conjuntos, a lógica *fuzzy* é uma generalização da lógica tradicional. Utilizando os operadores de complemento, união, interseção e implicação *fuzzy*, é possível realizar os processos de inferência já conhecidos na lógica tradicional com conjuntos *fuzzy* [Pedrycz and Gomide, 1998]. A lógica *fuzzy* pode ser utilizada para o desenvolvimento de sistemas de controle que lidam com informações imprecisas. Através de um conjunto de regras é possível realizar a inferência de informações que serão utilizadas na tomada de decisão de um sistema. Na figura 2.3 são ilustradas as operações básicas de um mecanismo de inferência *fuzzy*.

A vantagem da lógica *fuzzy* é que o conhecimento pode ser descrito na forma de regras SE-ENTÃO similares àquelas empregadas em uma linguagem natural [Sánchez-Solano et al., 2007]. Logo, uma regra *fuzzy* pode ser escrita da seguinte maneira:

```
SE temperatura do ambiente É alta, ENTÃO potência do ar condicionado
É máxima
```

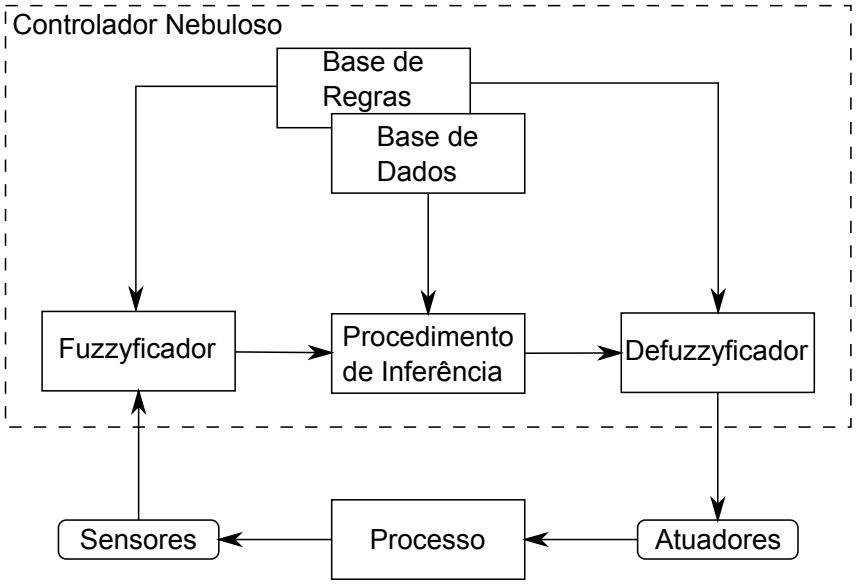


Figura 2.3: Mecanismo de inferência *fuzzy* - Adaptado de [Fabro, 1996].

Para exemplificar o processo de inferência *fuzzy* será considerado um sistema de frenagem automático de um carro. O sistema contém o seguinte conjunto de regras *fuzzy*:

1. `SE distância É curta E velocidade É média, ENTÃO frenagem É média`;
2. `SE distância É média E velocidade É baixa, ENTÃO frenagem É fraca`.

Este possui as variáveis linguísticas de entrada que representam a distância até um obstáculo e a velocidade do carro e a variável de saída de intensidade da frenagem conforme apresentado na figura 2.4.

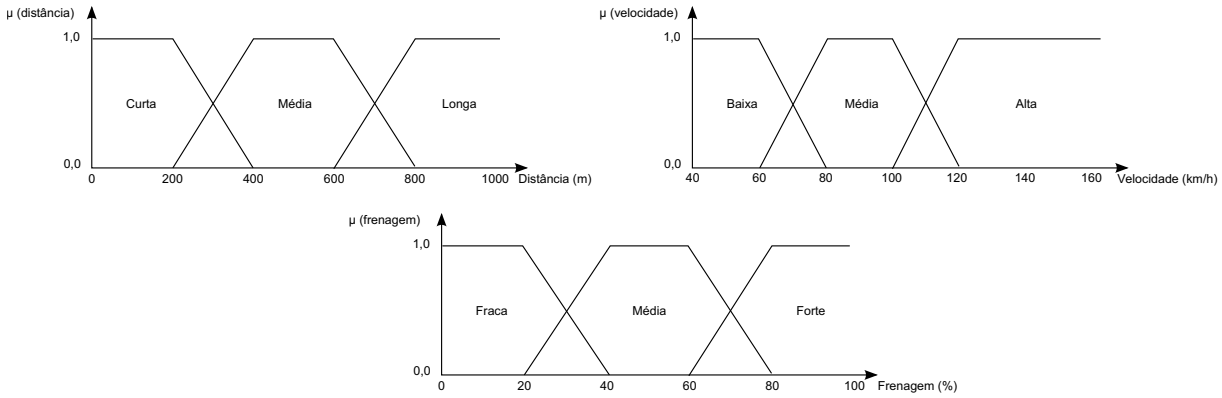


Figura 2.4: Variáveis linguísticas do sistema de frenagem.

Na maioria das aplicações práticas consideram-se dados de entrada como valores não-*fuzzy* ou *crisp* como, por exemplo, resultante de medições ou observações. Para utilização destes no mecanismo apresentado, é necessário efetuar um mapeamento destes dados para os conjuntos *fuzzy* descritos nas regras [Tanscheit, 2004]. Este mapeamento é chamado de "fuzzyficação" e retorna o grau de pertinência do valor *crisp* para o conjunto *fuzzy* em uma premissa da regra. O grau é calculado de acordo com a função de pertinência que representa o conjunto *fuzzy*.

Voltando ao exemplo citado e considerando o carro estando a 300 metros do obstáculo e a uma velocidade de 75 km/h, a "fuzzyficação" para a regra 1 irá calcular os graus de pertinência 0,5 e 0,75 para as premissas "distância é CURTA" e "velocidade é MÉDIA" respectivamente.

Os valores gerados a partir do cálculo dos graus de pertinência são utilizados para realizar as inferências *fuzzy* através da aplicação dos operadores sobre os conjuntos *fuzzy* sobre os antecedentes da regra. O processo de inferência resulta no cálculo do nível de ativação de cada regra. Neste âmbito, os operadores que foram definidos em [Zadeh, 1965] e utilizados posteriormente em [Mamdani, 1974] são os seguintes:

- Conectivo lógico **E** : Interseção (*norma-t min*);
- Conectivo lógico **OU** : União (*norma-s max*);
- Operação lógica de negação : Complemento.

A tabela 2.1 apresenta os valores de ativação das regras *fuzzy* do sistema citadas anteriormente e de seus antecedentes quando as entradas do sistema de frenagem estiverem ajustadas com os valores citados anteriormente (300 metros e 75 km/h).

Tabela 2.1: Valores dos graus de pertinência das premissas e do grau de ativação das regras.

Regra	μ(distância)	μ(velocidade)	μ(regra)
1	0,5	0,75	0,5
2	0,5	0,25	0,25

Após este cálculo, é feita a propagação dos valores verdade através das regras *fuzzy*, gerando conjuntos *fuzzy* representativos dos consequentes de cada regra. A contribuição de cada regra (conjunto *fuzzy*) é levada em consideração via, por exemplo, a união entre os conjuntos *fuzzy*. A partir deste ponto o operador já tem a resposta do sistema mas na forma de conjuntos *fuzzy*. A figura 2.5 demonstra o conjunto *fuzzy* resultante gerado com base nos valores de ativação das regras contidos na tabela 2.1 e nos operadores de Mandani.

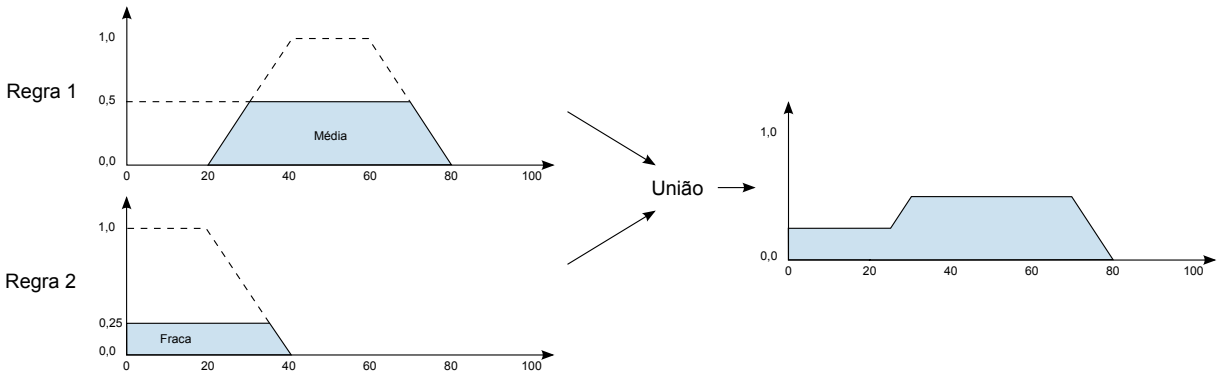


Figura 2.5: Geração do conjunto *fuzzy* resultante.

Uma vez obtido o conjunto *fuzzy* de saída através do processo de inferência, no estágio de "defuzificação" é efetuada uma interpretação dessa informação. Isto se faz necessário pois, em aplicações práticas, geralmente são requeridas saídas não-*fuzzy*. No caso de uma sistema de controle, por exemplo, em que o controle é efetuado por um sistema de inferência *fuzzy*

(ou controlador *fuzzy*), este deve fornecer à planta dados ou sinais *crisp* (i.é, não-*fuzzy*), já que a "apresentação" de um conjunto *fuzzy* à entrada da planta não teria significado algum [Tanscheit, 2004].

Existem vários métodos de "defuzificação" na literatura; dois dos mais empregados são o centroide e a média dos máximos. A equação 2.2 demonstra a fórmula para o cálculo do centroide do conjunto *fuzzy* resultante.

$$z = \frac{\sum_{k=1}^{q} z_k \mu(z_k)}{\sum_{k=1}^{q} \mu(z_k)} \tag{2.2}$$

A função de centroide faz a média ponderada dos valores *crisp* (z_k) que fazem parte do universo de discussão da variável de saída ponderados de acordo com seus respectivos graus de pertinência ($\mu(z_k)$) no conjunto *fuzzy* resultante. A quantidade de valores *crisp* a ser utilizado no cálculo é gerado a partir da granularidade g utilizado na função. Ela define o espaçamento entre os valores.

Para exemplificar, a figura 2.6 apresenta os parâmetros da função centroide no conjunto resultante do sistema de frenagem.

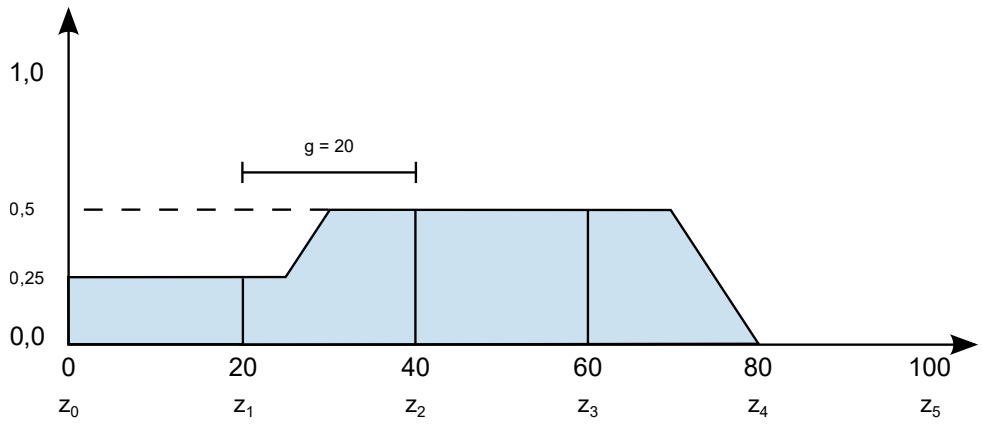


Figura 2.6: Conjunto *fuzzy* resultante no cálculo do centroide.

Aplicando os valores na fórmula 2.2 temos:

$$\begin{aligned} z = \frac{\sum_{k=1}^{q} z_k \mu(z_k)}{\sum_{k=1}^{q} \mu(z_k)} &\rightarrow \frac{z_0\mu(z_0) + z_1\mu(z_1) + z_2\mu(z_2) + z_3\mu(z_3) + z_4\mu(z_4) + z_5\mu(z_5)}{\mu(z_0) + \mu(z_1) + \mu(z_2) + \mu(z_3) + \mu(z_4) + \mu(z_5)} \\ z &= \frac{0 \times 0,25 + 20 \times 0,25 + 40 \times 0,5 + 60 \times 0,5 + 80 \times 0 + 100 \times 0}{0,25 + 0,25 + 0,5 + 0,5 + 0 + 0} \\ z &= \frac{55}{1,5} \rightarrow 36,666... \end{aligned} \tag{2.3}$$

Já na média dos máximos a saída precisa é obtida tomando-se a média entre os dois elementos extremos no universo que correspondem aos maiores valores da função de pertinência do consequente [Tanscheit, 2004].

2.1.3 Modelo de Takagi-Sugeno

O modelo de Takagi-Sugeno foi introduzido por [Sugeno, 1985] e é similar ao modelo de Mamdani descrito na seção 2.1 em muitos aspectos. As duas primeiras partes, a "fuzificação" e o procedimento de inferência das regras são exatamente os mesmos. A principal diferença entre os modelos é que as funções de pertinência das saídas do modelo Takagi-Sugeno são, em geral, lineares ou constantes. Uma regra típica deste modelo possui a seguinte forma [MathWorks, 2015]:

```
SE temperatura(x) É alta E umidade(y) É alta, ENTÃO pot.  AC(z) É 10x + 5y + 2
```

A saída do processo de inferência será a somatória dos valores calculados para cada regra ponderados pelos seus graus de ativação. A fórmula 2.4 apresenta o cálculo do resultado, sendo que z_i é o valor calculado para a regra com base nos valores de entrada e w_i é o grau de ativação da regra.

$$Resultado = \frac{\sum_{i=1}^{N} z_i w_i}{\sum_{i=1}^{N} w_i} \tag{2.4}$$

Considerando o exemplo do sistema de frenagem descrito na seção 2.1, as duas regras podem ser definidas da seguinte forma no modelo de Takagi-Sugeno:

1. `SE distância(x) É curta E velocidade(y) É média, ENTÃO frenagem É 0,05x + 0,3y`;
2. `SE distância(x) É média E velocidade(y) É baixa, ENTÃO frenagem É 0,02x + 0,1y`.

Para os valores de entrada definidos no exemplo (300 metros e 75 km/h) as regras terão os graus de ativação 0,5 e 0,25 respectivamente. Neste modelo, os valores *crisp* de entrada são utilizados nas fórmulas definidas na saída das regras. Neste caso, os valores de saída de cada uma das regras são os seguintes:

1. $0,05x + 0,3y \rightarrow 0,05 \times 300 + 0,3 \times 75 = 37,5$;
2. $0,02x + 0,1y \rightarrow 0,02 \times 300 + 0,1 \times 75 = 13,5$;

A partir destes valores é calculado a saída do processo conforme apresentado na fórmula 2.5.

$$Resultado = \frac{\sum_{i=1}^{N} z_i w_i}{\sum_{i=1}^{N} w_i} \rightarrow \frac{37,5 \times 0,5 + 13,5 \times 0,25}{0,5 + 0,25} \rightarrow 29,5 \tag{2.5}$$

2.1.4 Propostas de sistemas de controle *fuzzy*

Nesta seção estão contidos estudos sobre os trabalhos que utilizaram os conceitos da lógica *fuzzy*. Existem diversos trabalhos que contêm propostas de sistemas *fuzzy* disponíveis na bibliografia. A maioria das aplicações de sistemas *fuzzy* durante as últimas duas décadas pertencem à classe de controladores PID (Proporcional Integral Derivativo) *fuzzy*, sendo que estes recebem como entrada para a inferência *fuzzy* o valor do erro calculado com base em um valor pré-definido (*setpoint*).

Estes controladores ainda podem ser classificados em três tipos [Sulaiman et al., 2009]:

- Controladores de ação direta (*Direct Action* - DA): são controladores que são inclusos na malha de controle do *feedback* e produzem a saída do controlador PID através da inferência *fuzzy*;
- Controladores de agendamento de ganho (*Gain Scheduling* - GS): são controladores em que a inferência *fuzzy* é realizada para computar os ganhos instantâneos do controlador PID, individualmente;
- A combinação dos dois tipos citados anteriormente.

Esta seção foi elaborada para apresentar alguns exemplos de aplicações práticas de sistemas nas quais a possibilidade de implementação mais eficiente possa trazer benefícios. A seguir são apresentados os trabalhos que foram estudados.

Sistema *fuzzy* para estabilização de um pêndulo

Em [Oliveira et al., 2010] foi desenvolvido um sistema *fuzzy* utilizando um *hardware* reconfigurável. Neste trabalho o sistema proposto é para controle que estabilize um pêndulo com base em um ângulo de referência pré-definida. A entrada do sistema é lida através de um conversor A/D ligado a um potenciômetro que monitora o movimento angular do pêndulo. A saída do sistema controla um motor responsável por estabilizar o pêndulo. A figura 2.7 apresenta a arquitetura implementada para o sistema proposto.

A proposta utiliza as variáveis ERRO (representa a diferença entre a posição atual do pêndulo em relação ao ângulo de referência) e VERRO (variação do erro) como variáveis linguísticas de entrada do sistema. Os conjuntos *fuzzy*, regras e os processos de "fuzzyficação", inferência e "defuzzyficação" foram implementadas diretamente em *hardware*, o que limita o sistema *fuzzy* a apenas o sistema de controle do pêndulo.

O trabalho faz o comparativo entre o sistema *fuzzy* proposto e outro sistema *fuzzy* com as mesmas características desenvolvido utilizando o *toolbox* de lógica *fuzzy* do MATLAB. Os erros entre as respostas de cada um dos sistemas é apresentado. O valor do RMSE (*Root Mean Square Error*) foi de 0,77. Este valor foi justificado pelo autor do trabalho devido ao fato que o *hardware* reconfigurável trabalha apenas com número inteiros, portanto apresentando erros maiores que os obtidos no MATLAB.

Sistema *fuzzy* de controle de suspensão de carro

Em [Abu-Khudhair et al., 2010] é proposto um sistema *fuzzy* adaptativo para controle do sistema de suspensão semi-ativa de um carro. Para estabilizar a suspensão, o carro deve

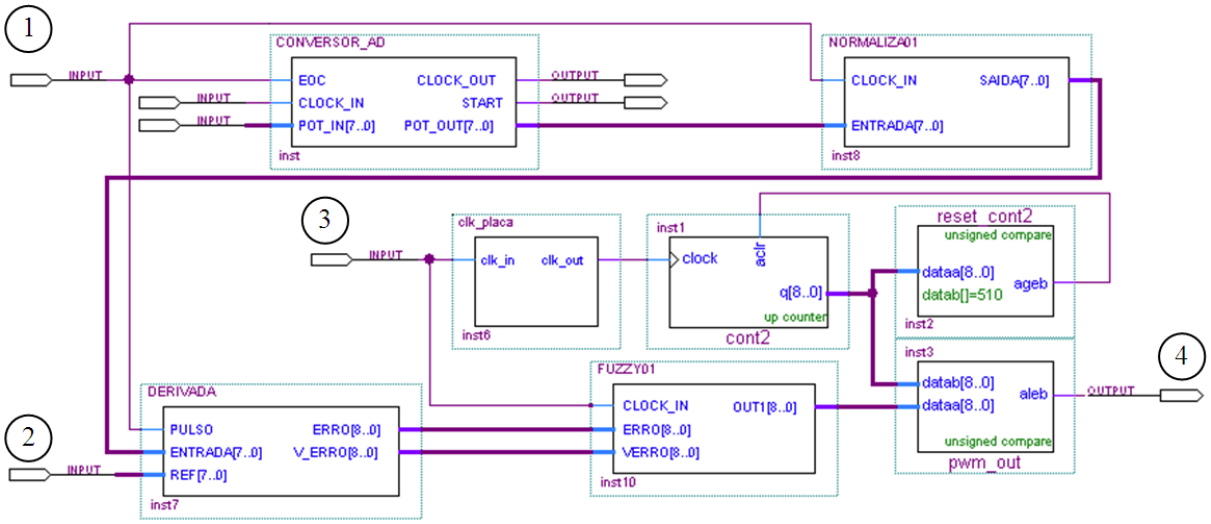


Figura 2.7: Arquitetura do sistema *fuzzy* implementado em hardware [Oliveira et al., 2010].

gerenciar o amortecedor variável no sistema. A figura 2.8 ilustra o diagrama de blocos do sistema de controle.

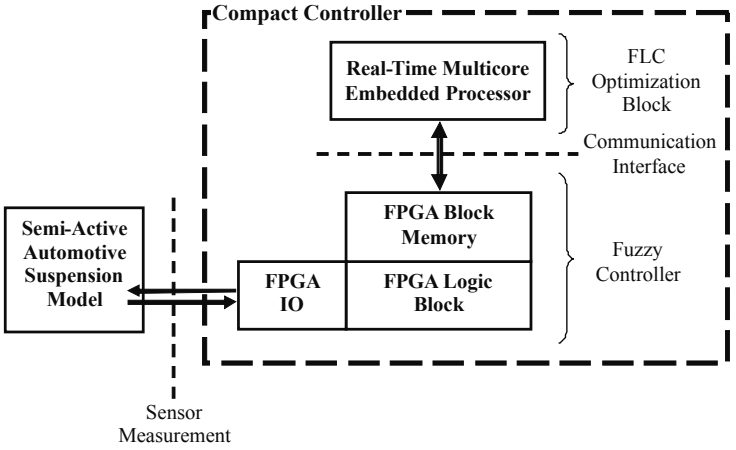


Figura 2.8: Diagrama de blocos do sistema de controle da suspensão de um carro [Abu-Khudhair et al., 2010].

Este sistema recebe duas variáveis de entrada para a realização da inferência e gera uma saída para o sistema da suspensão. As variáveis são as seguintes:

- Variáveis de entrada:
  - Deflexão da suspensão (cm);
  - Velocidade (m/s);
- Variável de saída:
  - Coeficiente de amortecimento.

Todo o processo de “fuzzyficação”, inferência e “defuzzyficação” do sistema é realizado diretamente pelo *hardware* utilizado pelo autor. A saída do sistema é ligada diretamente ao sistema de suspensão do veículo para que este realize o ajuste do amortecedor.

A diferença deste trabalho para outros é que ele utiliza um algoritmo adaptativo para modificar as funções de pertinência das variáveis de entrada e saída, e as regras de inferência do sistema. O autor propõe o uso de algoritmos genéticos ou redes neurais para a realização da otimização do sistema de controle. Este algoritmo de otimização é implementado em *software* e executado em um processador ligado ao sistema através do barramento.

Para os testes foram feitas comparações dos resultados com outros dois sistemas: um sistema de suspensão passivo e um sistema de suspensão semi-ativa baseado em regulador quadrático linear. Os resultados apresentados são referentes à saída produzida pelos sistemas, onde o sistema *fuzzy* demonstrou uma melhora de 45% em relação ao sistema passivo [Abu-Khudhair et al., 2010].

Controlador multi-camada para controle de hexacóptero

Em [Koslosky et al., 2015] foi proposto um controlador multi-camada para controle de hexacóptero com um peso pendular preso a sua carcaça. O sistema de controle é responsável pela movimentação e estabilização do hexacóptero utilizando múltiplos controladores *fuzzy*, descritos a seguir.

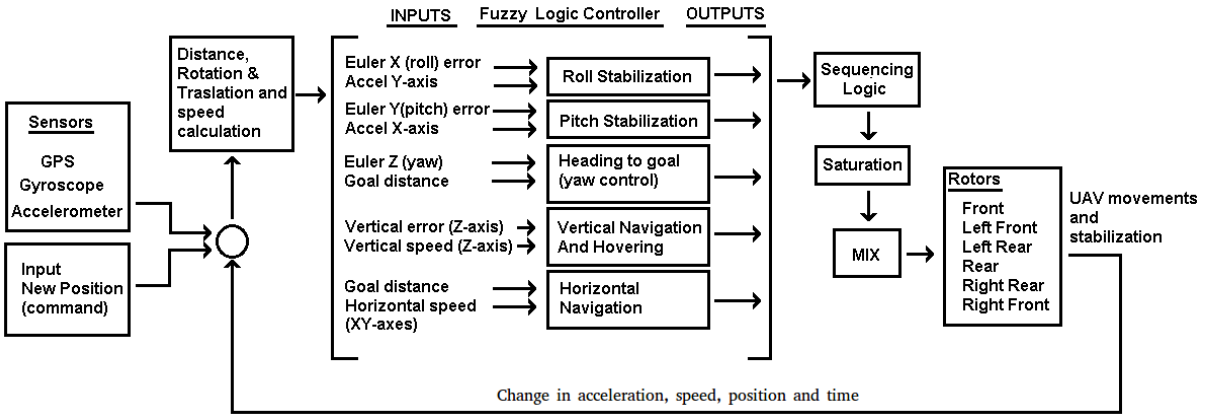


Figura 2.9: Arquitetura do controlador do hexacóptero [Koslosky et al., 2015].

A abordagem proposta foi validada através de simulação. Para tal, um modelo do hexacóptero foi criado no ambiente de simulação V-REP. Um pêndulo foi preso à carcaça do hexacóptero a fim de gerar instabilidade no voo. Isto foi feito para demonstrar que o controlador é capaz de controlar o movimento enquanto mantém o hexacóptero estável durante o voo [Koslosky et al., 2015].

Ao iniciar a simulação do modelo, o sistema de controle passa a ler as informações dos sensores ligados ao hexacóptero. Esta informação é lida utilizando a API disponibilizada pelo V-REP[1] para tal fim. A partir delas são feitos os cálculos dos valores que serão passados para as variáveis de entrada dos controladores *fuzzy* do sistema. O controlador possui cinco subcontroladores *fuzzy*, conforme pode ser visualizado na figura 2.9:

[1] *Virtual Robot Experimentation Platform*, software disponível para download de uso gratuito para fins educacionais e de pesquisa em: http://www.coppeliarobotics.com

- Estabilizador do ângulo no eixo X (*Roll*);
- Estabilizador do ângulo no eixo Y (*Pitch*);
- Controlador de direção (*Yaw*);
- Navegação vertical e flutuação;
- Navegação horizontal.

Os resultados das inferências dos controladores são utilizados para calcular a potência de cada um dos rotores do hexacóptero para a realização do movimento desejado. Estas velocidades são enviadas ao modelo utilizando a API do V-REP fazendo com que os valores das velocidades dos rotores sejam alterados, levando as grandezas medidas a serem alteradas, reiniciando assim o *loop* de controle.

Na próxima seção, os conceitos relativos ao PON são apresentados em mais detalhes.

2.2 Paradigma Orientado a Notificações

Nesta seção estão descritos os conceitos referentes ao Paradigma Orientado a Notificações (PON). Também é apresentado um breve histórico de trabalhos que serviram como base para criação do PON, e um breve resumo de algumas materializações do mesmo.

2.2.1 Histórico

O Paradigma Orientado a Notificações (PON) foi concebido a partir de esforços embrionários advindos da dissertação de mestrado[2] e tese de doutorado[3] de Jean Marcelo Simão na forma de um mecanismo de controle que suprisse as necessidades relacionadas com os sistemas modernos de produção. O autor propôs uma abordagem que permite organizar as colaborações entre os recursos (e.g. equipamentos) a fim de alcançar agilidade na produção. Esta abordagem refere-se a um modelo genérico de controle discreto que foi aplicado na simulação de sistemas inteligentes de manufatura [Simão, 2001, Simão, 2005, Banaszewski, 2009].

Posteriormente, de certa forma, o autor percebeu que este modelo poderia ser aplicado em diferentes domínios de problemas. Sendo assim, ele propôs o modelo como uma solução geral de controle discreto e também de inferência [Simão et al., 2003b, Simão et al., 2009]. Este modelo serviu como inspiração para a definição do PON. Nas sub-seções a seguir são descritos alguns trabalhos que utilizaram este modelo até a definição do paradigma.

Sistema de inferência orientado a agentes

Em [Simão and Stadzisz, 2002] é feito a proposta de um motor de inferências baseados em agentes reativos voltado para controle de sistemas automatizados de produção. O artigo foi baseado no trabalho anterior realizado em [Simão, 2001] e contém a proposta de um sistema

[2]Proposta de uma Arquitetura de Controle para Sistemas Flexíveis de Manufatura Baseada em Regras e Agentes [Simão, 2001]

[3]Contribution to the Development of a HMS Simulation Tool and Proposition of a Meta-Model for Holonic Control [Simão, 2005]

agente-reativos para o desenvolvimento de Sistemas Especialistas (SE) que representam fatos sobre o sistema a ser controlado, relações lógico-causais que especificam as diferentes formas de alterar os fatos e um motor de inferência para efetivamente alterá-los.

Os agentes apresentados representam os fatos e regras do sistema a ser controlado, sendo que o motor de inferência é alcançado através de seus relacionamentos [Simão and Stadzisz, 2002]. Estes compõem o conceito de Controle Baseado em Regras e Agentes (CBRA) que será definido posteriormente em [Simão, 2005] como controle holônico. Além disso, este modelo foi utilizado como base para a definição das entidades do paradigma conforme descrito na seção 2.2.2.

Os tipos de agentes apresentados e as entidades dos paradigmas aos quais eles estão relacionados são os seguintes:

- Agente “Base de fatos” (*fba*) - Entidade *Elemento da Base de Fatos*;
- Agente “Atributo” (*at*) - Entidade *Atributo*;
- Agente “Premissa” (*pa*) - Entidade *Premissa*;
- Agente “Condição” (*ca*) - Entidade *Condição*;
- Agente “Regra” (*ra*) - Entidade *Regra*;
- Agente “Ação” (*aa*) - Entidade *Ação*;
- Agente “Ordem” (*oa*) - Entidade *Instigação*;
- Agente “Método” (*ma*) - Entidade *Método*.

A figura 2.10 demonstra um exemplo de regra, na forma de agente, para controlar uma célula de manufatura. Nesta é possível visualizar as relações que cada um dos agentes citados acima possui.

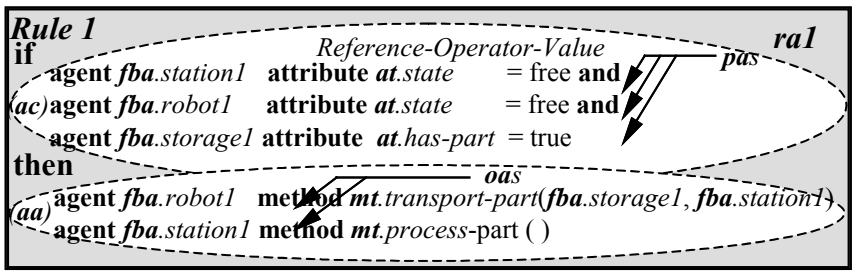


Figura 2.10: Instância de um agente “Regra” [Simão and Stadzisz, 2002]. Figura retirada de [Simão et al., 2003a].

O mecanismo de inferência apresentado no trabalho citado foi utilizado como base para a definição do mecanismo descrito na seção 2.2.3. A figura 2.11 demonstra o mecanismo de inferência citado. Nesta é possível ver que a cooperação entre os agentes permite que o mecanismo de inferência se sustente de forma autônoma.

No trabalho é prevista a possibilidade de haver conflitos entre os agentes de regras quando alguma de suas premissas possui a referência para um atributo em um agente de base de fatos exclusivo. Este agente representa um recurso exclusivo, isto é, um recurso que permite

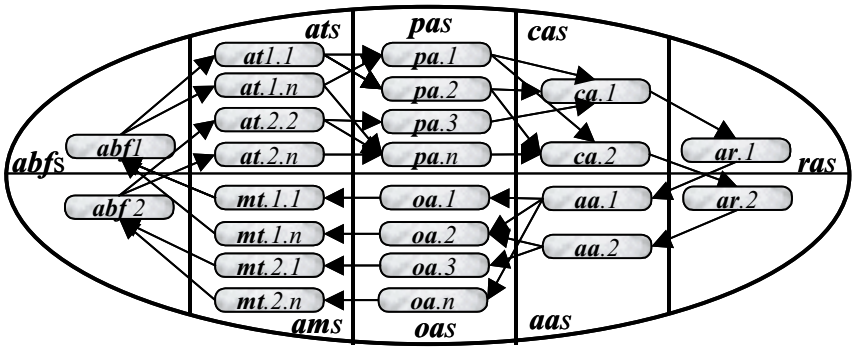


Figura 2.11: Mecanismo de inferência sustentado pela cooperação entre os agentes [Simão and Stadzisz, 2002]. Figura retirada de [Simão et al., 2003a].

um acesso por vez. Para solucionar este conflito, é escolhida apenas uma única regra para ser ativada. A decisão de qual regra será ativada é realizada por um agente solucionador de conflitos.

Sistema de inferência *fuzzy* orientado a agentes

Em [Simão et al., 2003a] é feita uma proposta de uma abordagem para o desenvolvimento de sistemas de inferência *fuzzy* (SIF) se utilizando dos conceitos de CBRA apresentados em [Simão and Stadzisz, 2002]. Este artigo descreve os conceitos de CBRA além de descrever as mudanças que foram realizadas para tornar possível o desenvolvimento de um SIF.

No trabalho, são propostas algumas mudanças para que o CBRA possa ser utilizado na criação dos SIF:

- Os agentes "Atributo" representam as variáveis linguísticas do sistemas e são compostos de sub-atributos, sendo que cada um destes representam um termo linguístico com sua respectiva função de pertinência;
- Os agentes "Premissa" serão ativos quando o grau de ativação dos atributos referenciados a eles tiverem um valor diferente de zero;
- Quando todos os agentes "Premissa" referenciados pelo agente "Condição" referenciado pelo agente "Regra" estiverem ativos, este último será ativo e realizará o cálculo do grau de ativação;
- Quando o grau de ativação for maior que zero, o agente "Ação" associado será ativado, o que fará com que o agente "Ordem" ative o agente "Método". O grau de ativação calculado é repassado para estes agentes.

Além destas mudanças, no trabalho é identificada a necessidade de se criar um agente conciliador para agregar as contribuições dos agentes "Regra" ativos no resultado da inferência. Este agente intercepta todos os agentes "Método" instigados e os agrupa de acordo com o agente "Atributo" referenciado nos consequentes dos agentes "Regra". Para cada agente "Atributo" referenciado, o agente conciliador modifica o seu valor usando uma função *fuzzy* de conciliação, sendo que esta recebe como parâmetro todas as mudanças a serem realizadas pelos agentes "Método" relacionados [Simão et al., 2003a]. Em outras palavras, este agente é responsável

em montar os conjuntos *fuzzy* resultantes e realizar a "defuzzyficação" para cada agente "Atributo" referenciado pelos agentes "Método" das regras ativas.

Meta-modelo para controle holônico

Em [Simão, 2005] é definido um meta-modelo para controle holônico baseado no modelo de agentes apresentado na seção 2.2.1.

Este meta-modelo de controle holônico foi definido para desenvolvimento de sistemas de manufatura holônicos, onde as entidades de produção, tais comos os recursos e produtos, possuem uma certa inteligência. Estas entidades são chamadas de *holons* cuja "inteligência" é relacionada às habilidades de autonomia e colaboração. O controle holônico é responsável em organizar apropriadamente as colaborações dos *holons* para alcançar agilidade [Simão, 2005].

No meta-modelo, os dados (atributos e métodos dos recursos ou elementos) e os comandos (premissas, condições e regras) são entidades independentes que apresentam características de autonomia e cooperação para realizar as avaliações lógicas [Banaszewski, 2009]. O mecanismo de inferência deste meta-modelo é semelhante ao mecanismo apresentado na seção 2.2.1, onde a inferência é realizada através da interação entre estas entidades.

Neste trabalho o meta-modelo é utilizado para desenvolvimento de sistemas que são simulados na ferramenta ANALYTICE II. Esta ferramenta separa as entidades de controle de alto nível dos recursos emulados da planta [Simão, 2005].

Definição e apresentação do PON

Incentivado pelos resultados obtidos e pelo fato das entidades do meta-modelo do controle holônico serem genéricas, o autor deste vislumbrou a possibilidade de aplicação do meta-modelo em domínios diferentes daquele de produção. Assim sendo, ele propôs o meta-modelo como uma solução geral de controle discreto, bem como uma solução geral de inferência. Ao decorrer do tempo, o autor percebeu que o modelo apresentado poderia ser utilizado para guiar o programador na concepção de programas, surgindo assim o PON [Simão and Stadzisz, 2008, Simao and Stadzisz, 2009, Banaszewski, 2009, Simão et al., 2012c].

A definição do paradigma é feita em [Simão and Stadzisz, 2008], onde as entidades do meta-modelo definidas em [Simão, 2005] são redefinidas genericamente como as entidades do paradigma. As relações de base entre as entidades e o modelo de inferência não sofreram alterações substanciais mas a compreensão de suas aplicabilidades bem como a generalidade dos elementos factuais.

Em [Banaszewski, 2009] é apresentado um estudo do paradigma PON para a elaboração da dissertação de mestrado do autor, que contém um estudo comparativo entre o PON e os outros paradigmas existentes, mais especificamente com os Paradigma Imperativo (PI) e o Paradigma Declarativo (PD). Estes possuem alguns elementos reaproveitados e adaptados no âmbito do PON. Conforme citado pelo autor, este trabalho é um dos pioneiros na apresentação e na demonstração dos conceitos do PON como um paradigma de programação.

Na dissertação, o PON é descrito como um paradigma inspirado em dois paradigmas: o imperativo e o declarativo. De forma mais estrita, o PON encontra alguma inspiração no Paradigma Orientado a Objetos e no Paradigma Lógico, com ênfase nos conceitos de sistemas baseados em regras [Banaszewski, 2009]. Nesta dissertação conclui-se que o PON oferece uma nova forma de computar, permitindo "curar" algumas das deficiências dos paradigmas atuais. A figura 2.12 mostra um esboço do relacionamento do PON com os paradigmas citados.

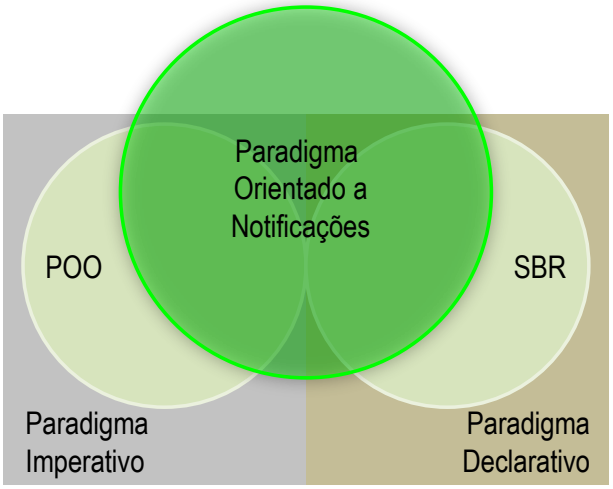


Figura 2.12: Relação entre o PON e os paradigmas imperativo e declarativo [Banaszewski, 2009].

O autor se baseou nos trabalhos anteriores em [Simão, 2001], [Simão, 2005] e [Simão and Stadzisz, 2008]. Entretanto, nesta dissertação foram feitas diversas contribuições para o avanço do paradigma. Um destes avanços foi a definição de um *framework* para linguagem C++ para utilização dos conceitos do PON mais genérico e melhor engenhado que as materializações precedentes [Linhares et al., 2011], incluindo o *framework* prototipal original na tese de Simão [Simão et al., 2012b].

Algumas destas contribuições também podem ser encontradas em [Simão et al., 2010], sendo que este é o pedido de patente referente às mesmas. Por ser o trabalho pioneiro na apresentação dos conceitos do PON, e por apresentar avanços para o mesmo, este é referenciado pela maioria dos trabalhos posteriores a ele.

2.2.2 Estrutura do paradigma

O PON introduz um novo conceito para conceber, construir e executar programas de computador. Ele é baseado na concepção de entidades pequenas, ativas e desacopladas que colaboram por meio de notificações para realização do cálculo lógico e causal existente no *software* [Simão and Stadzisz, 2008, Simão et al., 2012c]. O conhecimento no paradigma é representado através do uso de regras causais que são compreendidas naturalmente por desenvolvedores dos paradigmas atuais [Simão et al., 2012c]. A figura 2.13 contém um exemplo de uma regra decomposta nas entidades do paradigma.

A colaboração entre as entidades é a base para o mecanismo de inferência do paradigma, sendo que a colaboração entre elas é feita através de troca de notificações pontuais. A figura 2.14 mostra o diagrama de classes das entidades citadas e o relacionamento existente entre elas.

As entidades que compõem o paradigma serão descritas em mais detalhes nas seções a seguir.

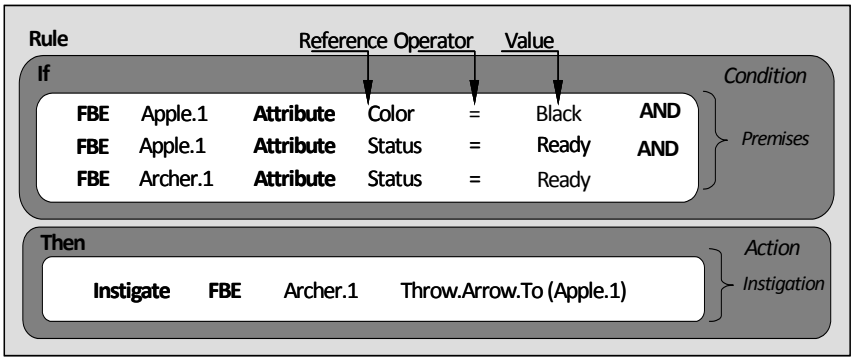


Figura 2.13: Representação de uma regra do PON [Simão et al., 2012c].

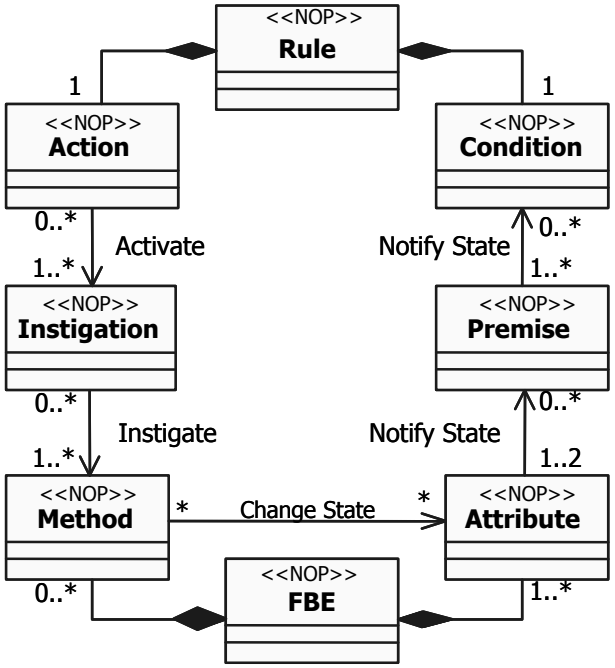


Figura 2.14: Diagrama de classes com as entidades do paradigma [Simão et al., 2012c].

Elemento da Base de Fatos - *Fact Base Element* (FBE)

O *FBE* é um elemento factual do PON que representa uma entidade do sistema observado. Este contém um conjunto de atributos e métodos que são representados pelas entidades *Atributos* e *Métodos* respectivamente. Eles representam os fatos/estados da entidade citada e suas funcionalidades.

Atributo

O *Atributo* representa um valor incluso no FBE que representa uma característica ou um estado do mesmo. Ao ter o seu valor alterado, ele notificará todas as *Premissas* às quais ele está associado, que são precisamente apenas as pertinentes, iniciando assim o processo de inferência do PON.

Premissa

A *Premissa* representa uma indagação lógica entre os *Atributos* de um *FBE* e um valor (por exemplo, "*X* é igual a 2 ?"). Estes são compostos de uma referência para um *Atributo*, o operador lógico de comparação e o valor a ser comparado, sendo que este último pode ser simples (um valor) ou composto (outro *Atributo*).

Condição

A *Condição* representa uma relação lógica entre as *Premissas* da *Regra*. A relação entre as *Premissas* pode ser denotada pelo conector lógico de conjunção (*E*), disjunção (*OU*) ou combinações de ambos.

Regra

A *Regra* representa uma regra da base de regras do sistema, sendo que esta entidade agrega uma *Condição* e uma *Ação*. A relação implementada por esta entidade é de implicação, sendo que esta pode ser lida na forma "Se a *Condição* estiver ativa (antecedente), então ative a *Ação* (consequente)". Este elemento também é responsável por fazer a ligação entre as partes ativa e reativa do processo de inferência.

Ação

A *Ação* representa uma ação a ser executada caso a *Regra* a qual esta entidade está associada for aprovada. Este elemento contém um conjunto de tarefas, representadas pela entidade *Instigação*, que deverão ser executadas simultaneamente ou sequencialmente quando a entidade for ativada.

Instigação

A *Instigação* representa uma tarefa a ser executada pela *Ação* quando ela for ativada. Ela realiza a atividade de indução a execução de um *Método* contido em um *FBE*.

Método

O *Método* representa uma função que poderá ser executada de um *FBE*. Esta função poderá realizar alterações sobre os *Atributos* do mesmo, o que pode ocasionar no inicio de um novo ciclo de notificações.

2.2.3 Mecanismo de notificações

O mecanismo de notificações consiste no processo interno de execução das instâncias do paradigma, o qual determina o fluxo de execução das aplicações. Por meio deste mecanismo, as responsabilidades de um programa são divididas entre as entidades, as quais cooperam por meio de notificações informando umas as outras sobre as parcelas de suas contribuições de inferência a fim de formar o fluxo de execução do programa [Banaszewski, 2009]. A figura 2.15 traz um exemplo de cadeia de notificações entre diversas instâncias de entidades do paradigma.

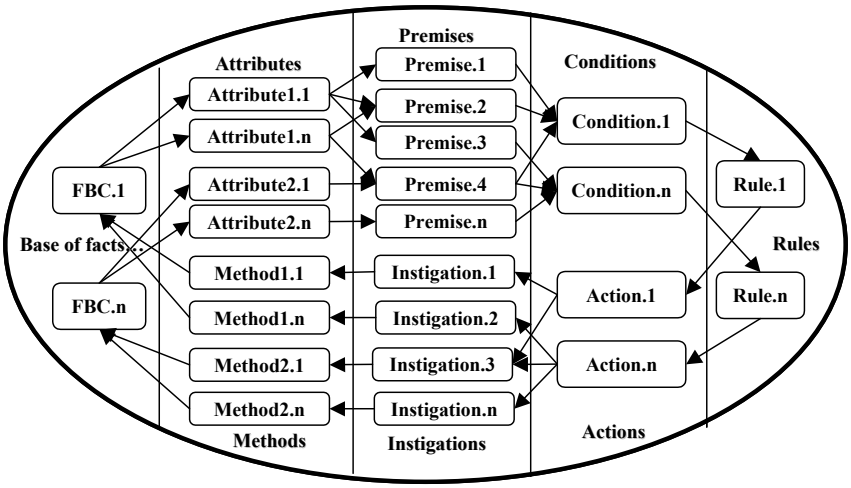


Figura 2.15: Cadeia de notificações do paradigma [Simão and Stadzisz, 2008].

O processo de inferência é iniciado quando o estado (valor) de um *Atributo* de um FBE é alterado(FBC.1 e FBC.n, lado esquerdo da figura). Cada alteração faz com que o *Atributo* notifique as *Premissas* que estão interessadas no mesmo.

As *Premissas*, após notificadas, realizam o cálculo lógico para determinar se ocorreu uma mudança de estado nas mesmas. Este cálculo é feito através da comparação do valor contido no *Atributo* com o valor da *Premissa* usando um operador lógico. As materializações atuais do paradigma suportam os seguintes operadores:

- Igual à ($=$);
- Diferente de ($\neq$);
- Maior que ($>$);
- Menor que ($<$);
- Maior ou igual à ($\geq$);
- Menor ou igual à ($\leq$).

Da mesma forma que um *Atributo* colabora com as *Premissas*, estas colaboram com as *Condições*, notificando-as quando ocorrem mudanças no valor booleano que representa o resultado do cálculo lógico. Ao receber notificações das *Premissas*, as *Condições* realizam seus cálculos lógicos a partir de uma expressão lógica causal. Esta expressão define o relacionamento entre as *Premissas* notificadoras através do uso de conectivos lógicos (“E” e “OU”). Uma expressão pode ser lida conforme o seguinte exemplo: “Premissa1 ativa ‘E’ Premissa2 ativa”. Caso houver uma mudança no estado de uma *Condição*, esta irá realizar a notificação da *Regra* a qual ela está associada. Uma mesma *Condição* pode estar associada, e portanto notificar, uma ou mais *Regras*.

A *Regra*, ao receber a notificação de uma *Condição*, irá verificar se a mesma foi ativa (isto é, se todas as *Condições* associadas a esta regra estão ativas). Caso afirmativo, a *Regra* irá executar a *Ação* associada. Esta irá realizar uma série de instigações através das entidades *Instigações* contidas nela.

Por último, as *Instigações* farão com que os *Métodos* dos FBEs associados sejam executados. A execução dos mesmos pode fazer com que os *Atributos* do FBE sejam alterados. Caso isto ocorrer, será iniciado um novo ciclo de notificações.

A complexidade assintótica polinomial do mecanismo de notificações do PON, no pior cenário, é representada por $O(n^3)$ ou $O(FactBaseSize \times nPremises \times nRules)$, onde *FactBaseSize* corresponde à quantidade máxima de *Atributos*, *nPremises* corresponde à quantidade máxima de *Premissas* notificadas por estes *Atributos* e *nRules* corresponde à quantidade máxima de *Condições* notificadas por estas *Premissas* [Simão, 2005, Banaszewski, 2009].

A função assintótica do PON representa a quantidade de notificações entre as instâncias colaboradoras que também corresponde à quantidade de avaliações lógicas. A figura 2.16 demonstra as relações por notificações entre instâncias colaboradoras. Nela os *Atributos*, *Premissas*, *Condições* e *Regras* correspondem respectivamente aos símbolos com abreviações *Att*, *Pr*, *Cd* e *Rl* [Banaszewski, 2009].

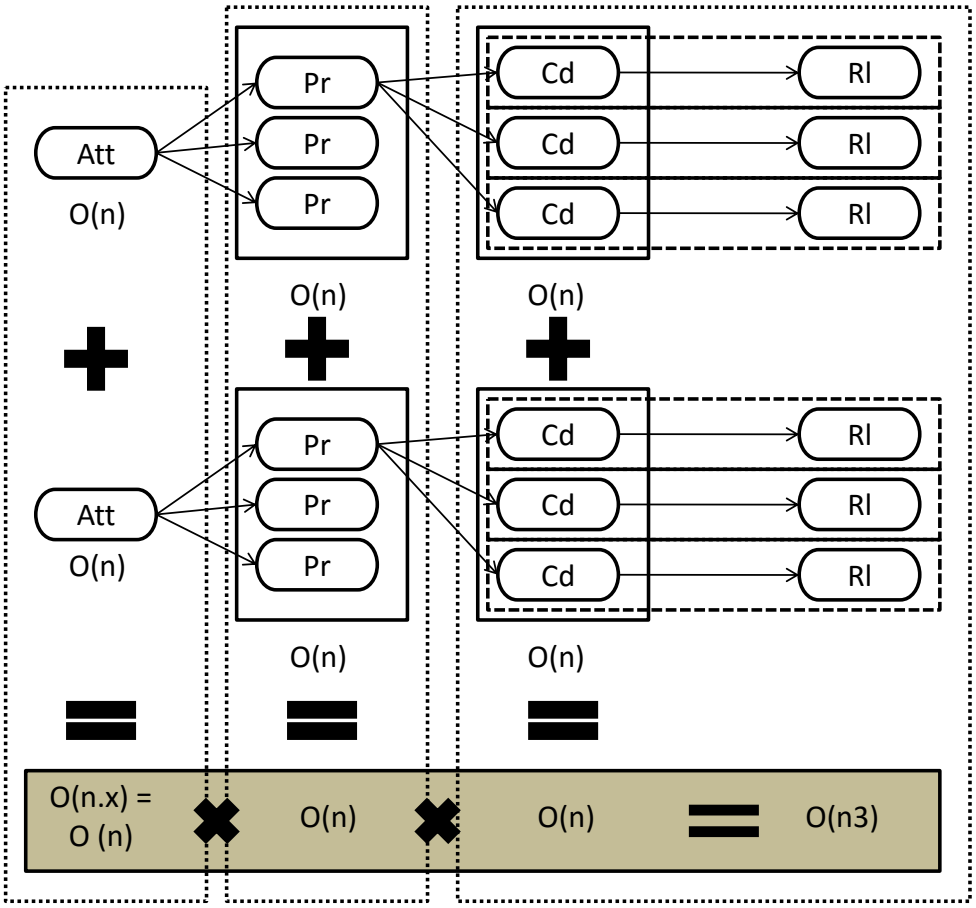


Figura 2.16: Cálculo assintótico do mecanismo de notificações [Banaszewski, 2009].

2.2.4 Mecanismo de resolução de conflitos

Um problema de sistemas computacionais em geral é a identificação e resolução de conflitos. Isto ocorre quando duas atividades (ou entidades) diferentes dependem de um mesmo elemento (ou recurso) compartilhado em um mesmo instante de tempo, sendo que este elemento em questão deve ser utilizado exclusivamente por uma das entidades naquele instante

[Simão and Stadzisz, 2010]. No PON este problema pode ser solucionado através da concepção das regras de forma que apenas uma delas será ativa por vez, gerando um inter-bloqueio entre elas através de verificações suplementares.

Em [Simão, 2005] é proposta a criação de um mecanismo que possa fazer a resolução destes conflitos sem necessitar de recorrer ao inter-bloqueio. Este mesmo mecanismo, que também está descrito em [Simão and Stadzisz, 2010], define que a *Condição* de uma *Regra* aprovada contra-notifique a *Premissa* se esta última estiver associada a um *Atributo* passível de conflito. A figura 2.17 mostra o processo de execução do mecanismo de resolução de conflitos.

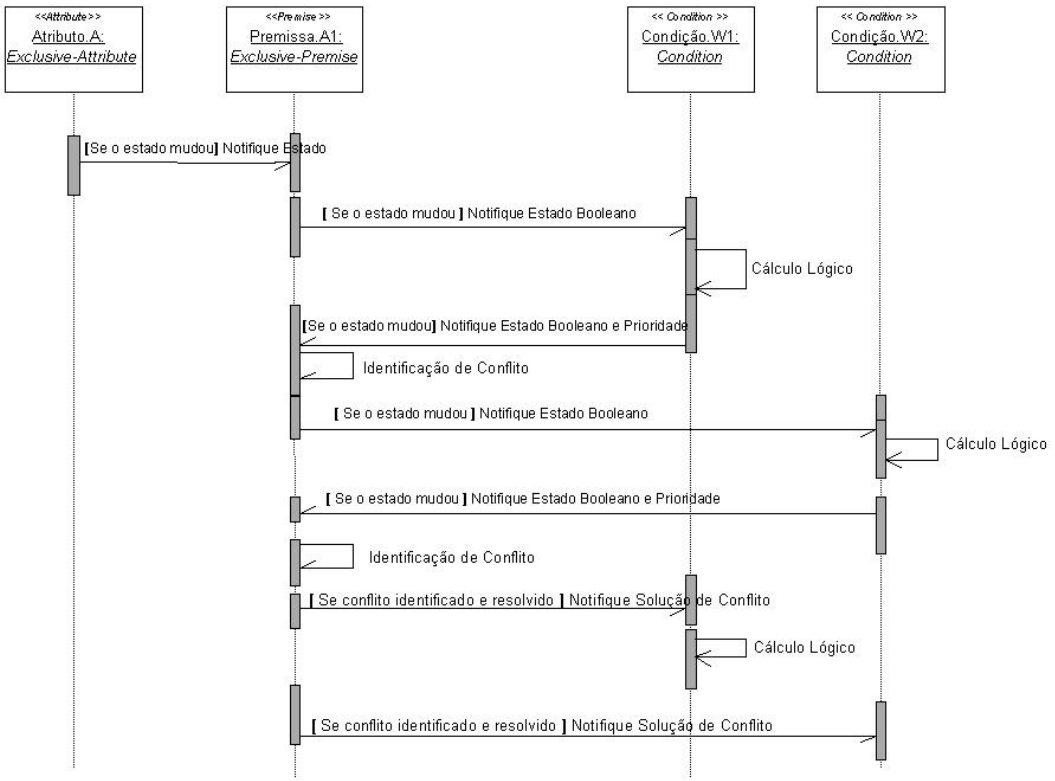


Figura 2.17: Processo de execução do mecanismo de resolução de conflitos [Simão and Stadzisz, 2010].

Ao receber uma contra-notificação, uma *Premissa* incrementa um contador quando o estado da *Condição* que o contra-notificou for verdadeiro, sendo que, caso contrário, este contador é decrementado. A *Premissa* identifica como uma situação de conflito quando este contador está com o valor maior que um, significando que duas ou mais *Condições* estão aptas para ativação. A partir deste instante, a *Premissa* pode realizar duas ações:

- Resolução por parte da própria *Premissa*: esta define qual *Condição* será ativa com base nos valores definidos de prioridade;
- Notificação de um resolutor de conflitos: a *Premissa* notifica uma entidade a parte que realizará a resolução do conflito. O mecanismo que define como será realizada esta resolução é deixado a cargo de cada implementação.

Após a definição da *Condição* que será ativa, as outras *Condições* que tiverem o seu estado com o valor verdadeiro serão desaprovadas. A vantagem deste mecanismo é que não

sobrecarrega o processo de inferência uma vez que gera algumas poucas notificações adicionais e facilita a composição das instâncias uma vez que dispensa a geração de conhecimento adicional nas *Regras* para tratar conflitos [Simão and Stadzisz, 2010]. Porém, conforme descrito em [Simão et al., 2010], a abordagem apresenta alguns problemas, sendo que estes são os seguintes:

- Geração de muitas contra-notificações não úteis em um ambiente monoprocessado;
- Indisponibilidade, *à priori*, de um mecanismo de concorrência em ambientes concorrentes (monoprocessado e multiprocessado).

Além da apresentação do paradigma, em [Banaszewski, 2009] também são propostas três abordagens para a resolução de conflitos que, conforme descrito em [Simão et al., 2010], solucionam os problemas citados acima. As propostas são as seguintes:

- Modelo centralizado de resolução de conflitos: emprega uma entidade centralizadora de conflitos que, através do emprego de alguma estrutura de dados linear (pilha, fila ou lista), armazena as regras ativas. As regras ativas, conforme a estratégia selecionada, serão executadas uma por vez;
- Modelo descentralizado de resolução de conflitos em ambientes não concorrentes monoprocessados: neste modelo não há um conjunto de conflitos uma vez que este se baseia no imediatismo da execução de uma regra. A regrá deverá ser executada no exato instante em que esta foi ativa;
- Modelo descentralizado de resolução de conflitos em ambientes concorrentes monoprocessados e multiprocessados: este modelo é uma evolução do modelo apresentado em [Simão and Stadzisz, 2010]. A diferença é que, para que uma *Condição* seja aprovada, além dela apresentar todas as suas *Premissas* com os estados lógicos verdadeiros, esta também deverá adquirir exclusividade de acesso aos *Atributos* passíveis de conflito.

2.2.5 Mecanismo de garantia de determinismo

Juntamente com a definição do mecanismo de resolução de conflitos citado anteriormente, em [Simão and Stadzisz, 2010] é definido um mecanismo para garantir o determinismo da execução do paradigma. O determinismo no PON significa que todas as entidades dedicadas às relações causais devem ter a mesma oportunidade para avaliar as mudanças de estados em entidades pertinentes, visando uma decisão determinística. Esta última, por sua vez, significa que uma decisão tomada deve ser a mesma dado um mesmo conjunto de dados [Simão and Stadzisz, 2010].

O mecanismo de garantia de determinismo funciona da seguinte maneira [Simão and Stadzisz, 2010]:

1. Cada *Condição* que foi notificada confirma, por meio de uma contra-notificação, o recebimento e utilização do estado da *Premissa* notificadora;
2. Cada *Premissa* que foi contra-notificada confirma, por meio de outra contra-notificação, o recebimento e utilização do valor atribuído no *Atributo* notificador;

3. As *Condições* aprovadas perguntam, através de notificações, para cada *Atributo* associado às suas *Premissas* se estes finalizaram a suas atividades de notificação;

4. Os *Atributos* notificados no passo anterior contra-notificam as *Condições* notificadoras para informar o fim das suas atividades de notificação;

5. As *Condições* aprovadas, após o recebimento das confirmações, notificam suas respectivas *Regras* sobre a sua aprovação.

A figura 2.18 mostra o processo de execução do mecanismo de garantia de determinismo do paradigma definido na patente.

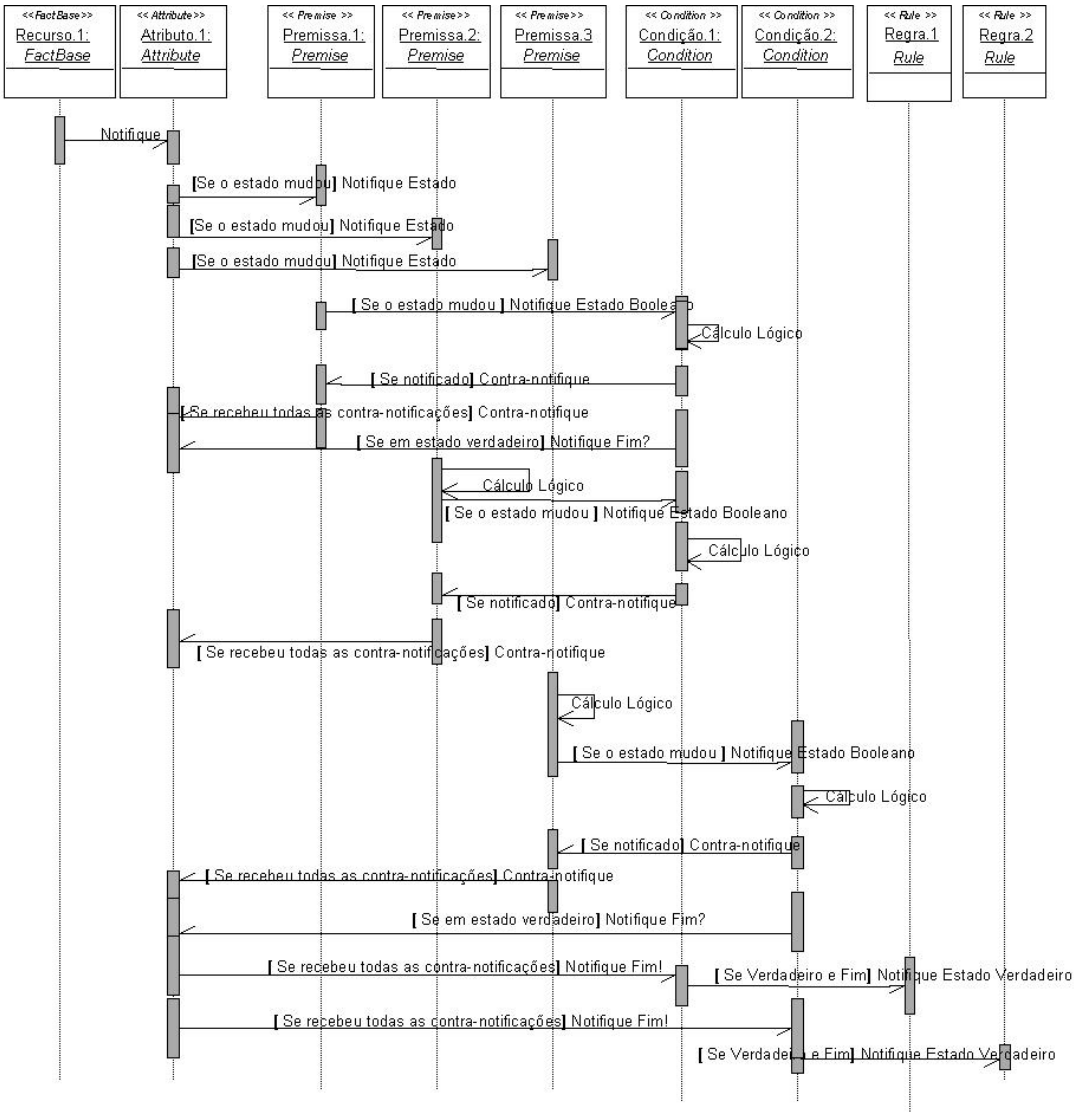


Figura 2.18: Processo de execução do mecanismo de resolução de conflitos [Simão and Stadzisz, 2010].

Deste modo, todas as *Condições* são obrigadas a esperar que todas as avaliações sejam realizadas antes de notificar as *Regras* para que estas possam ser ativadas.

2.2.6 *Frameworks* do PON

Os conceitos do PON propriamente dito foram materializados sobre o POO na forma de *frameworks* desenvolvidos na linguagem de programação C++. Ao longo do desenvolvimento do PON foram concebidas diversas versões de *frameworks*:

- A versão prototipal foi concebida por Simão em 2007 a partir da evolução da tecnologia de controle com agentes descritas em [Simão, 2001] e [Simão, 2005];
- A versão original do *framework* PON desenvolvida em [Banaszewski, 2009];
- A versão otimizada definida em [Valença, 2012] e incrementada em [Ronszcka, 2012].

Estruturalmente, o *framework* PON original materializa as entidades colaboradoras do paradigma em forma de classes/objetos relacionados através de estruturas de dados com referências às entidades interessadas em seus estados. Nesta materialização o desenvolvedor se preocupa somente em instanciar as *Regras* em alto nível no código C++ baseado no *framework* PON, onde o encadeamento das estruturas notificantes será realizada em tempo de compilação em cada aplicação PON [Valença, 2012].

Neste âmbito, uma estrutura de pacotes foi projetada, conforme apresentado na figura 2.19, para modelar (e explicar) esta materialização do PON. Isto dito, para desenvolver aplicações com o *framework* PON em questão, é necessário que o desenvolvedor estenda a classe `Application` contida no pacote `Application`, a qual define uma ponte de ligação entre uma aplicação PON e seu *framework*. Ainda, esta classe `Application` se relaciona com as classes contidas no pacote `Core`. Este por sua vez, contém as classes responsáveis pelo processo de notificação e realização do cálculo lógico causal do PON [Valença, 2012].

Conforme apresentado na figura citada, o *framework* PON é subdividido em alguns pacotes, porém as entidades notificadoras que dão "vida" às aplicações desenvolvidas nesse paradigma estão concentradas no pacote `Core`. Ele é formado pelas classes colaboradoras que realizam o processo de notificação do *framework* PON [Valença, 2012]. As classes contidas neste pacote são apresentadas na figura 2.20.

As classes `Rule` e `FBE` se apresentam nas extremidades opostas e se relacionam por meio de suas classes colaboradoras – `Attribute`, `Premise`, `Condition`, `Action`, `Instigation` e `Method` – sendo que a colaboração entre os objetos destas classes determina o fluxo de execução de uma aplicação do PON. Ademais, as classes `Method` e `Rule` que definem as entidades puras do PON são estendidas de modo a proporcionar funcionalidades adicionais [Valença, 2012].

Além do pacote `Core`, o *framework* é composto também pelos pacotes `Attributes` e `Conditions`. Conforme apresentado na figura 2.21, o pacote `Attribute` é formado pelas classes responsáveis por encapsular os tipos primitivos da POO. Estas classes (`Boolean`, `Char`, `Double`, `Integer` e `String`) introduzem reatividade aos tipos primitivos, permitindo que estes façam parte de estruturas causais do PON. Já o pacote `Conditions` é formado pelas classes que fazem parte da composição de um objeto do tipo `Condition`. Particularmente, a classe `LogicalOperator` e suas derivadas (`Conjunction`, `Disjunction` e `Single`), definem a operação lógica utilizada pelo objeto do tipo `Condition` de maneira a aprovar uma determinada *Regra* representada pelo objeto do tipo `Rule` [Valença, 2012].

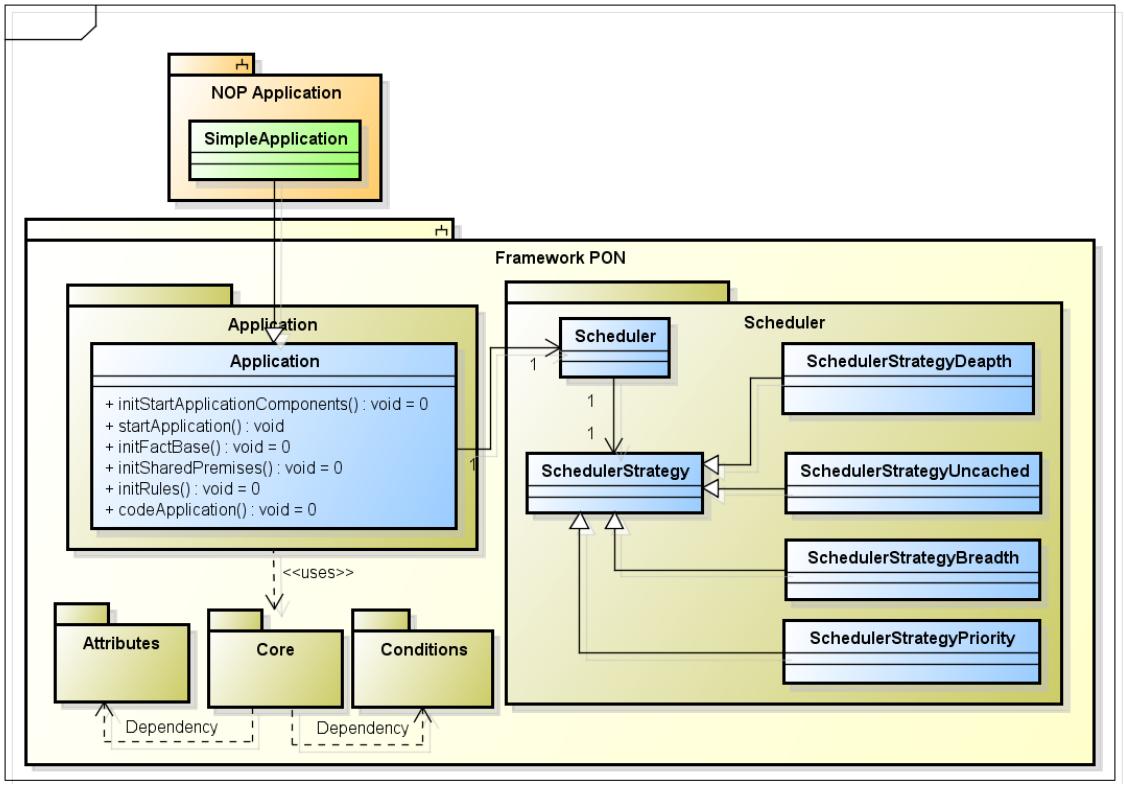


Figura 2.19: Estrutura do *framework* PON original [Valença, 2012].

O *framework*, por ser um conjunto de classe cujos métodos são materializados segundo o PI, tem sua implementação amplamente baseada em busca sobre estruturas de dados (originalmente fornecidas pela *Standard Template Library* - STL, biblioteca padrão da linguagem C++) para avaliação de relações lógico-causais e decisão sobre o envio de notificações. Esta forma de implementação do *framework* é desvantajosa à filosofia do PON, pois depende fundamentalmente de estruturas de dados e percurso sequencial sobre estas estruturas efetuado em PI [Linhares, 2015].

Dadas algumas questões de implementação do *framework* PON, em particular o *overhead* causado pela sua implementação baseada em estrutras de dados, em [Valença, 2012] foi realizada uma série de otimizações com o objetivo de melhorar o desempenho de aplicações sobre o *framework*. O resultado deste trabalho é uma versão do *framework* baseada em uma variedade de estruturas mais otimizadas do que as suas equivalentes baseadas em STL [Linhares, 2015].

Basicamente as classes que possuem manipulação (adição, remoção) e iteração (percorrimento) de elementos PON são especializadas para realizarem estas operações em suas respectivas estruturas de dados. Com isso, os métodos responsáveis por realizar a manipulação/iteração dos elementos nas estruturas de dados são sobrescritos e implementados em suas classes derivadas [Valença, 2012]. A figura 2.22 apresenta a estrutura de classes dos elementos iterados.

A estratégia é realizar a iteração dos elementos PON em cada instância de classe correspondente a uma respectiva estrutura de dado. Por exemplo, os *Atributos*, que possuem uma lista de *Premissas* a serem notificadas, possui cinco versões, sendo que cada uma é para uma estrutura de dados diferente [Valença, 2012]:

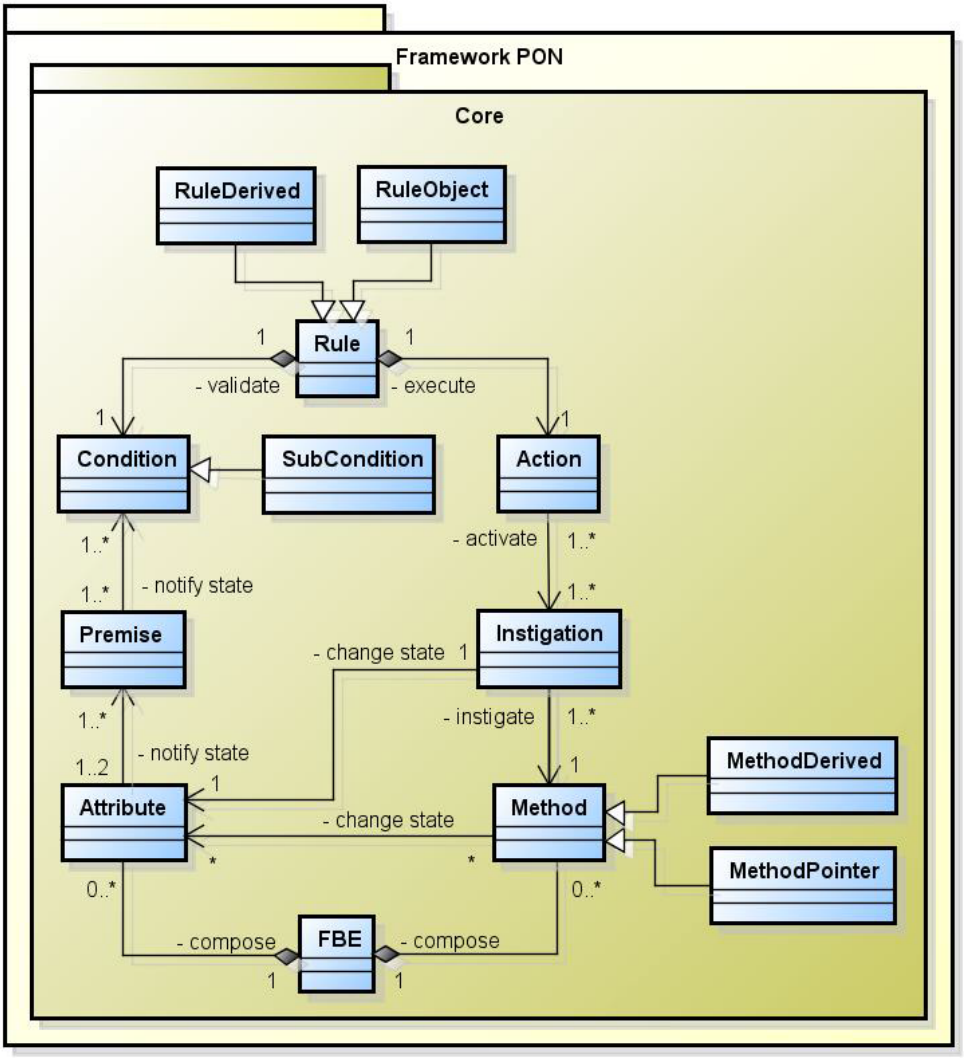


Figura 2.20: Estrutura do pacote `Core` [Valença, 2012].

- `PONHashAttribute`: referencia as *Premissas* usando uma tabela *hash*;
- `PONListAttribute`: referencia as *Premissas* usando uma implementação própria de uma lista encadeada;
- `PONVectorAttribute`: referencia as *Premissas* usando uma implementação própria de um *vector*, que é uma lista encadeada de vetores (*arrays*);
- `PONListSTLAttribute`: referencia as *Premissas* usando a implementação da STL de uma lista encadeada.

Assim os métodos de manipulação dos elementos PON das classes pertencentes ao pacote `Core`, são formadas por métodos virtuais puros, os quais são implementados em suas classes derivadas. Isto, de fato, flexibiliza o poder de manipulação dos elementos PON em cada estrutura de dados implementada sobre *framework* PON [Valença, 2012].

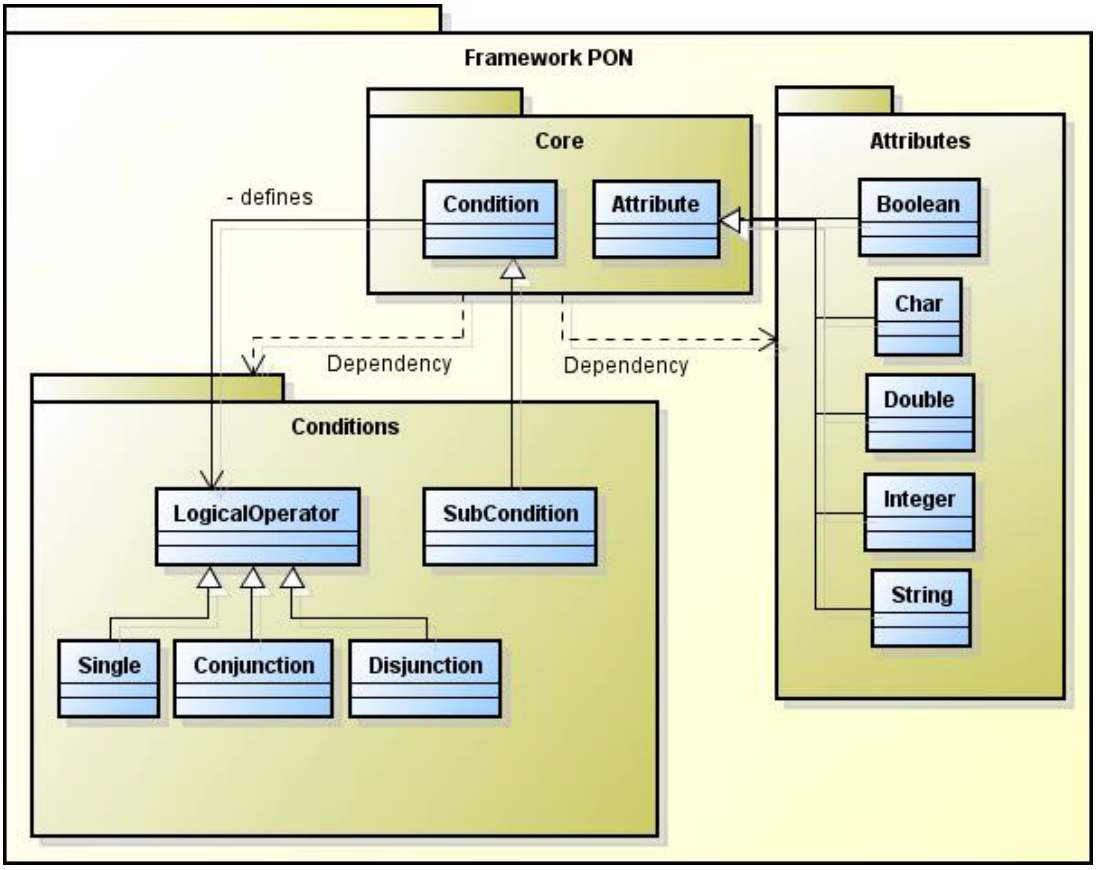


Figura 2.21: Estrutura dos pacotes `Attributes` e `Conditions` [Valença, 2012].

2.2.7 Linguagem para o PON

As materializações do PON se apresentavam como alternativas para o desenvolvimento de aplicações sob o viés deste paradigma. No entanto, conforme apresentado em trabalhos anteriores, elas não haviam alcançado resultados satisfatórios em termos de desempenho [Ferreira, 2015].

No tocante a facilidade de programação, por sua vez, o PON apresenta uma forma de compor software em alto nível pela representação do conhecimento lógico-causal com a utilização de regras lógico-causais, inspirando-se neste sentido nos SBRs. No entanto, com a utilização das materializações existentes na época, as regras podem ficar dispersas no código fonte, tornando difícil o entendimento do software composto. Ainda, a utilização delas, de certa forma, era uma tarefa difícil, pois o desenvolvedor ainda necessitava do conhecimento prévio do paradigma e de seus componentes para ser possível compor o software [Ferreira, 2015].

Sendo assim, com o advento da linguagem PON o desenvolvedor poderá compor as regras em alto nível utilizando elementos da linguagem para tal. Para dar suporte a linguagem PON em um compilador próprio, foi especificada uma gramática segundo a Backus-Naur Form (BNF) [Ferreira, 2015].

De modo geral, o código fonte da linguagem PON (conhecida também como Ling-PON) 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 *Regras*. Subsequentemente é necessário definir

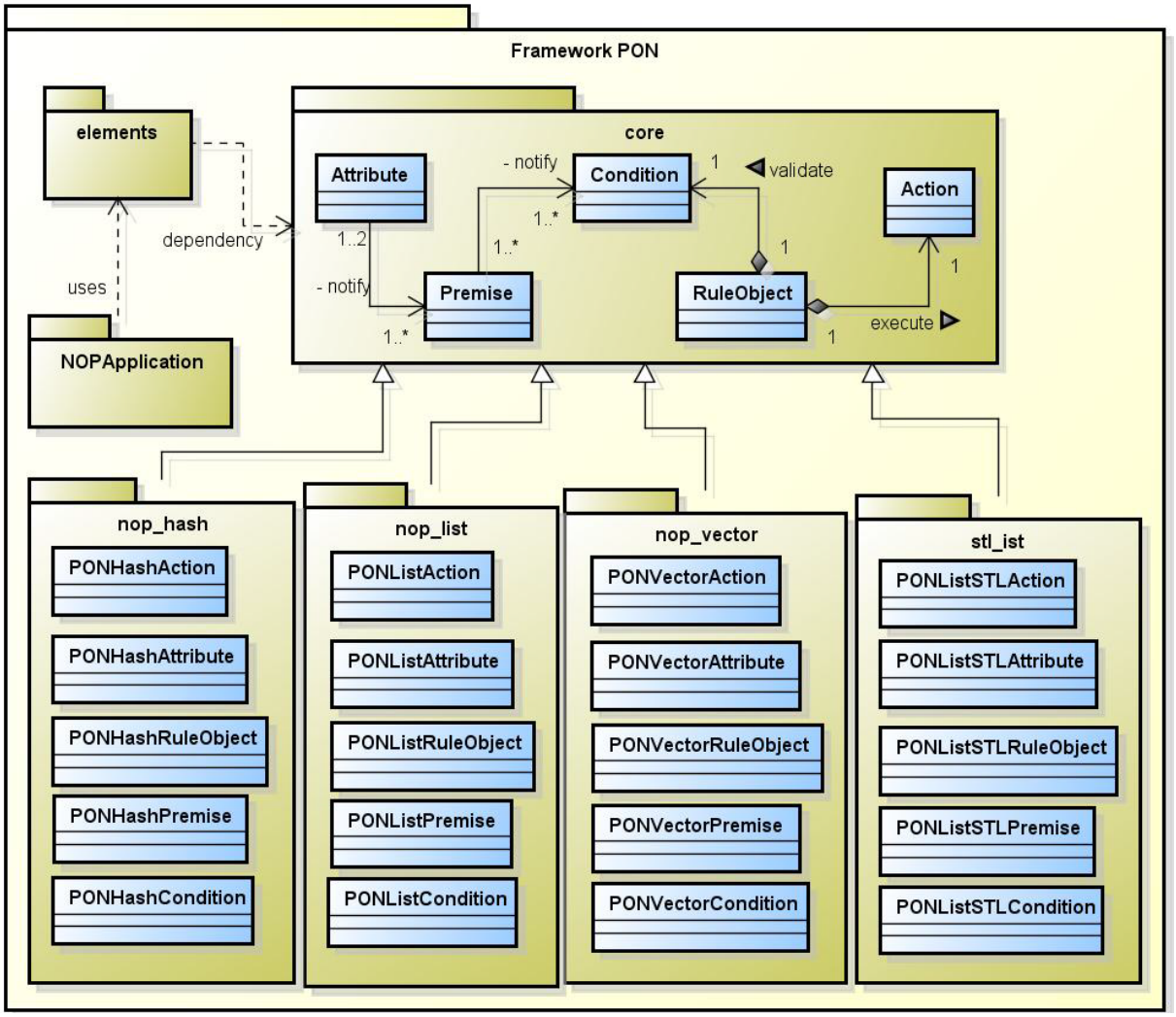


Figura 2.22: Estrutura de classes com elementos iterados [Valença, 2012].

as *Regras* para fins de avaliação lógico causal dos estados do FBEs por meio de notificações. Por fim, é possível adicionar código específico da linguagem alvo escolhida no processo de compilação (C ou C++) com a utilização do bloco de código *main* [Ferreira, 2015].

Conforme apresentado no algoritmo 2.1, o código PON é 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 *Regras*. Por fim, a quinta parte apresenta o bloco de código `main`.

Algoritmo 2.1: Padrão de declarações da linguagem PON [Ferreira, 2015].

```
1  fbe Apple
2     . . .
3  end_fbe
4
5  fbe Archer
6     . . .
7  end_fbe
8
9  fbe Controller
10    . . .
```

```
end_fbe

------------------

inst
    Apple apple, apple1
    Archer archer, archer1
    Controller controller, controller1
end_inst

------------------

strategy
        . . .
end_strategy

------------------

rule RlTurnOn1
   . . .
end_rule

------------------

main {
        . . .
}
```

A primeira parte do código deverá conter a lista de FBEs do sistema. Uma FBE é composta por duas partes. A primeira parte representa a declaração dos *Atributos*, enquanto a segunda parte representa a definição dos *Métodos*. Cada parte, assim como no código apresentado no algoritmo anterior, possui uma palavra reservada para a abertura (ex. `attributes`) do bloco e outra para o fechamento (ex. `end_attributes`) [Ferreira, 2015]. O algoritmo 2.2 demonstra um exemplo de criação de FBEs.

Algoritmo 2.2: Exemplo de criação de FBEs [Ferreira, 2015].

```
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
```

A segunda parte do código consiste na instanciação dos FBEs definidos na primeira parte. Vale lembrar que poderão existir múltiplas instâncias de um único FBE, sendo que cada uma delas deverá possuir um nome distinto. O algoritmo 2.3 demonstra o exemplo de criação das instâncias dos FBEs.

Algoritmo 2.3: Exemplo de criação das instâncias de FBEs [Ferreira, 2015].

```
inst
    Apple apple, apple1
    Archer archer, archer1
    Controller controller, controller1
end_inst
```

A terceira parte do código consiste na definição da estratégia de escalonamento das *Regras* ativadas definidas em [Banaszewski, 2009] a ser utilizada. Poderá ser selecionada uma

das duas opções (`BREATH` e `DEPTH`) de escalonamento ou a opção para executar as *Ações* das *Regras* no instante em que elas são ativadas (`NO_ONE`). O algoritmo 2.4 apresenta um exemplo de seleção da opção de escalonamento `NO_ONE`.

Algoritmo 2.4: Exemplo de seleção da estratégia de escalonamento [Ferreira, 2015].

```
strategy
   no_one
end_strategy
```

A quarta parte do código consiste na criação do conhecimento lógico-causal da aplicação por meio da criação de regras, onde as *Regras* farão as conexões entre as entidades PON [Ferreira, 2015]. O algoritmo 2.5 apresenta o padrão de implementação para a criação de uma *Regra*.

Algoritmo 2.5: Exemplo de criação de *Regras* [Ferreira, 2015].

```
rule RlTurnOn1
   condition
      subcondition A1
         premise PrIsCrossed apple.atIsCrossed == false and
         premise PrHasFired archer.atHasFired == false and
         premise PrFire controller.atFire == true
      end_subcondition
      or
      subcondition A2
         premise PrIsCrossed archer.atCount == 0
      end_subcondition
   end_condition
   action
      instigation inFire archer.mtFire1();
   end_action
end_rule
```

Basicamente, cada *Regra* é composta por três partes, que são as suas propriedades (opcional), a *Condição* (expressão lógica) e a *Ação* (execução). Nessa versão da linguagem, é obrigatória a utilização de *Sub-Condições*, seguidas de um identificador, mesmo para a composição de expressões simples. Uma *Sub-Condição* necessita de ao menos uma *Premissa*. No caso da utilização de mais de uma *Premissas*, estas devem ser aninhadas com operador de conjunção "E" (`AND`). Para utilização do operador de disjunção "OU" (`OR`) em uma *Regra* é necessário a criação de duas ou mais *Sub-Condições* conforme exemplificado no algoritmo anterior.

Por fim, 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. É importante ressaltar que código na linguagem alvo não é avaliado pelas regras de compilação da LingPON, sendo somente copiado ao código alvo gerado pelo compilador [Ferreira, 2015].

2.2.8 Propriedades inerentes ao PON

A essência da computação no PON está organizada e distribuída entre entidades autônomas e reativas que colaboram por meio de notificações pontuais. Esta organização 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 entidades do modelo, o que permitiria execução otimizada e minimamente acoplada. Neste âmbito, pode-se dizer que aplicações no PON possuem características apropriadas para a execução em ambientes multiprocessados, uma vez que é necessário apenas que as entidades noti-

ficantes conhecerem as entidades a serem notificadas para a inferência por notificação ocorrer [Banaszewski, 2009].

Neste sentido, este modelo 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 que evitaria uso desnecessário de recurso, como processamento e energia.

2.2.9 Materializações do PON

Nesta seção estão contidos estudos sobre os trabalhos que utilizaram os conceitos do PON. Estes trabalhos contêm algumas aplicações que foram realizadas para provar a usabilidade do PON em diversos ambientes diferentes.

Sistema de inferência *fuzzy* utilizando PON

Em [de Souza et al., 2009] é proposta uma máquina de inferência *fuzzy* utilizando os conceitos do PON. O trabalho adapta a proposta descrita em [Simão et al., 2003a] para o paradigma proposto e aplica estes conceitos no desenvolvimento de um sistema *fuzzy* para tratar a situação de um robô seguir uma bola em um campo de futebol de robôs. A adaptação feita foi estender o *framework* proposto em [Banaszewski, 2009] para que este suporte os objetos necessários, como, por exemplo, as variáveis linguísticas para realização de uma inferência *fuzzy*.

Os resultados apresentados foram obtidos a partir da execução de 10 mil processos de inferência em dois sistemas de controle *fuzzy* desenvolvidos: um utilizando uma abordagem tradicional e outro utilizando o *framework* adaptado. Os dois sistemas *fuzzy* utilizam um conjunto de 21 regras propostas no trabalho. Os testes foram executados em um simulador devido a limitações nos robôs disponíveis na época da elaboração do artigo.

A comparação de resultados foi feita com base em um sistema *fuzzy* proposto em [Fabro, 1996]. Os resultados demonstraram que o sistema *fuzzy* desenvolvido com PON realizou a avaliação de 40 mil regras enquanto o sistema *fuzzy* convencional avaliou 210 mil regras. Apesar disso, o desempenho do sistema PON *fuzzy* se mostrou inferior ao desempenho do sistema *fuzzy* puro.

Implementação do PON em *hardware*

Em [de Witt and Linhares, 2010] é apresentada a proposta de um modelo para a execução em *hardware* da cadeia de notificações do PON. Este modelo é constituído por blocos básicos de lógica reconfigurável que representam os objetos colaboradores da cadeia de notificações do PON e que podem ser combinados para a implementação de partes do sistema modelado em PON em um dispositivo de lógica reconfigurável [de Witt and Linhares, 2010, de Witt et al., 2011].

Devido a baixa complexidade do funcionamento das entidades do paradigma, estas podem ser modeladas com blocos lógicos simples de circuitos combinacionais. Isto viabiliza a implementação das mesmas diretamente em *hardware* reconfigurável [de Witt and Linhares, 2010]. No mesmo trabalho foi desenvolvido um estudo de caso referente

a um simulador de sistema de telefonia. O sistema é composto por dois terminais telefônicos simulados operados por comandos enviados via porta serial. A figura 2.23 apresenta a arquitetura do estudo de caso.

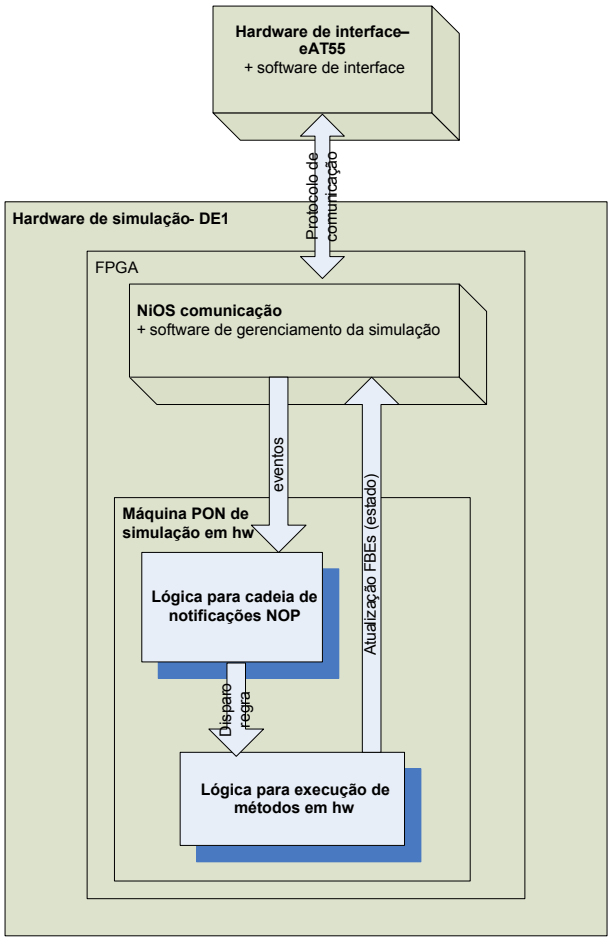


Figura 2.23: Arquitetura do sistema de telefonia [de Witt and Linhares, 2010].

Foram criadas duas versões deste sistema para a realização dos testes: uma utilizando o modelo proposto e outra utilizando o *framework* proposto em [Banaszewski, 2009]. Nos testes foi feita a contagem de ciclos de *clock* necessários para a execução de cada um dos casos definidos no trabalho. Os resultados obtidos demonstraram que a versão em *hardware* teve um desempenho muito superior comparado a versão em *software*. Apesar disso, a versão em *hardware* não dispõe da mesma flexibilidade da versão em *software*. O modelo apresentado neste trabalho foi utilizado para a elaboração de um pedido de patente [Simão et al., 2012a].

Framework para geração de *hardware* a partir do PON

Em [Jasinski, 2012] está contida a proposta de um *framework* que, a partir da descrição do sistema PON, gera o código-fonte em VHDL para ser embarcado em um FPGA. Neste trabalho é descrita a metodologia de desenvolvimento utilizada, os artefatos gerados como, por exemplo, os modelos das entidades do PON em FPGA, e os resultados obtidos nos testes realizados.

O *framework* inicia a geração do sistema em FPGA ao realizar a leitura da definição do sistema a partir de um arquivo de entrada. A descrição do sistema neste arquivo é feita no formato YAML, sendo que esta é uma linguagem de serialização de dados alternativa ao XML.

Após a leitura do arquivo de entrada e a identificação dos componentes (entidades) que foram utilizados, é feita a instanciação de cada um deles. Isto é feito através do uso de *templates*, sendo que estes consistem em trechos de código VHDL intercalados com código escrito na linguagem Ruby. Este último será executado pelo *framework* no instante em que os componentes forem instanciados, gerando o código VHDL com as entidades de acordo com a definição do sistema. O código gerado pode ser sintetizado para o *hardware* utilizando-se as ferramentas que os fabricantes de FPGA disponibilizam.

Co-processador otimizado para aplicações desenvolvidas utilizando PON

Em [Peters, 2012] é proposto o desenvolvimento de um coprocessador especializado em executar as lógicas de notificação definidas pelo paradigma. Neste trabalho o coprocessador proposto é implementado na forma de um periférico a ser ligado no barramento de um processador embarcável. A estrutura fundamental deste periférico se baseia em converter os componentes não-dinâmicos do paradigma em componentes de *hardware*. Isto livra o processador da tarefa de processar a cadeia de notificações do PON [Peters, 2012]. A figura 2.24 demonstra a arquitetura interna do coprocessador PON.

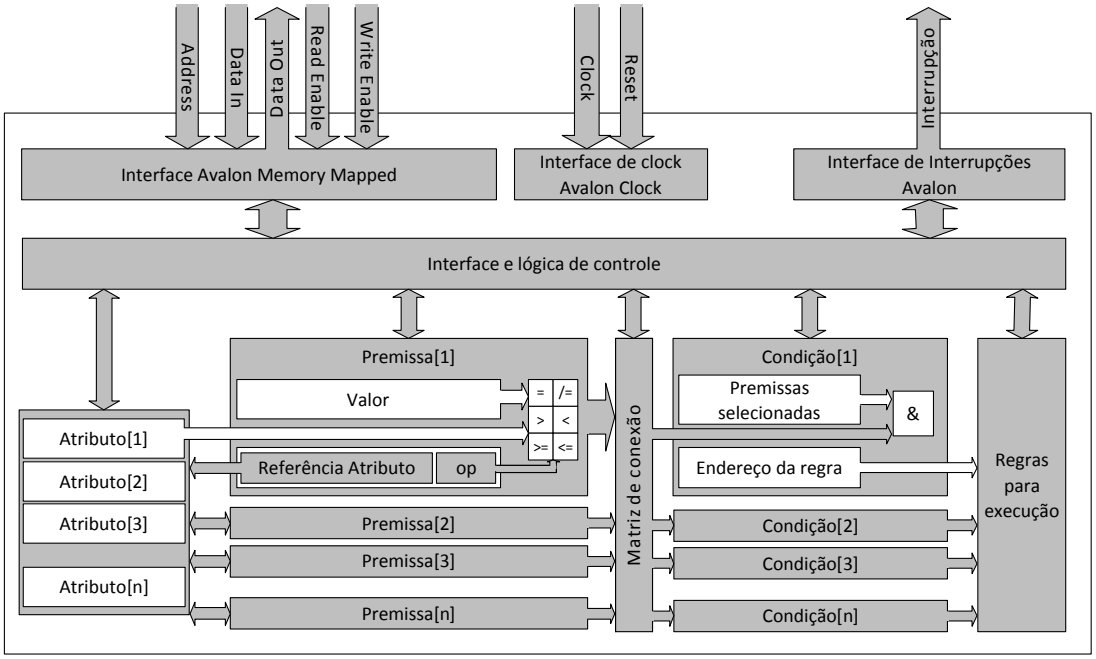


Figura 2.24: Arquitetura do coprocessador proposto [Peters, 2012].

Além da definição do coprocessador, este trabalho também propõe mudanças sobre o *framework* proposto em [Banaszewski, 2009] para que este se utilize das funcionalidades de tal coprocessador. A utilização do *framework* permitiu que não fosse necessário realizar alterações nas aplicações desenvolvidas para utilizar ou não os recursos do coprocessador.

O autor deste trabalho concluiu que os componentes não-dinâmicos (*Atributos*, *Premissas*, *Condições* e *Regras*) podem trazer benefícios se implementados em *hardware*, sem

trazer prejuízos à usabilidade do *framework* PON alterado. Os outros componentes (*Ações*, *Instigações* e *Métodos*) podem assumir diferentes formas e dependem da aplicação em questão para sua criação. Logo, optou-se por não implementá-los em *hardware* para não prejudicar a flexibilidade do *framework* [Peters, 2012].

Os trabalhos citados nesta seção indicam haver possibilidades de aumento de eficiência tanto energética quanto computacional ao utilizar os conceitos do PON para o desenvolvimento de sistemas tanto em software quanto em hardware.

Linguagem e compilador para o PON

Em [Ferreira, 2015] foi definida uma linguagem de programação que pode ser utilizada para o desenvolvimento de aplicações utilizando os conceitos do PON. A linguagem foi definida de acordo com o formato BNF (*Backus Normal Form* - Forma Normal de Backus) que é utilizada para a especificação de linguagens de programação, e posterior criação de um compilador. A linguagem está descrita com maiores detalhes na seção 2.2.7. A figura 2.25 apresenta as principais etapas de execução do compilador para a linguagem PON.

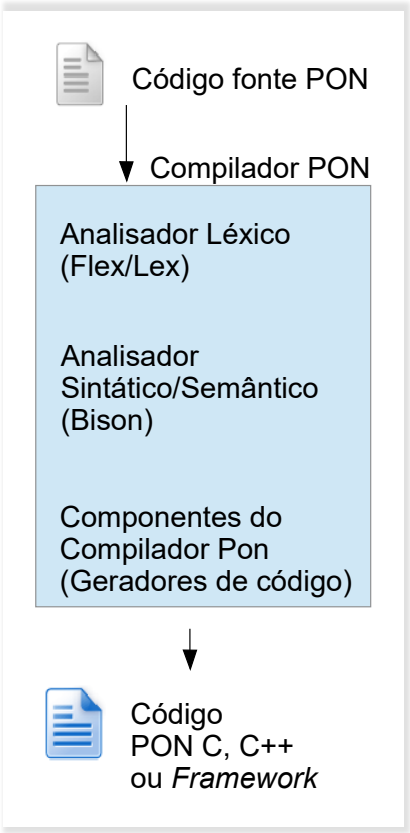


Figura 2.25: Etapas de execução do compilador PON [Ferreira, 2015].

A partir da definição da linguagem foi utilizada a ferramenta de software livre `flex` para a geração de um analisador léxico da linguagem. Este analisador lê a sequência de caracteres contidas em um arquivo fonte e realiza processamento sobre eles, verificando se esta sequência está de acordo com a definição da linguagem. A partir deste processamento é gerada uma sequência de *tokens*, que são símbolos que compõem a linguagem (tais como palavras

reservadas, identificadores de variáveis, símbolos separadores, operadores lógicos e matemáticos). Esta sequência de símbolos, denominados lexemas, será então processada pelo analisador sintático.

Para realizar a segunda etapa do processo de compilação, é utilizada a ferramenta de software livre `bison` para a geração do analisador sintático e semântico da linguagem. O papel do analisador sintático é verificar se a sequência de *tokens* corresponde a construções sintaticamente corretas da linguagem especificada pela BNF, e permite ao programador definir uma rotina a ser executada quando um determinado *token* da linguagem for encontrado. O analisador semântico verifica se o código fonte está correto também em questões de compatibilidade de tipos, adequação dos operadores aos operandos, entre outras verificações necessárias, não cobertas pela análise sintática. Neste trabalho, estas ferramentas foram utilizadas para criar as etapas de análise do compilador para a linguagem PON-Fuzzy aqui especificada e desenvolvida.

O compilador, a partir da execução dos analisadores léxico, sintático e semântico, gera estruturas de dados correspondentes, na medida que os elementos do paradigma são identificados no código fonte. Estas estruturas são utilizadas na etapa de geração de código por parte do compilador. O compilador PON apresentado por [Ferreira, 2015] é capaz de gerar código nas linguagens C e C++. Os códigos gerados contêm as estruturas declaradas no código-fonte PON, definidas na forma de estruturas internas das linguagens, onde o processo de inferência é realizado através da troca de mensagens (chamadas de funções) entre as estruturas. No caso do C++, o compilador também é capaz de gerar código utilizando-se das estruturas contidas no *framework* definido em [Valença, 2012, Ronszcka, 2012] (chamado de *framework* otimizado).

No código gerado para a linguagem C, os FBEs são declarados como estruturas (tipos de dados *struct*), onde cada campo representa um *Atributo*, e são alocados no segmento de dados da aplicação através do uso de variáveis globais. Os *Métodos* do FBE são funções que fazem acesso as variáveis que representam os FBEs para realizar alguma operação.

As *Premissas* são representadas por funções com implementações específicas que fazem a checagem dos valores dos *Atributos* conforme definido no código-fonte do PON. Os valores de ativação das *Premissas* também são armazenados no segmento de dados para consulta por parte das *Condições*.

As *Condições* também são implementadas como funções e armazenam o valor de ativação da *Regra* após serem chamadas. A *Regra*, por sua vez verifica se ela está aprovada e realiza a chamada dos *Métodos* do FBE sem o intermédio das *Ações* e *Instigações*.

No caso do código gerado para a linguagem C++ sem o uso do *framework*, os FBEs são declarados como classes, onde cada campo representa um *Atributo* e cada função representa um *Método*. Estas classes contêm as *Premissas* associadas a seus *Atributos*, sendo que estas possuem a referência para *Regra* que contém a *Condição*. A ativação da *Regra* faz com que os *Métodos* do FBE sejam chamados diretamente, sem o intermédio das *Ações* e *Instigações*.

O código gerado para o *framework* do PON gera as entidades utilizando as estruturas já definidas no mesmo. O FBE contém um conjunto de objetos que representam os *Atributos* que são associados as outras entidades através dos mecanismos internos do *framework*.

[Ferreira, 2015] apresenta testes comparativos entre os códigos gerados para as linguagens citadas. No geral, o código gerado para a linguagem C obteve um desempenho melhor em relação aos códigos gerados para a linguagem C++ (puro e *framework* otimizado). Este código gerado obteve um melhor resultado referente ao consumo de memória em relação aos outros dois.

2.3 Considerações

Nesta seção foram apresentados os conceitos que estão diretamente associados ao tema do trabalho. Assim como o PON, os sistemas *fuzzy* permitem a representação do conhecimento do programa através de regras SE-ENTÃO, entretanto relacionando variáveis que representam informações imprecisas.

Um ponto a se observar é que muitos trabalhos referentes a sistemas *fuzzy* utilizaram sistemas embarcados para implementar os seus conceitos. Devido às restrições de processamento, consumo de memória e energia, é interessante buscar uma forma de otimizar estes pontos. O PON soluciona estes problemas através do uso de entidades passivas que serão processadas apenas quando necessário.

Outra característica interessante é a facilidade de mapeamento das etapas do mecanismo de inferência *fuzzy* com o processamento das entidades do PON, onde o processamento de uma ou mais entidades (*Atributos* e *Premissas*) pode representar uma mesma etapa no mecanismo de inferência ("fuzificação").

Os conceitos levantados, referentes a sistemas *fuzzy* e PON, auxiliarão no entendimento dos capítulos de desenvolvimento e testes. O capítulo seguinte apresenta informações relacionadas às alterações realizadas sobre o paradigma durante o desenvolvimento do trabalho, visando integrar os conceitos do PON aos conceitos de sistemas de inferência *fuzzy*.

Capítulo 3

Alterações realizadas sobre o PON

Neste capítulo estão descritas as alterações realizadas sobre o paradigma, e suas materializações (novo *framework* e novo compilador) para que estes possam ser utilizados no desenvolvimento de sistemas *fuzzy*. A seção 3.1 apresenta as alterações conceituais realizadas para proporcionar o suporte aos conceitos *fuzzy* no Paradigma Orientado a Notificações, que são a mudança na representação do estado lógico das entidades (sub-seção 3.1.1), uma nova representação para variáveis linguísticas (sub-seção 3.1.2), mudança no cálculo lógico do estado das *Premissas* (sub-seção 3.1.3), readequação da expressão lógica da *Condição* (sub-seção 3.1.4), readequação das *Instigações* (sub-seção 3.1.5) e criação de *Métodos fuzzy* (sub-seção 3.1.6). Também é apresentada uma discussão sobre os atributos impertinentes e as premissas exclusivas para entidades *fuzzy* (sub-seção 3.1.7). Nas seções seguintes, são apresentados detalhes das implementações computacionais desenvolvidas, especificamente apresentando a criação do *framework PON Fuzzy* (seção 3.2), e das alterações realizadas na linguagem PON (seção 3.3) e no compilador LingPON (seção 3.4) para fornecer o suporte às construções definidas na seção 3.1, de modo a fornecer suporte completo ao desenvolvimento de sistemas *fuzzy* para o PON.

3.1 Alterações na estrutura de entidades do PON

A primeira etapa do desenvolvimento do trabalho foi levantar as mudanças necessárias sobre o PON para que este disponha dos elementos necessários para a realização de inferências *fuzzy*. Com base neste levantamento foram realizadas alterações sobre a estrutura de entidades do PON apresentada na seção 2.2.2. Estas modificações são descritas nas sub-seções a seguir.

3.1.1 Mudança na representação do estado lógico das entidades

Entidades do paradigma como, por exemplo, *Premissas*, *Condições* e *Regras* possuem um estado lógico que é propagado através das notificações. Este estado lógico possui, no paradigma originalmente proposto, apenas dois valores possíveis:

$$X(x) \in \{0, 1\}$$

Onde X representa o estado lógico do elemento x, sendo que o valor 0 (zero) denota que a entidade não está ativa e 1 (um) que a mesma está ativa. Para suportar a lógica *fuzzy*, as entidades citadas devem poder representar o seu estado lógico através de seu grau de ativação.

O grau de ativação pode assumir os valores reais na faixa $[0, 1]$, podendo representar graus de pertinência, como apresentado na seção 2.1. As entidades iniciarão o processo de notificações caso o grau de ativação calculado seja diferente do valor atual. Este grau de ativação é calculado de forma diferente para cada uma das entidades e cada uma das formas é explicada nas subseções a seguir.

3.1.2 Representação de uma variável linguística

Nas materializações atuais do PON (*framework* e compilador LingPON), cada *Atributo* representa um valor incluído no FBE, que representa uma característica ou um estado, podendo ter valores dos tipos *boolean* (booleano, verdadeiro ou falso), *integer* (número inteiro), *pfloat* (número real), *char* (um único caractere), ou *string* (uma sequência de caracteres). Para que o PON passe a suportar inferências *fuzzy* foi necessário implementar também o conceito de variável linguística, que representa um identificador que pode assumir um dentre vários valores linguísticos, definidos através de conjuntos *fuzzy*. Para tal, foi criada uma extensão do conceito de *Atributo* do PON, conforme apresentado na figura 3.1.

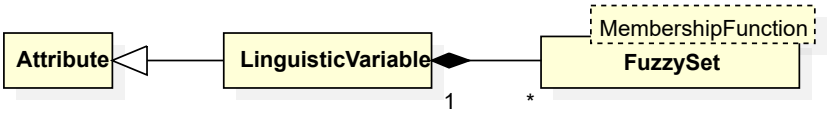


Figura 3.1: Definição do elemento "Variável linguística" no PON.

Este novo elemento, denominado *LinguisticVariable*, e derivado de *Atributo* por herança, contém uma lista de conjuntos *fuzzy* que representam os termos linguísticos possíveis da variável, sendo que cada variável poderá estar associada (por composição) a um ou mais conjuntos *fuzzy*. Os conjuntos são representados por funções de pertinência como aquelas exemplificadas na seção 2.1.1. Elas são utilizadas para calcular o grau de pertinência de um valor àquele conjunto.

As variáveis linguísticas se comportam como *Atributos*, notificando as *Premissas* associadas caso o valor ajustado seja diferente do valor atual. A partir desta notificação, as *Premissas* associadas realizam o processo de "fuzificação" do valor para o conjunto *fuzzy* aos quais elas estão associadas conforme descrito na seção 3.1.3.

3.1.3 Mudança no cálculo lógico do estado das *Premissas*

O estado da *Premissa* é determinado pelo resultado de uma operação lógica realizada sobre o *Atributo* e um valor. As operações lógicas do PON são representadas pelos operadores lógicos de comparação tradicionais ($=$, $\neq$, $>$, $<$, $\geq$ e $\leq$), sendo que estes são utilizados para comparar o valor do *Atributo* com outro valor, permitindo a avaliação do valor-verdade de cada *Premissa*. Na definição atual do paradigma as operações retornam os valores "verdadeiro" ou "falso" que definem se a *Premissa* está ou não ativa.

Devido às mudanças descritas na seção 3.1.1, o estado lógico da *Premissa* deve ser alterado, passando a ser representado pelo grau de ativação, sendo que o cálculo deste depende do tipo da *Premissa*. Os tipos de *Premissas* são os seguintes:

- *Premissa* comum (ou *crisp*): *Premissa* que faz a operação lógica entre um *Atributo* e um valor *crisp*. O resultado desta operação irá determinar o valor do grau de ativação da *Premissa*, que pode ser 'verdadeiro" ou "falso";

- *Premissa fuzzy*: *Premissa* que calcula o grau de pertinência do valor do *Atributo* no conjunto *fuzzy* associado à *Premissa*. O grau de pertinência calculado define o grau de ativação da *Premissa*.

Em outras palavras, a *Premissa*, após as alterações aqui propostas no paradigma, passa a realizar a operação de "fuzzyficação" do valor contido no *Atributo* referenciado por ela. No caso da *Premissa* comum o resultado da operação lógica define o grau de ativação da *Premissa*, onde para o resultado "verdadeiro" será atribuído o valor 1 (um) e "falso" o valor 0 (zero). Por exemplo, a *Premissa* "x é maior que 2 ?" poderia ser lida da seguinte forma: "x pertence ao conjunto de números maiores que 2 ?". A figura 3.2 ilustra as funções de pertinência consideradas para cada uma das operações lógicas suportadas pela *Premissa* comum.

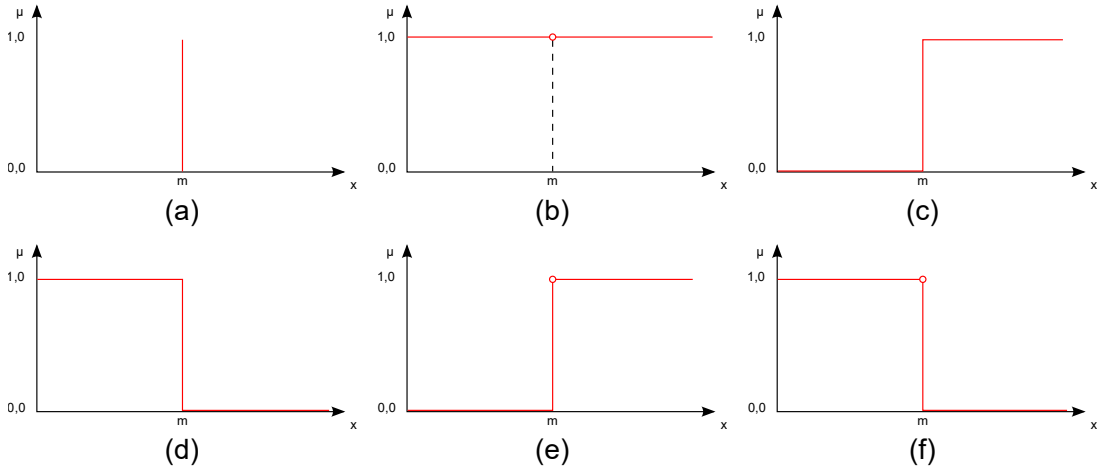


Figura 3.2: Funções de pertinência para as *Premissas*: (a) $x = m$, (b) $x \neq m$, (c) $x \geq m$, (d) $x \leq m$, (e) $x > m$ e (f) $x < m$.

No caso da *Premissa fuzzy* o grau de ativação será o resultado do cálculo do grau de pertinência do conjunto para o valor armazenado no *Atributo*. A operação de "fuzzyficação" destas *Premissas* é realizada conforme apresentado na seção 2.1.1.

3.1.4 Readequação da expressão lógica da *Condição*

A concepção original do paradigma apresentada em [Simão and Stadzisz, 2008] não permitia a criação de *Condições* com intercalamento de operadores em sua expressão causal, sendo portanto permitidas apenas *Condições* compostas por apenas um tipo de conectivo lógico (ou "E" ou "OU"). Desta forma, se houvesse a necessidade de intercalar os operadores na *Condição* de uma *Regra* seria necessário criar várias *Regras* com o mesmo consequente, o que gera um problema de replicação dos objetos que compõem as regras. Por exemplo, a seguinte regra:

```
SE (a > 1 E a < 3) OU (a = 5) ENTÃO ...
```

Deveria ser quebrada em duas regras:

```
1.  SE (a > 1 E a < 3) ENTÃO ...
2.  SE (a = 5) ENTÃO ...
```

Para solucionar este problema, em [Banaszewski, 2009] foi criada a estrutura de *Sub-Condição*, sendo que esta é uma especialização da *Condição*, que notifica as *Condições* as quais ela está associada quando o seu estado lógico é modificado. Entretanto, apesar de mais flexível, esta solução ainda não permite a criação de *Condições* mais complexas, com mais de dois níveis, que seria necessária se a *Condição* fosse, por exemplo:

```
SE (x > 10 E y<20) OU ((x < 5 E y < 10) E (z = falso)) ENTÃO ...
```

Desta forma, para este trabalho, a estrutura *Sub-Condição* foi removida e a entidade *Condição* foi modificada para suportar expressões causais complexas. A modificação consistiu na criação de um modelo que representa as expressões causais das *Condições*. Toda a complexidade adicionada pela *Sub-Condição* foi movida para este modelo. O modelo criado pode ser visualizado na figura 3.3.

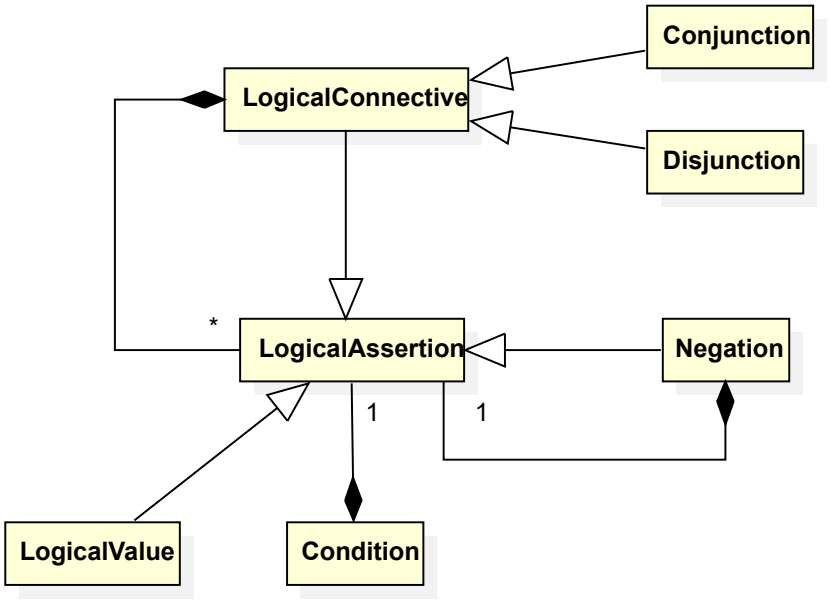


Figura 3.3: Modelo de expressão causal da *Condição*.

O modelo de *Expressão Lógica* (`LogicalAssertion`) elaborado para permitir o suporte à condições *fuzzy* no PON segue o padrão de projeto (*design pattern*) *composite* [Johnson et al., 1995]. Neste padrão de projeto, é possível criar expressões complexas a partir da "composição" de expressões mais simples. No modelo proposto, a classe `LogicalAssertion` representa uma expressão ou sub-expressão lógica. Cada *Condição* (classe `Condition` na figura 3.3) é composta de uma `LogicalAssertion`. Uma `LogicalAssertion` pode ser tanto um valor lógico (classe `LogicalValue`), quanto uma composição de diferentes valores lógicos, obtidos a partir da negação (classe `Negation`), conjunção (classe `Conjunction`) ou disjunção (classe `Disjunction`).

Neste modelo, a *Premissa* está associada ao valor lógico da expressão representado por um objeto do tipo `LogicalValue`, sendo que este valor será atualizado quando a *Premissa* notificar a *Condição*. Este objeto armazena o grau de ativação da *Premissa* a qual ele está associado e será utilizado como entrada para operações com os conectivos lógicos definidos no modelo. A notificação também irá iniciar o cálculo da expressão que irá determinar o valor do grau de ativação da *Condição*.

O cálculo é feito utilizando as definições das operações definidas por Mamdani [Mamdani, 1974] conforme descrito na seção 2.1. As classes de composição de outros valores lógicos utilizam os operadores para calcular seus graus de ativação da seguinte forma, onde $\mu_{a1}, \mu_{a2}, ...$ são os graus de ativação dos objetos `LogicalAssertion` contidos neles:

- `Conjunction` - conjunção - $\mu_{con} = min(\mu_{a1}, \mu_{a2}, ...)$;
- `Disjunction` - disjunção - $\mu_{dis} = max(\mu_{a1}, \mu_{a2}, ...)$;
- `Negation` - negação - $\mu_{neg} = 1 - \mu_a$;

A utilização das operações citadas permite que as características da lógica booleana sejam "emuladas", permitindo que *Regras* não-*fuzzy* funcionem com o mesmo modelo. Ao calcular a "tabela de ativações", conforme demonstrado na tabela 3.1, é possível verificar que os valores obtidos através do uso dos operadores são os mesmos para as mesmas operações na lógica booleana.

(a)

$min(\mu_c, \mu_l)$	**0**	**1**
0	0	0
1	0	1

(b)

$max(\mu_c, \mu_l)$	**0**	**1**
0	0	1
1	1	1

(c)

$1 - \mu_l$	μ_{neg}
0	1
1	0

Tabela 3.1: Tabela de ativações para as operações de (a) conjunção, (b) disjunção e (c) negação.

3.1.5 Readequação das *Instigações* e a criação da *Instigação fuzzy*

Com as mudanças citadas na seção 3.1.1 uma *Regra* passará a se ativar caso haja uma mudança de estado na *Condição*. Isto também reflete nas *Instigações* já que estas serão ativadas juntamente com as *Regras*.

Neste trabalho foi definido que as *Instigações* passarão a ter duas formas: a *Instigação* comum e a *Instigação fuzzy*. As *Instigações* comuns representam a atual definição de *Instigações* do paradigma e servem para manter o atual comportamento do paradigma. Estas só serão ativadas caso o grau de ativação da *Regra* seja igual a um, sendo que, caso contrário, a *Instigação* não será ativada.

A *Instigação fuzzy* será ativada a cada mudança no valor do grau de ativação da *Regra* a qual ela está associada, mesmo se o grau de ativação for zero. Esta ativação implicará na execução do *Método* ao qual ela está associada, sendo ele *fuzzy* ou não. Caso a *Instigação fuzzy* implique na execução de um *Método fuzzy*, ela deverá repassar o grau de ativação da *Regra* para que este possa realizar as operações citadas na seção 3.1.6.

3.1.6 Criação do *Método fuzzy*

O *Método fuzzy* foi criado para representar um consequente lógico da lógica *fuzzy*, associando o *Atributo* ao conjunto *fuzzy*. Ao ser chamado, o *Método* armazena a contribuição da *Regra* em uma lista associativa do *Atributo* que faz a associação entre o grau de ativação da *Regra* e o conjunto *fuzzy* definido no mesmo, permitindo assim a geração do conjunto *fuzzy* resultante da *Regra*.

Por exemplo, se existirem duas *Regras* (com graus de ativação $0,5$ e 0) com *Métodos fuzzy* que referenciam um mesmo *Atributo* (`temperatura é fria` e `temperatura é quente`) então a lista associativa para o *Atributo* `temperatura` conterá as seguintes entradas:

- Conjunto *fuzzy*: `frio` $\rightarrow 0,5$;
- Conjunto *fuzzy*: `quente` $\rightarrow 0$.

Depois de armazenar a contribuição de todas as regras, é feita a união destes conjuntos *fuzzy* para a realização da operação de "defuzificação" conforme exemplificado na seção 2.1.2. Para tal, a lista associativa é percorrida para montar o conjunto *fuzzy* resultante a partir dos conjuntos *fuzzy* e seus respectivos graus de ativação ali contidos. Caso o grau de ativação de um conjunto *fuzzy* seja proveniente de várias *Regras*, será considerado o maior grau de ativação.

Depois de gerar o conjunto *fuzzy* a partir das contribuições das *Regras* é realizada a operação de "defuzzyficação" e o resultado da operação é atribuído ao *Atributo*. Este processo de atribuição poderá então iniciar um novo ciclo de notificações.

3.1.7 *Atributos* impertinentes e *Premissas* exclusivas para entidades *fuzzy*

Os *Atributos* impertinentes, propostos por [Ronszcka, 2012], e as *Premissas* exclusivas, propostas por [Banaszewski, 2009], surgiram como ideias para otimização da execução dos programas desenvolvidos sob paradigma, cujo objetivo é reduzir a quantidade de notificações emitidas por estas entidades. Estas otimizações podiam ser aplicadas sobre o paradigma já que o resultado do cálculo lógico das entidades indicavam apenas se as *Regras* seriam ativadas e não influenciavam sobre o resultado da execução.

No caso da lógica *fuzzy*, o resultado do cálculo do estado lógico das entidades citadas irá influenciar o resultado da execução da inferência, podendo gerar um resultado errado caso alguma notificação seja omitida. Sendo assim, qualquer mudança de estado destas deverá ser notificada, mesmo que esta não ative as próximas entidades na cadeia de notificações. Logo, para que a lógica *fuzzy* seja suportada, estas funcionalidades foram desabilitadas para os *Atributos* e *Premissas fuzzy*.

3.2 Criação do *framework* PON Fuzzy

Após definir as alterações conceituais a serem feitas no paradigma, foi realizado um trabalho de adaptação do *framework* definido em [Valença, 2012, Ronszcka, 2012] e apresentado na seção 2.2.6, sendo então criada uma nova versão do *framework* com suporte ao mecanismo de inferência *fuzzy*. Isto foi feito para realizar a validação das alterações levantadas. Nas sub-seções a seguir estão descritas as alterações realizadas, que deram origem ao que doravante será denominado *Framework PON Fuzzy*.

3.2.1 Reestruturação das classes dos *Atributos* e *Premissas*

Foi realizada uma reestruturação das classes disponíveis no *framework* relacionadas ao *Atributo* para facilitar a realização de modificações sobre as mesmas. O *framework* original dispõe de diversas classes que representam especializações do *Atributo* para tratamento de tipos de dados. A figura 3.4 apresenta as classes citadas.

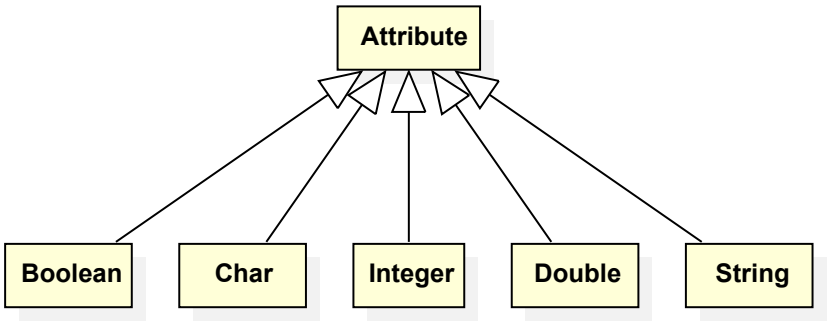


Figura 3.4: Classes derivadas de um *Atributo* PON.

Neste trabalho este modelo foi simplificado através da funcionalidade de *templates* da linguagem C++, sendo que esta é adequada para este caso tendo em vista que as classes especializadas possuem lógicas iguais, facilitando a realização de manutenção sobre o código. A figura 3.5 apresenta este novo modelo.

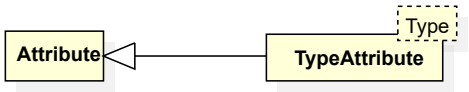


Figura 3.5: Novo modelo de classes dos *Atributos* PON.

A nova classe de tipo recebe um parâmetro de *template* que define o tipo do *Atributo*. Este parâmetro pode ser qualquer um dos tipos já suportados pelo *framework* original: `bool`, `char`, `int`, `double` e `string`. Por exemplo, para declarar um *Atributo* do tipo inteiro basta declarar uma variável do tipo `TypeAttribute<int>`. O processo de notificação se iniciará sempre que o valor atribuído for modificado pela função `setValue` da classe `TypeAttribute`.

A partir deste novo modelo de classes do *Atributo*, foi criada uma especialização da classe `TypeAttribute` para a representação da variável linguística da lógica *fuzzy*. Neste caso o parâmetro de *template* passado na classe representa o tipo do valor a ser ajustado nela. O tipo da variável pode ser apenas um tipo numérico como, por exemplo, `int` ou `double`. Na figura 3.6 é possível visualizar como a variável linguística se encaixa neste novo modelo.

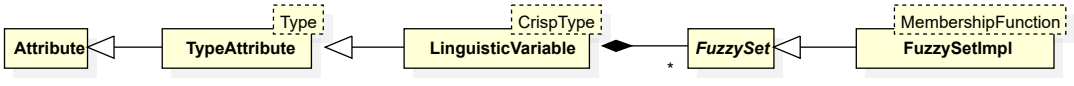


Figura 3.6: Variável linguística no novo modelo de classes dos *Atributos* PON.

A classe `LinguisticVariable` representa a variável linguística e contém os termos linguísticos representados pelos conjuntos *fuzzy* incluídos na mesma. Estes conjuntos são

definidos pela função de pertinência passada no parâmetro de *template* da classe de implementação `FuzzySetImpl`. A figura 3.7 apresenta uma interface mínima a ser implementada por classes que poderão ser passada como parâmetro da função de pertinência de um conjunto *fuzzy*.

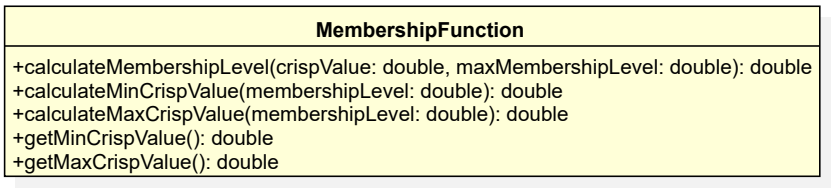


Figura 3.7: Interface mínima para uma função de pertinência de um conjunto *fuzzy*.

A implementação desta interface depende do tipo de função de pertinência utilizada. Por exemplo, para a função trapezoidal (com 4 parâmetros: $a \leq m \leq n \leq b$) apresentada na seção 2.1.1, as funções da interface deverão realizar as seguintes operações:

- `calculateMembershipLevel`: faz o cálculo do grau de pertinência do valor *crisp* passado. O valor máximo de grau será retornado caso o valor *crisp* estiver entre *m* e *n*;
- `calculateMinCrispValue`: faz o cálculo do menor valor *crisp* para o grau de pertinência passado. Para a função trapezoidal, o valor a ser retornado estará entre *a* e *m*;
- `calculateMaxCrispValue`: faz o cálculo do maior valor *crisp* para o grau de pertinência passado. Para a função trapezoidal, o valor a ser retornado estará entre *n* e *b*;
- `getMinCrispValue`: retorna o menor valor *crisp* do conjunto. Para a função trapezoidal, o valor a ser retornado será *a*;
- `getMaxCrispValue`: retorna o maior valor *crisp* do conjunto. Para a função trapezoidal, o valor a ser retornado será *b*.

Estas funções são utilizadas em várias etapas do processo de inferência. Por exemplo, a função `calculateMembershipLevel` é utilizada pela *Premissa fuzzy* para realizar a "fuzificação" do valor do *Atributo* e na realização da "defuzificação" através do cálculo da função centroide apresentada na seção 2.1.2.

Devido às mudanças realizadas sobre os *Atributos*, estas também foram realizadas sobre as *Premissas*, que passaram a ter um parâmetro de *template* indicando o tipo da mesma. A figura 3.8 apresenta o modelo de classes da *Premissa* desenvolvido neste trabalho.

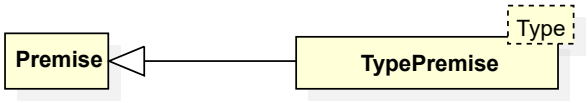


Figura 3.8: Novo modelo de classes da *Premissa*.

O desenvolvedor deverá passar o tipo da *Premissa* através do parâmetro do *template*, sendo que este deverá ser o mesmo que o tipo definido para o *Atributo* associado a esta. Por exemplo, se o *Atributo* for do tipo `double` (`TypeAttribute<double>`), então a *Premissa* deverá ser do mesmo tipo (`TypePremise<double>`).

Há o caso especial da *Premissa fuzzy* em que o tipo passado é o conjunto *fuzzy*. Neste caso, o tipo passado deverá ser a classe `FuzzySet *` para o parâmetro de *template* da classe `TypePremise` (`TypePremise<FuzzySet *>`). O valor de comparação da *Premissa fuzzy* deverá ser ajustado com um conjunto *fuzzy* para o qual será calculado o grau de pertinência do valor recebido do *Atributo*. Caso seja uma *Premissa* comum, o grau de ativação será o resultado da operação lógica definida pelo desenvolvedor na hora de instanciar a *Premissa*.

3.2.2 Elaboração de um mecanismo de notificações unificado

No *framework* original o mecanismo de notificações foi implementado através de um conjunto de classes que representam as entidades do PON, sendo que cada uma delas possui em seu código a implementação específica das notificações.

Para simplificar este mecanismo, foi proposta uma nova estrutura para o mecanismo de notificações, utilizando um único conjunto de classes para todas as entidades do paradigma, sendo que este conjunto é responsável em manter a ligação entre duas entidades do paradigma. O mecanismo é composto por duas entidades: um notificador e um notificado, sendo que estas são representada pelas classes `Notifier` e `Notified` respectivamente. A figura 3.9 apresenta o modelo lógico com as classe do notificador e do notificado.

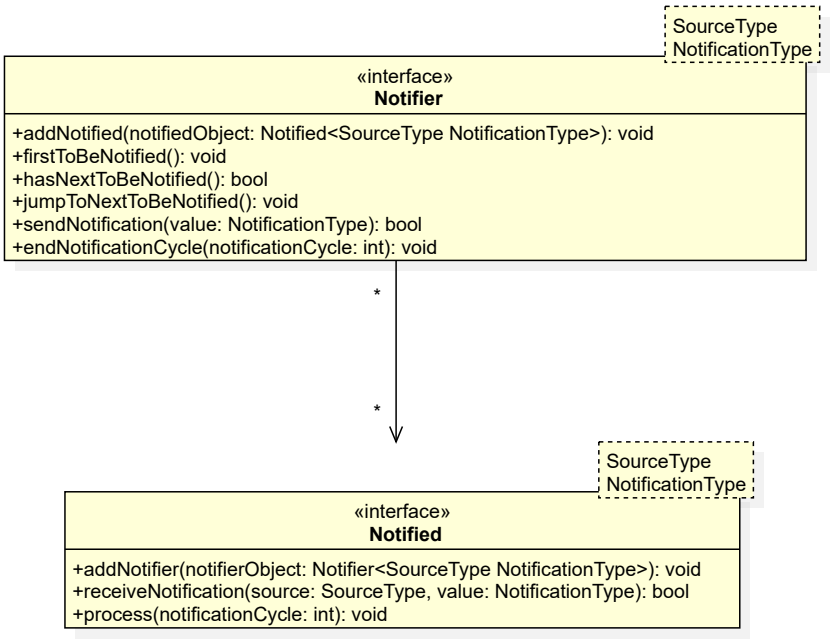


Figura 3.9: Modelo lógico do mecanismo de notificações.

Além disso, este mecanismo disponibiliza funcionalidades que permitem que as entidades do paradigma iniciem o processo de notificação de acordo com as lógicas específicas de cada entidade. Sendo assim, a lógica de notificação foi separada da lógica de conexão.

As duas classes possuem os mesmos parâmetros de *template*, sendo que estes devem receber os mesmos tipos para um conjunto notificador-notificado. Os parâmetros são os seguintes:

- `SourceType`: tipo da classe do objeto de origem;

- `NotificationType`: tipo do valor a ser notificado ao objeto.

Algumas entidades podem tanto enviar quanto receber notificações de outras entidades. Um exemplo é a *Premissa* que recebe notificações do *Atributo* e envia para a *Condição*. Neste caso, a classe da *Premissa* irá implementar as duas interfaces, sendo que as interfaces recebem os seguintes valores para os parâmetros de *template*:

- `Notified`:
  - `SourceType` a classe `Attribute`;
  - `NotificationType`: tipo do valor armazenado pelo *Atributo*;
- `Notifier`:
  - `SourceType`: a classe `Premise`;
  - `NotificationType`: tipo do grau de ativação da *Premissa*, que no caso desta implementação é o `double`.

Devido à característica do paradigma de ser paralelizado, o mecanismo foi elaborado para simular esta característica utilizando um processamento das entidades em largura. O *framework* original realiza o processamento em profundidade das notificações, conforme pode ser visualizado na figura 3.10. Os números na figura demonstram a ordem das transições de processamento das entidades. Neste exemplo, o *Atributo* **Attr1** deve notificar três *Premissas*, **Prem1**, **Prem2** e **Prem3**. Por suas vez, as *Premissas* **Prem1** e **Prem2** devem notificar a *Condição* **Cond1**, e a *Premissa* **Prem3** deve notificar a *Condição* **Cond2**. As *Condições* **Cond1** e **Cond2** notificam, respectivamente, as *Regras* **Rule1** e **Rule2**, que então, quando aprovadas, executam seus *Métodos*.

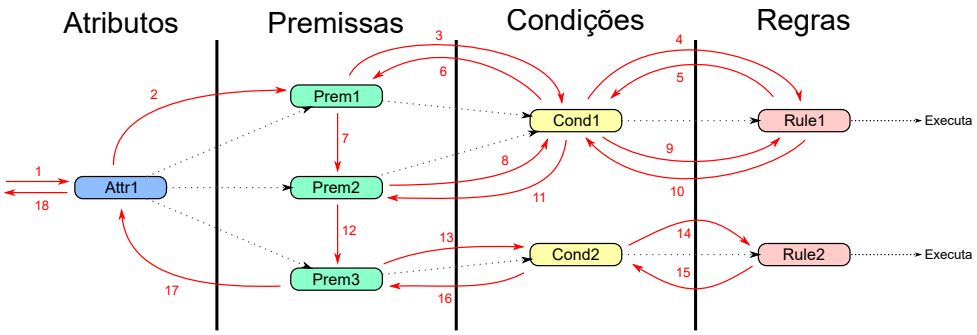


Figura 3.10: Processamento em profundidade das entidades.

A abordagem em profundidade faz com que uma *Premissa* notifique as *Condições* para que estas realizem o seu cálculo lógico antes que as outras *Premissas* estejam atualizadas, o que pode causar ativações indevidas das *Regras*. Por exemplo, caso o valor de `Attr1` seja alterado de 1 para 5, ele iniciará o ciclo de notificações notificando a *Premissa* `Prem1` (ex.: `Attr1 > 2` com estado inicial "falso"), fazendo com que o seu estado seja mudado para "verdadeiro".

Antes que a *Premissa* `Prem2` (ex.: `Attr1 < 4` com estado inicial "verdadeiro") pelo `Attr1`, a *Premissa* `Prem1` notificará a *Condição* `Cond1` (ex.: `Prem1 E Prem2`, ou

seja, `(Attr1 > 2) E (Attr1 < 4)`), fazendo com que ela calcule o seu estado lógico. Como a *Premissa* `Prem2` não teve a oportunidade de atualizar o seu estado lógico ("verdadeiro" sendo que deveria ser "falso"), o estado lógico da *Condição* será "verdadeiro", o que ocasionará a ativação incorreta da *Regra*.

Isto ocorre devido a característica do paradigma imperativo de ser sequencial, o que faz que as *Premissas* sejam processadas uma por vez. Já na abordagem em largura todas as *Premissas* são processadas antes das *Condições*, fazendo com que todas as *Premissas* sejam atualizadas antes da checagem das *Regras*. A figura 3.11 apresenta o processamento em largura das notificações.

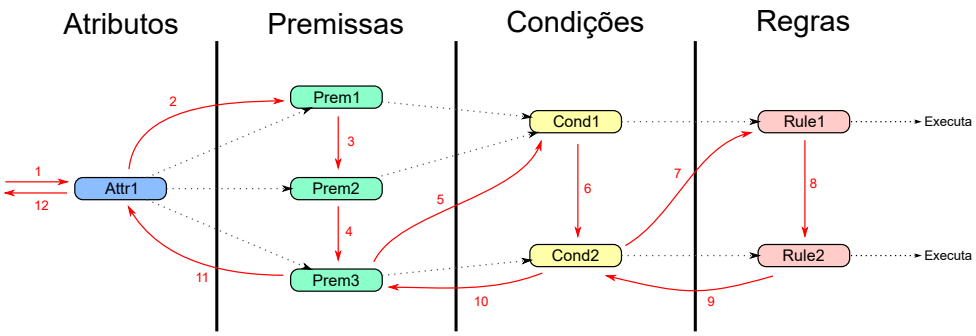


Figura 3.11: Processamento em largura das entidades.

Para realizar o processamento em largura, o mecanismo de notificação realiza duas etapas distintas:

1. Notifica as entidades para que estas atualizem o seu grau de ativação e;
2. Processa as entidades notificadas, fazendo com que as próximas entidades da cadeia de notificações realizem estas duas etapas.

No exemplo citado, ao alterar o valor de `Attr1` (de 1 para 5) serão notificadas as três *Premissas* que estão associadas a ele (`Prem1`, `Prem2` e `Prem3`) para que estas atualizem os seus respectivos estados lógicos. Isto fará com que as *Premissa* `Prem1` (`Attr1 > 2`) e `Prem2` (`Attr1 < 4`) passem a ter os estados "verdadeiro" e "falso", respectivamente. A notificação também faz com que as entidades notificadas sejam inseridas em uma fila que será processada após a finalização da notificação de todas as entidades daquela camada (neste caso, as *Premissas*).

Após a finalização das notificações, a fila é percorrida e as entidades contidas nela realizarão o processamento do estado lógico calculado anteriormente. No caso das *Premissas*, caso os seus estados lógicos sejam "verdadeiro", as *Condições* associadas serão notificadas e inseridas em uma fila para processamento posterior. No caso da *Condição* `Cond1` (`Prem1 E Prem2`) do exemplo citado, ao ser notificada, ela calculará o seu estado lógico a partir dos estados já atualizados das *Premissas* `Prem1` e `Prem2`. Depois de notificar todas as *Condições*, as que estão na fila serão processadas e, caso seus estados lógicos sejam "verdadeiro", notificarão e incluirão as *Regras* associadas na fila para serem processadas após o processamento das *Condições*.

O processamento em largura das notificações se assemelha ao mecanismo de garantia de determinismo proposto em [Simão et al., 2009] e apresentado na seção 2.2.5 deste trabalho.

A diferença entre os dois mecanismos é que no processamento em largura não há necessidade da realização da contra-notificação já que as camadas de entidades do paradigma (*Atributos*, *Premissas*, *Condições* e *Regras*) serão processadas de forma sequencial. Em outras palavras, o notificado será processado apenas após todos os elementos da mesma camada do notificante sejam processados.

No modelo lógico apresentado na figura 3.9, as entidades notificadoras deverão utilizar o método `sendNotification` para enviar uma notificação para outra entidade. No algoritmo 3.1 é possível visualizar um exemplo de lógica de notificação de uma entidade.

Algoritmo 3.1: Lógica de notificação de uma entidade.

```
1  // percorre os target que estão associados a esta entidade
2  this->firstToBeNotified ();
3
4  // enquanto tiver targets a serem checados
5  while( this->hasNextToBeNotified())
6  {
7          // notifica o target atual
8          this->sendNotification( valueToBeNotified );
9
10         // vai para o próximo target
11         this->jumpToNextToBeNotified();
12 }
```

Na entidade notificada será chamado o método `receiveNotification` com a referência da entidade notificadora e o valor que causou a notificação. Nesta chamada, a entidade notificada deverá apenas atualizar o seu estado lógico. Após notificar todas as entidades, a entidade notificadora deverá verificar se o ciclo atual de notificações para a camada de entidades foi processado. Isto é feito através da chamada do método `notificationCycleEnded`, sendo que, em caso verdadeiro, a entidade terá que chamar a função `endNotificationCycle` para oficializar a finalização e iniciar o processamento da próxima camada de entidades. Isto fará com que seja chamado o método `process` de cada uma das entidades notificadas, sendo que esta função deverá conter a lógica de notificação para a próxima camada de entidades e a verificação da finalização do ciclo.

3.2.3 Modelo de resolução de conflitos

No *framework* original é utilizado um escalonador para realizar a resolução de conflitos conforme descrito em [Banaszewski, 2009]. O modelo a ser utilizado (centralizado ou descentralizado) depende da estratégia selecionada no código (`NO_ONE` para descentralizado ou `BREADTH`, `DEPTH` ou `PRIORITY` para centralizado). Após a verificação da ativação, as *Regras* são incluídas nos escalonador para que sejam executadas. Este escalonador é instanciado externamente e passado no método construtor da classe da *Regra*.

Para o *framework* desenvolvido neste trabalho, devido ao processamento em largura das entidades, é utilizado um modelo centralizado de resolução de conflitos. Isto é necessário porque as *Ações* serão executadas apenas quando todas as *Regras* forem processadas.

A estrutura do escalonador utilizada difere da estrutura utilizada no *framework* original. O escalonador é instanciado juntamente com a *Regra* e o tipo do escalonador é definido por um parâmetro de *template* passado na classe de implementação da *Regra*.

A principal mudança do escalonador é que as *Ações* serão escalonadas em vez das *Regras*. Com as mudanças citadas na seção 3.1.1, a *Regra* irá adicionar a *Ação* no escalona-

dor para a execução sempre que o grau de ativação da mesma for modificado. A figura 3.12 apresenta o relacionamento entre o escalonador e as duas entidades.

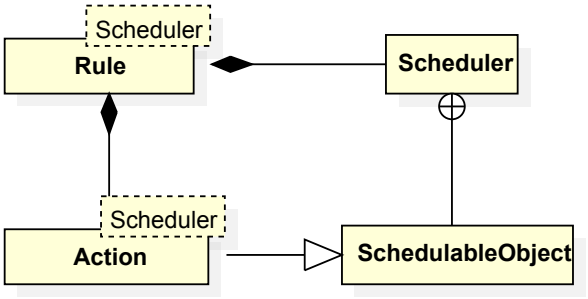


Figura 3.12: Relacionamento entre o escalonador e a *Regra* e a sua *Ação*.

A *Regra* (classe `Rule`) possui uma *Ação* (classe `Action`) que será executada quando a mesma for ativada. Ambas as classes das entidades partilham de um mesmo parâmetro de *template* (`Scheduler`) que define o tipo do escalonador a ser utilizado para organizar a execução das *Ações* de acordo com a sua política.

Para que uma classe seja um escalonador compatível com o *framework*, esta deverá possuir a declaração da classe aninhada `SchedulableObject`, a declaração de uma estrutura aninhada `Params` que armazena os parâmetros a serem passados para o escalonador e os métodos apresentados na figura 3.13.

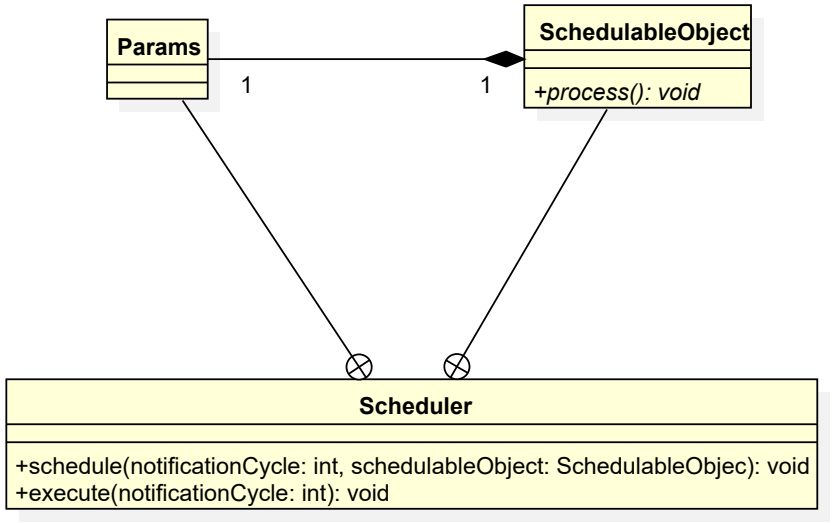


Figura 3.13: Modelo do escalonador.

Quando uma *Regra* torna-se ativa, esta executa o método `schedule` passando o ciclo de notificação da ativação e a *Ação* para ser executada. Após a finalização do processamento de todas as *Regras* ativas, o escalonador será ativado fazendo com que sejam executadas as *Ações* escalonadas de acordo com a política de escalonamento definida. Atualmente o *framework* possui duas politicas de escalonamento representadas pelas classes:

- `QueueScheduler`: escalonador do tipo FIFO (*First In, First Out*) que executa as *Ações* na ordem em que elas forem escalonadas;

- `PriorityScheduler`: escalonador que utiliza um critério de prioridade para execução das *Ações* escalonadas. As *Ações* com maior prioridade serão executadas primeiro. Caso existirem duas ou mais *Ações* com a mesma prioridade elas serão executadas na ordem de chegada.

Para que uma *Ação* seja escalonada, é necessário que esta receba o mesmo tipo de escalonador no seu parâmetro de *template* que foi atribuído no parâmetro da *Regra*. Isto se faz necessário pois a classe da *Ação*, por restrições da linguagem C++, precisa conhecer o tipo de escalonador selecionado. Além disso, ela herdará características da classe `SchedulableObject` definida dentro da classe do escalonador.

A classe da *Ação* contém um objeto da classe `Params`, sendo que esta também é definida dentro da classe o escalonador. Esta classe representa os parâmetros utilizados pelo escalonador para determinar qual será a próxima *Ação* que será executada. Por exemplo, a classe `Params` definida dentro da classe do escalonador que utiliza a política de prioridades irá conter o valor da prioridade de execução da *Ação*.

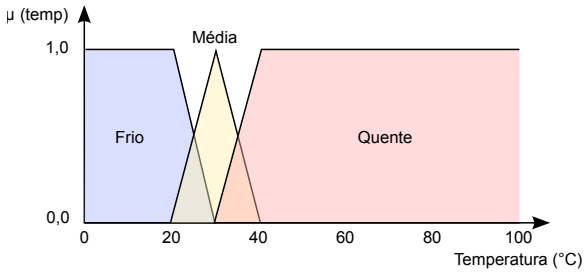
3.3 Adequação da linguagem PON

Em [Ferreira, 2015] foi definida uma linguagem de programação para desenvolvimento de aplicações utilizando o PON. Entretanto, esta não dispõe das extensões necessárias para o desenvolvimento de sistemas *fuzzy*. Sendo assim, foi necessário modificar a linguagem e seu compilador, para adicionar as estruturas necessárias para prover suporte para o desenvolvimento de sistemas utilizando conceitos *fuzzy*.

As mudanças iniciaram com a modificação da definição da gramática da linguagem para suportar as estruturas *fuzzy*. A gramática, definida em BNF (*Backus Normal Form* - Forma Normal de Backus) no trabalho citado, foi modificada para suportar as mudanças descritas nas seções a seguir. A especificação completa BNF modificada da LingPON com suporte à Lógica *Fuzzy* é apresentada no Apêndice A.

3.3.1 Declaração de variáveis linguísticas

As variáveis linguísticas foram modeladas como *Atributos* incluídos em um FBE. Para declarar uma variável linguística em um FBE na linguagem PON basta declarar um *Atributo* com o tipo `fuzzy` conforme exemplificado a seguir:




```
1 fuzzy temperatura 0.0 centroid 0.1
2 fuzzysets
3    trapezoidal frio 0.0 0.0 20.0 30.0
4    triangular media 20.0 30.0 40.0
5    trapezoidal quente 30.0 40.0 100.0 100.0
6 end_fuzzysets
```

Figura 3.14: Gráficos dos conjuntos nebulosos componentes da variável linguística “temperatura”.

Neste exemplo, a variável linguística *temperatura* é declarada através de três conjuntos *fuzzy*, um trapézio (frio, entre 0.0 e 30.0 graus, com pertinência máxima entre 0.0 e 20.0 graus), um triângulo (media, entre 20.0 e 40.0 graus, com pertinência máxima em 30.0 graus), e outro trapézio (quente , entre 30.0 e 100.0 graus, com pertinência máxima entre 40.0 e 100.0 graus), como apresentado graficamente na figura 3.14. Esta variável linguística é inicializada com ativações para a temperatura 0.0 graus, o método de "defuzificação" a ser utilizado é o de cálculo do centroide (`centroid`), com granularidade 0,1. Internamente, todas as variáveis linguísticas têm seus valores implementados como *Atributos* do tipo `double`, podendo ser utilizadas como tal. A principal diferença é que estes *Atributos* contêm uma lista de conjuntos *fuzzy* que representam os termos linguísticos da variável. Uma variável linguística possui a seguinte estrutura:

```
1 fuzzy <nome da variável> [<valor inicial>] [<método de defuzificação> [<argumentos>]]
2 fuzzysets
3     <função de pertinência> <nome do conjunto> <parâmetros>
4     ...
5 end_fuzzysets
```

Para definir uma variável linguística é necessário definir os seguintes parâmetros:

- **Nome da variável**: o nome da variável linguística. Ele deve ser único em relação aos *Atributos* contidos no FBE;
- **Valor inicial**: o valor inicial do atributo. Este parâmetro é opcional;
- **Método de "defuzzyficação"**: define o método de "defuzzyficação" a ser utilizado para atribuição de valor no *Atributo*. O método selecionado pode requerer que alguns **argumentos** sejam configurados. O método e a sua lista de **argumentos** são opcionais.

Os conjuntos *fuzzy* da variável linguística devem ser definidas na seção `fuzzysets` do código fonte. Estes conjuntos possuem os seguintes parâmetros:

- **Função de pertinência**: a função de pertinência que define o conjunto;
- **Nome do conjunto**: o nome do conjunto, sendo que este representa um termo linguístico da variável;
- **Parâmetros**: os parâmetros de definição do conjunto *fuzzy*. Estes parâmetros dependem da função de pertinência selecionada para o conjunto.

Atualmente a linguagem suporta três dos quatro tipos de funções de pertinência descritas na seção 2.1.1. Os conjuntos suportados possuem a seguinte lista de parâmetros:

- Trapezoidal (`trapezoidal`) - parâmetros: a, m, n, b
- Triangular (`triangular`) - parâmetros: a, m, b
- Singleton (`singleton`) - parâmetro: m

Outra diferença entre a variável linguística e um *Atributo* comum do PON é que o primeiro pode possuir um método de "defuzzyficação" para o cálculo de seu valor com base nos graus de ativação das regras. A linguagem PON *fuzzy* suporta dois métodos de "defuzificação":

- Centroide (`centroid`): realiza o cálculo do centroide do conjunto *fuzzy* resultante. Este método possui o parâmetro que define a granularidade do cálculo;
- Centro de massa (`masscenter`): realiza o cálculo do centro de massa do conjunto *fuzzy* resultante. Este método não possui parâmetros.

Caso o método de "defuzzyficação" não seja definido, o *Atributo* não poderá ser utilizado nos consequentes das *Regras*.

3.3.2 Declaração de *Métodos fuzzy*

Os *Métodos fuzzy* foram criados para permitir a definição dos consequentes de uma *Regra fuzzy*. Eles são declarados junto com os outros *Métodos* de um FBE. Um *Método fuzzy* é declarado utilizando a palavra-chave `fuzzy` ao lado da indicação do método conforme exemplificado a seguir:

```
1 method fuzzy temperaturaIsFrio(temperatura is frio)
```

Neste exemplo é declarado um *Método fuzzy* que pode ser utilizado no consequente de uma *Regra fuzzy*. Neste caso, a declaração deste pode ser lida da seguinte forma: "A temperatura é frio".

Eles foram elaborados de forma a manter uma estrutura semelhante aos *Métodos* PON, onde um *Atributo* recebe o resultado de uma expressão matemática. Porém, em vez do operador de atribuição, no *Método fuzzy* é utilizado o operador `is` para associar a variável linguística com o conjunto *fuzzy* do consequente. O *Método fuzzy* possui a seguinte estrutura:

```
1 method fuzzy <nome do método>(<nome da variável linguística> is <nome do conjunto fuzzy>)
```

Os *Métodos fuzzy* possuem os seguintes parâmetros:

- **Nome do método**: o nome do *Método fuzzy*. Ele deve ser único em relação aos *Métodos* contidos no FBE;
- **Nome da variável linguística**: o nome da variável linguística (*Atributo fuzzy*) do consequente da *Regra*. Esta variável deverá estar contida no mesmo FBE que o *Método fuzzy*;
- **Nome do conjunto *fuzzy***: o nome do conjunto *fuzzy* do consequente. Ele deverá estar contido no *Atributo fuzzy* referenciado no *Método*.

Vale lembrar que a definição da variável linguística deverá conter o método de "defuzificação" a ser utilizado. Caso ela não seja definida, o atributo não poderá ser utilizado em um *Método fuzzy*.

3.3.3 Declaração de *Premissas fuzzy*

Para as *Premissas fuzzy* foi criado o operador `is` que realiza a "fuzzyficação" do valor armazenado na variável linguística para o conjunto *fuzzy* referenciado pela *Premissa* conforme exemplificado a seguir:

```
1 premise nomeFbe.temperatura is frio
```

As *Premissas fuzzy* podem ser utilizadas juntamente com as *Premissas* comuns em uma *Condição* através dos conectivos lógicos definidos na linguagem. A *Premissa fuzzy* possui a seguinte estrutura:

```
premise <instância de FBE>.<nome da variável linguística> is <nome do conjunto fuzzy>
```

As *Premissas fuzzy* possuem os seguintes parâmetros:

- **Instância do FBE**: o nome da instância do FBE onde está contida a variável linguística;
- **Nome da variável linguística**: o nome da variável linguística;
- **Nome do conjunto *fuzzy***: o nome do conjunto *fuzzy* do consequente. Ele deverá estar contido no *Atributo fuzzy* referenciado na *Premissa*.

Vale lembrar que o PON permite a relação de composição (uma instância de FBE compõe outro FBE). Caso haja este tipo de relacionamento, a instância do FBE deverá conter o nome da instância do FBE-todo seguido pelo nome do *Atributo* do FBE-parte separados por . (ponto).

3.3.4 Declaração de *Instigações fuzzy*

No caso das *Instigações fuzzy* foi adicionada uma palavra-chave para identificar que a mesma é *fuzzy* conforme exemplificado a seguir:

```
instigation fuzzy nomeFbe.temperaturaIsFrio();
```

A diferença das *Instigações fuzzy* é que os *Métodos* referenciados serão executados a cada mudança no grau de ativação da *Regra*. A *Instigação fuzzy* possui a seguinte estrutura:

```
instigation fuzzy <instância de FBE>.<nome do método>();
```

As *Instigações fuzzy* possuem o mesmo conjunto de parâmetros que as *Instigações* comuns. A referência à instância do FBE contida na *Instigação* segue o mesmo formato definido para as *Premissas fuzzy*.

3.4 Compilador LingPON Fuzzy

Devido as mudanças realizadas na linguagem também foi necessário criar um novo compilador da linguagem para suportar a nova gramática. Ele foi criado com base no compilador criado em [Ferreira, 2015], onde algumas estruturas do mesmo foram reaproveitadas e readequadas para suportar as mudanças realizadas na linguagem. Nas sub-seções a seguir estão descritos os detalhes da elaboração do compilador.

3.4.1 Analisadores Léxico e Sintático da linguagem PON *Fuzzy*

Os analisadores léxico e sintático da linguagem são responsáveis por realizar a leitura e validação do código-fonte escrito na linguagem PON *Fuzzy*. Eles são responsáveis por verificar se todos os termos utilizados no código-fonte são válidos (analisador léxico), por verificar se as construções sintáticas estão corretas (analisador sintático) e verificar se os comandos estão

utilizando corretamente os conceitos (análise semântica). Além disto, deve-se também gerar o código intermediário do programa analisado.

Os analisadores léxico e sintático utilizados neste novo compilador foram reaproveitados do compilador original, sendo adaptados para suportar as novas construções de linguagem inseridas conforme descrito na seção 3.3. Para tal, foram modificadas as definições dos *tokens* a serem identificados pelo analisador léxico gerado pela ferramenta de software livre `flex` [1] para identificarem novas palavras-chaves da linguagem como, por exemplo, `fuzzy` e `fuzzysets`. O analisador sintático, gerado pela ferramenta de software livre `bison` [2], foi modificado para incluir as novas regras gramaticais da linguagem. Assim como no compilador original, o código intermediário é gerado no instante em que as entidades do paradigma são processadas no código fonte. É utilizada uma estrutura de pilha para armazenar as entidades do paradigma que estão sendo processadas a cada instante. Isto foi feito para facilitar a associação dos elementos mais internos (como *Atributos* e *Métodos*) com os elementos que os contêm (como FBEs). A figura 3.15 ilustra a forma de utilização da pilha.

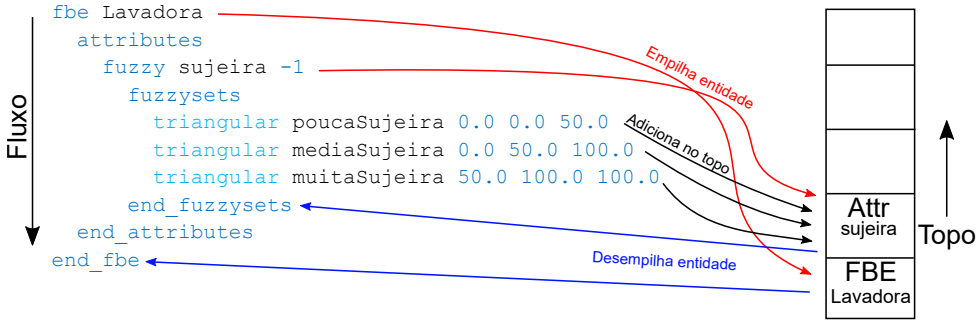


Figura 3.15: Fluxo de processamento do código fonte de um FBE.

Neste exemplo o fluxo inicia fazendo a interpretação do FBE `Lavadora` criando e empilhando o objeto que o representa. Depois o interpretador passa a interpretar o *Atributo* `sujeira`, onde o mesmo cria e empilha o objeto. Neste caso, antes de empilhar, o interpretador checa o objeto do topo da pilha (o FBE `Lavadora`) e adiciona o *Atributo* nele. Como o *Atributo* é *fuzzy*, o interpretador passa a processar os conjuntos *fuzzy* nele contidos. Para cada conjunto o interpretador irá gerar um objeto e adicionar no objeto no topo da pilha (o *Atributo* `sujeira`). Depois de finalizar a interpretação dos conjuntos, o interpretador desempilha o *Atributo* `sujeira` e o topo da pilha volta a ser o objeto do FBE `Lavadora`. Este processo será repetido caso existirem outros *Atributos* e é semelhante para os *Métodos*. Ao finalizar a interpretação do FBE, o objeto do mesmo é desempilhado e o interpretador passa para outro FBE se existir.

O analisador sintático realiza a leitura do arquivo fonte e gera o formato inicial do código intermediário. Este código conterá a definição de todas as entidades declaradas no código-fonte. Esta geração preliminar é necessária para a realização da solução das dependências entre FBEs, *Atributos* e *Métodos* contidas no código. Por exemplo, um FBE pode conter um *Atributo* cujo tipo é outro FBE. Como a interpretação é feita na ordem em que os FBEs são declarados, caso o segundo FBE não tiver sido declarado antes do primeiro, o compilador não teria como fazer a associação, ocasionando em um erro. Para evitar problemas deste tipo, as dependências

[1] Disponível em: http://flex.sourceforge.net/
[2] Disponível em: https://www.gnu.org/software/bison

são armazenadas em uma lista a medida que elas são interpretadas e resolvidas depois que toda a estrutura do código-fonte estiver montada.

Depois de resolvidas as dependências é realizada a análise semântica a fim de buscar problemas no código. Ela realiza as seguintes checagens:

- Compatibilidade de tipos (ex.: *Atributo* inteiro recebendo um valor em ponto flutuante);
- Checagem das *Premissas* (ex.: checar se o *Atributo* contém o conjunto *fuzzy* referenciado);
- Checagem dos *Atributos* (ex.: checar se o *Atributo fuzzy* contém conjuntos *fuzzy*);
- Checagem dos *Métodos* (ex.: checar se o *Atributo* contém um método de "defuzificação");

Depois da análise, o analisador sintático finaliza a montagem do código intermediário que será utilizado para a geração do código. Ele está descrito com maiores detalhes na seção a seguir.

3.4.2 Código intermediário

O analisador sintático gera um objeto que representa o código intermediário do compilador. Ele contém todos os objetos que representam as entidades do paradigma que foram identificadas no código-fonte. A estrutura escolhida para representar o código foi a de um grafo semântico abstrato (*Abstract Semantic Graph* - ASG) pois eles podem ter sub-termos compartilhados, o que é o caso das *Premissas*. O diagrama de classes do código intermediário pode ser visualizado na figura 3.16.

Para fins de facilidade de visualização, nesta figura não são apresentadas as relações que representam as dependências entre as entidades do paradigma de forma gráfica, apenas como atributos das classes. Estes estão marcados no diagrama de classes com o estereótipo que representa o tipo da referência. Nas seções a seguir estão contidos detalhes sobre o modelo definido para o código intermediário.

Referências à *Atributos* e *Métodos* do FBE

Algumas entidades do paradigma fazem referências a outras entidades, como é o caso da *Instigação* que contém a referência para o *Método* a ser executado. Estas referências podem ser de um dos dois tipos listados a seguir, sendo que os estereótipos citados indicam o atributo da classe da referência no diagrama de classes da figura 3.16:

- Referências internas do FBE, representadas pelo estereótipo `fbeRef`;
- Referências à instâncias de FBEs, representadas pelo estereótipo `instRef`.

Dada a estrutura de notificações do PON, apenas duas entidades do paradigma podem ser referenciadas por outras: os *Atributos* e *Métodos*. O tipo da entidade a ser referenciada é definido pelo parâmetro de *template* das classes que representam estas referências conforme apresentado na figura 3.17.

As referências internas do FBE são aquelas que referenciam entidades contidas no mesmo FBE como, por exemplo, a referência do *Método* PON ao *Atributo* do FBE que irá

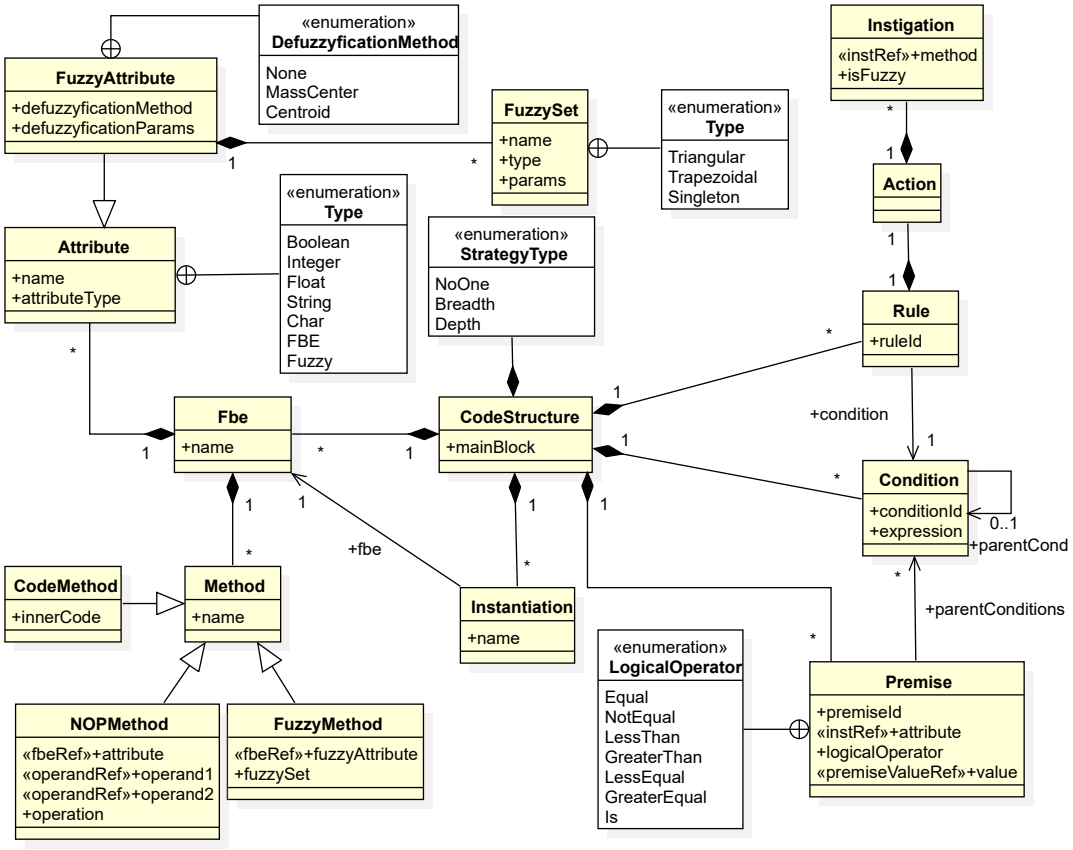


Figura 3.16: Diagrama de classes do código intermediário.

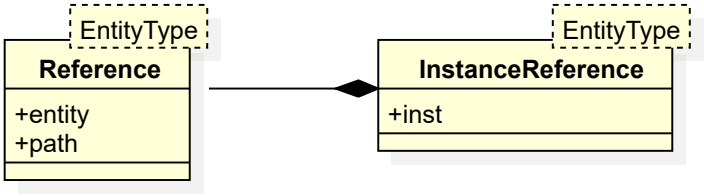


Figura 3.17: Classes que representam as referências a outras entidades.

receber o resultado da execução do mesmo. Elas contém o atributo de classe chamado `path` que é uma lista que contém o caminho a ser percorrido para acessar o *Atributo* ou *Método* que está contido em um FBE-parte que compõe o FBE. Este atributo irá conter os nomes dos *Atributos* que compõem o caminho. Caso o *Atributo* ou *Método* estiver no FBE-todo, a lista estará vazia. A figura 3.18 contém um exemplo do funcionamento desta lista.

Já as referências à instâncias de FBEs referenciam entidades contidas em um objeto instanciado. Além da referência interna ao *Atributo* ou *Método* do FBE, ela contém a instância do FBE onde eles estão armazenados. Estas são utilizadas, por exemplo, para referenciar o *Atributo* a ser checado por uma *Premissa*.

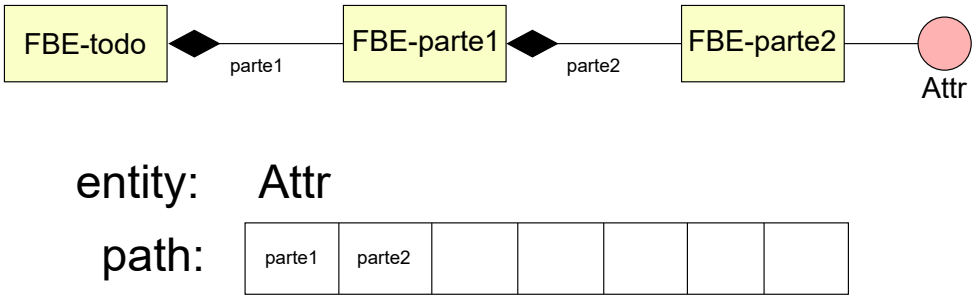


Figura 3.18: Exemplo de referência interna do FBE.

Referências opcionais

Existem casos especiais em que uma entidade poderá ou não referenciar uma outra entidade dependendo de como a primeira foi declarada. Existem dois casos na linguagem PON em que isso pode ocorrer:

- No valor a ser comparado na *Premissa* e;
- Nos operandos dos *Métodos* PON.

Em ambos os casos, o valor a ser utilizado poderá ser de um *Atributo* ou um literal. Para suportar tal funcionalidade, foram modeladas as classes apresentadas na figura 3.19, sendo que o atributo `type` determina o tipo do valor.

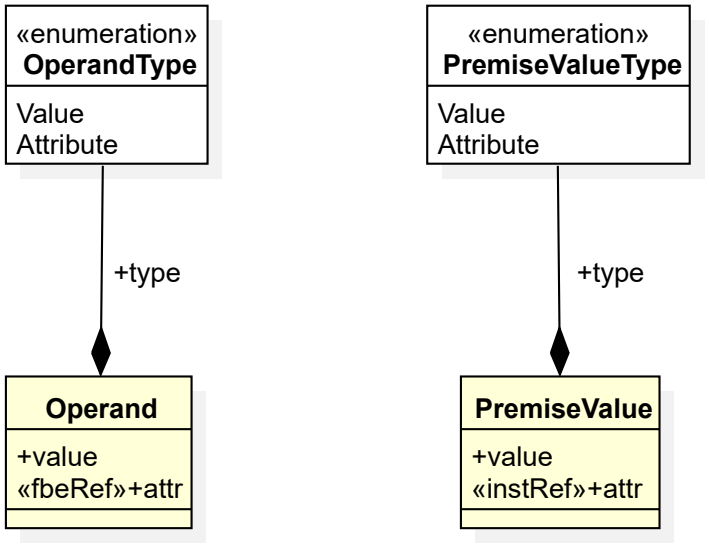


Figura 3.19: Classes dos valores a serem utilizado nas entidades citadas.

No caso das *Premissas* a referência ao *Atributo* contém a referência da instância do FBE de onde o valor será coletado. Já no caso dos operandos dos *Métodos* a referência é apenas interna do FBE. Caso o valor for um literal, este estará disponível no atributo `value`.

Modelo de expressão das *Condições*

A *Condição* conta com uma lista que representa a expressão lógica da mesma. Os elementos desta lista podem ser *Premissas* ou outras *Condições*. Ela é composta de uma cabeça, que representa o primeiro elemento, e a cauda com os elementos subsequentes. Cada elemento da cauda contém, além da referência para a entidade, o conectivo lógico indicando a operação lógica a ser realizada entre o elemento e o anterior. A figura 3.20 apresenta um exemplo de como a lista da expressão é montada a partir do código-fonte.

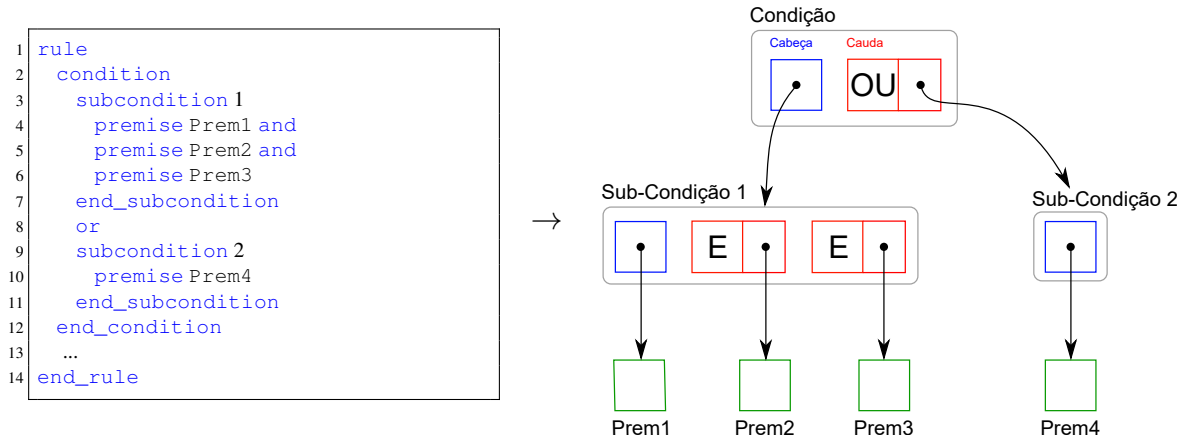


Figura 3.20: Exemplo da montagem da expressão a partir do código-fonte.

A ideia dessa estrutura é que ela seja uma forma intermediária entre a estrutura da expressão definida na seção 3.1.4 e a estrutura definida por [Banaszewski, 2009] e utilizada pela LingPON, sendo compatível com ambas as estruturas. Ela foi pensada para que, se a estrutura de *Regras* do LingPON for alterada, não seja necessário reescrever o algoritmo de geração de código do compilador.

3.4.3 Geração de código

A partir da estrutura do código intermediário gerado é feita a geração de código correspondente. O compilador é capaz de gerar código-fonte escrito na linguagem C++ com duas opções de formato de código:

- Código utilizando o *framework* PON Fuzzy descrito na seção 3.2;
- Código utilizando as bibliotecas padrões do C++.

Em ambas as opções, os programadores que desejarem acessar as instâncias dos FBEs podem acessá-las a partir das variáveis globais que o compilador gera. O nome da instância definida no código-fonte em LingPON será o nome da variável global no C++. Os FBEs são gerados como classes onde os *Atributos* são gerados como atributos privados de classe com *setters* e *getters*, e os *Métodos* não-*fuzzy* gerados como funções públicas. Os *Métodos fuzzy*, dependendo do formato do código gerado, poderão ser mapeados para funções da classe que serão chamadas durante o processo de inferência *fuzzy* do paradigma.

Os detalhes da implementação das classes muda de acordo com o formato de código gerado. Para garantir o funcionamento do código no bloco `main` do código em LingPON

independente do formato, foi definido que as classes dos FBEs devem ser compatíveis com uma interface mínima do FBE. Ela deve conter as funções públicas de `set` e `get` para todos os *Atributos* e funções sem retorno e parâmetros para os *Métodos* não-*fuzzy* do FBE. A figura 3.21 apresenta o formato que esta interface deverá ter para um exemplo em LingPON.

```
fbe Contador
    attributes
        integer contador 0
    end_attributes
    methods
        method inc(contador = contador + 1)
    end_methods
end_fbe
```

→

```
class Contador
{
public:
    int getContador();
    void setContador(int value);

    void inc();
};
```

Figura 3.21: Exemplo da interface mínima da classe para um FBE.

Após a compilação são sempre gerados dois arquivos: o *header* (`.h`) e o código-fonte com a implementação (`.cpp`). O terceiro arquivo (`main.cpp`) é gerado apenas se o bloco de `main` é declarado no código-fonte em LingPON. O *header* contém a definição das classes dos FBEs e as declarações das variáveis globais das instâncias deles. O código-fonte tem as implementações das classes dos FBEs (*Atributos* e *Métodos*), implementação das estruturas do mecanismo de inferência (*Premissas*, *Condições*, *Regras*, *Ações* e *Instigações*) e lógica de inicialização/finalização.

A forma de implementação do mecanismo de notificações muda conforme a opção de formato do código selecionada. Nas seções a seguir estão contidos os detalhes do formato do código gerado para cada uma das opções possíveis.

Geração de código para *framework* PON Fuzzy

No *framework* PON Fuzzy as entidades do paradigma são modeladas como classes. Logo, para montar a estrutura do programa basta instanciar os objetos das entidades e ajustar os relacionamentos entre elas, que é o que o código gerado pelo compilador irá realizar durante a inicialização do programa.

O programa, antes da execução da função `main`, executa a função de inicialização de todos os objetos usados no mecanismo de notificações do PON sendo que as instâncias dos FBEs são as primeiras a serem criadas. Nos construtores destes objetos serão definidas as lógicas para a criação dos objetos que representam os *Atributos* e *Métodos* do FBE.

Após a criação das instâncias dos FBEs são criados os objetos que representam o mecanismo de inferência do PON. Nesta etapa são realizadas as seguintes operações:

- Criação das *Premissas* e associação dos *Atributos* a elas;
- Criação das *Condições* e suas expressões lógicas a partir das *Premissas* previamente criadas;
- Criação das *Regras* e suas *Instigações* previamente inicializadas com as referências aos *Métodos* a serem executados.

O código gerado também conta com uma função de finalização que é executada quando o programa é finalizado. Ela realiza a limpeza dos objetos que foram alocados durante a inicialização.

Geração de código para C++ puro

Já o código C++ puro gerado pelo compilador precisa conter a implementação de todo o mecanismo de notificações para funcionar. Esta abordagem foi realizada para tentar eliminar a sobrecarga do *framework* PON Fuzzy identificada durante a primeira bateria de testes.

O mecanismo de notificações implementado, assim como o *framework* PON Fuzzy, realiza o processamento em largura das notificações, sendo que o controle é feito através do uso de *pipelines*. Ao atribuir o valor à um *Atributo*, o mecanismo irá adicionar as *Premissas* interessadas no *pipeline* das *Premissas*. Após isso, as *Premissas* no *pipeline* serão processadas, onde elas atualizam o seus respectivos estados lógicos e adicionam as *Condições* interessadas no *pipeline* das *Condições* interessadas. É tomado o cuidado para que uma entidade não seja inserida no *pipeline* caso ela já tenha sido inserida anteriormente, já que várias *Premissas* podem referenciar uma mesma *Condição*. Depois disso repete-se a lógica realizada no *pipeline* das *Premissas* sobre os *pipelines* subsequentes até que os *Métodos* sejam executados.

Os *pipelines* são implementados na forma de listas de ponteiros para funções. Durante a fase de processamento, a lista é percorrida e as funções ali inseridas são executadas. O código gerado contém três *pipelines* de entidades:

- *Pipeline* das *Premissas*;
- *Pipeline* das *Condições*;
- *Pipeline* das *Regras*.

As entidades que possuem *pipelines* são implementadas na forma de funções estáticas da linguagem. Os graus de ativação são armazenados em variáveis globais. As funções realizam o cálculo lógico da entidades, se tiver, e atualiza os graus de ativação nestas variáveis globais. Caso o grau de ativação calculado for diferente do valor anterior, a função adiciona as entidades interessadas na entidade notificante no respectivo *pipeline*. O algoritmo 3.2 contém o código da função gerado para a *Premissa* `premise prmOperationSum calculadora.attOperation == 1`.

Algoritmo 3.2: Código gerado para a *Premissa* citada.

```
1  void prmOperationSumAtualiza()
2  {
3      double newActivation =
4          ( calculadora->getAttOperation() == 1) ? 1.0 : 0.0;
5
6      if (prmOperationSumActivation != newActivation)
7      {
8          prmOperationSumActivation = newActivation;
9
10         for (auto &cond : prmOperationSumConditions())
11         {
12             // se já não estiver escalonado
13             if (! contains ( condPipeline , cond))
14                 // escalona
15                 condPipeline . push_back(cond);
16         }
17     }
18 }
```

A inicialização e finalização das estruturas do programa, assim como na opção de geração para *framework*, são executadas antes e depois da execução da função `main` respectivamente. A inicialização cria apenas as instâncias dos FBEs já que as estruturas do mecanismo de inferência são declaradas de forma estática.

3.5 Considerações

Neste capítulo foram apresentadas as mudanças realizadas sobre as entidades do paradigma para que este suporte a inferência *fuzzy*.

Com estas alterações, o PON passa a permitir o uso de *Atributos*, *Premissas*, *Condições*, *Regras*, *Instigações* e *Métodos Fuzzy*, fornecendo desta forma todo o suporte para a implementação de sistemas completos baseados em lógica *fuzzy*, e permitindo até a composição de regras com elementos *fuzzy* e não-*fuzzy* de forma transparente.

Na seção a seguir está contida a descrição dos testes que foram realizados tanto no novo *framework* em linguagem C++ aqui desenvolvido, quanto no compilador para a linguagem de programação *LingPONFuzzy*, de forma a validar as alterações realizadas.

Capítulo 4

Testes e resultados

Durante a execução deste trabalho foram realizadas duas baterias de testes. A primeira foi realizada a fim de validar as mudanças no paradigma através do uso do *framework* PON Fuzzy. A segunda foi realizada para validar a geração de código do compilador PON Fuzzy desenvolvido, e avaliar o funcionamento do código gerado na linguagem C++. Nas seções a seguir estão contidos os testes realizados nas baterias e seus respectivos resultados.

4.1 Primeira bateria de testes

Na primeira bateria de testes foi realizada uma série de testes comparativos do *framework* PON Fuzzy desenvolvido neste trabalho. As comparações foram feitas com base nos seguintes sistemas:

- Sistema desenvolvido utilizando o *framework* original definido em [Valença, 2012, Ronszcka, 2012];
- Sistema *fuzzy* desenvolvido na disciplina de Sistemas *Fuzzy* do Programa de Pós-Graduação em Computação Aplicada (PPGCA) da UTFPR.

Nas seções a seguir são descritos com mais detalhes os testes realizados.

4.1.1 Testes comparativo com o *framework* original

Estes testes foram realizados para validação das modificações que foram realizadas sobre o paradigma e sobre o *framework* citadas no decorrer deste trabalho. Os testes realizados foram os seguintes:

1. Teste de uma *Regra* com uma *Premissa* na *Condição*;
2. Teste de uma *Regra* com duas *Premissas* conectados pelos seguintes conectivos lógicos:
   - Conectivo “E”;
   - Conectivo “OU”;
3. Teste com 5 *Atributos* que notificam até 5 *Regras* com uma *Premissa* na *Condição*;

Para estes testes, foi elaborado um conjunto de valores a serem atribuídos no *Atributo* associado às *Regras*. Os valores do conjunto e a ordem de atribuição deles podem ser visualizados na tabela 4.1.

Tabela 4.1: Valores atribuídos nos testes realizados.

Ordem:	1	2	3	4	5	6	7	8
Valores:	1,0	2,0	3,0	1,0	2,0	3,0	3,5	1,0

Para o primeiro teste foi realizada a medição de tempo de execução para atribuir estes valores sobre um *Atributo* associado a uma *Premissa* da *Regra*. A medição foi feita utilizando o *High Precision Event Timer* - HPET, sendo que este é um contador de eventos de alta precisão presente nos processadores atuais [1]. A *Regra* elaborada foi a seguinte:

```
SE attr ≥ 2, ENTÃO attr2 = 2
```

Os valores citados na tabela 4.1 foram atribuídos 1000 vezes sobre o *Atributo* associado a *Regra* e o tempo medido após a atribuição do último valor em cada iteração. Na figura 4.1 é possível visualizar os valores médios de tempo de execução com os dois *frameworks* compilados com e sem as otimizações do compilador.

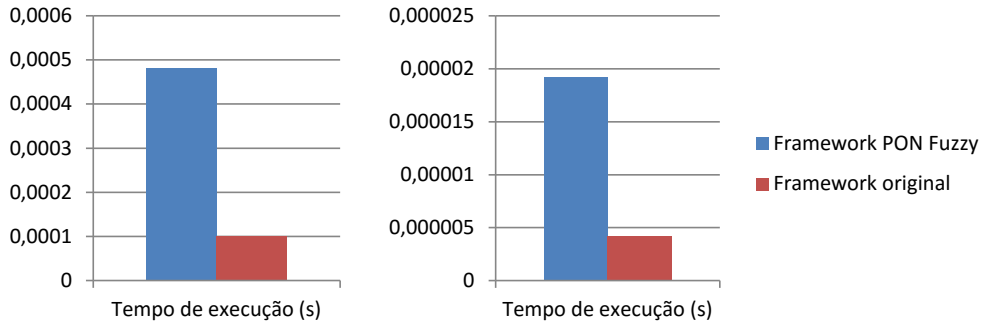


Figura 4.1: Tempo de execução do primeiro teste com as otimizações do compilador desativadas (esq.) e ativadas (dir.).

Com as otimizações do compilador desativadas o tempo médio de execução foi de 0,4824 milissegundos para o *framework PON Fuzzy* aqui desenvolvido e 0,1009 para o *framework* original. Com as otimizações ativadas os tempos foram 0,0192 e 0,0042 respectivamente. A proporção de tempo entre os dois *framework* foi muito semelhante com e sem as otimizações do compilador, sendo que os valores foram 4,777:1 e 4,546:1 respectivamente. Esta proporção de tempo ocorre pois o processamento em largura das notificações requer que o envio das notificações seja feito de forma separada do processamento delas, diferentemente do processamento em profundidade que realiza o processamento juntamente com a notificação.

O segundo teste foi elaborado para testar os operadores lógicos das *Condições*. Para tal, foram elaboradas duas *Regras* que foram testadas sobre o conjunto de dados contidos na

[1] O "temporizador de eventos de alta precisão" (HPET) é um temporizador de hardware usado em computadores pessoais. Ele foi desenvolvido em conjunto pela AMD e Microsoft e foi incorporada em chipsets de PC desde 2005. Este dispositivo faz a contagem de ciclos de *clock* que se passaram desde a inicialização do computador. Maiores detalhes podem ser encontrados no site: http://wiki.osdev.org/HPET.

tabela 4.1. Cada *Regra* foi testada de forma individual, e foi analisada a quantidade de vezes que o *Método* associada a cada uma delas é executado. Isto foi feito para verificar se ambos os *frameworks* estavam se comportando de forma adequada. As *Regras* elaboradas foram as seguintes:

```
1.  SE attr ≤ 1,5 OU attr ≥ 2,5 , ENTÃO contador++
2.  SE attr ≤ 1,5 E attr ≥ 2,5 , ENTÃO contador++
```

Antes de executar os testes foi feito o cálculo da quantidade de vezes que cada uma das *Regras* deveria se ativar. O resultado do cálculo foi o seguinte para as *Regras*:

- Primeira *Regra*: 38001 vezes;
- Segunda *Regra*: 0 vezes.

Ao executar os testes sobre a primeira *Regra* foi verificado que o *framework PON Fuzzy* desenvolvido ativou a *Regra* uma quantidade de vezes diferente do *framework* original. Após fazer o cálculo da quantidade de vezes que a *Regra* deveria ser ativa, foi verificado que o *framework* original ativou 22000 vezes a primeira *Regra* enquanto o *framework* desenvolvido ativou as 38001 vezes.

Logo em seguida foram feitos os testes sobre a segunda *Regra*, sendo que, devido ao seu conectivo lógico, ela não deveria ser ativada. O *framework PON Fuzzy* não ativou a *Regra*. Porém o *framework* original ativou a *Regra* durante a transição entre as atribuições 3 e 4, e 7 e 8, resultando em 16000 ativações. Logo, foi considerado que como os dois *frameworks* não se comportam de forma igual os tempos medidos não poderiam ser comparados para este caso.

O terceiro teste foi realizado para verificar o comportamento dos *frameworks* para um conjunto maior de *Regras*. Para tal, foram criados cinco *Atributos* sendo que cada um deles está associado a um número entre um e cinco de *Regras*. Todas as *Regras* são iguais mas não há compartilhamento de objetos entre elas, ou seja, cada *Regra* possui a sua *Premissa* e *Condição*. As *Regras* são iguais à *Regra* elaborada para o primeiro teste. Isto foi feito para que seja possível testar o pior caso, que é o de todas as *Regras* sendo ativadas.

Este teste foi executado sobre o conjunto de dados contidos na tabela 4.1 aplicados sobre cada um dos *Atributos*. A medição de tempo foi feita 1000 vezes e o tempo medido após a atribuição do último valor. Na figura 4.2 é possível visualizar os valores médios de tempo de execução com os dois *frameworks* compilados com e sem as otimizações do compilador.

Os tempos médios calculados para este teste podem ser visualizados na tabela 4.2. Nela é possível verificar que os tempos de ambos os *frameworks* apresentam o mesmo comportamento em relação a complexidade de tempo, com ambos apresentando um crescimento linear do tempo de processamento.

Na tabela também é possível ver que a diferença de desempenho para os *frameworks* sem as otimizações do compilador não varia significativamente a medida que a quantidade de *Regras* aumentava. Entretanto, com as otimizações ativadas, a diferença diminuía conforme a quantidade de *Regras* aumentava. A tabela 4.3 demonstra as proporções de tempo calculadas no teste.

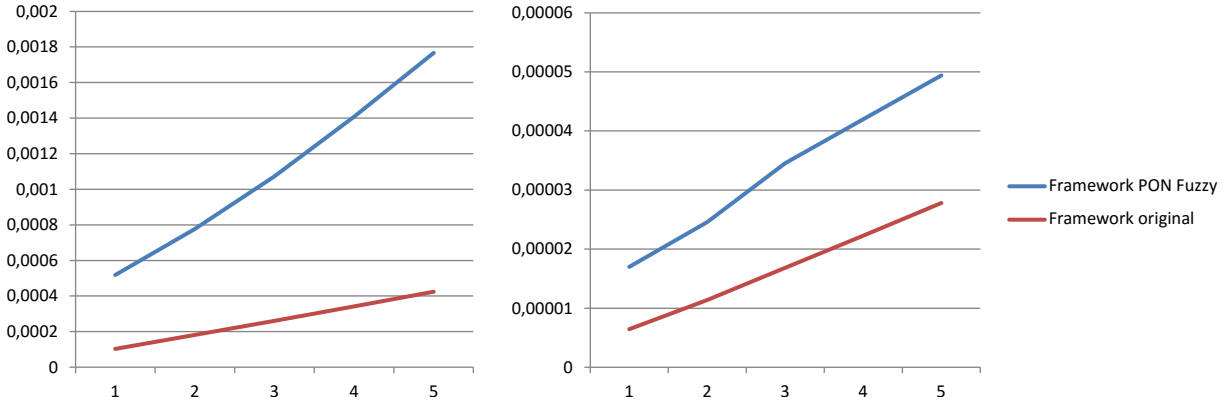


Figura 4.2: Tempo de execução em milissegundos do terceiro testes com as otimizações do compilador desativadas (esq.) e ativadas (dir.).

Tabela 4.2: Tempos médios em milissegundos calculados no terceiro teste.

Qtde. de *Regras* ativadas:	1	2	3	4	5
Framework original					
Tempo (s/ otimizações):	0,1024	0,1815	0,2611	0,3414	0,4241
Diferença:	-	0,0791	0,0795	0,0803	0,0827
Tempo (c/ otimizações):	0,0063	0,0113	0,0167	0,0222	0,0278
Diferença:	-	0,0049	0,0054	0,0055	0,0055
Framework PON Fuzzy					
Tempo (s/ otimizações):	0,5180	0,7772	1,0729	1,4082	1,7665
Diferença:	-	0,2591	0,2956	0,3353	0,3582
Tempo (c/ otimizações):	0,0168	0,0242	0,0337	0,0407	0,0489
Diferença:	-	0,0074	0,0094	0,0070	0,0082

4.1.2 Testes comparativos com o sistema *fuzzy*

Para os testes da funcionalidade da lógica *fuzzy* do *framework* foi realizada a comparação com um sistema desenvolvido na disciplina de Sistemas *Fuzzy* ofertada pela Prof. Dra. Myriam Regattieri Delgado para o PPGCA. O sistema consiste em um controlador de máquina de lavar que calcula o tempo de lavagem de acordo com a quantidade de sujeira e o tamanho da mancha contida na roupa. Sendo assim, as variáveis linguísticas do sistema são as seguintes:

- Grau de sujeira da roupa (variável de entrada);
- Manchas presentes na roupa (variável de entrada);
- Tempo de lavagem da máquina (variável de saída).

Também foram definidos os termos linguísticos que compõem cada variável. Os termos e os conjuntos *fuzzy* associados a ele são os seguintes:

Tabela 4.3: Proporção de tempo de processamento entre os *frameworks*.

Qtde. de *Regras* ativadas:	1	2	3	4	5
Proporção (s/ otimizações):	5,05:1	4,28:1	4,10:1	4,12:1	4,16:1
Proporção (c/ otimizações):	2,66:1	2,14:1	2,01:1	1,83:1	1,75:1

- Grau de sujeira da roupa:
  - **Pequena (PS)** - triangular
    A = 0,0; M = 0,0; B = 50,0
  - **Média (MS)** - triangular
    A = 0,0; M = 50,0; B = 100,0
  - **Grande (GS)** - triangular
    A = 50,0; M = 100,0; B = 100,0
- Manchas presentes na roupa:
  - **Sem (SM)** - triangular
    A = 0,0; M = 0,0; B = 50,0
  - **Média (MM)** - triangular
    A = 0,0; M = 50,0; B = 100,0
  - **Grande (GM)** - triangular
    A = 50,0; M = 100,0; B = 100,0
- Tempo de lavagem da máquina:
  - **Muito curto (MC)** - triangular
    A = 0,0; M = 0,0; B = 10,0
  - **Curto (C)** - triangular
    A = 0,0; M = 10,0; B = 25,0
  - **Médio (M)** - triangular
    A = 10,0; M = 25,0; B = 40,0
  - **Longo (L)** - triangular
    A = 25,0; M = 40,0; B = 60,0
  - **Muito longo (ML)** - triangular
    A = 40,0; M = 60,0; B = 60,0

A figura 4.3 ilustra os conjuntos *fuzzy* para cada uma das variáveis linguísticas definidas anteriormente.

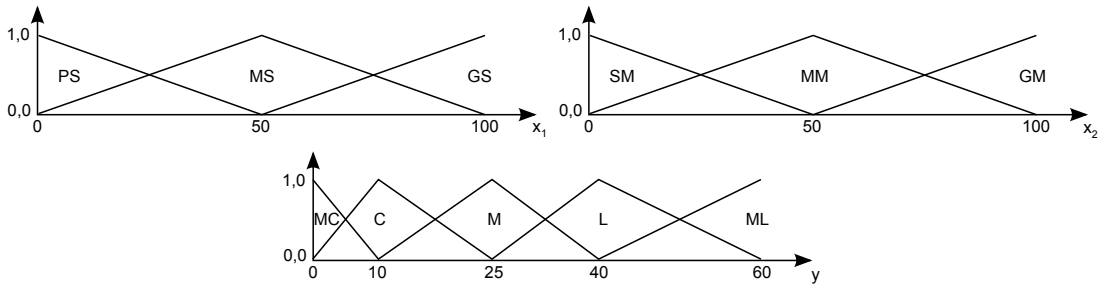


Figura 4.3: Os conjuntos *fuzzy* para as variáveis de sujeira (esq.), mancha (dir.) e tempo de lavagem (abaixo).

Tabela 4.4: Base de *Regras* do sistema da máquina de lavar.

Sujeira \Mancha	**SM**	**MM**	**GM**
PS	MC	M	L
MS	C	M	L
GS	M	L	ML

A base de *Regras* envolvendo as variáveis de entrada (grau de sujeira e manchas) e a variável de saída (tempo de lavagem) conta com nove *Regras* que foram definidas na disciplina, e podem ser visualizadas na tabela 4.4.

No sistema *fuzzy* desenvolvido utilizando o *framework PON Fuzzy* as *Regras* também incrementavam o valor de uma variável global que indica quantas *Regras* foram ativadas durante a execução do programa.

Para a "defuzzyficação" foi utilizada a função de centroide sobre o conjunto *fuzzy* resultante, sendo que em ambos os sistemas a granularidade da função foi ajustada em 0,1.

Para os testes foi utilizado o mesmo caso que havia sido definido para a disciplina de Sistemas *Fuzzy*, o que consistia em atribuir valores gerados e comparar os resultados com a implementação do mesmo sistema em MatLab. O sistema *fuzzy*, que foi desenvolvido em linguagem C++, apresentou os mesmos resultados que o sistema desenvolvido no Matlab. Os valores utilizados foram gerados através do pseudo-código exibido no algoritmo 4.1.

Algoritmo 4.1: Pseudo-código utilizado no teste da máquina de lavar.

```
1  para i = 0 até 100 incrementando de 10
2          para j = 0 até 100 incrementando de 10
3                  sujeira = i
4                  mancha = j
5
6                  realiza inferência
7
8                  escreve valor de tempo em arquivo
9          fim
10 fim
```

Em ambos os sistemas foram feitas medições de tempo de execução entre a atribuição do primeiro valor e a finalização da inferência. Para os testes será considerado o tempo total para a execução de todas as inferências, ou seja, a somatória de todos os tempos medidos.

Durante os testes ambos os sistemas obtiveram os mesmos resultados para cada valor de entrada apresentado. Após esta verificação, foi feita análise da quantidade de ativações das *Regras*. O pseudo-código apresentado deve executar 121 vezes o processo de inferência, o que significa que o sistema *fuzzy* da disciplina irá analisar as 9 *Regras* esta quantidade de vezes, resultando em 1089 verificações. Entretanto, no sistema desenvolvido usando o *framework PON Fuzzy*, a execução deste código resultou na ativação de 277 *Regras*, resultado semelhante aos apresentados em [de Souza et al., 2009]. Apesar deste resultado, o tempo de execução do sistema da disciplina foi de 0,366214 segundos enquanto o sistema com o *framework* foi de 1,12508, resultando em uma proporção de tempo de execução de 3,072:1.

4.2 Segunda bateria de testes

Após a realização da primeira bateria de testes foram realizados novos avanços no desenvolvimento do trabalho. Neste tempo foi elaborada a linguagem PON Fuzzy e seu respectivo

compilador, além de correções e adequações no *framework* PON Fuzzy. Com estas mudanças foi realizada uma nova bateria de testes para validar o trabalho desenvolvido. Os casos de testes nesta bateria foram elaborados utilizando a linguagem PON Fuzzy. Os sistemas testados foram os seguintes:

- Sistema *fuzzy* da máquina de lavar utilizado na bateria de teste anterior;
- Sistema *fuzzy* de controle do hexacóptero definido em [Koslosky et al., 2015].

Nas seções a seguir são descritos em mais detalhes os testes realizados sobre estes sistemas.

4.2.1 Reexecução dos testes comparativos com o sistema *fuzzy* da máquina de lavar

Para ser um dos testes da nova linguagem, o sistema da máquina de lavar utilizado na bateria de testes anterior foi reescrito utilizando a linguagem desenvolvida. Logo, julgou-se interessante realizar novamente os testes sobre este sistema utilizando os mesmos parâmetros e metodologia utilizada no teste anterior.

Desta vez foram utilizados três versões do sistema para comparação:

- O sistema *fuzzy* sem a utilização do PON;
- O sistema *fuzzy* gerado pelo compilador para o *framework PON Fuzzy*;
- O sistema *fuzzy* gerado pelo compilador para C++ utilizando apenas os recursos padrões da linguagem.

Na bateria anterior os testes foram realizado apenas utilizando programas compilados sem que as otimizações estivessem ativas. Para esta bateria também foram considerados os tempos médios dos programas compilados com otimizações para velocidade.

Assim como na bateria anterior, os três sistemas obtiveram os mesmos valores de saída para cada um dos valores de entrada apresentados. A tabela 4.5 apresenta os tempos médios de execução de cada um dos programas utilizado no teste.

Tabela 4.5: Tempos médios de execução do sistema da máquina de lavar em segundos.

	S/ otimizações	**C/ otimizações**
Fuzzy	0,387872	0,00369194
PON Framework	2,01338	0,0136884
PON C++	0,851679	0,00785601

Conforme demonstrado na tabela houve uma piora no tempo de execução do sistema na versão gerada para o *framework PON Fuzzy* em relação a bateria de testes anterior enquanto não houve uma piora significativa no tempo do sistema *fuzzy* da disciplina. Já a versão gerada para C++ obteve um tempo de execução menor que a versão para o *framework*, apesar de ainda ser maior que o tempo de execução em comparação com o sistema *fuzzy* da disciplina. A quantidade de *Regras* ativadas nos dois sistemas *PON Fuzzy* foi a mesma da quantidade alcançada na bateria de testes anterior (era para se ativar 1089 vezes mas se ativaram 277 vezes).

4.2.2 Teste comparativos com o sistema *fuzzy* de controle do hexacóptero

Para esta bateria de testes foi incluído mais um caso de teste: o sistema de controle do hexacóptero definido em [Koslosky et al., 2015] e descrito na seção 2.1.4. O teste consistiu em escrever o controlador *fuzzy* que controla a rotação do hexacóptero no eixo X em LingPON Fuzzy e realizar a comparação dos valores gerados e de desempenho com o controlador original. Para este teste foram utilizadas as mesmas versões utilizadas no teste com o sistema da máquina de lavar.

O controlador *fuzzy* citado conta com duas variáveis de entrada: o erro do valor do ângulo de rotação no eixo X em relação ao *setpoint* e a aceleração do hexacóptero no eixo Y. Na figura 4.4 é possível visualizar os conjuntos *fuzzy* que compõem as variáveis de entrada.

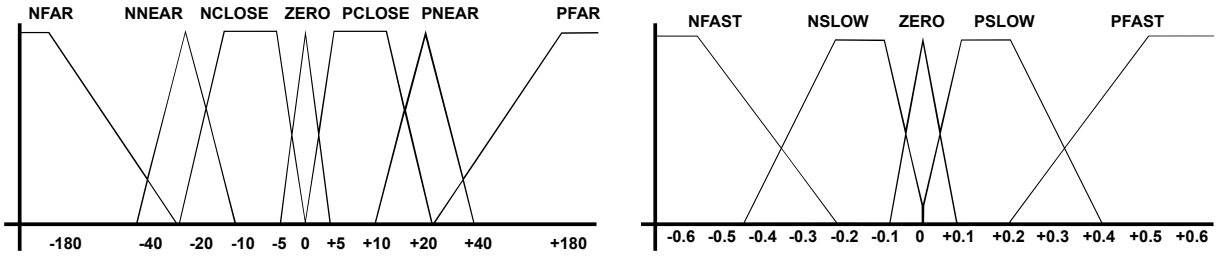


Figura 4.4: Conjuntos *fuzzy* da variável do erro do ângulo de rotação no eixo X (esq.) e da variável de aceleração no eixo Y (dir.) [Koslosky et al., 2015].

A variável de saída do controlador irá armazenar o valor de ajuste do ângulo de rotação no eixo X. O sistema de controle do hexacóptero irá utilizar este valor para aumentar ou diminuir o ângulo de rotação. A figura 4.5 ilustra os conjuntos *fuzzy* da variável de saída.

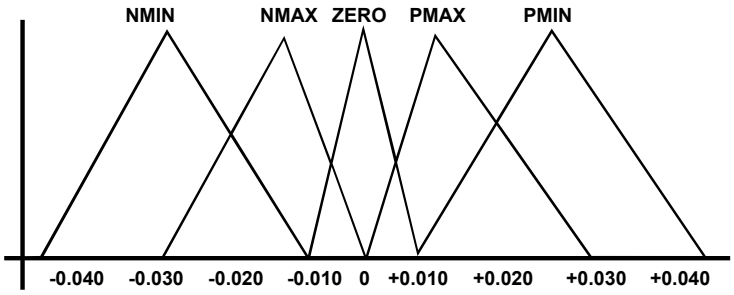


Figura 4.5: Conjuntos *fuzzy* da variável do valor de ajuste do ângulo de rotação no eixo X [Koslosky et al., 2015].

A base de regras do controlador envolvendo as variáveis de entrada com a variável de saída conta com 35 regras. Estas podem ser visualizadas na tabela 4.6.

O conjunto de dados de entrada foi elaborado de acordo com os limites definidos para as variáveis de entrada. Foram testados todos os valores dentro destes limites, onde os valores das variáveis foram incrementados de 0,05 a cada iteração.

Durante os testes foi verificado que as duas versões feitas em LingPON Fuzzy (*framework* e C++ puro) calcularam os mesmos valores. Porém houve diferenças em relação aos valores calculados pelo controlador *fuzzy* original. A figura 4.6 apresenta os valores de saída para o conjunto de valores de entrada e um gráfico mostrando a diferença entre os valores calculados.

Tabela 4.6: Base de *Regras* do controlador da rotação no eixo X [Koslosky et al., 2015].

Ângulo / Aceler.	NFAR	NNEAR	NCLOSE	ZERO	PCLOSE	PNEAR	PFAR
NFAST	PMAX	PMIN	PMIN	NMIN	NMIN	NMIN	NMAX
NSLOW	PMAX	PMIN	PMIN	ZERO	NMIN	NMIN	NMAX
ZERO	PMAX	PMIN	PMIN	ZERO	NMIN	NMIN	NMAX
PSLOW	PMAX	PMIN	PMIN	ZERO	NMIN	NMIN	NMAX
PFAST	PMAX	PMIN	PMIN	PMIN	NMIN	NMIN	NMAX

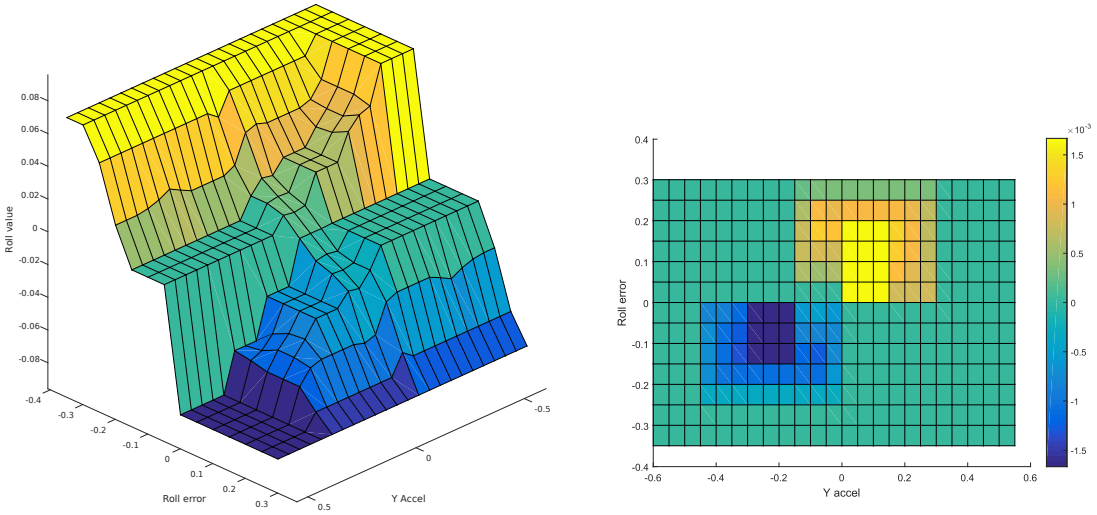


Figura 4.6: Saída do controlador da rotação no eixo X (esq.) e diferença entre as saídas das versões dos controladores (dir.).

Apesar da diferença, o resultado foi considerado bom pois a diferença foi pequena conforme visualizado na figura citada. Ela foi causada por arredondamentos ocasionados pela diferença no tipo de dados utilizado em cada implementação (os sistemas em LingPON Fuzzy utilizaram o tipo `double` enquanto o controlador original utiliza `float`, sendo que este último é mais impreciso que o primeiro). Em termos de tempo de execução, os resultados tiveram um comportamento esperado dado os resultados dos testes anteriores. A tabela 4.7 contém os tempos médios de execução dos testes para cada uma das versões testadas.

Tabela 4.7: Tempos médios de execução do controlador do hexacóptero em segundos.

	S/ otimizações	**C/ otimizações**
Fuzzy	0,0004494643	0,0001193158
PON Framework	0,0088071178	0,0069502413
PON C++	0,0025351361	0,0004610116

Apesar do número maior de avaliações das regras, o controlador *fuzzy* obteve um tempo de execução melhor em relação aos controladores escritos em LingPON Fuzzy. A versão utilizando o *framework* PON Fuzzy apresentou um tempo de execução 19,6 vezes maior sem as otimizações e 58,2 vezes maior com as otimizações que o sistema *fuzzy* original. Já a versão em

C++ puro apresentou um tempo de execução 5,6 vezes maior sem as otimizações e 3,8 vezes com as otimizações.

Capítulo 5

Conclusões e trabalhos futuros

De maneira geral, o objetivo deste trabalho, definido na seção 1.3, de adaptar o PON para suporte ao desenvolvimento de sistemas *fuzzy* foi alcançado. Os objetivos específicos definidos na mesma seção também foram alcançados.

O primeiro objetivo específico foi alcançado através da definição das modificações realizadas sobre o PON descritas na seção 3.1. As modificações citadas neste trabalho permitiram que o paradigma pudesse ser utilizado para o desenvolvimento de sistemas *fuzzy*. Além disso, estas modificações permitem que sejam desenvolvidos sistemas mistos, onde *Premissas crisp* e *Premissas fuzzy* podem compor uma mesma *Regra* através dos conectivos lógicos definidos por Mamdani.

Para realizar o segundo objetivo específico do trabalho, que é validar as modificações do PON, foi realizada a etapa de validação destas modificações através do desenvolvimento e testes com alguns sistemas de inferência *fuzzy*. Para tal, o *framework* do PON, definido em [Valença, 2012, Ronszcka, 2012], e a linguagem e seu compilador PON, definidos em [Ferreira, 2015], foram modificados para suportar as mudanças propostas neste trabalho. Como as duas materializações foram adaptadas em momentos diferentes, os testes foram divididos em duas baterias distintas.

Durante a primeira bateria de testes, criada para testar o *framework* PON Fuzzy, foi verificado que houve uma redução significativa na quantidade de regras avaliadas, o que resultaria em uma redução na computação necessária para a execução do caso de teste citado. Isto se deve ao mecanismo de inferência do paradigma, onde o processamento ocorre apenas quando houver uma mudança de estado no sistema. Entretanto, foi verificado que, apesar da quantidade menor de regras *fuzzy* verificadas, o tempo de execução do sistema desenvolvido com o *framework PON Fuzzy* foi aproximadamente três vezes maior que o do sistema desenvolvido diretamente em linguagem C++. Isto ocorreu pois o processamento em largura das entidades, necessário para a correta avaliação das regras *fuzzy*, gera um *overhead* de trabalho ao realizar as notificações das entidades em duas passadas: a primeira notificando as entidades propriamente ditas, e a segunda realizando o processamento das entidades notificadas.

Para o terceiro objetivo foi feita a adaptação da linguagem PON para incluir os conceitos e estruturas necessárias para a realização de inferência *fuzzy*. Esta nova linguagem permite que desenvolvedores criem sistemas *fuzzy* mais facilmente, o que auxiliou na criação de novos casos de teste. Além disso, este nível de abstração permitiu que o mecanismo de notificações do PON Fuzzy fosse otimizado sem adicionar complexidade no código-fonte do sistema (tornando mais simples adicionar instruções específicas para geração de *hardware* reconfigurável,

modificar o conjunto de regras, entre outras vantagens em relação às abordagens que não são baseadas em PON).

Com a criação da linguagem foram estudadas formas para que um compilador possa gerar um código PON mais otimizado que aquele desenvolvido utilizando o *framework* PON Fuzzy. Para tal foi criado um compilador que, além de gerar código para o *framework*, gera também código em linguagem C++ pura.

A partir deste ponto foi realizada uma nova bateria de testes com o código gerado pelo compilador. Nos testes realizados verificou-se que o código em C++ puro possui um desempenho superior ao código utilizando o *framework* PON Fuzzy. Porém o código C++ puro continua sendo mais lento que sistemas desenvolvido utilizando bibliotecas de lógica *fuzzy* convencionais, indicando que o código gerado deve ser otimizado mais ainda pelo compilador.

Ainda assim, a utilização do PON para a realização de inferência *fuzzy* se mostra promissora graças às propriedades do paradigma orientado a notificações, onde os elementos que compõem o sistema são processados apenas quando necessário, economizando recursos de *hardware*. Além disso, o desacoplamento das entidades facilita a distribuição do processamento entre diversos processadores em paralelo ou até mesmo entre diversos computadores em uma rede local, por exemplo.

Por fim, com as duas baterias de testes foi possível alcançar o quarto objetivo específico do trabalho, que é a realização de testes comparativos, verificando a factibilidade da adequação do PON para o desenvolvimento de sistemas que possuem regras de inferência *fuzzy*.

Sendo assim, os objetivos almejados para este trabalho foram alcançados em sua totalidade. Os avanços realizados por este trabalho permitirão que desenvolvedores criem sistemas *fuzzy* de forma mais fácil podendo até criar regras com elementos *fuzzy* e não-*fuzzy* de forma transparente. Apesar dos problemas de desempenho encontrados durante os testes, os resultados se mostraram promissores já que a quantidade de regras avaliadas foi reduzida consideravelmente. Com a evolução e uso da linguagem PON Fuzzy os sistemas desenvolvidos poderão se aproveitar das vantagens do PON como, por exemplo, a economia de recursos computacionais, reduzindo assim os custos de energia e de *hardware*.

5.1 Trabalhos futuros

Durante o desenvolvimento do trabalho foram identificados alguns trabalhos futuros que poderiam dar continuidade ao que foi aqui desenvolvido. Estes trabalhos são:

- Geração de código mais otimizado: os testes revelaram que, apesar do menor número de regras *fuzzy* avaliadas quando se utiliza o PON, o código gerado pelo compilador LingPON Fuzzy é mais lento que códigos equivalentes utilizando bibliotecas *fuzzy* convencionais. Sendo assim, é interessante realizar um estudo para otimizar mais ainda o código gerado pelo compilador;

- Geração de código para ambientes multiprocessados: o compilador LingPON Fuzzy gera código para ambientes monoprocessados, não se aproveitando das propriedades de desacoplamento do paradigma. Uma versão futura dele poderia realizar a geração de código que utilize diversas *threads*, por exemplo, para a realização em paralelo do processo de inferência do paradigma;

- Melhorias da LingPON Fuzzy: a linguagem desenvolvida permite que desenvolvedores criem sistemas *fuzzy* mais facilmente. Porém foram identificados alguns elementos nela que poderiam ser simplificados para tornar o desenvolvimento ainda mais fácil. Um exemplo disto é a quantidade de código necessário para se declarar uma *Regra*. A versão atual da linguagem requer muito código para declarar os escopos internos da *Regra* (*Condições*, *Sub-condições* e *Premissas*) de regras simples.

Referências Bibliográficas

[Abu-Khudhair et al., 2010] Abu-Khudhair, A., Muresan, R., and Yang, S. X. (2010). Fpga based real-time adaptive fuzzy logic controller. In *Automation and Logistics (ICAL), 2010 IEEE International Conference on*, pages 539–544.

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

[da Silva Delgado, 2002] da Silva Delgado, M. R. D. B. (2002). *Projeto Automático de Sistemas Nebulosos: Uma Abordagem Co-Evolutiva*. PhD thesis, Universidade Estadual de Campinas (UNICAMP), Campinas, SP.

[de Souza et al., 2009] de Souza, J. T. S., Fabro, J. A., Simão, J. M., and Banaszewski, R. F. (2009). Proposta de uma máquina de inferência fuzzy utilizando o paradigma orientado a notificações (pon). In *XIV Seminário de Iniciação Científica e Tecnológica da UTFPR*, Pato Branco, PR.

[de Witt and Linhares, 2010] de Witt, F. A. and Linhares, R. R. (2010). Implementação do paradigma orientado a notificações em hardware. Relatório interno restrito para a disciplina de Lógica Reconfigurável por Hardware. Prof. Dr. C. R. Erig Lima.

[de Witt et al., 2011] de Witt, F. A., Simão, J. M., Linhares, R. R., Stadzisz, P. C., and Lima, C. R. E. (2011). Comparação entre o paradigma orientado a objetos (poo) e o paradigma orientado a notificações (pon) em um controle discreto em lógica reconfigurável. *XVI Seminário de Iniciação Científica e Tecnológica da UTFPR*.

[Fabro, 1996] Fabro, J. A. (1996). Grupos neurais e sistemas nebulosos: Aplicação à navegação autônoma. Master's thesis, Universidade Estadual de Campinas (UNICAMP), Campinas, SP.

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

[Jasinski, 2012] Jasinski, R. P. (2012). Framework para geração de hardware em vhdl a partir de modelos em pon. Relatório interno restrito para a disciplina de Lógica Reconfigurável por Hardware. Prof. Dr. C. R. Erig Lima.

[Johnson et al., 1995] Johnson, R., Gamma, E., Vlissides, J., and Helm, R. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[Koslosky et al., 2015] Koslosky, E., Wehrmeister, M. A., Fabro, J. A., and de Oliveira, A. S. (2015). On using fuzzy logic to control a simulated hexacopter carrying an attached pendulum. In *2nd Latin-American Congress on Computational Intelligence*, pages 1–6, Curitiba, PR.

[Linhares, 2015] Linhares, R. R. (2015). *Contribuição para o Desenvolvimento de uma Arquitetura de Computação Própria ao Paradigma Orientado a Notificações*. PhD thesis, Universidade Tecnológica Federal do Paraná (UTFPR), Curitiba, PR.

[Linhares et al., 2011] Linhares, R. R., Ronszcka, A. F., Z.Valença, G., Batista, M. V., Witt, F. A. D., Lima, C. R. E., Simão, J. M., and Stadzisz, P. C. (2011). Comparações entre o paradigma orientado a objetos e o paradigma orientado a notificações sob o contexto de um simulador de sistema telefônico. *III Congreso Internacional de Computación y Telecomunicaciones*.

[Mamdani, 1974] Mamdani, E. (1974). Application of fuzzy algorithms for control of simple dynamic plant. *Proceedings of the Institution of Electrical Engineers*, 121(12):1585.

[MathWorks, 2015] MathWorks (2015). What is sugeno-type fuzzy inference?

[Oliveira et al., 2010] Oliveira, D. N., de Souza Braga, A. P., and da Mota Almeida, O. (2010). Fuzzy logic controller implementation on a fpga using vhdl. In *2010 Annual Meeting of the North American Fuzzy Information Processing Society (NAFIPS)*, pages 1–6, Toronto.

[Pedrycz and Gomide, 1998] Pedrycz, W. and Gomide, F. (1998). *An Introduction to Fuzzy Sets: Analysis and Design*. MIT Press.

[Peters, 2012] Peters, E. (2012). Coprocessador para aceleração de aplicações desenvolvidas utilizando paradigma orientado a notificações. Master's thesis, Universidade Tecnológica Federal do Paraná (UTFPR), Curitiba, PR.

[Peters et al., 2012] Peters, E., Jasinsk, R. P., Pedroni, V. A., and Simão, J. M. (2012). A new hardware coprocessor for accelerating notification-oriented applications. In *2012 International Conference on Field-Programmable Technology*, pages 257 – 260, Seoul.

[Ronszcka, 2012] Ronszcka, A. F. (2012). 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 (UTFPR), Curitiba, PR.

[Ross, 2010] Ross, T. J. (2010). *Fuzzy logic with engineering applications*. John Wiley & Sons, Ltd., third edit edition.

[Sánchez-Solano et al., 2007] Sánchez-Solano, S., Cabrera, A. J., Baturone, I., Moreno-Velo, F. J., and Brox, M. (2007). Fpga implementation of embedded fuzzy controllers for robotic applications. *Industrial Electronics, IEEE Transactions on*, 54(4):1937–1945.

[Simao and Stadzisz, 2009] Simao, J. and Stadzisz, P. (2009). Inference based on notifications: A holonic metamodel applied to control issues. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 39(1):238–250.

[Simão et al., 2009] Simão, J., Tacla, C., and Stadzisz, P. (2009). Holonic control metamodel. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 39(5):1126–1139.

[Simão, 2001] Simão, J. M. (2001). Proposta de uma arquitetura de controle para sistemas flexíveis de manufatura baseada em regras e agentes. Master's thesis, Universidade Tecnológica Federal do Paraná (UTFPR), Curitiba, PR.

[Simão, 2005] Simão, J. M. (2005). *A Contribution to the Development of a HMS Simulation Tool and Proposition of a Meta-Model for Holonic Control*. PhD thesis, Universidade Tecnológica Federal do Paraná (UTFPR), Curitiba, PR.

[Simão et al., 2003a] Simão, J. M., Fabro, J. A., Stadzisz, P. C., Arruda, L. V. R., and Ishimatsu, S. (2003a). An agent-oriented fuzzy inference engine. In *VI Simpósio Brasileiro de Automação Inteligente*, pages 585–590, Bauru, SP.

[Simão and Stadzisz, 2002] Simão, J. M. and Stadzisz, P. C. (2002). An agent-oriented inference engine applied for supervisory control of automated manufacturing systems. *J. Abe, & J. Silva Filho, Advances in Logic, Artificial Intelligence and Robotics*, 85:234–241.

[Simão and Stadzisz, 2008] Simão, J. M. and Stadzisz, P. C. (2008). Paradigma orientado a notificações (pon) - uma técnica de composição e execução de software orientado a notificações. Patente pendente submetida ao INPI/Brazil em 2008 e Agência de Inovação da UTFPR.

[Simão and Stadzisz, 2010] Simão, J. M. and Stadzisz, P. C. (2010). Mecanismo de resolução de conflito e garantia de determinismo para o paradigma orientado a notificações (pon). Patente pendente submetida ao INPI/Brazil em 2010 e Agência de Inovação da UTFPR.

[Simão et al., 2010] Simão, J. M., Stadzisz, P. C., Banaszewski, R. F., and Tacla, C. A. (2010). Mecanismo de inferência otimizado do paradigma orientado a notificações (pon) e mecanismos de resolução de conflitos para ambientes monoprocessados e multiprocessados aplicados ao pon. Patente pendente submetida ao INPI/Brazil em 2010 e Agência de Inovação da UTFPR.

[Simão et al., 2003b] Simão, J. M., Stadzisz, P. C., Künzle, L. A., et al. (2003b). Rule and agent-oriented architecture to discrete control applied as petri net players. *Frontiers in Artificial Intelligence and Applications (FAAI)-Advances in Intelligent Systems and Robotics' LAPTEC 2003*, pages 121–129.

[Simão et al., 2012a] Simão, J. M., Stadzisz, P. C., Lima, C. R. E., de Witt, F. A., and Linhares, R. R. (2012a). Paradigma orientado a notificações em hardware digital. Pedido de Proteção Industrial e Pedido de Patente enviados à Agência de Inovação da UTFPR respectivamente em 11/05/2012 e 17/07/2012, Curitiba - PR, Brasil - Aguardando Aprovação da Agência para eventual envio para o INPI.

[Simão et al., 2012b] Simão, J. M., Stadzisz, P. C., Tacla, C. A., Linhares, R. R., Belmonte, D. L., and Banaszewski, R. F. (2012b). Comparações entre duas materializações do paradigma orientado a notificações (pon): Framework pon prototipal versus framework pon primário. In *IV Congreso Internacional de Computación y Telecomunicaciones*, pages 13–20, Lima, Peru.

[Simão et al., 2012c] Simão, J. M., Tacla, C. A., Banaszewski, R. F., and Stadzisz, P. C. (2012c). Notification oriented paradigm (nop) and imperative paradigm: A comparative study. *Journal of Software Engineering and Applications (JSEA)*.

[Sugeno, 1985] Sugeno, M. (1985). *Industrial Applications of Fuzzy Control*. Elsevier Science Inc., New York, NY, USA.

[Sulaiman et al., 2009] Sulaiman, N., Obaid, Z. A., Marhaban, M. H., and Hamidon, M. N. (2009). FPGA-Based Fuzzy Logic : Design and Applications - a Review. *IACSIT International Journal of Engineering and Technology*, 1(5):491–503.

[Tanscheit, 2004] Tanscheit, R. (2004). Sistemas fuzzy. *Inteligência computacional: aplicada a administraçao, economia e engenharia em Matlab*, pages 229–264.

[Valença, 2012] Valença, G. Z. (2012). Contribuição para a materialização do paradigma orientado a notificações (pon) via framework e wizard. Master's thesis, Universidade Tecnológica Federal do Paraná (UTFPR), Curitiba, PR.

[Xavier, 2012] Xavier, R. D. (2012). Paradigmas de desenvolvimento de software: comparação entre abordagens orientada a eventos e orientada a notificações. Master's thesis, Universidade Tecnológica Federal do Paraná (UTFPR), Curitiba, PR.

[Zadeh, 1965] Zadeh, L. A. (1965). Fuzzy sets. *Information and control*, 8(3):338–353.

Apêndice A

BNF da LingPON Fuzzy

```
<PROGRAM>              ::= <fbes> <inst> <strategy> <rules> <main>
                         | <fbes> <inst> <strategy> <rules>
                         | <fbes> <inst> <rules> <main>
                         | <fbes> <inst> <rules>

<inst>                 ::= <INST> <inst_declarations> <END_INST>

<strategy>             ::= <STRATEGY> <strategy_declaration> <END_STRATEGY>

< strategy_declaration > ::= <NO_ONE>
                         | <BREADTH>
                         | <DEPTH>

< inst_declarations >  ::= < inst_declaration >
                         | < inst_declaration > < inst_declarations >

< inst_declaration >   ::= <fbe_name> <inst_names>

<inst_names>           ::= <inst_name>
                         | <inst_name> <COMMA> <inst_names>

<rules>                ::= <rule>
                         | <rule> <rules>

<rule>                 ::= <RULE> <rule_id> <rule_depends> <rule_body> <END_RULE>

<rule_depends>         ::=
                         | <DEPENDS> <rule_ref>

<rule_body>            ::= <decl_condition> <decl_action>
                         | < decl_rule_properties > <decl_condition> <decl_action>

< decl_rule_properties > ::= <PROPERTIES> <rule_properties_body> <END_PROPERTIES>

<rule_properties_body> ::= <rule_property_body>
                         | <rule_properties_body> <rule_property_body>

<rule_property_body>   ::= <PRIORITY> <INTEGERVALUE>
                         | <KEEPER> <INTEGERVALUE>

<decl_condition>       ::= <CONDITION> <cond_id> <condition_body> <END_CONDITION>

<condition_body>       ::= <subcondition> < logical_operator > <condition_body>
                         | <subcondition>

< logical_operator >   ::= <AND>
                         | <OR>

<subcondition>         ::= <SUBCONDITION> <cond_id> <subcondition_body> <END_SUBCONDITION>

<subcondition_body>    ::= <premise> < logical_operator > <subcondition_body>
```

```
52                              | <premise>
53
54  <premise>                   ::= <PREMISE> <imp> <premise_id> <premise_exp>
55
56  <imp>                       ::=
57                              | <IMP>
58
59  <premise_exp>               ::= <ref_ext> <premise_comp> <premise_comp_value>
60                              | <ref_ext> <premise_comp> <ref_ext>
61
62  <premise_comp>              ::= <EQ>
63                              | <NE>
64                              | <LT>
65                              | <GT>
66                              | <LE>
67                              | <GE>
68                              | <IS>
69
70  <premise_comp_value>        ::= <number>
71                              | <boolean>
72                              | <STRINGVALUE>
73                              | <CHARVALUE>
74                              | <fuzzyset_id>
75
76  <decl_action>               ::= <ACTION> <action_id> <action_body> <END_ACTION>
77
78  <action_body>               ::= < instigation > <action_body>
79                              | < instigation >
80
81  < instigation >             ::= <INSTIGATION> <fuzzy_instigation> <instigation_id> <ref_ext> <LP> <RP> <SEMICOLON>
82
83  < fuzzy_instigation >       ::=
84                              | <FUZZY>
85
86  <fbes>                      ::= <fbe>
87                              | <fbe> <fbes>
88
89  <fbe>                       ::= <FBE> <fbe_name> <fbe_body> <END_FBE>
90
91  <fbe_body>                  ::= < decl_attributes > <decl_methods>
92                              | < decl_attributes >
93
94  < decl_attributes >         ::= <ATTRIBUTES> <attributes> <END_ATTRIBUTES>
95
96  < attributes >              ::= < attribute_body >
97                              | < attribute_body > < attributes >
98
99  < attribute_body >          ::= < attr_type > <attr_name> < attr_value >
100                             | <fbe_name> <attr_name>
101                             | <FUZZY> <attr_name> <attr_value> <decl_defuzzyfication> < decl_fuzzysets >
102
103 < attr_type >               ::= <BOOLEAN>
104                             | <INTEGER>
105                             | <PFLOAT>
106                             | <STRING>
107                             | <CHAR>
108
109 < attr_value >              ::=
110                             | <number>
111                             | <boolean>
112                             | <STRINGVALUE>
113                             | <CHARVALUE>
114
115 < decl_defuzzyfication >    ::=
116                             | <MASSCENTER>
117                             | <CENTROID> <FLOATVALUE>
118
119 < decl_fuzzysets >          ::= <FUZZYSETS> <fuzzysets> <END_FUZZYSETS>
120
121 < fuzzysets >               ::= <fuzzyset_body>
```

```
122                         | <fuzzyset_body> <fuzzysets>
123
124 <fuzzyset_body>          ::= <TRIANGULAR> <fuzzyset_id> <number> <number> <number>
125                         | <TRAPEZOIDAL> <fuzzyset_id> <number> <number> <number> <number>
126                         | <SINGLETON> <fuzzyset_id> <number>
127
128 <decl_methods>           ::= <METHODS> <methods> <END_METHODS>
129
130 <methods>                ::= <method_body> <methods>
131                         | <method_body>
132
133 <method_body>            ::= <METHOD> <method_name> <LP> <RP> <INNER_CODE_METHOD>
134                         | <METHOD> <method_name> <LP> <ref_local> <ASSIGN> <method_assign_expr> <RP>
135                         | <METHOD> <FUZZY> <method_name> <LP> <ref_local> <IS> <fuzzyset_id> <RP>
136
137 <method_assign_expr>     ::= <method_value>
138                         | < ref_local >
139                         | < ref_local > <method_operator> <method_value>
140                         | < ref_local > <method_operator> < ref_local >
141
142 <method_operator>        ::= <PLUS>
143                         | <MINUS>
144                         | <MULT>
145                         | <DIV>
146
147 <method_value>           ::= <number>
148                         | <boolean>
149                         | <STRINGVALUE>
150                         | <CHARVALUE>
151
152 <main>                   ::= <MAIN> <INNER_CODE_MAIN>
153
154 <fbe_name>               ::= < identifier >
155
156 <attr_name>              ::= < identifier >
157
158 <method_name>            ::= < identifier >
159
160 <inst_name>              ::= < identifier >
161
162 <rule_id>                ::=
163                         | < identifier >
164
165 <cond_id>                ::=
166                         | < identifier >
167
168 <premise_id>             ::=
169                         | < identifier >
170
171 <action_id>              ::=
172                         | < identifier >
173
174 < instigation_id >       ::=
175                         | < identifier >
176
177 <fuzzyset_id>            ::= < identifier >
178
179 < ref_local >            ::= < identifier >
180                         | <ID_POINT>
181                         | <ID_POINT> <POINT> <ref_local>
182
183 <ref_ext>                ::= <ID_POINT>
184                         | <ID_POINT> <POINT> <ID>
185                         | <ID_POINT> <POINT> <ref_ext>
186
187 < rule_ref >             ::= < identifier >
188
189 < identifier >           ::= <ID>
190
191 <boolean>                ::= <TRUE>
```

```
192                         | <FALSE>
193
194 <number>                ::= <INTEGERVALUE>
195                         | <FLOATVALUE>
```

Apêndice B

Códigos-fonte em LingPON Fuzzy utilizado nos testes

Neste apêndice estão contidos os códigos-fonte dos sistemas *fuzzy* que foram utilizados nos testes.

B.1 Sistema da máquina de lavar

```
fbe Lavadora
  attributes
    integer contador 0

    // se o método de defuzzyficação não estiver ajustado, significa
    // que o atributo não pode ser utilizado no consequênte de uma regra
    fuzzy sujeira -1
      fuzzysets
        triangular poucaSujeira 0.0 0.0 50.0
        triangular mediaSujeira 0.0 50.0 100.0
        triangular muitaSujeira 50.0 100.0 100.0
      end_fuzzysets
        // o valor de inicialização éopcional
    fuzzy mancha -1
      fuzzysets
        triangular semMancha 0.0 0.0 50.0
        triangular mediaMancha 0.0 50.0 100.0
        triangular grandeMancha 50.0 100.0 100.0
      end_fuzzysets

    // defuzzifica usando a fórmula de centro de massa
    fuzzy tempo 0 centroid 0.1
      fuzzysets
        triangular muitoCurto 0.0 0.0 10.0
        triangular curto 0.0 10.0 25.0
        triangular medio 10.0 25.0 40.0
        triangular longo 25.0 40.0 60.0
        triangular muitoLongo 40.0 60.0 60.0
      end_fuzzysets
  end_attributes

  methods
    // incrementa o contador de vezes que as regras se ativaram
    method incContador(contador = contador + 1)

    // métodos fuzzy não poderão ser chamados a partir das linguagens OO
    method fuzzy tempoIsMuitoCurto(tempo is muitoCurto)
    method fuzzy tempoIsCurto(tempo is curto)
    method fuzzy tempoIsMedio(tempo is medio)
```

```
40      method fuzzy tempoIsLongo(tempo is longo)
41      method fuzzy tempoIsMuitoLongo(tempo is muitoLongo)
42    end_methods
43  end_fbe
44
45  inst
46    Lavadora lavadora
47  end_inst
48
49  strategy
50    no_one
51  end_strategy
52
53  rule
54    condition
55      subcondition
56          // caso já existir uma premissa com o mesmo nome que esta, ela será
57          // utilizada e o restante da declaração desta será ignorada. Caso
58          // não seja definido um nome para a premissa, o compilador irá criar
59          // uma premissa que será utilizada apenas nesta condição
60        premise premPoucaSujeira lavadora.sujeira is poucaSujeira and
61        premise premSemMancha lavadora.mancha is semMancha
62      end_subcondition
63    end_condition
64
65    action
66      // as instigações fuzzy vão executar a cada mudança no grau de ativação
67      // da regra. As instigações normais vão executar apenas quando o grau
68      // for igual a 1.
69      instigation fuzzy lavadora.tempoIsMuitoCurto();
70      instigation fuzzy lavadora.incContador();
71    end_action
72  end_rule
73
74  rule
75    condition
76      subcondition
77        premise premMediaSujeira lavadora.sujeira is mediaSujeira and
78        premise premSemMancha lavadora.mancha is semMancha
79      end_subcondition
80    end_condition
81
82    action
83      instigation fuzzy lavadora.tempoIsCurto();
84      instigation fuzzy lavadora.incContador();
85    end_action
86  end_rule
87
88  rule
89    condition
90      subcondition
91        premise premMuitaSujeira lavadora.sujeira is muitaSujeira and
92        premise premSemMancha lavadora.mancha is semMancha
93      end_subcondition
94    end_condition
95
96    action
97      instigation fuzzy lavadora.tempoIsMedio();
98      instigation fuzzy lavadora.incContador();
99    end_action
100 end_rule
101
102 rule
103   condition
104     subcondition
105       premise premPoucaSujeira lavadora.sujeira is poucaSujeira and
106       premise premMediaMancha lavadora.mancha is mediaMancha
107     end_subcondition
108   end_condition
109
```

```
110   action
111     instigation fuzzy lavadora.tempoIsMedio();
112     instigation fuzzy lavadora.incContador();
113   end_action
114 end_rule
115
116 rule
117   condition
118     subcondition
119       premise premMediaSujeira lavadora.sujeira is mediaSujeira and
120       premise premMediaMancha lavadora.mancha is mediaMancha
121     end_subcondition
122   end_condition
123
124   action
125     instigation fuzzy lavadora.tempoIsMedio();
126     instigation fuzzy lavadora.incContador();
127   end_action
128 end_rule
129
130 rule
131   condition
132     subcondition
133       premise premMuitaSujeira lavadora.sujeira is muitaSujeira and
134       premise premMediaMancha lavadora.mancha is mediaMancha
135     end_subcondition
136   end_condition
137
138   action
139     instigation fuzzy lavadora.tempoIsLongo();
140     instigation fuzzy lavadora.incContador();
141   end_action
142 end_rule
143
144 rule
145   condition
146     subcondition
147       premise premPoucaSujeira lavadora.sujeira is poucaSujeira and
148       premise premGrandeMancha lavadora.mancha is grandeMancha
149     end_subcondition
150   end_condition
151
152   action
153     instigation fuzzy lavadora.tempoIsLongo();
154     instigation fuzzy lavadora.incContador();
155   end_action
156 end_rule
157
158 rule
159   condition
160     subcondition
161       premise premMediaSujeira lavadora.sujeira is mediaSujeira and
162       premise premGrandeMancha lavadora.mancha is grandeMancha
163     end_subcondition
164   end_condition
165
166   action
167     instigation fuzzy lavadora.tempoIsLongo();
168     instigation fuzzy lavadora.incContador();
169   end_action
170 end_rule
171
172 rule
173   condition
174     subcondition
175       premise premMuitaSujeira lavadora.sujeira is muitaSujeira and
176       premise premGrandeMancha lavadora.mancha is grandeMancha
177     end_subcondition
178   end_condition
179
```

```
  action
    instigation fuzzy lavadora.tempoIsMuitoLongo();
    instigation fuzzy lavadora.incContador();
  end_action
end_rule
```

B.2 Controlador da rotação do eixo X do hexacóptero

```
// estabilizador do roll
fbe RollStabilizator
      attributes
            // Variáveis de entrada --------------------------------

            // Erro do ângulo de rotação no eixo Y
            fuzzy rollAngleError 0
            fuzzysets
                  trapezoidal N_FAR −0.436331 −0.349066 −0.349066 −0.261755
                  triangular N_NEAR −0.349066 −0.261755 −0.174533
                  triangular N_CLOSE −0.261755 −0.174533 0
                  triangular ZERO −0.174533 0 +0.174533
                  triangular P_CLOSE 0 +0.174533 +0.261755
                  triangular P_NEAR +0.174533 +0.261755 +0.349066
                  trapezoidal P_FAR +0.261755 +0.349066 +0.349066 +0.436331
            end_fuzzysets

            // aceleração no eixo Y
            fuzzy yAxisAccel 0
            fuzzysets
                  trapezoidal N_FAST −1000000 −0.60000 −0.60000 −0.20000
                  trapezoidal N_SLOW −0.40000 −0.20000 −0.10000 0.00000
                  triangular ZERO −0.10000 0.00000 0.10000
                  trapezoidal P_SLOW 0.00000 0.10000 0.20000 0.40000
                  trapezoidal P_FAST 0.20000 0.60000 0.60000 1000000
            end_fuzzysets

            // Variáveis de saída ----------------------------------

            // ajuste no roll
            fuzzy omegaRoll 0 masscenter
            fuzzysets
                  triangular N_MAX −0.10 −0.075 −0.05
                  triangular N_MID −0.075 −0.05 −0.03
                  triangular N_MIN −0.05 −0.03 0
                  triangular ZERO −0.03 0 0.03
                  triangular P_MIN 0 0.03 0.05
                  triangular P_MID 0.03 0.05 0.075
                  triangular P_MAX 0.05 0.075 0.10
            end_fuzzysets
      end_attributes

      methods
            // métodos fuzzy para omegaRoll
            method fuzzy omegaRollIsNMax(omegaRoll is N_MAX)
            method fuzzy omegaRollIsNMid(omegaRoll is N_MID)
            method fuzzy omegaRollIsNMin(omegaRoll is N_MIN)
            method fuzzy omegaRollIsZero(omegaRoll is ZERO)
            method fuzzy omegaRollIsPMin(omegaRoll is P_MIN)
            method fuzzy omegaRollIsPMid(omegaRoll is P_MID)
            method fuzzy omegaRollIsPMax(omegaRoll is P_MAX)
      end_methods
end_fbe

inst
      RollStabilizator rollStabilizator
end_inst

```

```
strategy
      no_one
end_strategy

// conjunto de regras

// regras para oRoll ----------------------------------------------------

// if roll_error is P_FAR then Oroll is N_MAX
rule
  condition
    subcondition
      premise premRollPFar rollStabilizator.rollAngleError is P_FAR
    end_subcondition
  end_condition

  action
    instigation fuzzy rollStabilizator.omegaRollIsNMax();
  end_action
end_rule

// if roll_error is P_NEAR then Oroll is N_MAX
rule
  condition
    subcondition
      premise premRollPNear rollStabilizator.rollAngleError is P_NEAR
    end_subcondition
  end_condition

  action
    instigation fuzzy rollStabilizator.omegaRollIsNMax();
  end_action
end_rule

// if roll_error is P_CLOSE and accelY is N_FAST then Oroll is N_MAX
rule
  condition
    subcondition
      premise premRollPClose rollStabilizator.rollAngleError is P_CLOSE and
      premise premYAccNFast rollStabilizator.yAxisAccel is N_FAST
    end_subcondition
  end_condition

  action
    instigation fuzzy rollStabilizator.omegaRollIsNMax();
  end_action
end_rule

// if roll_error is P_CLOSE and accelY is N_SLOW then Oroll is N_MIN
rule
  condition
    subcondition
      premise premRollPClose rollStabilizator.rollAngleError is P_CLOSE and
      premise premYAccNSlow rollStabilizator.yAxisAccel is N_SLOW
    end_subcondition
  end_condition

  action
    instigation fuzzy rollStabilizator.omegaRollIsNMin();
  end_action
end_rule

// if roll_error is P_CLOSE and accelY is ZERO then Oroll is N_MIN
rule
  condition
    subcondition
      premise premRollPClose rollStabilizator.rollAngleError is P_CLOSE and
      premise premYAccZero rollStabilizator.yAxisAccel is ZERO
    end_subcondition
  end_condition
```

```
129
130   action
131     instigation fuzzy rollStabilizator.omegaRollIsNMin();
132   end_action
133 end_rule
134
135 // if roll_error is P_CLOSE and accelY is P_SLOW then Oroll is ZERO
136 rule
137   condition
138     subcondition
139       premise premRollPClose rollStabilizator.rollAngleError is P_CLOSE and
140       premise premYAccPSlow rollStabilizator.yAxisAccel is P_SLOW
141     end_subcondition
142   end_condition
143
144   action
145     instigation fuzzy rollStabilizator.omegaRollIsZero();
146   end_action
147 end_rule
148
149 // if roll_error is P_CLOSE and accelY is P_FAST then Oroll is ZERO
150 rule
151   condition
152     subcondition
153       premise premRollPClose rollStabilizator.rollAngleError is P_CLOSE and
154       premise premYAccPFast rollStabilizator.yAxisAccel is P_FAST
155     end_subcondition
156   end_condition
157
158   action
159     instigation fuzzy rollStabilizator.omegaRollIsZero();
160   end_action
161 end_rule
162
163 // ----------
164
165 // if roll_error is ZERO and accelY is ZERO then Oroll is ZERO
166 rule
167   condition
168     subcondition
169       premise premRollZero rollStabilizator.rollAngleError is ZERO and
170       premise premYAccZero rollStabilizator.yAxisAccel is ZERO
171     end_subcondition
172   end_condition
173
174   action
175     instigation fuzzy rollStabilizator.omegaRollIsZero();
176   end_action
177 end_rule
178
179 // ----------
180
181 // if roll_error is N_CLOSE and accelY is P_FAST then Oroll is P_MAX
182 rule
183   condition
184     subcondition
185       premise premRollNClose rollStabilizator.rollAngleError is N_CLOSE and
186       premise premYAccPFast rollStabilizator.yAxisAccel is P_FAST
187     end_subcondition
188   end_condition
189
190   action
191     instigation fuzzy rollStabilizator.omegaRollIsPMax();
192   end_action
193 end_rule
194
195 // if roll_error is N_CLOSE and accelY is P_SLOW then Oroll is P_MIN
196 rule
197   condition
198     subcondition
```

```
199       premise premRollNClose rollStabilizator.rollAngleError is N_CLOSE and
200       premise premYAccPSlow rollStabilizator.yAxisAccel is P_SLOW
201     end_subcondition
202   end_condition
203
204   action
205     instigation fuzzy rollStabilizator.omegaRollIsPMin();
206   end_action
207 end_rule
208
209 // if roll_error is N_CLOSE and accelY is ZERO then Oroll is P_MIN
210 rule
211   condition
212     subcondition
213       premise premRollNClose rollStabilizator.rollAngleError is N_CLOSE and
214       premise premYAccZero rollStabilizator.yAxisAccel is ZERO
215     end_subcondition
216   end_condition
217
218   action
219     instigation fuzzy rollStabilizator.omegaRollIsPMin();
220   end_action
221 end_rule
222
223 // if roll_error is N_CLOSE and accelY is N_SLOW then Oroll is ZERO
224 rule
225   condition
226     subcondition
227       premise premRollNClose rollStabilizator.rollAngleError is N_CLOSE and
228       premise premYAccNSlow rollStabilizator.yAxisAccel is N_SLOW
229     end_subcondition
230   end_condition
231
232   action
233     instigation fuzzy rollStabilizator.omegaRollIsZero();
234   end_action
235 end_rule
236
237 // if roll_error is N_CLOSE and accelY is N_FAST then Oroll is ZERO
238 rule
239   condition
240     subcondition
241       premise premRollNClose rollStabilizator.rollAngleError is N_CLOSE and
242       premise premYAccNFast rollStabilizator.yAxisAccel is N_FAST
243     end_subcondition
244   end_condition
245
246   action
247     instigation fuzzy rollStabilizator.omegaRollIsZero();
248   end_action
249 end_rule
250
251 // ----------
252
253 // if roll_error is N_NEAR then Oroll is P_MAX
254 rule
255   condition
256     subcondition
257       premise premRollNNear rollStabilizator.rollAngleError is N_NEAR
258     end_subcondition
259   end_condition
260
261   action
262     instigation fuzzy rollStabilizator.omegaRollIsPMax();
263   end_action
264 end_rule
265
266 // if roll_error is N_FAR then Oroll is P_MAX
267 rule
268   condition
```

```
269     subcondition
270       premise premRollNFar rollStabilizator.rollAngleError is N_FAR
271     end_subcondition
272   end_condition
273
274   action
275     instigation fuzzy rollStabilizator.omegaRollIsPMax();
276   end_action
277 end_rule
```

Apêndice C

Artigo publicados

Durante a execução do trabalho foram elaborados artigos contendo os avanços do mesmo. Neste apêndice estão contidos os artigos publicados em eventos e revistas da área.

C.1 Artigo publicado no III CBFS

Este artigo foi publicado no III Congresso Brasileiro de Sistema Fuzzy (CBFS) que ocorreu no ano de 2014 em João Pessoa/PB.

Adaptação do Paradigma Orientado a Notificações para desenvolvimento de sistemas *fuzzy*

Luiz Carlos Viana Melo, Jean Marcelo Simão, and João Alberto Fabro

Departamento Acadêmico de Informática (DAINF)
Universidade Tecnológica Federal do Paraná (UTFPR)
Av. Sete de Setembro, 3165 – Rebouças CEP 80230-901 – Curitiba – PR – Brasil
amomrabr@gmail.com,jeansimao@utfpr.edu.br,
fabro@utfpr.edu.br
http://www2.dainf.ct.utfpr.edu.br/

Resumo O Paradigma Orientado a Notificações - PON combina a programação baseada em eventos e a programação declarativa a fim de solucionar os problemas de ambos. Ao decompor uma aplicação em uma rede de entidades computacionais menores que são executadas apenas quando necessário, o PON elimina a necessidade de realizar computações desnecessárias e melhora a reusabilidade do código. Os sistemas *fuzzy* são sistemas que utilizam a lógica *fuzzy* para a realização de inferências sobre o conhecimento, utilizando regras SE-ENTÃO sobre conceitos que envolvem imprecisão. Devido a similaridade na forma de representação de conhecimento de sistemas desenvolvidos utilizando o PON e sistemas *fuzzy*, foi realizado um estudo para verificar as mudanças necessárias a serem realizadas sobre o paradigma para que este possa ser utilizado no desenvolvimento de sistemas *fuzzy*.

Keywords: Paradigma orientado a notificações, sistemas *fuzzy*, sistemas baseados em regras

1 Introdução

Um *software* dispõe de duas maneiras para detectar mudanças nos valores de uma variável: através de *polling* e notificações de eventos. No *polling*, o *loop* de um programa realiza uma leitura contínua do valor das variáveis do sistema, realiza computação sobre elas e dispara ações caso certas condições sejam atendidas. Esta abordagem é considerada sequencial já que apenas uma condição é analisada por vez. Devido ao fato que o *loop* é executado mesmo quando não há modificação nos valores das variáveis, esta abordagem desperdiça recursos como tempo do processador e energia elétrica [1].

Uma alternativa ao *polling* é a programação orientada a eventos, onde a computação é realizada apenas quando ocorre um evento. Em algumas abordagens, isto elimina a necessidade de se ter um *loop* que monitora o estado de uma variável, o que reduz a quantidade de computação desnecessária. Porém

esta alternativa torna o desenvolvimento de aplicações complexo, normalmente resultando em programas maiores. Além disso, dadas as restrições de *hardware*, o sistema de controle de eventos pode realizar um *polling* para fazer o despacho dos mesmos [1].

Tendo estes problemas como motivação, o Paradigma Orientado a Notificações - PON - combina a programação baseada em eventos e a programação declarativa a fim de solucioná-los. Ao decompor uma aplicação em uma rede de entidades computacionais menores que são executadas apenas quando necessário, o PON elimina a necessidade de realizar computações desnecessárias e melhora a reusabilidade do código. O PON agrega as principais vantagens dos paradigma declarativo e sistemas baseados em regras (maior abstração e linguagem mais próxima a forma de cognição humana), e dos paradigmas imperativo e orientado a objetos (reusabilidade do código, flexibilidade e abstração através de classes e objetos) [1].

Baseada na teoria dos conjuntos *fuzzy* definida por Lofti A. Zadeh (*Fuzzy Sets* [2]), a lógica *fuzzy* proporciona os mecanismos para realizar inferências baseadas em informações nebulosas. Os sistemas *fuzzy* são sistemas que utilizam a lógica *fuzzy* para a realização de inferências sobre o conhecimento representado pelas implicações descritas anteriormente. A vantagem destes é que o conhecimento pode ser descrito na forma de regras SE-ENTÃO similares aquelas empregadas em uma linguagem natural [3].

Devido a similaridade na forma de representação de conhecimento de sistemas desenvolvidos utilizando o PON e sistemas *fuzzy*, foi realizado um estudo para verificar as mudanças necessárias a serem realizadas sobre o paradigma para que este possa ser utilizado no desenvolvimento de sistemas *fuzzy*. Após esta breve introdução, que apresentou as motivações do estudo, as seções a seguir demonstram os conceitos relacionados ao PON e sistemas *fuzzy*, as modificações realizadas sobre o paradigma e os testes com os resultados obtidos.

2 Paradigma Orientado a Notificações

O paradigma foi concebido por Jean Marcelo Simão na forma de um mecanismo de controle que suprisse as necessidades relacionadas com os sistemas modernos de produção. Posteriormente, o autor percebeu que este modelo poderia ser aplicado em diferentes domínios de problemas. Sendo assim, ele propôs o modelo como uma solução geral de controle discreto. Por fim, ele também percebeu que o modelo poderia ser utilizado para guiar o programador na concepção de programas, surgindo assim o PON [4]. O paradigma será descrito com maiores detalhes nas sub-seções a seguir.

2.1 Estrutura do paradigma

O PON introduz um novo conceito para conceber, construir e executar programas de computador. Ele é baseado na concepção de entidades pequenas, ativas e desacopladas que colaboram por meio de notificações para realização do cálculo

lógico e causal existente no *software* [5,6]. O conhecimento no paradigma é representado através do uso de regras causais que são compreendidas naturalmente por programadores dos paradigmas atuais [6]. A figura 1 demonstra um exemplo de uma regra decomposta nas entidades do paradigma.

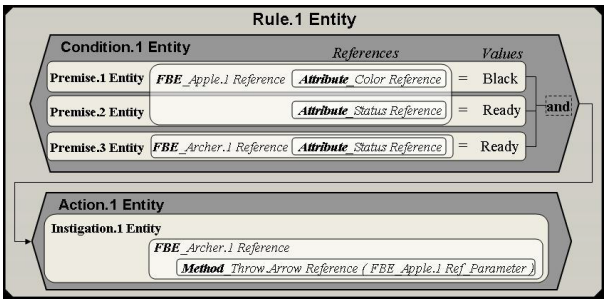


Figura 1. Representação de uma regra do PON [6].

As entidades que compõem o paradigma são as seguintes:

- *Elemento da Base de Fatos - Fact Base Element* (FBE): representa uma entidade do sistema observado. Este agrega os *Atributos* e os *Métodos* que representam os fatos da entidade citada;
- *Atributo*: representa um valor incluso no FBE que representa uma característica ou um estado do mesmo;
- *Premissa*: representa uma indagação lógica entre os *Atributos* de um FBE e um valor (por exemplo, "X é igual a 2 ?"). Estes são compostos de uma referência para um *Atributo*, o operador lógico de comparação e o valor a ser comparado, sendo que este último pode ser simples (um valor) ou composto (outro *Atributo*);
- *Condição*: representa uma relação lógica entre as premissas da regra. A relação entre as premissas pode ser denotada pelo conector lógico de conjunção (E), disjunção (OU) ou combinações de ambos;
- *Regra*: representa uma regra da base de regras do sistema, sendo que esta entidade agrega uma *Condição* e uma *Ação*. A relação implementada por esta entidade é de implicação, sendo que esta pode ser lida na forma "Se a *Condição* estiver ativa (antecedente), então ative a *Ação* (consequente)". Este elemento também é responsável por fazer a ligação entre as partes ativa e reativa do processo de inferência;
- *Ação*: representa uma ação a ser executada caso a *Regra* a qual esta entidade está associada for aprovada. Este elemento contém um conjunto de *Instigações* que deverão ser processadas;
- *Instigação*: representa a atividade de indução a execução de um *Método* sobre um FBE;
- *Método*: representa um método de um FBE que pode realizar alterações sobre os *Atributos* do mesmo.

A figura 2 demonstra o diagrama de classes das entidades citadas e o relacionamento existente entre elas.

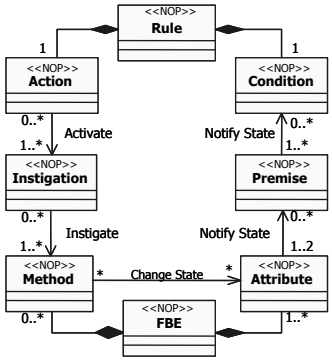


Figura 2. Diagrama de classes com as entidades do paradigma [6].

A colaboração entre as entidades é a base para o mecanismo de inferência do paradigma, sendo que a colaboração entre elas é feita através de troca de notificações pontuais. Na seção a seguir será demonstrado o mecanismo de notificações do paradigma através da demonstração do papel realizado por cada entidade na realização de inferências sobre o conhecimento.

2.2 Mecanismo de notificações

O mecanismo de notificações consiste no processo interno de execução das instâncias do paradigma, o qual determina o fluxo de execução das aplicações. Por meio deste mecanismo, as tarefas de um programa são divididas entre as entidades, as quais cooperam por meio de notificações informando umas as outras sobre as parcelas de suas contribuições a fim de formar o fluxo de execução do programa [4]. A figura 3 demonstra um exemplo de cadeia de notificações entre diversas instâncias de entidades do paradigma.

O processo de inferência é iniciado quando o estado (valor) de um *Atributo* de um FBE é alterado(FBC.1 e FBC.n, lado esquerdo da figura). Cada alteração faz com que o *Atributo* notifique as *Premissas* que estão interessadas no mesmo. As *Premissas*, após notificadas, realizam o cálculo lógico para determinar se ocorreu uma mudança de estado nas mesmas. Este cálculo é feito através da comparação do valor contido no *Atributo* com o valor da *Premissa* usando um operador lógico. Da mesma forma que um *Atributo* colabora com as *Premissas*, estas colaboram com as *Condições*, notificando-as quando ocorrem mudanças no valor booleano que representa o resultado do cálculo lógico. Ao receber notificações das *Premissas*, as *Condições* realizam seus cálculos lógicos a partir de uma expressão lógica causal. Esta expressão define o relacionamento entre as *Premissas* notificadoras através do uso de conectivos lógicos (“E” e “OU”). Uma

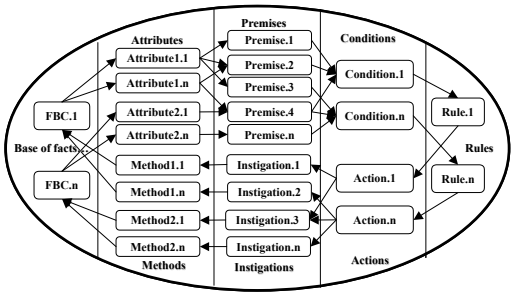


Figura 3. Cadeia de notificações do paradigma [5].

expressão pode ser lida conforme o seguinte exemplo: "Premissa1 ativa 'E' Premissa2 ativa". Caso houver uma mudança no estado de uma *Condição*, esta irá realizar a notificação da *Regra* a qual ela está associada. Uma mesma *Condição* pode estar associada, e portanto notificar, uma ou mais *Regras*.

A *Regra*, ao receber a notificação de uma *Condição*, irá verificar se a mesma foi ativa (isto é, se todas as *Condições* associadas a esta regra estão ativas). Caso afirmativo, a *Regra* irá executar a *Ação* associada. Esta irá realizar uma série de instigações através das entidades *Instigações* contidas nela. Por último, as *Instigações* farão com que os *Métodos* dos FBEs associados sejam executados, podendo fazer com que os *Atributos* destes últimos sejam alterados, iniciando novamente o ciclo de notificações.

3 Sistemas *Fuzzy*

Os estudos sobre sistemas *fuzzy*, também conhecidos como sistemas nebulosos, iniciaram a partir de 1965 quando Lofti A. Zadeh desenvolveu sua teoria dos conjuntos *fuzzy* ([2]). Esta teoria generaliza a teoria clássica dos conjuntos, permitindo representar conceitos que não podem ser bem representados usando limites claramente definidos, ou *crisp* [7].

Um conjunto *fuzzy* é um conjunto contendo os elementos do universo de discussão que possuem graus variáveis de adesão neste conjunto. Esta ideia difere dos conjuntos clássicos já que nestes o elemento só fará parte de um conjunto caso o seu grau de adesão seja completo. Elementos em um conjunto *fuzzy*, devido ao fato de poderem não ser completamente membros de um conjunto, poderão também ser membros de outros conjuntos no mesmo universo [8]. A relação entre um elemento e os conjunto aos quais ele pertence é representada pelo seu grau de pertinência àquele conjunto. A equação 1 demonstra a notação matemática do grau de pertinência μ_A de um elemento x em um conjunto A, onde o valor "1.0" indica adesão completa do elemento ao conjunto e o valor "0.0" indica que não há qualquer adesão do elemento ao conjunto.

$$\mu_A(x) \in [0, 1] \tag{1}$$

Um conceito relacionado com conjuntos *fuzzy* é o de variável linguística que representa um identificador que pode assumir um dentre vários valores [7]. Desta forma, uma variável linguística pode assumir um valor linguístico que representa um conjunto *fuzzy*. Na figura 4 é possível visualizar um exemplo de variável linguística que representa a altura de uma pessoa com os conjuntos *fuzzy* que representam seus possíveis valores (Baixo, Médio e Alto).

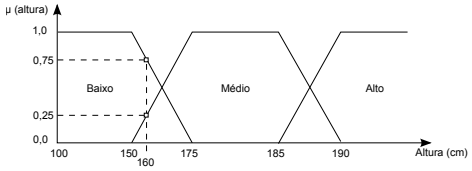


Figura 4. Variável linguística que representa a altura de uma pessoa [7].

Baseada na teoria dos conjuntos *fuzzy*, a lógica *fuzzy* proporciona os mecanismos para realizar inferências baseadas em informações imprecisas. Analogamente a teoria dos conjuntos, a lógica *fuzzy* é uma generalização da lógica tradicional. Utilizando os operadores de complemento, união e interseção *fuzzy*, é possível realizar todos os processos de inferência já conhecidos na lógica tradicional com conjuntos *fuzzy* [7]. Os operadores sobre os conjuntos *fuzzy* definidos por Zadeh a partir do grau de pertinência são os seguintes [9]:

- Interseção : $\mu_{(A \cap B)}(x) = min[\mu_A(x), \mu_B(x)]$;
- União : $\mu_{(A \cup B)}(x) = max[\mu_A(x), \mu_B(x)]$;
- Complemento : $\mu_{(\bar{A})}(x) = 1 - \mu_A(x)$.

A lógica *fuzzy* pode ser utilizada para o desenvolvimento de sistemas de controle que lidam com informações imprecisas. Através de um conjunto de regras é possível realizar a inferência de informações que serão utilizadas na tomada de decisão de um sistema.

Na maioria das aplicações práticas consideram-se dados de entrada como valores precisos ou não *fuzzy* como, por exemplo, resultante de medições ou observações. Para utilização destes dados pelo sistema de inferência *fuzzy*, é necessário efetuar um mapeamento destes dados de entrada para os conjuntos *fuzzy* relevantes [10]. Este mapeamento é chamado de "fuzzyficação" e retorna o grau de pertinência do dado a cada conjunto *fuzzy*.

Os conjuntos gerados são utilizados para realizar as inferências *fuzzy*. A etapa de inferência se dá através da comparação destes com os antecedentes das regras armazenadas na base de regras. Este processo resulta no cálculo do nível de ativação de cada regra. Este cálculo é feito através da aplicação dos operadores sobre os conjuntos *fuzzy* que são propostos por diversos trabalhos disponíveis na bibliografia. Os operadores que foram definidos em [2] são utilizados em [11] da seguinte forma:

– Interseção : conectivo lógico **E**;
– União : conectivo lógico **OU**;
– Complemento : operação lógica de negação.

Após este cálculo, é feita a propagação dos valores verdade através das regras *fuzzy*, gerando conjuntos *fuzzy* representativos dos consequentes de cada regra. A contribuição de cada regra (conjunto *fuzzy*) é levada em consideração via, por exemplo, a união entre os conjuntos *fuzzy*. A partir deste ponto o operador já tem a resposta do sistema na forma de conjuntos *fuzzy* [7].

Uma vez obtido o conjunto fuzzy de saída através do processo de inferência, no estágio de "defuzzificação" é efetuada uma interpretação dessa informação. Isto se faz necessário pois, em aplicações práticas, geralmente são requeridas saídas exatas, não *fuzzy* [10].

Na seção seguinte, são detalhadas as modificações realizadas na implementação do PON para permitir a realização de inferências baseadas em regras e conceitos *fuzzy*.

4 Alterações realizadas sobre o PON

Foram necessárias diversas modificações no paradigma para que este possa ser utilizado para o desenvolvimento de sistemas de lógica *fuzzy*. Parte das modificações realizadas já foram feitas sobre o modelo multi-agentes que antecedeu a definição do paradigma conforme descrito em [12], sendo que este também foi base para as modificações descritas em [13]. No presente trabalho foram inclusas novas modificações e realizadas alterações sobre algumas das modificações descritas nos trabalhos citados. Estas modificações são descritas nas sub-seções a seguir.

4.1 Mudança na representação do estado lógico das entidades

Entidades do paradigma como, por exemplo, *Premissas*, *Condições* e *Regras* possuem um estado lógico que é propagado através das notificações. Este estado lógico possui, no paradigma originalmente proposto, apenas dois valores possíveis:

$$X(x) \in \{0, 1\}$$

Onde X representa o estado lógico do elemento x, sendo que o valor 0 (zero) denota que a entidade não está ativa e 1 (um) que a mesma está ativa. Para suportar a lógica *fuzzy*, as entidades citadas devem poder representar o seu estado lógico através de seu grau de ativação. O grau de ativação pode assumir os valores reais na faixa $[0, 1]$, podendo representar graus de pertinência, como apresentado na seção 3. As entidades iniciarão o processo de notificações caso o grau de ativação calculado seja diferente do valor atual. Este grau de ativação é calculado de forma diferente para cada uma das entidades e cada uma das formas será explicada nas seções a seguir.

4.2 Representação de uma variável linguística

Uma variável linguística foi implementada através de uma extensão de um *Atributo* do PON, conforme demonstrado pela figura 5. Este novo elemento, derivado de *Atributo* por herança, contém uma lista de conjuntos *fuzzy* que representam os termos linguísticos da variável. Cada conjunto é representado por uma função de pertinência utilizada para calcular o grau de pertinência de um valor àquele conjunto.

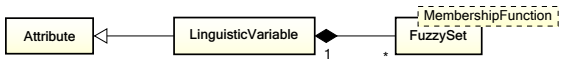


Figura 5. Definição do elemento "Variável linguística"no paradigma PON.

Ao atribuir um valor *crisp* a esta variável, a mesma se comportará como um *Atributo* e notificará as *Premissas* caso o valor ajustado seja diferente do valor atual. O cálculo do grau de ativação para um determinado conjunto *fuzzy* é feito em cada *Premissa* conforme descrito na seção 4.3.

4.3 Mudança no cálculo lógico do estado das *Premissas*

Devido as mudanças descritas na seção 4.1, o estado lógico da *Premissa* passa a ser representado pelo grau de ativação, sendo que o cálculo deste depende do tipo da *Premissa*. Os tipos de *Premissas* são os seguintes:

- *Premissa* comum (ou *crisp*): *Premissa* que faz a operação lógica entre um *Atributo* e um valor *crisp*. O resultado desta operação irá determinar o valor do grau de ativação da *Premissa*;
- *Premissa fuzzy*: *Premissa* que calcula o grau de pertinência do valor do *Atributo* no conjunto *fuzzy* associado a *Premissa*. O grau de pertinência calculado define o grau de ativação da *Premissa*.

No caso da *Premissa* comum o resultado da operação lógica define o grau de ativação da *Premissa*, onde para o resultado "verdadeiro" será atribuído o valor 1 (um) e "falso" o valor 0 (zero). No caso da *Premissa fuzzy* o grau de ativação será o resultado do cálculo do grau de pertinência do conjunto para o valor armazenado no *Atributo*.

4.4 Readequação da expressão lógica da *Condição*

A modificação sobre a *Condição* consistiu na criação de um modelo que representa as expressões causais das *Condições*. O modelo criado pode ser visualizado na figura 6.

Neste modelo a *Premissa* estará associada ao valor lógico da expressão, sendo que este valor será atualizado quando a *Premissa* notificar a *Condição*. Este valor

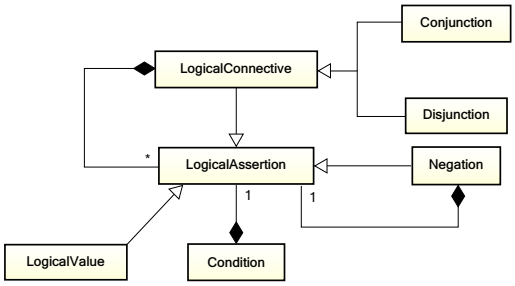


Figura 6. Modelo de expressão causal da *Condição*.

armazena o grau de ativação da *Premissa* a qual ele está associado e será utilizado como entrada para operações com os conectivos lógicos definidos no modelo. A notificação também irá iniciar o cálculo da expressão que irá determinar o valor do grau de ativação da *Condição*.

O cálculo é feito utilizando as definições das operações definidas por Mamdani [11] conforme descrito na seção 3. A utilização das operações citadas permite que as características da lógica booleana sejam "emuladas", permitindo que *Regras* não-*fuzzy* funcionem com o mesmo modelo.

4.5 Readequação das *Instigações* e a criação da *Instigação fuzzy*

Com as mudanças citadas na seção 4.1 uma *Regra* passará a se ativar caso haja uma mudança de estado na *Condição*. Isto também reflete nas *Instigações* já que estas serão ativadas juntamente com as *Regras*.

Neste trabalho foi definido que as *Instigações* passarão a ter duas formas: a *Instigação* comum e a *Instigação fuzzy*. As *Instigações* comuns representam a atual definição de *Instigações* do paradigma e servem para manter o atual comportamento do paradigma. Estas só serão ativadas caso o grau de ativação da *Regra* seja igual a um, sendo que, caso contrário, a *Instigação* não será ativada.

A *Instigação fuzzy* será ativada a cada mudança no valor do grau de ativação da *Regra* a qual ela está associada. Esta ativação implicará na execução do *Método* a qual ela está associada, sendo ele *fuzzy* ou não. Caso *Instigação fuzzy* implique na execução de um *Método fuzzy*, ela deverá repassar o grau de ativação da *Regra* para que este possa realizar as operações citadas na seção 4.6.

4.6 Criação do *Método fuzzy*

O *Método fuzzy* foi criado para representar um consequente lógico da lógica *fuzzy*. Ele associa o *Atributo* ao conjunto *fuzzy* e realiza a operação de "defuzzyficação". O resultado da operação é atribuído ao *Atributo*, sendo que esta etapa pode encadear um novo ciclo de notificações.

Apenas com estas alterações, o PON passa a permitir o uso de *Atributos*, *Premissas*, *Condições*, *Regras*, *Instigações* e *Métodos Fuzzy*, fornecendo desta

forma todo o suporte para a implementação de sistemas completos baseados em lógica *fuzzy*, e permitindo até a composição de regras com elementos *fuzzy* e não-*fuzzy* de forma transparente. Na próxima seção, são apresentados experimentos que comprovam a funcionalidade da proposta apresentada.

5 Testes e resultados

Os testes foram realizados utilizando a materialização do paradigma na forma de um *framework* definido em [14], que foi adaptado para suportar o desenvolvimento de sistemas de inferência baseados em lógica *fuzzy*. Foram feitas comparações com um sistema desenvolvido na linguagem C++ sem a utilização do PON. O sistema consiste em um controlador de máquina de lavar que calcula o tempo de lavagem de acordo com a quantidade de sujeira e o tamanho da mancha contida na roupa. A base de *Regras* envolvendo as variáveis de entrada (grau de sujeira e manchas) e a variável de saída (tempo de lavagem) conta com nove *Regras*, que podem ser visualizadas na tabela 1.

Tabela 1. Base de *Regras* do sistema da máquina de lavar.

	Sem Mancha - SM	**Média Mancha - MM**	**Grande Mancha - GM**
Pequena Sujeira - PS	Muito Curto - MC	Médio - M	Longo - L
Média Sujeira - MS	Curto - C	Médio - M	Longo - L
Grande Sujeira - GS	Médio - M	Longo - L	Muito Longo - ML

Também foram definidos os termos linguísticos que compõem cada variável. A figura 7 demonstra os conjuntos *fuzzy* para cada uma das variáveis linguísticas definidas.

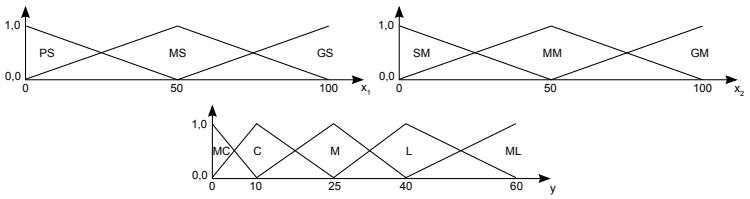


Figura 7. Os conjuntos *fuzzy* para as variáveis de sujeira (esq.), mancha (dir.) e tempo de lavagem (abaixo).

No sistema *fuzzy* desenvolvido utilizando o *framework* PON estendido para suportar conceitos *fuzzy*, as *Regras* também incrementavam o valor de uma variável global que indica quantas *Regras* foram ativadas durante a execução do programa. Para a "defuzzyficação" foi utilizada a função de centroide sobre o conjunto *fuzzy* resultante, sendo que em ambos os sistemas a granularidade

da função foi ajustada em 0,1. Para os testes foi utilizado o caso que consistia em atribuir valores gerados e comparar os resultados entre os dois sistemas. Os valores utilizados foram os seguintes: $[0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]$. Cada um destes valores foram ajustados nas duas variáveis de entrada, resultando na execução do processo de inferência 121 vezes. Em ambos os sistemas foram feitas medições de tempo de execução entre a atribuição do primeiro valor e a finalização da inferência. Para os testes foi considerado o tempo total para a execução de todas as inferências, ou seja, a somatória de todos os tempos medidos.

Durante os testes ambos os sistemas calcularam os mesmos valores para o conjunto de entrada passado. Além disto, foi feita análise da quantidade de ativações das *Regras*. Conforme dito anteriormente, o processo de inferência foi executado 121 vezes, o que significa que o sistema *fuzzy* em C++ analisou as nove regras esta quantidade de vezes, resultando em 1089 verificações. Entretanto, no sistema desenvolvido usando o PON, a execução deste teste resultou na ativação de 277 *Regras*, resultado semelhante aos apresentados em [13]. Apesar deste resultado, o tempo de execução do sistema em C++ foi de 0,366214 segundos enquanto o sistema com o *framework* do PON foi de 1,12508, resultando em uma proporção de tempo de execução de 3,072:1.

6 Conclusão

Com o presente trabalho foi possível verificar a possibilidade de adaptar o paradigma orientado a notificações para que este suporte o desenvolvimento de sistemas de lógica *fuzzy*. As modificações citadas neste trabalho permitiram que o paradigma possa ser utilizado para o desenvolvimento de sistemas *fuzzy*. Além disso, estas modificações permitem que sejam desenvolvidos sistemas mistos, onde *Premissas crisps* e *Premissas fuzzy* podem compor uma mesma *Regra* através dos conectivos lógicos definidos por Mamdani. Também foi possível fazer a validação destas modificações através do desenvolvimento e testes de sistemas. Para tal, o *framework* atual do PON foi modificado para suportar as mudanças citadas no trabalho.

Durante os testes foi verificado que houve uma redução significativa na quantidade de regras avaliadas, o que poderia resultar em uma redução na computação necessária para a execução do caso de teste citado. Isto se deve ao mecanismo de inferência do paradigma, onde o processamento ocorrerá apenas quando houver uma mudança de estado no sistema. Porém foi verificado que, apesar da quantidade reduzida de regras verificadas, o tempo de execução do sistema desenvolvido com o *framework* do PON foi três vezes maior que o do sistema desenvolvido em C++. Isto ocorreu pois o *framework* foi desenvolvido para simular o paralelismo do PON em um ambiente de execução sequencial. Para solucionar este problema, a próxima etapa do trabalho será realizar um estudo para implementar o paradigma em um ambiente com processamento altamente paralelizado.

Referências

1. Peters, E., Jasinsk, R.P., Pedroni, V.A., Simão, J.M.: A new hardware coprocessor for accelerating notification-oriented applications. In: Proceedings in 2012 International Conference on Field-Programmable Technology, Seoul (2012) 257 – 260
2. Zadeh, L.A.: Fuzzy sets. Information and Control **8** (1965) 338–352
3. Sánchez-Solano, S., Cabrera, A.J., Baturone, I., Moreno-Velo, F.J., Brox, M.: Fpga implementation of embedded fuzzy controllers for robotic applications. Industrial Electronics, IEEE Transactions on **54**(4) (2007) 1937–1945
4. Banaszewski, R.F.: Paradigma orientado a notificações - avanços e comparações. Master's thesis, Universidade Tecnológica Federal do Paraná (UTFPR), Curitiba, PR (2009)
5. Simão, J.M., Stadzisz, P.C.: Paradigma orientado a notificações (pon) - uma técnica de composição e execução de software orientado a notificações (2008) Patent pending submitted to INPI/Brazil in 2008 and UTFPR Innovation Agency.
6. Simão, J.M., Tacla, C.A., Banaszewski, R.F., Stadzisz, P.C.: Notification oriented paradigm (nop) and imperative paradigm: A comparative study. Journal of Software Engineering and Applications (JSEA) (2012)
7. Fabro, J.A.: Grupos neurais e sistemas nebulosos: Aplicação à navegação autônoma. Master's thesis, Universidade Estadual de Campinas (UNICAMP), Campinas, SP (1996)
8. Ross, T.J.: Fuzzy logic with engineering applications. Third edn. John Wiley & Sons, Ltd. (2010)
9. da Silva Delgado, M.R.D.B.: Projeto Automático de Sistemas Nebulosos: Uma Abordagem Co-Evolutiva. PhD thesis, Universidade Estadual de Campinas (UNICAMP), Campinas, SP (2002)
10. Tanscheit, R.: Sistemas fuzzy. Inteligência computacional: aplicada a administraçao, economia e engenharia em Matlab (2004) 229–264
11. Mamdani, E.: Application of fuzzy algorithms for control of simple dynamic plant. Proceedings of the Institution of Electrical Engineers **121**(12) (1974) 1585
12. Simão, J.M., Fabro, J.A., Stadzisz, P.C., Arruda, L.V.R., Ishimatsu, S.: An agent-oriented fuzzy inference engine. In: Proceedings in VI Simpósio Brasileiro de Automação Inteligente, Bauru, SP (2003) 585–590
13. de Souza, J.T.S., Fabro, J.A., Simão, J.M., Banaszewski, R.F.: Proposta de uma máquina de inferência fuzzy utilizando o paradigma orientado a notificações (pon). In: XIV Seminário de Iniciação Científica e Tecnológica da UTFPR, Pato Branco, PR (2009)
14. Ronszcka, A.F.: 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 (UTFPR), Curitiba, PR (2012)

C.2 Artigo publicado na revista MSC

Este artigo foi publicado na revista *Mathware & Soft Computing Magazine* (MSC), que é uma revista publicada pela EUSFLAT (*European Society For Fuzzy Logic And Tecnology*). A artigo foi publicado na edição de junho de 2015, volume 22, número 1.

Adaptation of the Notification Oriented Paradigm (NOP) for the Development of Fuzzy Systems

Luiz Carlos Viana Melo, Jean Marcelo Simão*, and João Alberto Fabro

Graduation Program on Applied Computing (PPGCA)
Graduation Program on Electrical Engineering and Computer Science (CPGEI)*
Federal University of Technology - Paraná (UTFPR)
Av. Sete de Setembro, 3165 – 80230-901 – Curitiba – PR – Brazil
luizmelo@alunos.utfpr.edu.br,jeansimao@utfpr.edu.br,
fabro@utfpr.edu.br
http://ppgca.ct.utfpr.edu.br/

Abstract. The Notification Oriented Paradigm (NOP) combines and evolves the event based programming with the declarative programming in order to solve some problems of both paradigms. Breaking down one application into a network of smaller computational entities, such as logic-causal and factual notifier entities that processes only when needed, the NOP eliminates the need to perform unnecessary computations and improves the code reusability. Fuzzy systems, in turn, perform inference based on knowledge bases (IF-THEN rules) that can cope with problems involving imprecision. Since NOP uses IF-THEN rules in an alternative way, by reducing evaluation redundancy and coupling, this research was conducted to identify, propose and evaluate the needed changes on the NOP to allow its use in the development of fuzzy systems. The tests results showed a significant reduction in the number of evaluated rules, which may represent improvement in performance of the applications.

Keywords: Notification Oriented Paradigm, Fuzzy Systems, Rule Based Systems

1 Introduction

The Notification Oriented Paradigm (NOP) was proposed by Simão [1] as a control solution that later evolved into a programming paradigm. NOP sought inspiration from both imperative and declarative paradigms, more precisely from the object oriented and logic sub-paradigms, at same time that it aims to solve some of their deficiencies [2].

NOP uses the main advantages of the declarative paradigm, namely the expressiveness of causal rules from rule based systems, which has abstraction and language closer to the form of human cognition. It also uses the advantages of the imperative paradigm, namely the code reusability, flexibility and abstraction

through classes and objects from the object oriented sub-paradigm [2]. However, NOP presents a new form of inference, which differs from current paradigms.

The main idea behind NOP is the way the software detects changes in variables and makes inferences about it. In the current imperative paradigms, there are two forms to detect changes in values of a variable: through polling or event notification. In polling, the program loop performs successive evaluations of the system variables, performing some logic evaluations on the values and triggering actions when certain conditions are met. This approach is considered sequential because only one condition is checked at a time in a given thread. Due to the fact that the loop is executed even when the variables do not change their values, this approach wastes computing resources [3].

An alternative to polling is event oriented programming. In this approach, any processing depends on events. Events can be triggered by user's actions or other situations that can provoke changes in the internal state of the program. In some approaches this eliminates the need to have a loop that checks the states of variables, which reduces the unnecessary computing. However, this alternative makes the application development usually more complex, resulting in bigger programs. In addition, given the hardware constraints, the event controller system may perform polling to dispatch events [3].

In the case of declarative programming, the programmer must focus on what a program should accomplish instead of how it should be accomplished. This frees the programmer from handling many unimportant details. However, they are usually slower to execute and less flexible than imperative programming [3,4].

Having these problems as motivation, NOP combines and evolves ideas of both event based and the declarative programming in order to solve them. Actually, NOP eliminates the need to perform unnecessary computing and enhance modular decoupling, thereby facilitating code reusability for example. NOP achieves this by breaking down one application into a network of smaller computational notifier entities that are executed only when needed [5].

Besides, in the past, some preliminary studies were conducted to evaluate if it would be possible to extend the NOP paradigm to fuzzy systems [6,7]. These studies were motivated due to the fact that in both NOP and fuzzy systems the knowledge can be described in the IF-THEN rules format such as those used in natural language. Those preliminary studies used as case studies applications in the field of robotics.

The results of the previous studies on fuzzy NOP demonstrated a significant reduction in the number of evaluated rules, which could improve the application performance.

However, they did not describe in details the changes performed, which makes difficult to replicate them in others materialization of the paradigm. Consequently, this ended up causing the lack of support to the development of fuzzy systems by current implementations of NOP. Also the results in terms of processing time were not good due to the style of materializations of NOP at the time, i.e. frameworks in C++ with expensive data structures.

In addition, many works on fuzzy systems use embedded systems to implement their concepts [8, 9]. Due to the restrictions of processing, memory and energy consumption, it would be interesting to find a way to optimize these factors. NOP can be an alternative to this since it solves these problems through the use of passive entities that will be processed only when necessary.

In the following sections, this paper presents the related concepts of NOP and fuzzy systems, the modifications performed on the paradigm to provide fuzzy concepts and inference, and a simple application, to evaluate its applicability.

2 Notification Oriented Paradigm (NOP)

This paradigm basis was first proposed as a control mechanism to supply the needs related to modern production systems [2]. Later, the author realized that this model could be applied on many problem domains. Therefore, he proposed and adjusted the mechanism to provide a general solution for discrete control. In addition, he also realized that the model could be used to guide programmers in the conception of applications, resulting in the so-called Notification Oriented Paradigm (NOP) [4]. This paradigm is detailed in the following subsections.

2.1 Paradigm structure

NOP introduces a set of new concepts that can be applied to design, build and execute computer programs. The main concept is the use of small, active and decoupled entities that collaborates by means of notifications in order to perform the logical and causal calculation existing in the software [2, 10]. The knowledge in the paradigm is represented by causal rules and factual base elements that are naturally understood by programmers of the current paradigms [2].

The entities that compose the paradigm are as follows:

- *Fact Base Element* (FBE): is an entity of the observed system. It aggregates the *Attributes* that represents the facts about the cited entity and the *Methods* that allow for the execution of functionalities associated to this element;
- *Attribute*: is a value of the FBE that represents one of its features, composing its state;
- *Premise*: is a logical inquiry between an *Attribute* (that belong to some FBE) and a value (for instance, "Is X equal to 2 ?"). A *Premise* is composed of an *Attribute*, a comparison operator, and a third element, that can be a constant value, or another *Attribute*;
- *Condition*: is a logical relationship between the *Rule's Premises*. This relationship is usually denoted by the conjunction (AND) and disjunction (OR) logical connectors, or a combination of both;
- *Rule*: is a rule in the system's rule set; this entity associates a *Condition* to an *Action*. The relationship implemented by this entity is casual implication, which can be read as "IF the *Condition* is active (antecedent), THEN activate the *Action* (consequent)";

– *Action*: is an action to be executed when the associated *Rule* was been approved. This element contains a set of *Instigations* that must be processed;
– *Instigation*: is the activity that induces the execution of a *Method* over a FBE;
– *Method*: is a FBE's method, that can perform changes over *Attributes*.

Figure 1 displays the class diagram of the cited entities and the relationship among them.

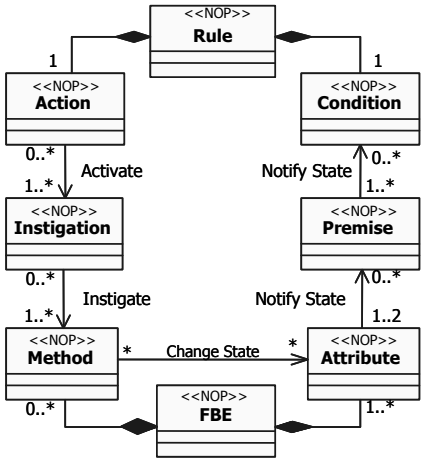


Fig. 1. Paradigm entities represented in a class diagram [2].

The collaboration between the entities is the basis for the paradigm inference mechanism, and the collaboration between them is done through an exchange of punctual notifications. The following section shows the paradigm notification mechanism by displaying the role of each entity.

2.2 NOP notification engine

The notification engine is the internal process to execute the NOP instances, which determines the application execution flow. Through this mechanism, the program tasks are split among the entities, which cooperate through notifications telling each other about the share of their contributions to form the application execution flow [4]. Figure 2 shows an example of the notification chain.

The inference process starts when the state (value) of an FBE's *Attribute* is changed (e.g. FBC.1 and FBC.n, seen in figure 2, left side). Each change causes the *Attribute* to notify the interested *Premises*. The *Premises*, after receiving the notification, perform logical calculation to determine if their state have to change (from *false* to *true*, for example). This calculation is done by comparing the *Attribute* value with the *Premise* value using a comparison operator.

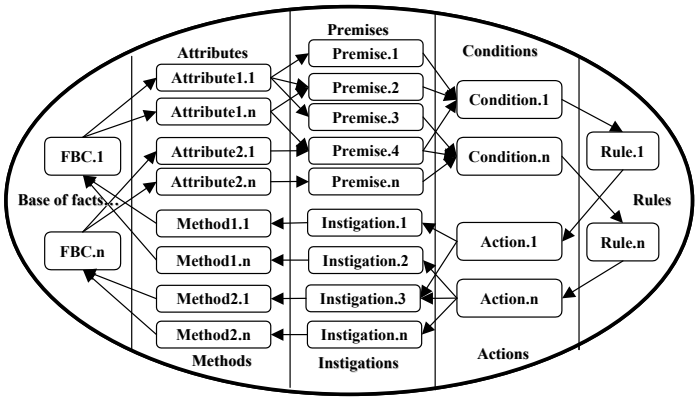


Fig. 2. Paradigm notification chain [10].

Using the same notification procedure, *Premises* that have changed their logical values notify the interested *Conditions*. Upon receiving the *Premises* notifications, the *Conditions* do their logical calculation from a causal logical expression. Each *Conditions* defines the relationship between the notifying *Premises* by the use of logical connectives ("AND" and "OR"). An expression can be read as the following example: "Premise1 activated 'AND' Premise2 activated". If the *Condition* state changes, it will notify the *Rule* to which it is associated.

The *Rule*, after receiving a notification from the *Condition*, will check if the associated *Condition* is not in conflict[1]. If this is the case, the *Rule* will notify the associated *Action*. An *Action* will perform a number of instigations through the *Instigation* entities within it. In the last step of the notification cycle, the *Instigations* will cause the FBE *Methods* to be executed, which can alter the FBE *Attributes* values, starting a new cycle of Notification Oriented Inference [13].

Through this notification engine the NOP entities will only execute after receiving a notification from another entity, which may results in resources saving and speed up in application performance [2, 4].

In the next section, a brief introduction of the main concepts of fuzzy systems is presented. In the following sections, the adaptation of the NOP paradigm to handle with fuzzy concepts and inference mechanisms is proposed and detailed.

3 Fuzzy systems

The studies on fuzzy systems started in 1965 when Zadeh [14] proposed his fuzzy sets theory. This theory generalizes the classical set theory [15], allowing the representation of concepts which cannot be represented using well defined (or crisp) limits.

A fuzzy set is the one containing the elements of the universe of discourse which has variable membership degree to this set. The equation $\mu_A(x) \in [0, 1]$

[1] There is mechanisms to solve conflicts as detailed in [11] and [12]

represents the mathematical notation of the membership degree μ_A of an element x in a set A, where the value "1.0" indicates the complete compatibility of an element with the concept represented by the set, and the value "0.0" indicates there isn't any compatibility of the element with the set.

Another concept related with fuzzy sets is the linguistic variable, which represents an identifier that can assume one of several values. Thus, a linguistic variable can take a linguist value which represents a fuzzy set.

Fuzzy logic can be used to develop control systems that deal with imprecise information. Through a set of rules, it is possible to perform inferences that will be used in the decision making process of a system. In the most practical applications the input data is composed of values provided by sensors, that result from measurements and observations. For the use of such data by fuzzy inference systems, it is required to perform a mapping of those measurements into relevant activation of fuzzy sets [16]. This mapping is called "fuzzyfication" and results in a value of membership degree of the data to each fuzzy set.

The generated sets are used to perform the fuzzy inference. The inference step is processed through the comparison of the rules stored in the rule database. This process results in the calculation of the activation level of each rule. After the calculations the results are spread through the fuzzy rules, producing fuzzy sets that represent the consequents of each rule. The contribution of each rule (fuzzy set) is taken into account through, for instance, the union of the fuzzy sets. Beyond this point the user already has the result of the system in the format of fuzzy sets.

Once the output fuzzy set is obtained by the inference process, in the "defuzzyfication" stage it is converted into a crisp value using a defuzzyfication function like the centroid function. This is necessary because usually in practical applications exact outputs are required, not fuzzy ones [16].

The following section details the changes made in the NOP implementation to perform inferences based in fuzzy rules and concepts. Since originally NOP accepts only "crisp" activations (i.e. each NOP entity can have only one of two activation values, "true" or "false"), several modifications were necessary to allow the propagation of fuzzy activations.

4 Changes made over NOP to provide fuzzy inference capabilities

Several modifications were required in the paradigm implementation in order to allow fuzzy logic systems development. The proposed modifications where applied over the current implementation of the NOP Framework in C++ language, as presented in [17], and draw inspiration from the first fuzzy implementation developed in previous work [6].

4.1 Changes in the representation of logical state of the entities

Paradigm entities like *Premises*, *Conditions* and *Rules* have each one a logical state that is spread through notifications. This logical state has, in the originally

proposed paradigm, only two possibles values: *true*(active) or *false* (inactive), represented by the integer values in $\{0, 1\}$. The value 0 (zero) indicates that the entity is not active, and 1 (one) indicates that it is active.

To extend the representation to support fuzzy logic, the mentioned entities must represent the logical state by their activation level. The activation level can be represented as real values in the range $[0, 1]$, representing the membership degree as shown in section 3. The notification process is started when the activation level of any entity changes. These changes to the original NOP implementation could only be from inactive to active, or vice-versa, but with the extension to fuzzy activations, any change in the value can start the notification. This activation level is calculated differently for each entity as described in the following sub-sections.

4.2 Linguistic variable representation

The linguistic variable was implemented as an extension of an NOP *Attribute*, as shown in figure 3. This new element derived from the *Attribute* by C++ class inheritance has a list of fuzzy sets that represents the linguistic terms of the variable. Each set is represented by a membership function used to calculate the membership degree of a value to that set.

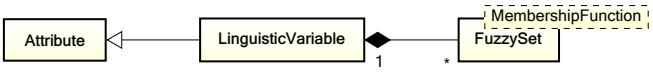


Fig. 3. Definition of element "Linguistic Variable" in the NOP paradigm.

By assigning a value to this variable, it will behave as an *Attribute* and will notify the *Premises* if the assigned value is different from current value. The membership degree calculation for a given fuzzy set performed by each *Premise* as described in the section 4.3.

4.3 Changes in the calculation of the *Premises* logical state

Due the changes described in section 4.1, the *Premise* logical state is now represented by activation levels. This calculation depends on the *Premise* type. The *Premises* types are:

- Common (or crisp): *Premise* in which all logical operations are performed on non-fuzzy *Attributes*. The result of this operation will determine the *Premise* activation ("true" or "false" , represented by either 0.0 or 1.0);
- Fuzzy *Premise*: *Premise* that calculates the membership degree of the *Attribute* value in the fuzzy set associated with that *Premise*. The calculated membership degree defines the *Premise* activation level (a real number between 0.0 and 1.0).

In the case of fuzzy *Premise* the activation level will be the result of the calculation of the membership degree for the value stored in the *Attribute* to the fuzzy set referred by the *Premise*.

4.4 Readjustment of the *Condition* logical expression

The change in the *Condition* consisted in the creation of a model that represent the *Conditions* causal expressions. The model is presented in figure 4.

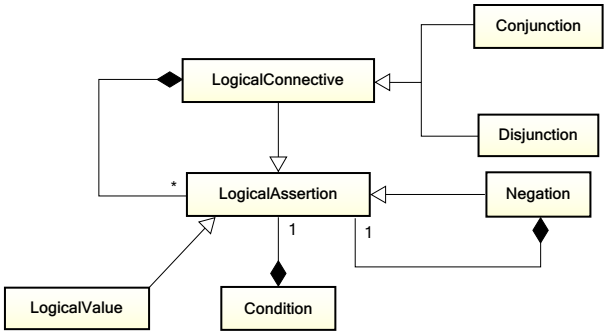


Fig. 4. *Condition* casual expression model.

In this model the *Premise* is associated to an expression with a logical value. This value is updated when the *Premise* notifies the *Condition*, and the activation level of the *Premise* are then used as input for the operations with the logical connectives defined in the model. The notification also starts the calculation of the activation level of this *Condition*.

The calculation is performed using the definitions of the operations defined by Mamdani in [18] as follows:

- Intersection $(\mu_{(A\cap B)}(x) = min[\mu_A(x), \mu_B(x)])$: logical connective **AND**;
- Union $(\mu_{(A\cup B)}(x) = max[\mu_A(x), \mu_B(x)])$: logical connective **OR**;
- Complement $(\mu_{(\bar{A})}(x) = 1 - \mu_A(x))$: negation logic operation.

The use of the mentioned operations allows the characteristics of the boolean logic to be "emulated", allowing for the use of *Rules* with both fuzzy and non-fuzzy *Premises*.

4.5 Changes in the active part of the inference process

With the changes mentioned in section 4.1 a *Rule* activates when there is a change in the *Condition* state. This also reflects in the *Instigations* since they are activated along the *Rules*.

In this work, it was defined that the *Instigations* now have two forms: the common *Instigation* and the fuzzy *Instigation*. The common *Instigation* represents the current definition of the paradigm *Instigation* and it maintains the current paradigm behavior. These will only activate when the *Rule* activation level is equal to one, otherwise the *Instigation* is not activated.

The fuzzy *Instigation* is activated at every change in the *Rule* activation level which it is related to. This activation executes the *Methods* associated to it, fuzzy or not. If the fuzzy *Instigation* involves the execution of a fuzzy *Method*, the *Rule* activation level is passed to it.

The fuzzy *Method* was created to represent the consequent of fuzzy logic rules. It associates the *Attribute* to the fuzzy set and performs the "defuzzyfication" operation. The operation result can be assigned to an *Attribute*, which can then trigger a new notification cycle.

With those changes, NOP can now allow the use of fuzzy versions of its entities, providing all of the necessary support for implementation of complete systems based in fuzzy logic and even allowing the composition of rules with fuzzy and non-fuzzy elements transparently. In the next section, some developed experiments to evaluate these modifications are presented.

5 Tests and results

Tests were performed using the implementation of paradigm in the form of a C++ framework defined in [17], after its adjusts to support fuzzy concepts. Comparisons were performed with a system developed in C++ without using NOP. The system consists of a simulated washing machine controller, that defines the wash time according to the amount of dirt and the size of stain in clothes. It's rules and variables were provided by a fuzzy system expert as an example, not representing a real system. The Fuzzy Rulebase relating the input variables and the output variable has nine *Rules* as shown in table 1.

Table 1. Washing machine controller rulebase (Input variables: Dirt level and Stain sizes; Output variable: Wash Time).

	No Stain - NS	**Medium Stain - MS**	**Big Stain - BS**
Little Dirt - LD	Very Short - VS	Average - A	Long - L
Average Dirt - AD	Short - S	Average - A	Long - L
Much Dirt - MD	Average - A	Long - L	Very Long - VL

Figure 5 shows the fuzzy sets for each defined linguistic variable.

In the developed fuzzy system using the NOP framework, extended to support the fuzzy concepts, the *Methods* executed by the activated *Rules* also increments the value of a global variable that indicates how many *Rules* where activated during the program execution. For "defuzzyfication" it was utilized the centroid function over the resulting fuzzy set and the function granularity was adjusted to 0.1 in both systems.

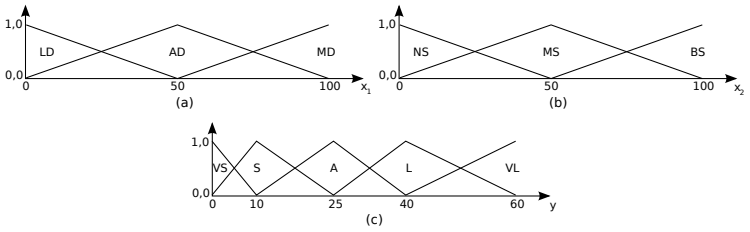


Fig. 5. The fuzzy sets for the dirt (a), stain (b) and wash time (c) variables.

The test case consists in assigning artificially generated values and comparing the results of both systems. The following code was used in the tests:

```
for i = 0 to 100 inc 10
        for j = 0 to 100 inc 10
                dirt = i
                stain = j

                perform inference

                write execution time in file
        end
end
```

This program resulted in the execution of the inference process 121 times. In both systems (fuzzy C++ and fuzzy NOP), measurements of the execution time between the assignment of the first value and the completion of inference were performed. For the tests, it was considered the total execution time of all inferences, that is, the sum of all measured times.

During the tests both systems calculated the same values for the passed input data set. Furthermore, an analysis of the amount of the *Rules* activated was performed. As the inference process was executed 121 times and the C++ fuzzy system analyzed the nine rules this amount of times, this resulted in 1089 verifications. However, in the developed NOP system, the execution of this test resulted in activation of only 277 *Rules*.

Despite this result, the C++ system average execution time was 0.366214 seconds while the NOP system developed with the modified framework was 1.12508 seconds, resulting in an execution ratio of 3.072:1. This was caused by the framework overhead since it was developed to simulate the NOP parallelism in a sequential execution environment by processing one layer of NOP entities (*Attributes*, *Premises*, ...) at each time. Furthermore, the framework was not optimized to use multiple threads.

6 Conclusion

With this work it was possible to verify the possibility of adapting the notification oriented paradigm to support the development of fuzzy logic systems. The changes mentioned in this work allowed the paradigm to be used for an

proof-of-concept fuzzy system development. Furthermore, those changes allow the development of mixed systems, where crisp *Premises* and fuzzy *Premises* can compose the same *Rule* through the use of the logical connectives defined by Mamdani, and this can be seen as a contribution of this paper.

During the tests it was verified that there was a reduction in the amount of evaluated rules, which could result in reduction of necessary computation to execute the mentioned test case. This is due the inference mechanism of the paradigm, where the processing will occur only if there's a change of state in the system. However, it was also verified that, although the reduced amount of verified rules, the execution time of the system developed with the fuzzy NOP framework was three times bigger than the system developed directly in C++. This happened because the framework was developed to simulate the NOP parallelism in a sequential execution environment.

The next step of this work is to extend the NOP language [5], which is already under development, to include the proposed fuzzy concepts and structures. This new language will allow developers to easily create fuzzy systems, which will help in creation of new test cases and in improvement of the fuzzy NOP framework, in order to reduce its overhead.

References

1. Simão, J.M.: A Contribution to the Development of a HMS Simulation Tool and Proposition of a Meta-Model for Holonic Control. PhD thesis, Universidade Tecnológica Federal do Paraná (UTFPR), Curitiba, PR (2005)
2. Simão, J.M., Tacla, C.A., Banaszewski, R.F., Stadzisz, P.C.: Notification oriented paradigm (nop) and imperative paradigm: A comparative study. Journal of Software Engineering and Applications (JSEA) (2012)
3. Peters, E., Jasinsk, R.P., Pedroni, V.A., Simão, J.M.: A new hardware coprocessor for accelerating notification-oriented applications. In: Proceedings - International Conference on Field-Programmable Technology, Seoul (2012) 257 – 260
4. Banaszewski, R.F.: Notification oriented paradigm - advances and comparisons (in portuguese). Master's thesis, Universidade Tecnológica Federal do Paraná (UTFPR), Curitiba, PR (2009)
5. Xavier, R.D.: Software development paradigms: comparing event-driven and notification oriented approaches (in portuguese). Master's thesis, Universidade Tecnológica Federal do Paraná (UTFPR), Curitiba, PR (2014)
6. Simão, J.M., Fabro, J.A., Stadzisz, P.C., Arruda, L.V.R., Ishimatsu, S.: An agent-oriented fuzzy inference engine. In: VI Simpósio Brasileiro de Automação Inteligente, Bauru, SP (2003) 585–590
7. de Souza, J.T.S., Fabro, J.A., Simão, J.M., Banaszewski, R.F.: Proposal of a fuzzy inference engine using notification oriented paradigm (nop) (in portuguese). In: XIV Seminário de Iniciação Científica e Tecnológica da UTFPR, Pato Branco, PR (2009)
8. Sánchez-Solano, S., Cabrera, A.J., Baturone, I., Moreno-Velo, F.J., Brox, M.: Fpga implementation of embedded fuzzy controllers for robotic applications. Industrial Electronics, IEEE Transactions on **54**(4) (2007) 1937–1945

9. Sulaiman, N., Obaid, Z.A., Marhaban, M.H., Hamidon, M.N.: FPGA-Based Fuzzy Logic : Design and Applications - a Review. IACSIT International Journal of Engineering and Technology **1**(5) (2009) 491–503
10. Simão, J.M., Stadzisz, P.C.: Notification oriented paradigm (nop) - a composition technique and notification oriented software execution (in portuguese) (2008) Patent pending submitted to INPI/Brazil in 2008 and UTFPR Innovation Agency.
11. Simão, J.M., Stadzisz, P.C., Banaszewski, R.F., Tacla, C.A.: Optimized inference engine for notification oriented paradigm (nop) and mechanisms for conflict resolution to uniprocessor and multiprocessor environments applied to nop (in portuguese) (2010) Patent pending submitted to INPI/Brazil in 2010 and UTFPR Innovation Agency.
12. Simão, J.M., Stadzisz, P.C.: Conflict resolution mechanism and determinism warranty for notification oriented paradigm (nop) (in portuguese) (2010) Patent pending submitted to INPI/Brazil in 2010 and UTFPR Innovation Agency.
13. Simao, J., Stadzisz, P.: Inference based on notifications: A holonic metamodel applied to control issues. Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on **39**(1) (Jan 2009) 238–250
14. Zadeh, L.A.: Fuzzy sets. Information and Control **8** (1965) 338–352
15. Pedrycz, W., Gomide, F.: An Introduction to Fuzzy Sets: Analysis and Design. MIT Press (1998)
16. Ross, T.J.: Fuzzy Logic with Engineering Applications. John Wiley & Sons (2004)
17. Ronszcka, A.F.: Contribution to the design of applications in notification oriented paradigm (nop) under the perspective of patterns (in portuguese). Master's thesis, Universidade Tecnológica Federal do Paraná (UTFPR), Curitiba, PR (2012)
18. Mamdani, E.: Application of fuzzy algorithms for control of simple dynamic plant. Proceedings - Institution of Electrical Engineers **121**(12) (1974) 1585–1588