Introdução ao teste funcional de software no Paradigma Orientado a Notificações

Clayton Kossoski[1], Jean Marcelo Simão[1], Paulo Cézar Stadzisz[1]

claytonkssk@gmail.com, jeansimao, stadzisz {@utfpr.edu.br}

Universidade Tecnológica Federal do Paraná – UTFPR
[1] Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial – CPGEI

Resumo: *O Paradigma Orientado a Notificações (PON) para desenvolvimento de software surgiu como uma solução que aplica os melhores conceitos do Paradigma Imperativo (PI) e Paradigma Declarativo (PD), ao mesmo tempo em que objetiva resolver suas principais deficiências no tocante ao cálculo lógico-causal. Entre as atividades envolvidas no desenvolvimento de software, a atividade de testes possui grande relevância e deve ser feita durante todo o desenvolvimento de software. Sistemas mal testados estão mais propensos a conter erros gerando retrabalho e custos mais elevados. Entretanto, a atividade de testes ainda não foi explorada suficientemente no PON. Assim, neste trabalho é apresentada uma introdução ao teste de software em PON. Primeiramente, é realizada uma breve revisão dos conceitos sobre um dos principais métodos de teste do PI, como o teste funcional de software, também conhecido como* caixa-preta*. Subsequentemente, discute-se a aplicação dos testes funcionais em PON, com a apresentação de exemplos e casos de teste. Os resultados preliminares apresentados indicam que é possível aplicar o teste funcional em PON. Trabalhos futuros deverão estudar e apresentar outros critérios de teste do PI e do PD, como o teste estrutural, teste baseado em erros (como teste de mutação) e verificação e validação com redes de Petri, abrindo caminho para mais pesquisas sobre o assunto.*

Palavras chave: Paradigma Orientado a Notificações (PON), Teste de Software, Teste Funcional de Software.

Abstract: *The Notification Oriented Paradigm (NOP) for software development emerged as a solution that uses the best concepts of the Imperative Paradigm (IP) and Declarative Paradigm (DP) and help solving their main deficiencies with regard to the logical-causal calculation. Among the activities involved in the software development process, test has an important impact and must be carried out during the entire development process. Poorly tested systems are more likely to contain errors, generating rework and higher costs. However, testing activity has not been sufficiently explored in NOP. Thus, in this paper, an introduction to software testing in NOP is presented. First, it held a brief review of the main concepts on one of the main Methods of testing in IP, the functional testing, also known as black-box testing. The paper also discusses the application of functional test in NOP, presenting samples and test cases. Preliminary results presented indicate that it is possible to apply functional testing in NOP. Future works should study and apply other test criteria of IP and DP, such as structural testing, based on errors testing (mutation testing) and verification and validation with Petri nets, opening way for further research on this subject.*

Keywords: Paradigm Oriented Notices (PON), Software Testing, Functional Test Software.

1. Introdução

A atividade de desenvolvimento de software envolve intenso esforço desde sua concepção, com o estabelecimento de requisitos, modelagem do sistema, construção da arquitetura, programação e testes. Em todas estas etapas da construção de um software há a possibilidade de ocorrerem erros [1].

Os erros podem propagar-se para atividades futuras, acarretando problemas, resultados incorretos, estados inconsistentes e, por consequência, o *software* pode não realizar a tarefa que deveria com precisão. Por isto, a atividade de testes é tão importante e é necessário que seja feita a verificação e validação dos artefatos pertinentes ao desenvolvimento de um sistema, desde os requisitos até as informações processadas e apresentadas pelo *software*.

Na verdade, esta atividade de testes é importante independentemente das facilidades de desenvolvimento do paradigma escolhido, salientado aqui o Paradigma Orientado a Objetos (POO), que faz parte do Paradigma Imperativo (PI), e o Paradigma Declarativo (PD) [2]. Isto inclui, ademais, os paradigmas emergentes como o chamado Paradigma Orientado a Notificações (PON).

O PON baseia-se no conceito de pequenas entidades dissociadas que colaboram por meio de notificações precisas para realizar a inferência de *software* [3][4]. Isto permite melhorar o desempenho e compor aplicações mais facilmente tanto em ambientes distribuídos, quanto em não distribuídos [4].

Como em qualquer outro paradigma de desenvolvimento de software, a atividade de teste possui também grande impacto em PON. Na literatura, principalmente, vários trabalhos investigam o teste funcional [5] [6] e o teste estrutural [7] [8] [9] [10]. No entanto, no contexto do PON, estes e outros tipos de testes não têm sido ainda explorados suficientemente. Basicamente, os testes têm sido aplicados sem considerar as especificidades do PON, como sua organização em forma de regras.

Neste trabalho, é apresentada uma introdução ao teste de *software* em PON com base em critérios de teste funcional utilizados por outros paradigmas.

Este artigo é estruturado da seguinte forma: a Seção 2 sucintamente apresenta o estado da arte do PON, Seção 3 apresenta os principais conceitos do o teste de software, a Seção 4 introduz teoricamente o teste funcional de software em PON. A Seção 5 apresenta um caso de estudo de aplicação do teste funcional em PON. A Seção 6, por fim, apresenta as conclusões e perspectivas de trabalhos futuros.

2. Paradigma Orientado a Notificações (PON)

O Paradigma Orientado a Notificações surgiu como uma solução que usa os melhores conceitos do Paradigma Imperativo e Paradigma Declarativo, ao mesmo tempo em que objetiva resolver suas deficiências no tocante ao cálculo lógico-causal. O PON propõe que toda a inferência ocorra por meio de entidades notificantes mínimas e colaborativas que tratam de processamento factual, lógico e causal [3] [11] [12].

Com isso, o PON apresentaria resposta a vários problemas desses paradigmas, como repetição de expressões lógicas e reavaliações desnecessárias (i.e. redundâncias estruturais e temporais) e, particularmente, o acoplamento forte de entidades com relação às avaliações ou cálculo lógico-causal [13].

O estado da técnica do PON atualmente é um *framework* em C++, enquanto uma linguagem e um compilador especializados estão em desenvolvimento. Este *framework* foi projetado para fornecer uma *Application Programming Interface* (API) e estruturas que facilitam o desenvolvimento de software segundo a orientação do PON [12] [14].

2.1. Mecanismo de Notificações do PON

Uma aplicação desenvolvida em PON é constituída por um conjunto de Regras (*Rules*) e Entidades Factuais (*FBE – Fact Base Element*). A dinâmica da execução da aplicação PON se dá pela mudança de estado dos *FBEs* que provocam um fluxo de notificações por meio das regras que levam o sistema a reagir e alcançar novos estados.

O fluxo de iterações das aplicações do PON é realizado de maneira transparente ao desenvolvedor, graças ao funcionamento da cadeia de notificações pontuais entre as entidades PON. Desta forma, a inferência ocorre de maneira diferente do fluxo de iterações encontrado em aplicações do PI, como o POO, nas quais o desenvolvedor informa de maneira explícita o laço de iteração através de comandos como *while* e *for* [13]. A Figura 5 esboça o diagrama de classes conceitual do PON.

A classe *FBE* representa entidades do sistema que juntas representam seu estado e comportamento. Quanto há uma mudança de estado no sistema, um objeto *Attribute* de um determinado *FBE* sofrerá mudança de seu valor (estado). Neste momento ele notifica todas as *Premisses* a ele relacionadas, para que estas reavaliem seus estados lógicos. Se o valor lógico da *Premise* é alterado, a *Premise* colabora com a avaliação lógica de uma ou de um conjunto de *Conditions* relacionadas, o que ocorre por meio da notificação sobre a mudança de seu estado lógico a elas [11].

Na sequência, cada *Condition* notificada avalia o seu valor lógico de acordo com as notificações da *Premise* e com o operador lógico (de conjunção ou disjunção) utilizado. Então, no caso de uma conjunção, quando todas as *Premisses* que integram uma *Condition* são satisfeitas, a *Condition* também é satisfeita, resultando na aprovação da sua respectiva *Rule* para a execução [11][12].

Quando uma *Rule* aprovada está pronta para executar, a sua *Action* é ativada. Uma *Action* é conectada a um ou vários *Instigations*. Os *Instigations* notificados pelas *Actions* acionam a execução de algum serviço de um objeto *FBE* por meio dos *Methods*. Normalmente, as chamadas para os *Methods* também alteram os estados dos *Attributes* e o ciclo de notificação pode recomeçar [11].

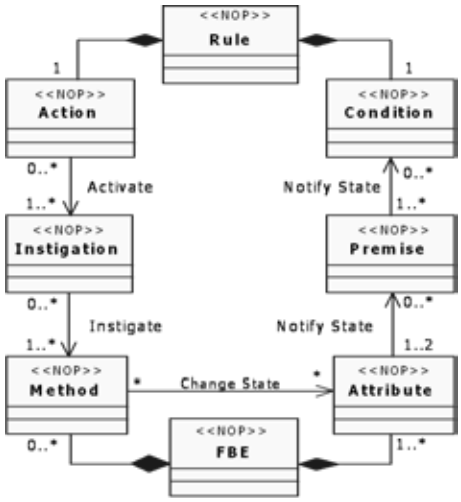

Figura 5 – Diagrama de classes da *Rule* e suas entidades [13]

Cada estado que se altera, calcula o seu valor booleano pela conjunção dos valores das *Premisses*. Quando todas as *Premisses* de uma *Condition* são satisfeitas, a *Condition* também está satisfeita e notifica a respectiva *Rule* para executar. A colaboração entre entidades PON relatadas, por meio de comunicações, pode ser observada no esquema representado na Figura 6. Neste esquema, o fluxo de notificações é representado por setas ligadas a retângulos que simbolizam as entidades PON [13].

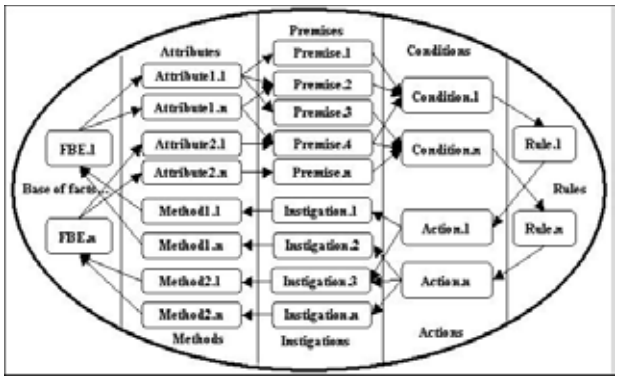

Figura 6 – Cadeia de notificação da *Rule* e de seus colaboradores [13]

A Figura 7 apresenta um exemplo de *Rule*, na forma de uma regra causal. Cada *Rule* é uma entidade computacional composta por outras entidades, conforme ilustrado na Figura 6, que podem ser vistas como objetos e/ou classes.

Por exemplo, a *Rule* apresentada é composta por um objeto *Condition* e um objeto *Action*. A *Condition* trata da decisão da *Rule*, enquanto a *Action* trata da execução das ações da *Rule*. Assim sendo, *Condition* e *Action*

trabalham juntas para realizar o conhecimento lógico e causal da *Rule* [13].

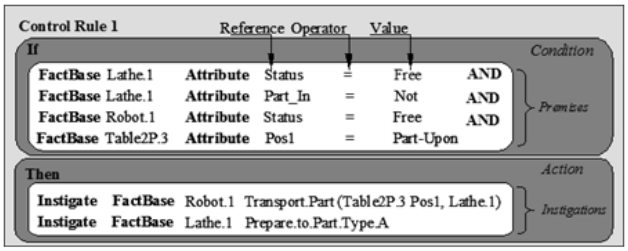

Figura 7 – Exemplo de uma *Rule* em PON [11]

A *Rule* apresentada na Figura 7 faz parte de um controle de manufatura inteligente. A *Condition* desta *Rule* trabalha com a decisão de transporte de peça a partir de uma 'Mesa' (Table) para um 'Torno' (Lathe) utilizando um 'Robô' (Robot). Esta *Condition* apresenta quatro *Premisses* que se constituem em outros tipos de objetos inteligentes. Essas *Premisses* fazem as seguintes avaliações sobre as *FBEs* [11]: a) o Torno está livre? b) há peça no Torno? C) o Robô está livre? e d) há peça na posição 1 da Mesa?

Percebe-se que a essência da computação no PON está organizada e distribuída entre entidades autônomas e reativas que colaboram por meio de notificações pontuais. A natureza do PON leva a uma nova maneira de compor software, na qual os fluxos de execução são distribuídos e colaborativos nas entidades [15].

3. Teste de software

O teste de software é a execução de código usando combinações de entrada e estados selecionados para revelar defeitos [16]. Resumidamente, quatro fases ou tipos de testes são efetuados durante o processo de desenvolvimento do software [17] [18]:

- Teste de Unidade: testa a menor unidade do software (e.g. função, trecho de código ou classe), examinando a estrutura de dados, identificando erros de lógica e cálculo.
- Teste de Integração: teste realizado na integração das unidades ou módulos do software, identificando erros de interface entre os mesmos.
- Teste de Validação: testa o software de acordo com os requisitos do usuário, identificando erros de requisitos informativos, funcionais, comportamentais e de desempenho.
- Teste de Sistema: o software é testado com outros elementos do sistema, verificando erros de funcionamento, tolerância a falhas e segurança. É nesta etapa que o software é testado por completo.

Técnicas e critérios de testes têm sido propostos para nortear a atividade de testes, auxiliando a seleção e avaliação de conjuntos de casos de teste, visando determinar aqueles que aumentam as chances de revelar defeitos [19].

Entre as técnicas de teste existentes, podem ser citadas: a *técnica estrutural*, que deriva casos de teste a partir do código fonte do software; a *técnica funcional*, que usa a especificação funcional do software para derivar casos de teste; e a *técnica baseada em defeitos*, que deriva casos de teste para mostrar a presença ou ausência de defeitos típicos no programa.

O teste funcional não está relacionado com o comportamento interno ou estrutura do software. Em vez disto, concentra-se na busca de circunstâncias em que o programa não apresenta comportamento de acordo com as especificações [17]. As técnicas de teste funcional propõem a derivação de casos de teste por meio de fornecimento de condições de entrada que exercitam completamente todos os requisitos funcionais de um software [18]. O particionamento em classes de equivalência é uma abordagem eficiente para teste funcional porque auxilia o encontro de erros no processamento de entradas no sistema.

Testes funcionais mais comuns que podem revelar defeitos incluem [19]:

- Teste de software com sequências que contém um único valor.
- Uso de diferentes sequências de diferentes tamanhos em testes diferentes.
- Derivar testes de modo que o primeiro, o intermediário e o último elemento da sequência esperada sejam testados.

Neste sentido, com o surgimento de novos paradigmas de desenvolvimento de sistemas, salientando aqui o PON, tem-se igualmente a necessidade de verificar e validar os programas para que sejam confiáveis e alcancem o resultado esperado [1]. Dado que o PON foi concebido usando inclusive alguns elementos do Paradigma Orientado a Objetos e dos Sistemas Baseados em Regras (SBR), os procedimentos de teste para software em PON podem ser calcados nas proposições existentes na literatura. Entretanto, os detalhes do paradigma PON deverão ser considerados.

4. Teste funcional em PON

O teste funcional em um programa PON pode ser efetuado em todas as fases de testes, contudo, neste artigo, é dada atenção especial para as fases de teste de unidade e de teste de integração.

Os passos genéricos para a realização de testes em PON são:

- Entender as funcionalidades requeridas elencadas no documento de requisitos;
- Compreender o fluxo de execução de notificações do PON;
- Preparar um ambiente de teste e determinar todas as suas dependências;
- Preparar planos de teste;
- Executar os casos de teste;
- Confrontar os resultados obtidos com os esperados;
- Informar os casos que passaram e os que falharam.

As subseções a seguir irão apresentar os conceitos teóricos sobre o teste unitário, análise de valor limite e teste de integração em software PON.

4.1. Teste unitário

As menores unidades testáveis de um software em PON são os *FBEs* e as *Rules* com suas *Premisses*, *Conditions, Actions* e *Methods*. O método funcional proposto para teste unitário de *Rule* consiste na análise do valor limite suportado pelas *Premisses*, conforme discutido a seguir.

A Análise do Valor Limite é uma técnica de teste de software utilizada para exercitar os limites do domínio de entrada. Considerada um complemento ao Particionamento de Equivalência (ou Classes de Equivalência), esta análise foca a seleção de Casos de Teste nas bordas da classe, ou seja, nos valores próximos às fronteiras das classes. A análise do valor-limite considera também o domínio de saída para derivar casos de teste [17].

Classes de equivalência são identificadas pela interpretação de cada condição de entrada (usualmente uma sentença ou frase na especificação) e particionadas em dois ou mais grupos, por exemplo, classes de valores válidos e classes de valores inválidos [17].

Os valores limites em PON referem-se aos *Attributes* referenciados nas *Premisses*. Deverão ser testados valores que as *Premisses* suportarão e os que não suportarão, a fim de se verificar o comportamento da *Rule*. A *Rule* deverá executar apenas quando tiver suas condições satisfeitas, de acordo com sua especificação. Além disso, como a execução da uma *Rule* provoca um conjunto de ações que podem afetar novos *Attributes*, os Casos de Teste deverão verificar as ações executadas e atributos alterados.

Assim, será elaborado um plano de testes unitários para cada regra (*Rule*) que compõe a aplicação PON. Em cada plano de teste de uma *Rule*, serão relacionados os casos de teste unitários para verificação dos conjuntos de valores limites compostos a partir do conjunto de *Premisses* daquela *Rule*.

Para a realização de testes unitários dos *FBEs*, são adotadas as práticas convencionais, pois cada *FBE* é implementada na forma de uma única classe com atributos e métodos.

4.2. Teste de integração

Neste tipo de teste são verificados todos os *processos* necessários para implementar cada caso de uso do sistema. Deve-se verificar o alcance do ciclo de notificações que cada mudança de estado (em *Attributes* dos *FBEs*) pode desencadear. Uma *Premise* pode ser compartilhada com várias *Rules* podendo ativar e executar mais de uma *Rule,* quando seu estado lógico passa a ser verdadeiro em razão da mudança de estado de um *Attribute*. Quando uma *Rule* é executada, ela pode alterar o valor de outros *Attributes* e, consequentemente, o estado de outras *Premisses*, recomeçando ou dando seguimento ao ciclo de notificações.

Para melhorar a visualização da alcançabilidade de alterações que poderão ocorrer no sistema, é sugerida uma representação gráfica simplificada das interconexões entre *FBE*, *Premisses* e suas respectivas *Rules*. Este instrumento permitirá determinar os casos de teste de integração necessários que serão especificados no plano de testes de integração.

5. Caso de estudo: exemplo de jogo

De modo a facilitar a compreensão do teste funcional em PON, apresenta-se, a seguir, um caso de estudo que consiste na implementação simplificada de um simulador de jogo de combate aéreo em duas dimensões (Figura 8). A seguir são apresentados os requisitos deste *software*:

- O Avião será movimentado pelo jogador com possibilidade de movimentação para direita ou para esquerda.
- O ataque do Avião é feito por meio de disparos de projéteis.
- O Avião poderá atacar quando tiver munição disponível.
- O Avião poderá atacar ou ser movimentado quando tiver pontos de vida suficientes.
- Haverá um contador de pontos de vidas e de munição que o jogador poderá disparar.
- Haverá inimigos do tipo Helicóptero.
- O Helicóptero será controlado pelo sistema, e possuirá deslocamento a partir da margem superior da tela em direção à margem inferior.
- O projétil disparado pelo Avião poderá colidir com o Helicóptero, resultando no decremento de pontos de vida do Helicóptero.

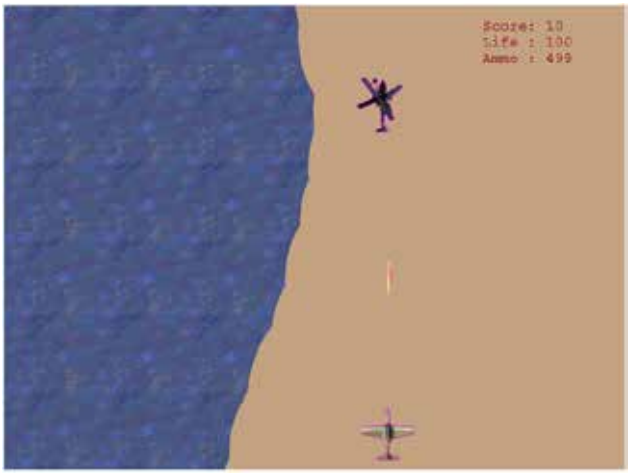

Figura 8 – Tela do jogo em execução

A Figura 9 apresenta o diagrama de casos de uso do software destacando os quatro principais casos que envolvem o controle do avião e do disparo pelo usuário e, ainda, o controle automático do inimigo (helicóptero) e a detecção de colição dos disparos.

O software é composto por cinco classes, conforme ilustrado na Figura 10. A classe GameControl é uma

classe derivada de NOP_Application e herda a estrutura do *framework* PON. Assim, esta classe permitirá incorporar o sistema de notificações na aplicação. A classe Plane representa o avião e a classe Helicopter representa o inimigo. Ambas são especializações da classe Character. Por fim, a classe Bullet representa o disparo feito pelo avião ou inimigo.

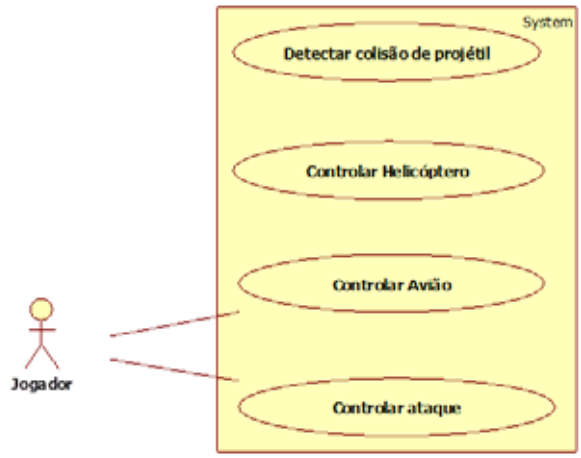

Figura 9 – Casos de uso do jogo

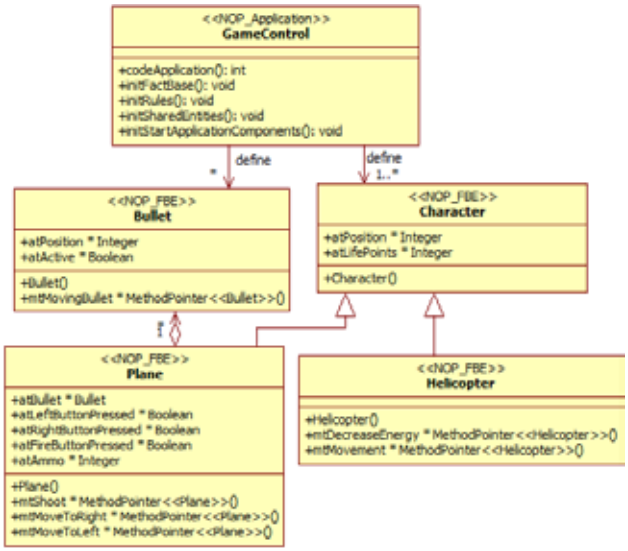

Figura 10 – Diagrama de classes do jogo

5.1. Teste unitário da aplicação

A partir dos requisitos foram identificadas as seguintes classes: Plane (Algoritmo 1), Character (Algoritmo 2), Helicopter (Algoritmo 3) e Bullet (Algoritmo 4). As *Premisses* são: prPlaneIsAlive, prHelicopterIsAlive, prRightButtonPressed, prLeftButtonPressed, prFireButtonPressed, prContainsAmmo, e prColisionDetection (Algoritmo 5). As *Rules* são: rlMovePlaneToRight, rlMovePlaneToLeft, rlPlaneShoots, rlMoveHelicopter e rlDecreaseHelicopterEnergy (Algoritmo 6). Os algoritmos a seguir detalham estes componentes do software.

Algoritmo 1 – *FBE* Plane

```
class Plane : public FBE
{
public:
    Plane();
    ~Plane();
public:
     Bullet * atBullet;
     Boolean * atFireButton;
     Boolean * atLeftButton;
     Boolean * atRightButton;
     Integer * atAmmo;
     MethodPointer<Plane> * mtShoot;
     MethodPointer<Plane> * mtMoveToRight;
     MethodPointer<Plane> * mtMoveToLeft;
public:
     shoot();
     moveToRight();
     moveToLeft();
};
```

Algoritmo 2 – *FBE* Character

```
class Character : public FBE
{
public:
    Character();
    ~Character();
public:
     Integer * atPosition;
     Integer * atLifePoints;
};
```

Algoritmo 3 – *FBE* Helicopter

```
class Helicopter : public Character
{
public:
    Helicopter();
    ~Helicopter();
public:
     MethodPointer<Helicopter> * mtDecreaseEnergy;
     MethodPointer<Helicopter> * mtMovement;
public:
     decreaseEnergy();
     movement();
};
```

Algoritmo 4 – *FBE* Bullet

```
class Bullet : public FBE
{
public:
    Bullet();
    ~Bullet();
public:
     Integer * atPosition;
     Boolean * atActive;
public:
     MethodPointer<Bullet> * mtMovingBullet;
public:
     movingBullet();
};
```

Algoritmo 5 – Principais *Premisses* do jogo

```
PON_PREMISE (prPlaneIsAlive, plane -> atLifePoints, 0,
Premise::GREATERTHAN, Premise::STANDARD, false);

PON_PREMISE (prRightButtonPressed, plane -> atRightButton,
true, Premise::EQUAL, Premise::STANDARD, false);

PON_PREMISE (prLeftButtonPressed, plane -> atLeftButton,
true, Premise::EQUAL, Premise::STANDARD, false);

PON_PREMISE (prFireButtonPressed, plane -> atFireButton,
true, Premise::EQUAL, Premise::STANDARD, false);

PON_PREMISE (prPlaneContainsAmmo, plane -> atBullet, 0,
Premise::GREATERTHAN, Premise::STANDARD, false);

PON_PREMISE (prHelicopterIsAlive, helicopter -> atLifePoints,
0, Premise::GREATERTHAN, Premise::STANDARD, false);

PON_PREMISE (prColisionDetection, helicopter -> atPosition,
bullet-> atPosition, Premise::EQUAL, Premise::STANDARD,
false);
```

Algoritmo 6 – Principais *Rules* para o desenvolvimento do jogo

```
rlMovePlaneToRight = new RuleObject ("rlMovePlaneToRight",
SingletonScheduler::getInstance(), Condition::CONJUNCTION);
rlMovePlaneToRight -> addPremise (prPlaneIsAlive);
rlMovePlaneToRight -> addPremise (prRightButtonPressed);
rlMovePlaneToRight -> addMethod (plane -> mtMoveToRight);
rlMovePlaneToRight -> end();

rlMovePlaneToLeft = new RuleObject ("rlMovePlaneToLeft",
SingletonScheduler::getInstance(), Condition::CONJUNCTION);
rlMovePlaneToLeft -> addPremise (prPlaneIsAlive);
rlMovePlaneToLeft -> addPremise (prLeftButtonPressed);
rlMovePlaneToLeft -> addMethod (plane -> mtMoveToLeft);
rlMovePlaneToLeft -> end();

rlPlaneShoots = new RuleObject("rlPlaneShoots ",
SingletonScheduler::getInstance(), Condition::CONJUNCTION);
rlPlaneShoots -> addPremise (prPlaneIsAlive);
rlPlaneShoots -> addPremise (prFireButtonPressed);
rlPlaneShoots -> addPremise (prContainsAmmo);
rlPlaneShoots -> addMethod (plane -> mtShoot);
rlPlaneShoots -> end();

rlMoveHelicopter = new RuleObject ("rlMoveHelicopter ",
SingletonScheduler::getInstance(), Condition::CONJUNCTION);
rlMoveHelicopter -> addPremise (prHelicopterIsAlive);
rlMoveHelicopter -> addMethod (helicopter -> mtMovement);
rlMoveHelicopter -> end();

rlDecreaseHelicopterEnergy = new RuleObject
("rlDecreaseHelicopterEnergy",
SingletonScheduler::getInstance(), Condition::CONJUNCTION);
rlDecreaseHelicopterEnergy -> addPremise
(prHelicopterIsAlive);
rlDecreaseHelicopterEnergy -> addPremise
(prColisionDetection);
rlDecreaseHelicopterEnergy -> addMethod (helicopter ->
mtDecreaseEnergy);
rlDecreaseHelicopterEnergy -> end();
```

A Tabela 1 apresenta o particionamento em classes de equivalência para condições das *Premisses*. Foram definidos dois conjuntos de classes (valores inválidos e inválidos).

Tabela 1 – Apresentação do particionamento de classes de equivalência com análise de valor limite

Entrada ou condição externa – *Premise* e respectivo *Attribute*	Classes de equivalência válidas	Classes de equivalência inválidas
prPlaneIsAlive plane -> atLifePoints	1; 2	-1; 0
prPlaneContainsAmmo plane -> atBullet	1; 2	-1; 0; a-z
prFireButtonPressed plane -> atFireButtonPressed	true	false, 0-9; a-z
prLeftButtonPressed plane -> atLeftButtonPressed	true	false, 0-9; a-z
prRightButtonPressed plane -> atRightButtonPressed	true	false, 0-9; a-z
prHelicopterIsAlive helicopter -> atLifePoints	1; 10	-1; 0
prColisionDetection helicopter -> atPosition == bullet -> atPosition	Iguais	Diferentes; primeiro maior que segundo; segundo maior que primeiro.

É possível observar que as *Premisses* apresentadas no Algoritmo 5 são utilizadas na composição de *Rules* no Algoritmo 6. Por exemplo, a *Premise* prPlaneIsAlive é compartilhada entre três *Rules*. Portanto, para cada mudança de estado do *Attribute* atLifePoints do *FBE* Plane, é realizada uma avaliação lógica sobre a *Premise* prPlaneIsAlive para conferência se o mesma possui valor superior a zero, requerido na *Condition*. A *Condition* da *Premise* prPlaneIsAlive, por sua vez, requer que atLifePoints seja superior a zero.

Com base na análise dos valores limite das *Conditions* das *Premisses*, são apresentados alguns casos de teste em *Rules* na Tabela 2.

Tabela 2 – Casos de teste para *Rules*

Caso de teste 1

plane -> atLifePoints -> setValue(1);
plane -> atBullet -> setValue(1);
plane -> atFireButton -> setValue(true);
plane -> atRightButton -> setValue(false);
plane -> atLeftButton -> setValue(true);
Executa as seguintes *Rules*:
rlPlaneShoots
rlMovePlaneToLeft

Caso de teste 2

plane -> atLifePoints -> setValue(1);
plane -> atBullet -> setValue(-1);
plane -> atFireButton -> setValue(true);
plane -> atRightButton -> setValue(true);
plane -> atLeftButton -> setValue(false);
Executa a seguinte *Rule*:
rlMovePlaneToRight

Caso de teste 3
plane -> atLifePoints -> setValue(100); plane -> atBullet -> setValue(0);

plane -> atFireButton -> setValue(true);
plane -> atRightButton -> setValue(false);
plane -> atLeftButton -> setValue(true);
Executa a seguinte *Rule*
rlMovePlaneToLeft

Caso de teste 4

plane -> atLifePoints -> setValue(0);
plane -> atBullet -> setValue(1);
plane -> atFireButtonPressed -> setValue(true);
plane -> atRightButtonPressed -> setValue(true);
Nenhuma *Rule* será executada

5.2. Teste de integração da aplicação

A Tabela 3 apresenta a relação das *Premisses*, *Rules* e *Instigations* que realizam cada caso de uso apresentado na Figura 9. Assim, é possível analisar os pequenos processos que acontecerão para cada caso de uso.

Tabela 3 – Casos de uso e as *Premisses* e *Rules* que os implementam

Caso de uso	*Premisses* e *Rules*	Instigações
Controlar Avião	prPlaneIsAlive prRightButtonPressed prLeftButtonPressed rlMovePlaneToRight rlMovePlaneToLeft	mtMoveToRight mtMoveToLeft
Controlar ataque	prContainsAmmo prPlaneIsAlive prFireButtonPressed rlPlaneShoots	mtShoot mtMovingBullet;
Detectar colisão de projétil	prHelicopterIsAlive prColisionDetection rlDecreaseHelicopter Energy	mtDecreaseHelicopt erEnergy
Controlar Helicóptero	prHelicopterIsAlive rlMoveHelicopter	mtMovement

A alcançabilidade da alteração de estado de *Premise*, mencionada na seção 0, pode ser observada na *Premise* prPlaneIsAlive. Esta *Premise* é parte de três *Rules* (Figura 11). Portanto, a cada alteração de valor do *Attribute* relacionado à *Premise* é preciso acompanhar o estado das *Rules* que a contém.

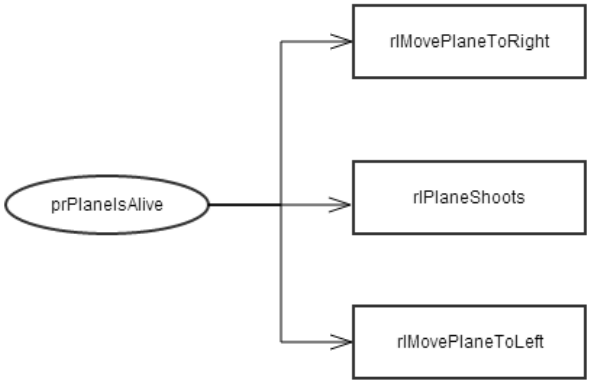

Figura 11 – Exemplo de uma *Premise* compartilhada com várias *Rules*

Na Figura 12 é apresentado todo o sistema e suas interconexões, de modo simplificado, entre *FBE*, *Premisses* e *Rules*.

Algoritmo 7 – Implementação do *Method* mtShoot (da *Rule* rlPlaneShoots)

```
void Plane::mtShoot()
{
atBullet->setValue(atBullet->getValue()-1); //subtract one
bullet per shoot

bullet = new Bullet();
bullet->atActive->setValue(true);

prActiveBullet = new Premise();
PON_PREMISE(prActiveBullet, bullet -> atActive, true,
Premise::EQUAL, Premise::STANDARD, false);

rlMovingBullet = new RuleObject ("rlMovingBullet",
SingletonScheduler::getInstance(), Condition::CONJUNCTION);
rlMovingBullet->addPremise(prActiveBullet);
rlMovingBullet->addMethod(bullet->mtMovingBullet);
rlMovingBullet->end();
}
```

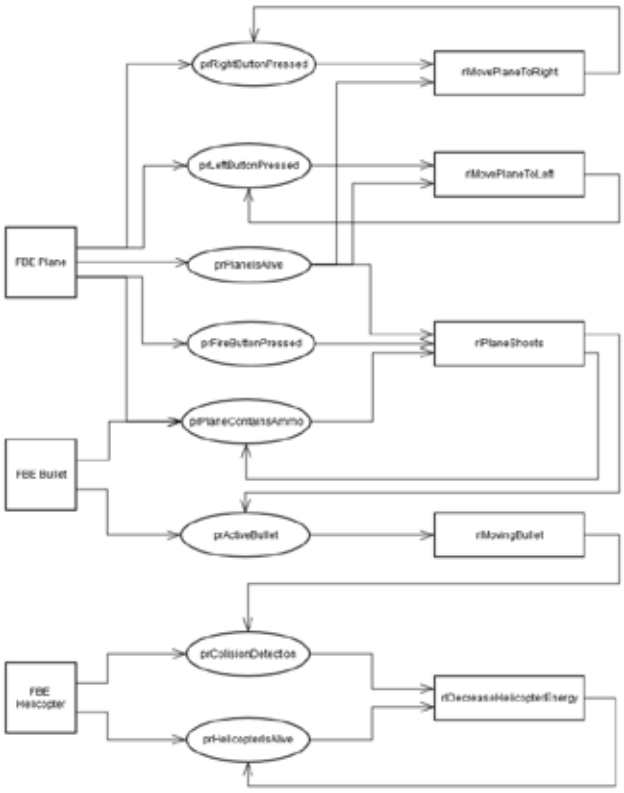

Figura 12 – Exemplo reduzido de alcançabilidade que uma *Rule* com outras *Premisses*

É possível notar que o ciclo de notificações apresenta pequenos *processos* relacionados à execução de uma determinada atividade. Por exemplo, as *Premisses* prPlaneIsAlive, prFireButtonPressed e prPlaneContainsAmmo quando verdadeiras permitem a execução da *Rule* rlPlaneShoots. Por sua vez, a *Rule* rlPlaneShoots quando executada (Algoritmo 7), decrementa uma unidade de atBullet e cria uma nova *Premise*, chamada prActiveBullet, e uma nova *Rule*

rlMovingBullet que pode alterar a *Premise* prColisionDetection (quando houver colisão com Helicóptero será decrementada seus pontos de vida). A *Rule* rlDecreaseHelicopterEnergy quando executada causa a reavaliação da *Premise* prHelicopterIsAlive e assim por diante.

6. Conclusões e trabalhos futuros

Qualquer sistema computacional em qualquer paradigma está propenso a conter erros. Portanto, faz-se necessário criar e aplicar critérios de testes de software para tentar diminuir os problemas provenientes dos erros e enganos cometidos pelo programador no desenvolvimento de um software. Apenas a atividade de *debugging* não é suficiente e, ainda, não é considerada uma ferramenta de testes [16].

Com o estudo da aplicabilidade descrita neste artigo, foi possível constatar que é possível aplicar o teste funcional em software PON. Entretanto, foi necessário fazer considerações particulares em razão das especificidades deste paradigma. Em especial, foi necessário tratar os casos de teste unitários separadamente por regras (*Rules*) e organizar os testes de integração pela análise de alcançabilidades das regras em cada caso de uso.

Com o avanço do estudo cabe investigar a aplicabilidade de outros métodos de teste como, por exemplo, o teste estrutural e o baseado em erros (v.g. teste de mutação).

De modo natural, adaptações, melhorias e desenvolvimentos futuros de critérios de teste são necessários para comportar as peculiaridades deste paradigma emergente.

Referências bibliográficas

1. DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. **Introdução ao Teste de Software**. 4. ed. Rio de Janeiro: Elsevier, 2007.
2. SEBESTA, R. W. **Concepts of Programming Languages**. 10th. ed. [S.l.]: Pearson, 2012. ISBN: 0273769103, 9780273769101.
3. SIMÃO, J. M.; STADZSIZ, P. C. **Paradigma Orientado a Notificações (PON) Uma Técnica de Composição e Execução de Software Orientado a Notificações**, PI015080004262, 2008.
4. SIMÃO, J. M.; STADZISZ, P. C. Inference Based on Notifications: A Holonic Meta-Model Applied to Control Issues. **IEEE Transaction on System, Man, and Cybernetics**, p. 238-250, 2009.
5. HOWDEN, W. E. Functional Program Testing. **Software Engineering, IEEE Transactions on**, v. SE-6, n. 2, p. 162-169, March 1980. ISSN: 0098-5589 DOI: 10.1109/TSE.1980.230467.
6. BEIZER, B. **Black-box testing:** techniques for functional testing of software and systems. [S.l.]: John Wiley & Sons, Inc., 1995.
7. RAPPS, S.; WEYUKER, E. J. Selecting Software Test Data Using Data Flow Information. **Software Engineering, IEEE Transactions on**, v. SE-11, n. 4, p. 367-375, April 1985. ISSN: 0098-5589 DOI: 10.1109/TSE.1985.232226.
8. RAPPS, S.; WEYUKER, E. Data Flow Analysis Techniques for Test Data Selection. **Department of Computer Science, Courant Institute of Mathematical Sciences**, New York, 1982.
9. MALDONADO, J. C. **Critérios potenciais usos: Uma contribuição ao teste estrutural de software**. Faculdade de Engenharia Elétrica, UNICAMP. [S.l.]. 1991.
10. MCCABE, T. J. A Complexity Measure. **Software Engineering, IEEE Transactions on**, v. SE-2, n. 4, p. 308-320, Dec 1976. ISSN: 0098-5589 DOI: 10.1109/TSE.1976.233837.
11. BANASZEWSKI, R. F. **Paradigma Orientado a Notificações - Avanços e Comparações**. Dissertação (Mestrado) – Curso de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI), Universidade Tecnológica Federal do Paraná (UTFPR). Curitiba, Brasil. 2009.
12. RONSZCKA, A. F. **Contribuição para a Concepção de Aplicações no Paradigma Orientado a Notificações (PON) sob o viés de Padrões**. Dissertação (Mestrado) – Curso de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI), Universidade Tecnológica Federal do Paraná (UTFPR). Curitiba, Brasil. 2012.
13. SIMÃO, J. M. et al. Notification Oriented Paradigm (NOP) and Imperative Paradigm: A Comparative Study. **Journal of Software Engineering and Applications (JSEA)**, v. 5, p. 402, 2012.
14. VALENÇA, G. Z. **Contribuição para materialização do Paradigma Orientado Notificações (PON) via Framework e Wizard**. Dissertação (Mestrado) – Curso de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI), Universidade Tecnológica Federal do Paraná (UTFPR). Curitiba, Brasil. 2012.
15. RONSZCKA, A. F. et al. Comparações quantitativas e qualitativas entre o Paradigma Orientado a Objetos e o Paradigma Orientado a Notificações sobre um simulador de jogo. **III Congresso Intern. de Computación y Telecom - COMTEL**, 2011.
16. BINDER, R. V. **Testing Object-oriented Systems:** Models, Patterns, and Tools. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN 0-201-80938-9.
17. MYERS, G. J. et al. **The Art of Software Testing**. 2. ed. New York, NY, EUA: John Wiley and Sons, 2004.
18. PRESSMAN, R. **Software Engineering:** A Practitioner's Approach. 7. ed. New York, NY, USA: McGraw-Hill, Inc., 2010. ISBN: 0073375977, 9780073375977.
19. SOMMERVILLE, I. **Software Engineering (9th Edition)**. [S.l.]: Addison-Wesley, 2011.
20. WIECHETECK, L. V. B.; STADZISZ, P. C.; SIMÃO, J. M. Um Perfil UML para o Paradigma Orientado a Notificações (PON). **III Congresso Intern. de Computación y Telecom - COMTEL**, 2011.