UNIVERSIDADE FEDERAL DE GOIÁS – CAMPUS CATALÃO

DEPARTAMENTO DE CIÊNCIAS DA COMPUTAÇÃO

Curso de Bacharelado em Ciências da Computação

MODELAGEM DE JOGOS: UMA ABORDAGEM BASEADA EM WORKFLOW-NETS PARA AGENTES JOGADORES

Thiago de Almeida Bastos

CATALÃO – GO

2013

THIAGO DE ALMEIDA BASTOS

MODELAGEM DE JOGOS: UMA ABORDAGEM BASEADA EM WORKFLOW-NETS PARA AGENTES JOGADORES

Monografia apresentada ao Curso de Bacharelado em Ciências da Computação da Universidade Federal de Goiás – Campus Catalão, como requisito parcial para a obtenção do Grau de Bacharel em Ciências da Computação.

Orientadora:
Liliane do Nascimento Vale

Área:
Engenharia de Software

CATALÃO – GO

2013

Bastos, Thiago de Almeida

Modelagem de Jogos: Uma abordagem baseada em Workflow-Nets para agentes jogadores/ Thiago de Almeida Bastos. – Catalão – GO, 2013.

88 f. ; 30 cm.

Orientadora: Liliane do Nascimento Vale.

Monografia (Graduação) – Universidade Federal de Goiás – Campus Catalão, Departamento de Ciências da Computação, Curso de Ciências da Computação, 2013.

1. Modelagem de jogos. 2. Sistemas de WorkFlow. 3. Redes de Petri. 4. Agentes inteligentes. I. Vale, Liliane do Nascimento. II. Universidade Federal de Goiás – Campus Catalão. Curso de Bacharelado em Ciências da Computação. III. Título.

THIAGO DE ALMEIDA BASTOS

MODELAGEM DE JOGOS: UMA ABORDAGEM BASEADA EM WORKFLOW-NETS PARA AGENTES JOGADORES

Monografia apresentada ao Curso de Bacharelado em Ciências da Computação pela Universidade Federal de Goiás – Campus Catalão.

Trabalho aprovado em 20 de março de 2013.

Área: Engenharia de Software

Liliane do Nascimento Vale
Orientadora

Vaston Gonçalves da Costa
Universidade Federal de Goiás

Fábio Gomes de Assunção
Universidade Federal de Goiás

Catalão – GO

2013

Aos meus familiares e amigos pelo eterno carinho e incentivo.

AGRADECIMENTOS

Agradeço a todos que de alguma forma contribuiu para o sucesso deste projeto, e que de forma direta ou indireta foi importante em minha caminhada, em especial, agradeço...

Primeiramente a Deus, por tudo que Ele é e faz por mim.

Aos meus pais, e de tudo que foram capazes de me proporcionar.

De forma especial a todos os meus familiares que estiveram juntos comigo.

Aos amigos, da faculdade e da vida, que sob diversos aspectos me ajudaram e sobretudo pela sua amizade.

A Liliane minha orientadora e parceira que muito me ajudou com sua incansável paciência e sabedoria.

De coração a todos estes, o meu muito obrigado!

"A maneira como você coleta, gerencia e utiliza as informações determina se você vai vencer ou perder" (Bill Gates)

RESUMO

BASTOS, T. DE A.. *Modelagem de Jogos: Uma abordagem baseada em Workflow-Nets para agentes jogadores.* 2013. 88 f. Monografia (Graduação) – Departamento de Ciências da Computação, Universidade Federal de Goiás – Campus Catalão, Catalão – GO.

O aumento considerável nos últimos tempos da produção de jogos digitais relacionados a diversão e educação tem voltado a atenção de desenvolvedores para a utilização de técnicas de modelagem que resulte em jogos de boa qualidade. O foco deste trabalho relaciona a aplicação de técnicas de modelagem para a representação de agentes inteligentes inserindo elementos da lógica linear no jogo, capazes de desempenhar funções de adversários nos jogos digitais. Para a formalização dos agentes inteligentes é proposto um modelo híbrido com redes de Petri a objetos (Workflow-Nets-objetos) no contexto de softwares orientados a objetos combinados com elementos da lógica linear. Inicialmente, as Workflow-Nets a objetos são usadas para especificar formalmente modelos de agentes de diversas funcionalidades existentes em jogos. Os modelos propostos permitem tanto a representação de estrutura de dados complexos, a especificação de atividades a serem realizadas bem como a representação de regras através da lógica linear que determinam o comportamento do agente durante a execução. A execução do agente inteligente é dado por meio de instanciação deste agente. Para ilustrar a utilização do modelo proposto é realizado um estudo de caso aplicado ao comportamento do agente para uma dada situação em um jogo conhecido como jogo dos pontinhos.

Palavras-chaves: Modelagem de jogos, Sistemas de WorkFlow, Redes de Petri, Agentes inteligentes.

LISTA DE ILUSTRAÇÕES

Figura 1 – Exemplo de um fluxograma para um ambiente de jogo de guerra. 28
Figura 2 – Exemplo de classes UML. 29

Figura 3 – Exemplo de WorkFlow-net 34

Figura 4 – Exemplo, elementos de uma rede de Petri. 36
Figura 5 – Exemplo de rede de Petri com fichas 37
Figura 6 – Exemplo de rede de Petri marcada 37
Figura 7 – Exemplo de disparo de uma rede de Petri 38
Figura 8 – Exemplo de rede de Petri a objetos 43
Figura 9 – Exemplo de rede de Petri a objetos após o disparo de fichas 44
Figura 10 – Descrição da semântica de transições das WorkFlow-net 46

Figura 11 – Modelo gráfico para descrição sumária de um agente. 48
Figura 12 – Ambiente de tarefa de um agente ASPIRADOR-DE-PÓ. 49
Figura 13 – Descrição gráfica de um agente reativo simples. 52

Figura 14 – (a) Resumo de operadores da lógica linear. (b) Gramática de proposições dos operadores 54
Figura 15 – Regras da lógica linear. 56
Figura 16 – Exemplo de transições para aplicação em fórmulas da lógica linear. 57
Figura 17 – Rede de Petri para exemplificar a construção da árvore de prova canônica. 59
Figura 18 – Arvore de Prova canônica das sequentes da Equação6.3 59

Figura 19 – Representação simples do ambiente de jogo dos pontinhos 64
Figura 20 – Plataforma utilizada no estudo de caso para o ambiente de jogo dos pontinhos 64
Figura 21 – Plataforma de jogo com uma situação que representa a marcação de arestas e contagem de pontos dos jogadores. 65
Figura 22 – Marcação de zonas de jogadas do jogo dos pontinhos 66
Figura 23 – Sequencia de jogadas ideal para vencer situação de jogo envolvendo as zonas da Figura22 66
Figura 24 – Disparo 1 69

Figura 25 – Disparo 2 69
Figura 26 – Disparo 3 70
Figura 27 – Disparo 4 70
Figura 28 – Disparo 5 71
Figura 29 – Disparo 6 71
Figura 30 – Disparo 7 72
Figura 31 – Disparo 8 72
Figura 32 – Grafo de alcançabilidade para a rede mostrada nos disparos 74
Figura 33 – Árvore de prova para a rede de alcançabilidade. 74
Figura 34 – Representação gráfica do ambiente para construção da classe No() do Agente-jogador 76
Figura 35 – Situação/teste em um cenário$A$ de jogo para o Agente jogador. 78
Figura 36 – Resultado das jogadas do agente para o cenário$A$. 79
Figura 37 – Exemplo de uma boa marcação para um cenário possível do jogo. 79

LISTA DE QUADROS

Quadro 1 – Exemplo de Sequência de percepções do agente ASPIRADOR-DE-PÓ . . . 49
Quadro 2 – Sequência de regras para formação inicial do agente adversário para o Jogo dos Pontinhos 66
Quadro 3 – Estratégias de acordo com o retorno da função fatorPerca 77

LISTA DE ALGORITMOS

Algoritmo 1 – Algoritmo para agente ASPIRADOR-DE-PÓ(posicao,estado) 49
Algoritmo 2 – Função Fatorperca(int matriz[][]) 80
Algoritmo 3 – Função ataca() 81

SUMÁRIO

1 Introdução **21**

1.1 Descrição do Problema 22

1.2 Organização do Trabalho 23

2 Modelagem de Jogos **25**

2.1 Especificação de Requisitos 25

2.1.1 Modelos de Contexto 26

2.1.2 Modelos Estruturais 26

2.1.3 Modelos Comportamentais 26

2.2 Modelagem de Jogos 27

3 Sistemas de WorkFlow **31**

3.1 Conceitos Sobre Workflow 31

3.2 Modelagem de Workflows 33

4 Redes de Petri **35**

4.1 Definição 35

4.2 Formalizando Redes de Petri 36

4.3 Transição Sensibilizada 37

4.4 Classificação das Redes de Petri 39

4.5 Redes de Petri a Objetos 40

4.6 WorkFlow-net 44

4.6.1 Propriedades das WorkFlow-nets 45

5 Técnicas para Produção de Agentes Inteligentes **47**

5.1 Agentes Inteligentes 47

5.2 Racionalidade 49

5.3 Estrutura de Agentes 50

5.4 Agente Reativo Simples 51

6 Lógica Linear **53**

6.1 Contextualização 53

6.2 Conectivos da Lógica Linear 54

6.3 Regras da Lógica Linear 55

6.4 Tradução de Redes de Petri para Lógica Linear 57

6.5 Cálculo de Sequentes para Lógica Linear 58

6.6 Provando a Corretude das WorkFlow nets 60

7 Estudo de Caso **63**
7.1 Jogo dos pontinhos 63
7.2 Modelo de Especificação do Jogo Baseado em Agente 67
7.3 Verificação do Modelo Utilizando Lógica Linear 73
7.4 Agente Jogador Para Jogo Dos Pontinhos 75
7.5 Testando o Agente Jogador 78

8 Conclusão **83**

Referências **87**

CAPÍTULO 1

INTRODUÇÃO

A fase de projeto de todo e qualquer software, bem como a produção de jogos digitais envolve várias etapas que incluem, por exemplo, definição de seu estilo, contexto, objetivos e também em nível de design, como definição de espaço virtual e principais ações. A modelagem de software, ou modelagem de jogos para este contexto envolve princípios, conceitos, métodos e ferramentas aplicados por engenheiros de software ao longo de todo este processo de desenvolvimento (PRESSMAN, 2011).

A comunicação visual é essencial no projeto de software. No desenvolvimento de jogos, a regra não foge a realidade, como por exemplo: Na identificação e construção de casos de uso, o fluxo entre os menus, a modelagem de sequência de eventos, ou a representação do relacionamento entre os atores do jogo. Além disso, existe a necessidade de verificar, validar e simular as funcionalidades do jogo. Neste caso o uso das redes de Petri(CARDOSO; VALETE, 1997) são consideradas, uma vez que seu foco é voltado para a checagem da consistência e sintaxe de diversas características.

Neste trabalho, são abordado as WorkFlow-nets a objetos, um tipo particular de redes de Petri orientados a objetos que permite que casos (objetos/fichas) sejam instanciados e ao longo da execução as estruturas de dados carregadas por este objeto serão manipulados pela rede por meio das regras da lógica linear que serão associados as transições da rede para análise, e satisfação das condições para disparo das transições.

Dessa forma, um modelo híbrido composto pelas WorkFlow-nets a objetos e a lógica linear é proposto para simular a inteligência do agente, bem como suas estratégias de jogadas. Então um planejamento de rotas é considerado para especificar o conjunto de comportamentos que o agente terá, ilustrando situações como, os relacionamentos, a concorrência, paralelismo, etc. Além do suporte matemático que permite a análise qualitativa e quantita-

tiva das propriedades.

1.1 Descrição do Problema

A modelagem de softwares é uma representação, geralmente em meios informais, das principais características de um objeto ou sistema através da análise do modelo. Um sistema real pode ser compreendido sem o perigo, o custo ou a inconveniência da manipulação de seus elementos.

Além de preparar a equipe de desenvolvimento para a produção de todo e qualquer projeto é necessário usar elementos que garantam a validação do resultado esperado. A modelagem semi formal das características do sistema, com o uso de diagramas da UML (*Unified Modeling Language*) (GUEDES, 2011), impede a correção e simulação dos modelos.

A UML é um conjunto de diagramas que tem como objetivo representar as diversas visões do sistema. Entretanto, quando aplicados com objetivo de compreender um software, o resultado é a divisão do projeto em um grande número de tipos de diagramas que obriga desenvolvedores a conhecer todos os elementos notacionais, bem como as inconsistências semânticas, que permite uma gama de interpretações em vez de um significado exato.

Assim diagramas da UML são usados no nível conceitual, sendo necessários outros modelos para análise, simulação e execução das funcionalidades consideradas, o uso de modelos formais que garantem a análise matemática e gráfica, como por exemplo as redes de Petri.

As redes de Petri (MURATA, 1989) um modelo matemático com representação gráfica formada por nós que indicam estados(lugares), e transição, representando ações, para se esquematizar o consumo de recursos (fichas).

O uso das redes de Petri, que permitem representar características oriundas de sistemas de WorkFlow como por exemplo, o conceito de rotas, tarefas e recursos participantes, ilustra as características complexas presentes em um jogo.

Além disso a capacidade de representar e manipular estruturas de dados, é outra particularidade observada nas redes de Petri, sendo possível também considerar regras representadas por fórmulas de lógica linear.

Assim, a produção de uma agente inteligente é baseado no ambiente em que o mesmo está incluso e nas perspectivas e reações que o mesmo terá que desempenhar para que seu objetivo seja alcançado. O maior problema é definir todas estas informações de forma precisa para que em situação alguma o agente tenha comportamentos que não satisfaz a determinação prévia ou até mesmo fuja todos os padrões aceitáveis. Por isso, a modelagem formal torna-se uma ferramenta de grande utilidade, pois com ela a gama de informações disponíveis para a modulação do agente será mais completa e irá oferecer suporte maior no

seu desenvolvimento.

Este projeto tem o propósito de relacionar o uso da modelagem de software para um estudo de caso que engloba o desenvolvimento de um agente jogador inteligente baseado nos conceitos da inteligência artificial.

O modelo a ser analisado será representado e posteriormente validado por meio da implementação do jogo dos pontinhos (SANTOS, 2006) que será nosso estudo de caso. O emprego do mecanismo de modelar formalmente terá a função de garantir a eficiência do desenvolvimento do agente.

A teoria dos jogos citada no trabalho de Bazzan (2001) geralmente não é suficiente para a produção de agentes para o núcleo de jogos e o resultado deste serviço nem sempre é um agente capaz de realizar suas tarefas com precisão. Buscando melhorar a produção de agentes que integrarão os jogos iremos adicionar à produção destes agentes a fase de modelagem para representação formal de informações.

1.2 Organização do Trabalho

No capítulo 2 se abordará conceitos básicos da Modelagem de software no contexto de jogos, evidenciando sua aplicação e de que forma este aparato é vantajoso. O desenvolvimento de um software de qualquer outra natureza assim como de jogos pode estar relacionado a atividade de modelagem tornando possível aplicar as técnicas de modelagem ao seu contexto como será demonstrado.

No capítulo 3 é feita a fundamentação teórica. A teoria de sistemas WorkFlow será demonstrado em uma sequência de passos necessários para que se possa atingir a automação de processos de negócio, e é feita de acordo com um conjunto de regras definidas, envolvendo a noção de processos.

No capitulo 4 será revisada a teoria de redes de Petri que apresenta um caso particular para a implementação gráfica e matemática dos sistemas Workflow, originando as WorkFlow-nets.

Em sequência no capitulo 5 serão considerados técnicas de inteligência artificial para a produção de agentes jogadores que irão compor o produto do estudo de caso deste trabalho. As técnicas usadas terão de determinar uma ação, para o jogo baseada na percepção atual, diante de uma situação de jogo possível.

Para a verificação de habilidades o modelo obtido no estudo de caso irá se submeter a técnicas de verificação de ações através da lógica linear. Este tema será abordado no capítulo 6 com o objetivo de provar caráter de solidez em um critério de corretude por meio de um tipo de análise qualitativa para as especificações formais do jogo. Caracterizada como lógica de recursos, a lógica linear trata sentenças como ações ou consequências lógicas entre

estados, se tornando propicio à aplicação das redes de Petri para análise de comportamentos ao longo da rede.

No capítulo 7 é feito o estudo de caso com o jogo utilizado como ambiente de teste, sua funcionalidade e suas peculiaridades. A forma como o agente foi desenvolvido e as técnicas de inteligência artificial utilizadas também serão demonstradas. Além é claro da modelagem envolvida no processo de produção, com a geração de formalismo através das WorkFlow-nets e a verificação imposta sobre este recurso com o uso da lógica linear sobre os ambientes de jogo descritos nos gráficos.

O capítulo 8 apresentará a conclusão do trabalho além de uma análise dos resultados obtidos e sugestões para trabalhos futuros.

CAPÍTULO 2

MODELAGEM DE JOGOS

O processo de desenvolvimento de modelos abstratos de um sistema compõe a modelagem de sistema, onde cada modelo apresenta uma visão ou perspectiva do sistema projetado. São usadas neste processo geralmente elementos gráficos para descrever as propriedades e/ou características do produto de software que se busca atingir.

Uma das características de um modelo de sistema é não dar tanta importância aos detalhes, porém uma abstração do sistema a ser estudado forma uma representação alternativa que é considerada importante para a equipe envolvida no desenvolvimento do sistema. Dentre as perspectivas de projeto existem alguns modelos que compõe a modelagem de software para um projeto de software eficaz que serão abordados neste capítulo.

2.1 Especificação de Requisitos

A modelagem de um sistema inclui o processo de especificar os requisitos de usuário e de sistema em um documento de requisitos. Idealmente, os requisitos de usuário e de sistema devem ser claros e inequívocos, de fácil compreensão, completos e consistentes.

A especificação de requisitos é algo difícil de se atingir com riqueza de detalhes no ponto de partida do projeto, isso porque geralmente os *stakeholders*, pessoas envolvidas no processo de modelagem, interpretam de maneiras diferentes cada ponto que forma o objetivo do sistema.

Todos os requisitos do sistema devem descrever apenas o comportamento externo do sistema e suas particularidades ou restrições operacionais contundentes a sua produção inicial, apesar de que, devem se preocupar com a forma em que o sistema será projetado e implementado. Para se atingir o nível de detalhe necessário para especificar completamente

um sistema de software, é preciso se atentar a alguns elementos, como:

- Projetar uma arquitetura inicial do sistema de forma a estruturar a especificação de requisitos.
- Interoperar o sistema com outros sistemas existentes.
- Garantir que o sistema seja seguro e identifique um projeto já certificado com a arquitetura adequada.

A seguir, são apresentados as principais perspectivas de modelagem de software.

2.1.1 Modelos de Contexto

Os modelos de contexto formam uma especificação inicial do sistema, onde os *stakeholders* do sistema estão envolvidos para decidir qual funcionalidade dever ser incluída no sistema e o que é fornecido pelo ambiente do sistema. Geralmente, os modelos de contexto mostram que o ambiente inclui vários outros sistemas, enquanto os relacionamentos de um sistema para o outro não são previstos.

Os modelos de contexto também são usados com outros modelos, como modelos de processo de negócio (SOMMERVILLE, 2011) . Um exemplo que pode descrever suas funcionalidades são os diagramas de atividades da UML que compõem um processo de sistema e o fluxo de controle de uma atividade para outra.

2.1.2 Modelos Estruturais

Além dos modelos de contexto existem também os modelos estruturais que por sua vez exibem a organização de um sistema em termos de seus componentes e seus relacionamentos de forma estática, descrevendo a estrutura do projeto(SOMMERVILLE, 2011) .

No desenvolvimento de um modelo de sistema orientado a objetos um diagrama de classes mostra as suas propriedades denotando suas classes e as associações entre essas classes, como pode ser verificado na. Uma classe pode ser um objeto participante do sistema para uma definição geral, enquanto uma associação é um link entre classes indicando seu relacionamento.

2.1.3 Modelos Comportamentais

Modelos comportamentais são responsáveis por mostrar o comportamento dinâmico do sistema enquanto está em execução (PRESSMAN, 2011). Estes modelos mostram o que acontece, ou pelo menos, o que deveria acontecer quando o sistema responder a qualquer estímulo de seu ambiente. Estes estímulos podem ser dados processados pelo sistema

ou eventos que disparam um processamento no sistema. Um modelo comportamental pode ser dividido em modelo dirigido a dados e modelo dirigido a eventos.

Modelos dirigidos a dados mostram a sequência de ações envolvidas no processamento de dados de entrada e a geração de uma saída associada. Estes diagramas formados por fluxos de dados são simples e intuitivos, e normalmente, é possível explicá-los aos potenciais usuários do sistema, que então, podem participar na validação do modelo.

Modelos dirigido a eventos mostra como o sistema reage a eventos externos e internos. Ele é baseado na suposição de que um sistema tem um número finito de estados e que os eventos(estímulos) podem causar uma transição de um estado para outro.

2.2 Modelagem de Jogos

Os jogos de computador são um segmento de rápido crescimento da indústria de entretenimento. Concepção e desenvolvimento de jogos de computadores modernos podem ser uma atividade complexa que envolve muitos participantes em uma variedade de ramos. No entanto, o projeto de jogo de computador apresenta abordagens normalmente que parecem ser menos formal do que aquelas utilizadas para outros tipos de sistemas de software.

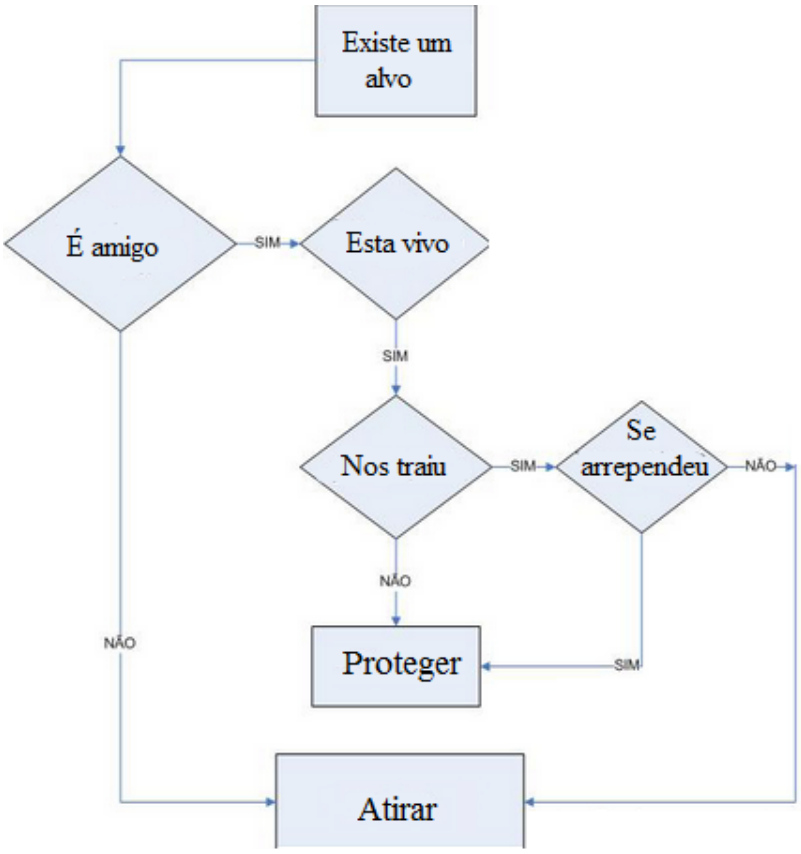
O processo de criação de um jogo de computador do ponto de vista clássico da indústria de vídeo games é essencialmente decomposto em duas fases: *Game Design* e *Level Design*. O *Game Design* é a fase que define a época, estilo, contexto, objetivos a serem alcançados, principal tipo de objetos como cenário, personagens, elementos de apoio, envolvidos, assim como a percepção do jogo pelos utilizadores. O *Level Design* consiste na definição de um espaço virtual, a criação de quebra-cabeças, e a definição das principais ações do jogador (OLIVEIRA; JúLIA; PASSOS, 2011).

Os jogos de computador são aplicações muito popular e bem-sucedida. Envolvem o desenvolvimento de softwares de grande porte que podem ser compostos por centenas de milhares, ou mesmo milhões de linhas de código. O desenvolvimento de jogos de computador é diferente de outros tipos de desenvolvimento de software, porque os jogos de computador incluem conteúdo artístico. Além disso, a maioria dos jogos de computador têm um sistema de controle, ou seja inter operação de ambientes e tarefas, que é bem diferente de outras aplicações de software (TAYLOR; GRESTY; BASKETT, 2006).

O uso de métodos de desenvolvimento para design de jogo, desde métodos sistemáticos à repetitivos permitem a retenção do que funciona atualmente e a melhoria do que não funciona bem. No entanto, a ausência de metodologias estabelecida de design de jogos de computador são um fator que atrapalha a concepção atual de jogos mais robustos e com características de inteligência artificial.

Em muitos casos as empresas de desenvolvimento de jogos não elaboram um processo de projeto para o desenvolvimento de software. A indústria de jogos de computador tem criatividade de sobra para gerar cada vez mais situações de grande gama de complexidade e desafio, se tornando maiores, mais rápidos e mais complexos.

Figura 1 – Exemplo de um fluxograma para um ambiente de jogo de guerra.



Fonte: O autor.

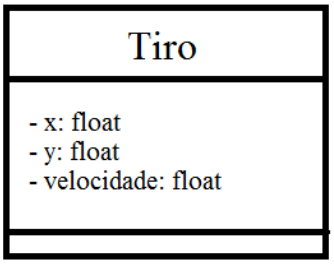
Diagramas de casos de uso da UML e outras linguagens formais como as redes de Petri em geral podem ser estendidos e adaptados ao incorporar aspectos de decisão para fornecer um meio de projetar jogos de computador que não são úteis apenas para programadores de jogos, mas também para outros profissionais envolvidos no desenvolvimento de outras partes do jogo (por exemplo, a história, o nível, os designers de personagens).

O uso de fluxogramas para design de jogos de computador simplesmente mostram como cada cena ou missão em um jogo de computador passa para a próxima cena ou a missão, por meio de interpolações de movimento descritos graficamente entre objetos e setas em um nível de design e modelagem com pouca formalidade. No entanto, fluxogramas não representam todas as visões no processo de design de jogos, com exceção de auxiliar na comunicação do progresso do jogo para os outros membros da equipe de desenvolvimento.

Na Figura 1 pode-se observar um exemplo de fluxograma para um jogo de ação(BUCKLAND, 2002).

O desenvolvimento de jogos digitais apontam variações do conceito básico de casos de uso na modelagem de sistemas envolvidos, particularmente para o suporte específico dos aspectos do desenvolvimento de software, tais como os *scripts* de tarefas. Uma classe de UML é uma descrição de um conjunto de objetos que compartilham os mesmos atributos, operações, relacionamentos e semântica (RUCKER, 2003). Uma classe em um diagrama mostra um conjunto de classes, interfaces e colaborações de seus relacionamentos, como pode ser visto no exemplo da Figura2.

Figura 2 – Exemplo de classes UML.



Fonte: O autor.

Apesar da contribuição valiosa os modelos apresentados como suporte para a modelagem de jogos, como fluxogramas e diagramas UML, não apresentam formalismo evidente para a verificação de corretude na produção inteligente de agentes para núcleo de jogos.

Métodos formais são linguagem/métodos matemáticos, muitas vezes apoiadas por ferramentas, para desenvolver software e hardware. O rigor matemático permite aos desenvolvedores analisar e verificar esses modelos em qualquer parte do ciclo de vida do projeto de software: engenharia de requisitos, arquitetura, design, implementação, teste, manutenção e evolução (WOODOCK; LARSEN; BICARSEGUI, 2009).

Por sua vez as redes de Petri oferecem suporte a verificação da corretude do sistema especificado, além disso, proporcionam um bom nível de abstração em relação a outros modelos gráficos. O emprego das redes de Petri para modelagem e especificação do comportamento dinâmico de sistemas é efetivado por diferentes razões e apresenta formalismo que serão evidenciadas nos próximos capítulos.

CAPÍTULO 3

SISTEMAS DE WORKFLOW

Este capítulo apresenta as principais definições teóricas sobre os sitemas de WorkFlow, que se caracterizam pela capacidade de execução de um conjunto de processos de negócios e são aplicados a uma variedade de áreas que vai desde componentes da administração, como também em sistemas de informação que necessitam de organização de objetivos em fluxos de dados para gerenciamento e controle de trabalho.

3.1 Conceitos Sobre Workflow

O contexto de sistemas de Workflow apresenta ferramentas que objetivam a automação e gestão de processos e que possibilitam o acompanhamento e distribuição de atividades que compõe fluxo de trabalho ao longo de sua execução. De maneira geral um WorkFlow é relacionado ao tratamento de processos.

As tecnologias de Workflow tem sido utilizadas para modelar e automatizar processos de negócio. Processos de negócio consistem em um conjunto de atividades relacionadas que, coletivamente, visam atingir um objetivo, dentro do contexto de uma estrutura organizacional (AALST V. D., 1998). Tendo em vista a automação total ou parcial de um processo de negócio, um Workflow pode ser visto como análise em um formato compreensível por uma máquina.

Workflow é a automação de um processo de negócio, por inteiro ou por partes, durante o qual documentos, informações e atividades são passadas de um participante para outro para que estes desenvolvam ações respeitando um conjunto de regras que irá compor o sistema em sua totalidade.

Sistemas de gerenciamento de WorkFlow são, em geral, ferramentas que promovem

de forma colaborativa a automação de procedimentos para o gerenciamento de processos de negócios (AALST; HEE, 2002).

Sistemas de WorkFlow tem se tornado uma solução para ambientes de gestão de processo onde de forma eficaz consegue gerenciar, por exemplo, sequência de atividades selecionando e atribuindo recursos que irão compor futuramente processos organizados.

A descrição (modelo) de um processo a ser automatizado em um sistema de Workflow deve conter todas as informações necessárias sobre os processos a serem executados pelo sistema. Estas informações incluem dados sobre as atividades que compõem os processos, suas condições de início e finalização, regras para sua efetivação, usuários, encarregados, documentos manipulados em cada atividade, aplicações a serem empregadas, etc. Apresentamos, a seguir, uma breve descrição dos componentes do sistema de Workflow (AALST; HEE, 2002) :

- **Caso**: É representado por uma ficha sendo um caso a ser tratado pelo sistema.
- **Tarefa**: Unidade lógica de trabalho que é executada integralmente por um recurso(ficha/caso).
- **Atividade**: Consiste na execução de uma tarefa realizada por um recurso. Uma atividade corresponde a um conjunto de procedimentos que colaboram para que o processo alcance determinado objetivo.
- **Participante**: Também conhecido como ator, agente ou usuário do WorkFlow tem a capacidade de assumir um papel e realizar uma atividade durante a execução de uma instância do WorkFlow.
- **Regra**: A definição de um fluxo de trabalho adota um conjunto de regras para sua execução. As regras determinam quais informações transitarão pelo fluxo de trabalho e sob quais condições.
- **Rota**: Consiste no caminho lógico que, sob regras específicas, uma informação passa por transferência da informação dentro do processo. No roteamento são considerados quatro tipos básicos de rotas:

  Rota serial: apresenta fluxo linear e direto onde uma tarefa é executada após a outra;

  Rota paralela: grupo de atividades que possuem execução simultânea;

  Rota iterativa: a mesma atividade é executada varias vezes.

  Rota condicional: múltiplas rotas podem ser utilizadas e a escolha é feita por meio de determinada regra, procedimento ou variável;

Um processo de WorkFlow pode tratar vários casos, que representam uma ficha em disparo no sistema WorkFlow. Assim uma mesma tarefa pode ser executada por vários casos. A representação de processos de WorkFlow por meio dos modelos, devem ser capazes de representar situações como atividades e suas sincronizações, tarefas e recursos determinados para realizar atividades, como também regras de negócio e tratamento de exceções para aspectos temporais(deadline, durações).

3.2 Modelagem de Workflows

De forma geral, sistema de WorkFlow pode ser representado por redes de Petri, mas conhecidas como Workflow-Nets, um caso particular de Workflow que apresenta apenas um lugar de início (start) e um de término (end). O conceito de *soundness*(corretude) é um critério de verificação de correção, ou seja, uma Workflow-net é sound(correto) se, e somente se, três condições são válidas (AALST; HEE, 2002):

- Para cada ficha colocada no lugar de início, apenas uma aparecerá no lugar de término;
- Quando uma ficha aparecer no lugar de término, os demais lugares estão vazios;
- Para toda a transição (tarefa), é possível evoluir da marcação inicial até a marcação que sensibiliza tal transição.

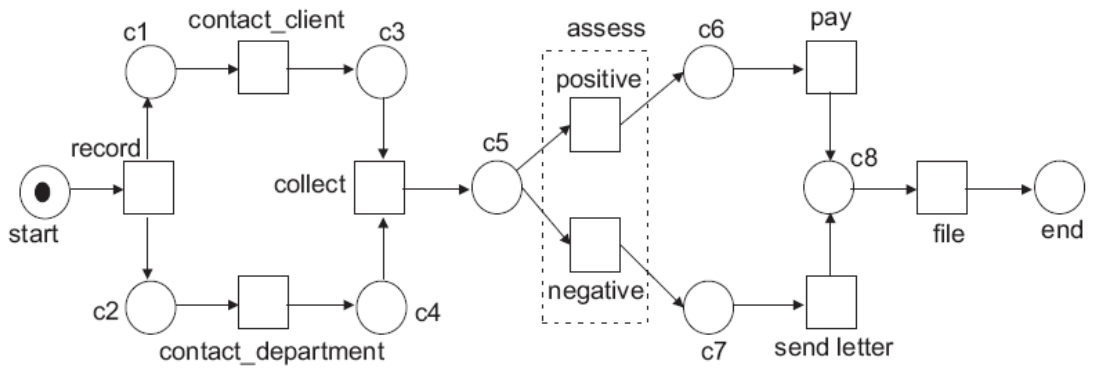
Alguns elementos, como caso, processo, tarefa, etc., presentes em sistemas de Workflow devem ser considerados na representação usando as redes de Petri. Um processo especifica as tarefas e define a ordem em que elas são executadas. Em uma rede de Petri, um processo tem um lugar de entrada, sem arcos de entrada, e um lugar de saída, sem arcos de saída. Os lugares (componentes passivos) em uma rede de Petri representam as condições e as transições (componentes ativos), as tarefas.

Na Figura 3 cada uma das tarefas *record, contactClient, contactDepartament, pay e file* é representada na rede de Petri por uma transição em forma de retângulos.

Os lugares *start* e *end* indicam o início e o fim do processo modelado. Os demais lugares da rede referem-se a condições que não podem ser cumpridas por todos os casos em andamento. Estas condições desempenham duas funções importantes: assegurar que as tarefas avancem na ordem correta e estabelecer o estado do processo. Considere-se por exemplo, o lugar c8: este lugar é responsável por indicar o estado de que a reclamação será arquivada quando for completamente resolvida.

Os casos são representados pelas fichas presentes nas redes. No lugar *start* existe uma ficha, indicando a presença de um caso. Se a transição *record* é disparada, duas fichas (uma em c1 e a outra em c2) representam o mesmo caso. Quando um caso é tratado, o

Figura 3 – Exemplo de WorkFlow-net



Fonte: (VALE, 2009)

número de fichas pode variar. Mas, a quantidade de fichas que representa um caso particular é sempre igual ao número de suas condições que devem ser satisfeitas. No lugar *end* deverá haver uma ficha, quando o caso for concluído.

Cada processo deve cumprir dois requisitos: em qualquer momento, poder atingir um estado, por meio da execução de tarefas, e, quando existir uma ficha em *end*, verificar se as demais desapareceram do restante dos processos. Dessa forma, é possível garantir que todo caso que se iniciou no lugar *start*, será completado corretamente e finalizado no lugar *end*.

As fichas que se referem a casos particulares são separadas pelo sistema de gerenciamento do WorkFlow. Os sistema de gerenciamento de Workflow são capazes de organizar e gerar fluxos de controles de forma automática para processos de negócios. As fichas presentes no sistema WorkFlow pertencem a casos diferentes e não podem influenciar uma à outra, pode-se gerar uma cópia para cada caso, que, assim, terá seu próprio processo. Ao se representar essa característica em uma rede de Petri, podem ser usadas as redes de Petri de alto nível por exemplo, em que cada ficha possui uma identificação, a fim de identificar a que caso a ficha se refere.

CAPÍTULO 4

REDES DE PETRI

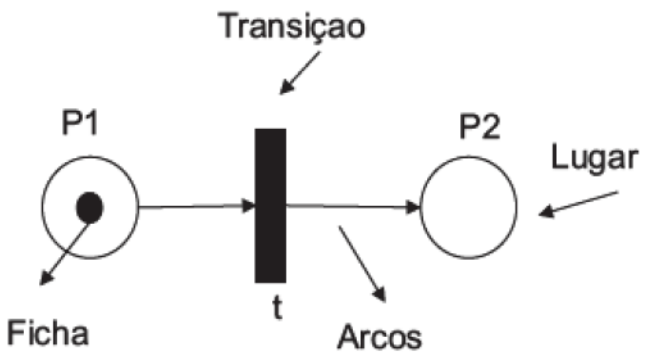
A rede de Petri é um modelo gráfico e matemático que vem sendo amplamente utilizada, em vários domínios de atuação, entre os quais destacam-se os sistemas de informação, de comunicação, logísticos e, de forma geral, todos os sistemas a eventos. Especificar, analisar o comportamento lógico, avaliar o desempenho de sistemas são as principais motivações para o uso da Rede de Petri (MURATA, 1989).

4.1 Definição

Inicialmente Carl Adam Petri postulou as redes de Petri em *Comunicação com Autômatos* (Petri, 1966) (DAVID, 2004). Simplificando uma rede de Petri como mostrado na Figura 4, obtem-se um grafo orientado contendo os seguintes elementos:

- **Lugar**: representa um estado, uma espera, um recurso etc.;
- **Ficha**: indica uma condição associada ao lugar, ou um objeto;
- **Transição**: representa ação ou evento que ocorre em um sistema;
- **Arcos orientados**: conectam os lugares às transições e estas aos lugares;

Figura 4 – Exemplo, elementos de uma rede de Petri.



Fonte: (VALE, 2009)

4.2 Formalizando Redes de Petri

Formalmente uma rede de Petri é definida como uma quádrupla da Equação4.1 contendo(MURATA, 1989):

$$R =< P, T, Pre, Post > \tag{4.1}$$

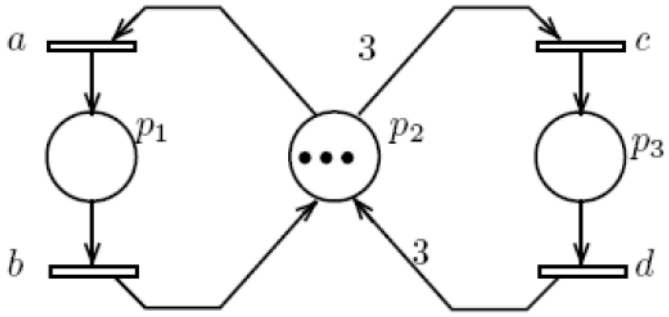
onde:

- P é um conjunto finito de lugares em dimensão n;
- T é um conjunto finito de transições de dimensão m;
- *Pre* de P x T em $\mathbb{N}$ é uma aplicação de entrada (lugares precedentes ou incidência anterior), com $\mathbb{N}$ sendo o conjunto dos números naturais;
- *Post* de P x T em $\mathbb{N}$ é uma aplicação de saída (lugares seguintes ou incidência posterior).

A quádrupla $R =< P, T, Pre, Post >$ com $P = (p1, p2p3)$, $T = (a, b, c, d)$ e os valores das aplicações de entrada e saída dados por: $Pre(p2, c) = 3$, $Pre(p1, b) = Pre(p2, a) = Pre(p3, d) = 1$, $Post(p2, d) = 3$ e $Post(p1, a) = Post(p2, b) = Post(p3, c) = 1$, indica uma rede de Petri como demonstrada na Figura 5.

A vantagem das redes de Petri em relação a outros modelos, deve-se em oferecer uma representação gráfica para especificação formal, sendo possível obter informações sobre o comportamento do sistema modelado, através da análise de suas propriedades, gerais ou estruturais (CARDOSO; VALETE, 1997). Uma rede de Petri marcada pode ser definida por N onde $N =< R, M >$, sendo que:

Figura 5 – Exemplo de rede de Petri com fichas

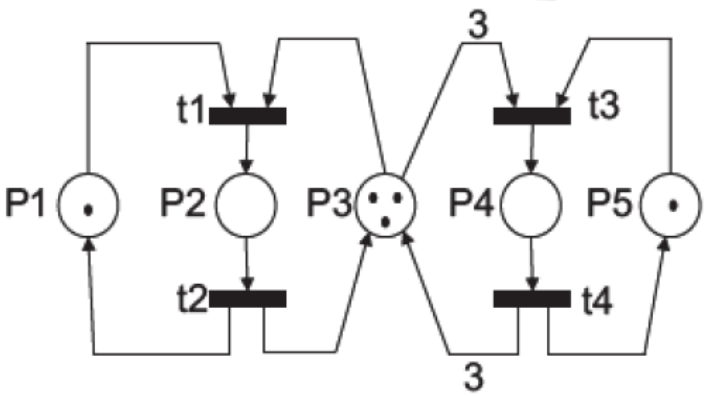


Fonte: O autor.

- R uma rede de Petri.
- M a marcação inicial dada pela aplicação M : P em ℕ.

As fichas presentes em cada lugar *P* são tidos através de *M(P)* representados por tokens. A marcação *M* é a distribuição das fichas nos lugares, sendo representada por um vetor coluna, que demonstra os valores de fichas em cada lugar, cuja dimensão é o número de lugares e elementos *M(P)*. Um exemplo de rede de Petri marcada é mostrado na Figura 6 em que a marcação é dada por $M(t) = (10301)$.

Figura 6 – Exemplo de rede de Petri marcada



Fonte: (VALE, 2009)

4.3 Transição Sensibilizada

Uma transição t está sensibilizada ou habilitada se e somente se:

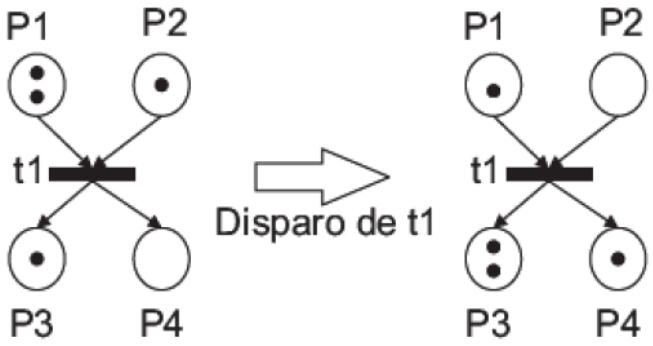
$$\forall p \epsilon P, M(p) \geq Pre(p,t) \tag{4.2}$$

Sendo assim, uma transição está sensibilizada, como mostra a Equação4.2, se o número de fichas em cada um dos seus lugares de entrada for maior (ou igual) ao peso do arco que liga este lugar à transição. A equação pode ser escrita na forma *M >= Pre(n , t)* onde o vetor coluna *Pre(n , t)* é a coluna da matriz Pre referente à transição *t* e *M*, o vetor marcação inicial e n, um lugar qualquer na rede. Um exemplo de disparo de transição é dado na Figura7 onde a notação matricial para condições *Pre* e *Post* podem ser verificadas por:

Pre(p1, t1)= 1
Pre(p2, t1)= 1
Pre(p3, t1)= 0
Pre(p4, t1)= 0

Post(p1, t1)= 0
Post(p2, t1)= 0
Post(p3, t1)= 1
Post(p4, t1)= 1

Figura 7 – Exemplo de disparo de uma rede de Petri



Fonte: (VALE, 2009)

Os lugares das fichas e suas variações são constatadas na Figura 7, representam o comportamento dinâmico que o sistema modelado pode apresentar. Vale ressaltar que as interpretações dos lugares e fichas são variadas, ou seja, dependem do contexto em que são criados (CARDOSO; VALETE, 1997).

Uma rede de Petri pode apresentar características de autonomia através de propriedades que podem ser verificadas pela dependência que apresentam na marcação inicial e a forma em que são agrupadas. Dentre elas estão:

- **Alcançabilidade**: uma marcação *M(n)*, sendo n o número de lugares, é dita alcançável se existe uma sequência finita de disparos de transições que possibilita a chegada a *Mn*

a partir da marcação inicial *Mo*. Essa propriedade garante que determinados estados sempre serão atingidos.

- **Limitabilidade**: uma rede é limitada ou k-limitada se o número de fichas em cada lugar não excede um número finito k para qualquer marcação alcançável a partir da marcação inicial.

- **Vivacidade**: uma rede é considerada viva se toda transição t pode ser sensibilizada a partir de qualquer marcação M' do grafo de marcações alcançáveis. Esse conceito, na prática garante que o sistema será livre de bloqueios (*deadlock free*).

- **Reiniciabilidade**: uma rede de Petri é reiniciável se a partir de qualquer marcação acessível, existe uma sequência de disparo que leva à marcação inicial Mo.

4.4 Classificação das Redes de Petri

A solução para a maioria dos problemas da grande diversidade de sistemas que necessitam ser representados de forma clara e sem ambiguidades estão nas várias extensões das redes de Petri. As redes de Petri de alto nível[1] são classificadas da forma que segue (MURATA, 1989).

- Redes de Petri interpretadas: são associadas variáveis às transições da rede, que representam condições e ações existentes no sistema. As variáveis podem indicar, ainda, o estado de atuadores, sensores, etc. Além de representarem o estado do sistema através da distribuição de fichas estas redes conseguem também carregar informações.

- Redes de Petri Coloridas: são atribuídas cores as fichas destas redes no intuito de diferenciar umas das outras, o que permite representar diferentes processos e recursos. Além disso, as marcas representam outros recursos, como por exemplo a existência de uma outra rede associada às marcas (rede dobrada). Dessa forma, permite reduzir o tamanho da rede (DAVID, 2004).

- Redes de Petri Híbridas e Redes de Petri Contínuas: em uma rede de Petri contínua, a marcação de um lugar e a taxa de disparo corresponde a uma variável contínua (número real não negativo). A marcação contínua é movida de um lugar para outro determinado por uma velocidade (DAVID, 2004). Esse modelo foi essencialmente desenvolvido para a modelagem e simulação de sistemas de produção híbridos que envolvia fluxos contínuos de produtos.

[1] Redes que possuem fichas associadas aos lugares, no qual, carregam informações que serão manipulados ao longo da execução da rede.

- Redes de Petri Temporais/Temporizadas: são usadas na modelagem de sistemas que levam em consideração o fator tempo. Suas principais aplicações são nas simulações, no diagnóstico, na supervisão e na análise de desempenho de sistemas. Nestas redes, o tempo é representado por durações associadas ao lugar (p-temporizadas) ou à transição (t-temporizadas), (TAZZA, 1987).

- Redes de Petri Estocásticas: é incluído um tempo aleatório associado ao disparo de uma transição. A definição do tempo nestas redes é definida através de uma distribuição que segue uma lei exponencial permitindo atribuir um processo Markoviano equivalente para o tempo de falha de uma máquina, por exemplo.

- Redes de Petri Predicado/Transição: descrevem de maneira estruturada o conjunto controle e dados. Nestas redes, os lugares são chamados de predicados, as marcas são condições válidas do predicado e as transições são consideradas como regras da lógica de primeira ordem, isto é, regras com variáveis. Por exemplo, em Champagnat, Pingaud e Alla (1998) as redes de Petri predicado/transição são abordadas para serem integradas a uma sequência de equações algebro-diferenciais para a modelagem de estocagem de gás.

- Redes de Petri a Objetos: podem ser consideradas como uma extensão das redes de Petri predicado/transição que aborda o conceito de paradigma orientado a objetos (SIBERTIN-BLANC, 1985). A principal motivação do uso das redes de Petri a objetos se deve principalmente a capacidade de representar características de concorrência, paralelismo, fluxo de controle e estruturas de dados apresentados nos sistemas. As redes de Petri a Objetos são apresentadas especificadamente no próximo tópico.

4.5 Redes de Petri a Objetos

Buscando entender melhor o conceito de redes de Petri a objetos considera-se primeiro as principais características do conceito de orientação a objetos.

Inicialmente, para superar algumas deficiências do paradigma procedimental criou-se o paradigma orientado a objetos. A produção de um sistema na forma estruturada era feita a partir de procedimentos/funções que manipulavam um determinado conjunto de dados. No entanto quando existiam grandes problemas a serem solucionados, a complexidade do programa se tornava tão alta a ponto de que a necessidade de uma simples alteração implicava em profundas alterações em tais procedimentos e funções que tinham acesso aos dados (DELAMARO; JINO, 2007).

Buscando diminuir a complexidade destes programas e facilitar a manipulação de dados, surgiu o paradigma orientado a objetos. Esta é na verdade uma forma de trabalhar os dados, porém , de maneira isolada (encapsulada), ou seja, cria-se uma entidade (classe),

onde os dados (atributos) e as funções/procedimentos (métodos) são agrupados para a realização de serviços (operações) sobre os atributos. Desta forma o objetivo deste paradigma é forçar o desenvolvedor a pensar numa abstração de mais alto nível em termos de classes e objetos (BEZERRA, 2007).

Os principais elementos que compõe o paradigma orientado a objetos é dado a seguir:

- **Objetos**: caracterizados pela instanciação de uma classe, sendo esta portanto, apenas uma definição. Aos atributos dos objetos são atribuídos valores que são manipulados e executados por operações (métodos).
- **Encapsulamento**: confere aos objetos certa independência funcional e proteção contra acessos não autorizados. Dessa forma, os dados só podem ser alterados, através de métodos definidos pela classe ou para a classe.
- **Abstração**: enfatiza detalhes significantes e suprime outros em um dado momento, relacionando-se assim, com a interface do objeto.
- **Polimorfismo**: referente ao fato de uma mesma mensagem poder resultar em eventos diferentes quando recebida por objetos diferentes ou quando enviada com parâmetros diferentes.
- **Modularidade**: consiste em dividir um programa em partes como funções individuais a fim de reduzir a complexidade do código. Duas características de modularidade são coesão, confere independência funcional ao software, e acoplamento que diz respeito à interconexão entre módulos para a execução de tarefas. O ideal é um software que apresente alta coesão e baixo acoplamento.
- **Persistência**: (tempo de vida) propriedade de um objeto continuar a existir, mesmo após o seu criador não mais existir.

As redes de Petri a objetos (SIBERTIN-BLANC, 1985) relacionam as redes de Petri predicado/transição e o conceito de paradigma orientado a objetos. Estas redes consideram as fichas como n-uplas de instâncias de classes de objetos e transportam verdadeiras estruturas de dados definidas como conjuntos de atributos de classes específicas. Nas transições, por sua vez, são associadas pré-condições e ações, que respectivamente, atuam sobre atributos das fichas e modificam seus valores.

Na estrutura de redes de Petri a objetos, os lugares estão associados as classes; as transições são associadas as operações que atuam sobre os atributos localizados nos lugares de entrada; e os objetos correspondem as fichas da rede. Dessa forma, a operação de uma determinada transição t só poderá ser executada por um objeto se estiver num lugar de entrada de t. Formalizando redes de Petri a objetos, tem-se:

Uma rede de Petri a objetos marcada é definida pela 9-upla conforme a Equação 4.3:

$$No = < P, T, Class, V, Pre, Post, Atc, Ata, Mo > \tag{4.3}$$

onde:

- *Class* representa um conjunto finito de classes de objetos: para cada classe é definido também um conjunto de atributos;
- *P* é um conjunto finito de lugares, cujos tipos são dados por *Class*;
- *T* é um conjunto finito de transições;
- *V* é um conjunto de variáveis, cujos tipos são dados por *Class*;
- *Pre* é a função lugar precedente, que a cada arco de entrada de uma transição, faz corresponder uma soma formal de elementos de *V*;
- *Post* é a função lugar seguinte, que a cada arco de saída de uma transição faz corresponder uma soma formal de elementos de *V*;
- *Atc* é uma aplicação que a cada transição associa uma condição que envolve os atributos das variáveis formais associados aos arcos de entrada;
- *Ata* é uma aplicação, que acada transição associa uma ação que envolve os atributos das variáveis formais associadas aos arcos de entrada e atualiza os atributos das variáveis formais associadas aos arcos de saída;
- *Mo* é a marcação inicial, que associa a cada lugar uma soma formal de objetos (n-uplas de instâncias de classes que pertencem a *Class*).

O exemplo de rede de Petri da Figura 8 é definido pelo conjunto de classes por: *Class = Posicao, Jogadas*.

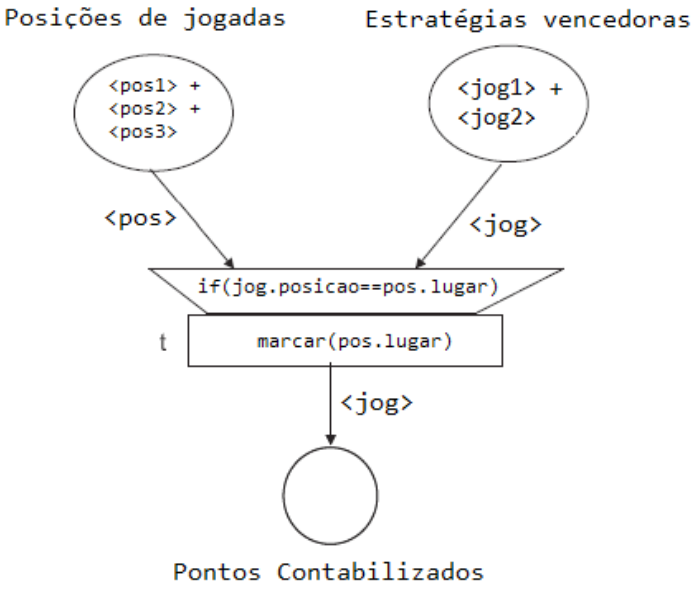
A classe *Posicao* possui os seguintes atributos:

- *lugar = boolean;*
- *valor = integer;*

A classe *Jogadas* possui os seguintes atributos:

- *posicao=integer* ;
- *pontuacao=integer*;

Figura 8 – Exemplo de rede de Petri a objetos



Fonte: O autor.

A variável (classe) *pos* é do tipo *Posicao* e a variável (classe) *jog* é do tipo *Jogadas*. O lugar *Posição de jogadas* é do tipo *Posicao* e retem os objetos da classe que indicam quais possíveis posições de um jogo de tabuleiro para serem marcados, o lugar *Estratégias vencedora* é do tipo *Jogadas* e possui uma matriz com posições de estratégias para vencer a disputa do jogo de tabuleiro. O lugar *Pontos contabilizados* é do tipo *Jogadas* e contém a marcação de pontos efetuados com marcações em posições ganhadoras. A marcação inicial *Mo* é dada pelos objetos que se encontram nos lugares.

Os valores atribuídos a cada ficha nos estados listados podem ser atribuídos por valores de atributos como por exemplo:

Posicao pos1;
lugar: false
valor: 2

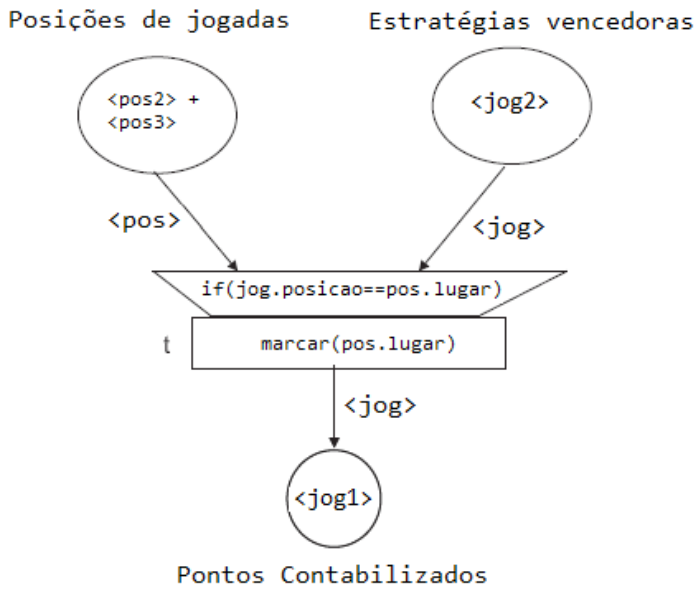
Jogada jog1;
posicao: 2
pontuacao: 2

Com estes valores citados no exemplo, onde, os atributos da classe indicam um ponto, já que o atributo *posicao* da classe Jogada indicava valor igual a 2 do tabuleiro, como uma boa

opção vencedora. Por sua vez o tabuleiro estava com esta posição desmarcada como indica a classe *Posicao*, com o atributo valor indicando uma posição de marcação 2 no tabuleiro e atributo lugar igual a false. Isso indicava que a posição ainda estava desmarcada.

Com a sensibilização da transição *t* o objeto *jog1* assumiria o estado *Pontos contabilizados*, desta forma o atributos *pontuação* da classe *Jogada* seria acrescido de mais um ponto. Enquanto isso o atributo *lugar* da classe *Posicao* assumiria valor 'true' indicando posição marcada. A rede de petri da Figura8 após sua sensibilização pode ser vista pela Figura9. A definição detalhada que fixa regras de sensibilização e disparo das transições das redes de Petri a objetos encontram-se em (SIBERTIN-BLANC, 1985).

Figura 9 – Exemplo de rede de Petri a objetos após o disparo de fichas



Fonte: O autor.

4.6 WorkFlow-net

Aalst v. d. (1998) definiu o uso de redes de Petri para a formalização dos processos de WorkFlow, originando as WorkFlow-nets, em seu trabalho, para que a verificação de propriedades qualitativas fossem possíveis através das especificações formais e matemáticas das redes de Petri no contexto como Workflow-Nets.

As Workflow-Net apresentam as seguintes propriedades:

- Redes de Petri com apenas um lugar de entrada (denominado *Start*) e apenas um lugar de saída (denominado *End*). Estes dois lugares são tratados como lugares especiais, o lugar *Start* tem apenas arcos de saída e o lugar *End* apenas arcos de entrada.

- Uma ficha em *Start* representa um caso que precisa ser tratado e uma ficha em *End* representa um caso que já foi tratado.

- Toda tarefa(transição) e condição(lugar) deve estar em um caminho que se encontra entre o lugar *Start* e o lugar *End*.

Em uma WorkFlow-Net o conceito de processo determina categoria de casos que devem ser tratadas. O processo define quais as tarefas devem ser realizadas pelos casos, que são representados pelas fichas. De forma mais simplificada o caminho que um caso percorre do lugar *Start* até o destino *End* é alocado a um processo.

Alguns elementos apresentam as condições que constituem a Workflow-Net. Para a transição de uma Workflow-Net em uma rede de Petri ou até mesmo aplicação de verificação qualitativa do modelo com a lógica linear como será visto nas próximas seções deste trabalho, é usada semântica de transição, descrita na Figura 10. Onde:

- *Or-Split*: apontamento dentro do WorkFlow onde uma única linha de controle decide sobre qual ramificação escolher quando encontrar múltiplas alternativas em um WorkFlow.

- *And-Split*: apontamento dentro do Workflow onde uma única linha de controle se divide em duas ou mais tarefas, as quais são executadas em paralelo dentro do Workflow.

- *Or-Join*: apontamento dentro do Workflow onde duas ou mais ramificações de atividades alternativas se convergem para uma única atividade comum com a próxima etapa dentro do Workflow.

- *And-Join*: apontamento no Workflow onde duas ou mais atividades executando paralelamente se convergem em uma única tarefa de controle.

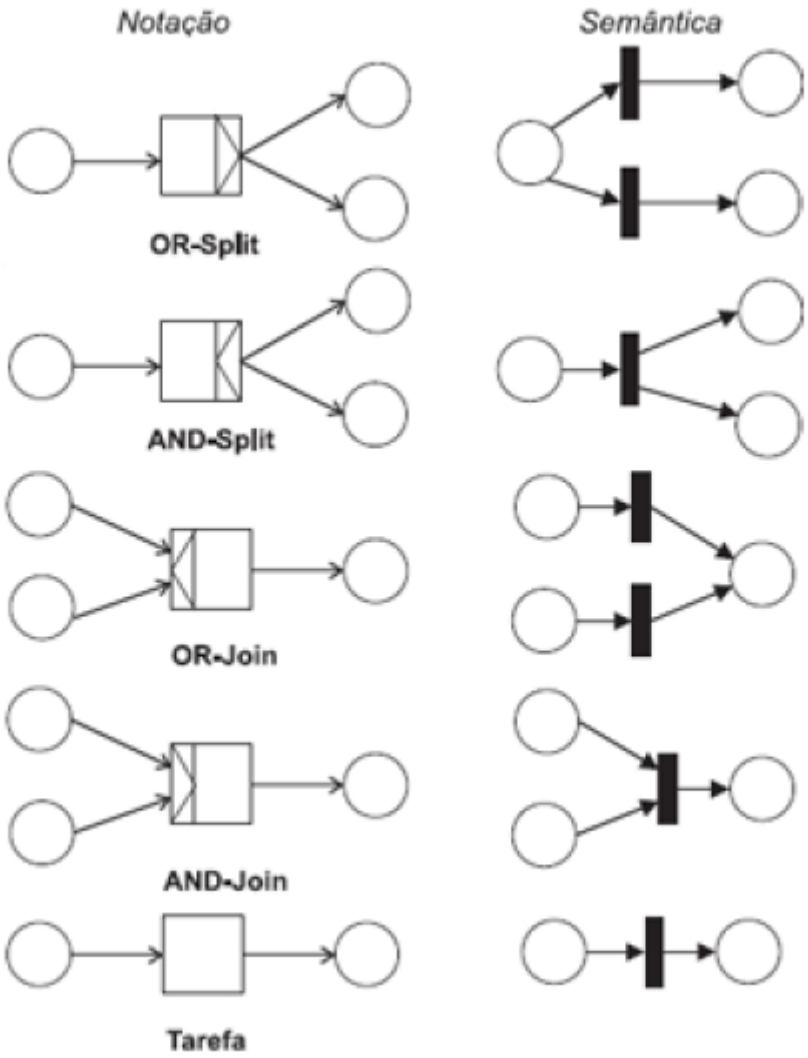
4.6.1 Propriedades das WorkFlow-nets

Um critério para verificação de correção para as WorkFlow-nets são definidos pela propriedade *Soundness*. Em particular uma WorkFlow-net é *sound* se, e somente se, as três condições a seguir forem satisfeitas:

- Para cada ficha colocada no lugar de início, uma (e apenas uma) ficha aparecerá no lugar de término.

- Quando uma ficha aparece no lugar de término, todos os outros lugares estão vazios, considerando o caso em questão.

Figura 10 – Descrição da semântica de transições das WorkFlow-net



Fonte: (AALST; HEE, 2002)

- Considerando uma tarefa associada à uma transição, é possível evoluir da marcação inicial até uma marcação que sensibiliza tal transição, ou seja, não deve haver nenhuma transição 'morta' na WorkFlow-net.

A corretude é um critério importante a ser satisfeito quando se trata de processos de WorkFlow e a prova desse critério está relacionada com a análise qualitativa, no contexto das WorkFlow-nets. A verificação de corretude será realizada mais a frente por meio do uso de conceitos da lógica linear e testes com usuários quando o agente jogador estiver implementado.

CAPÍTULO 5

TÉCNICAS PARA PRODUÇÃO DE AGENTES INTELIGENTES

Este capítulo tem por objetivo apresentar mais um fundamento que envolve este trabalho, completando o aparato teórico para a concepção de uma abordagem positiva através de modelagem formal com redes de Petri para um agente jogador. Iremos mostrar como a inteligência artificial se apresenta dentre seus conceitos e aplicações para produção de agentes inteligentes.

5.1 Agentes Inteligentes

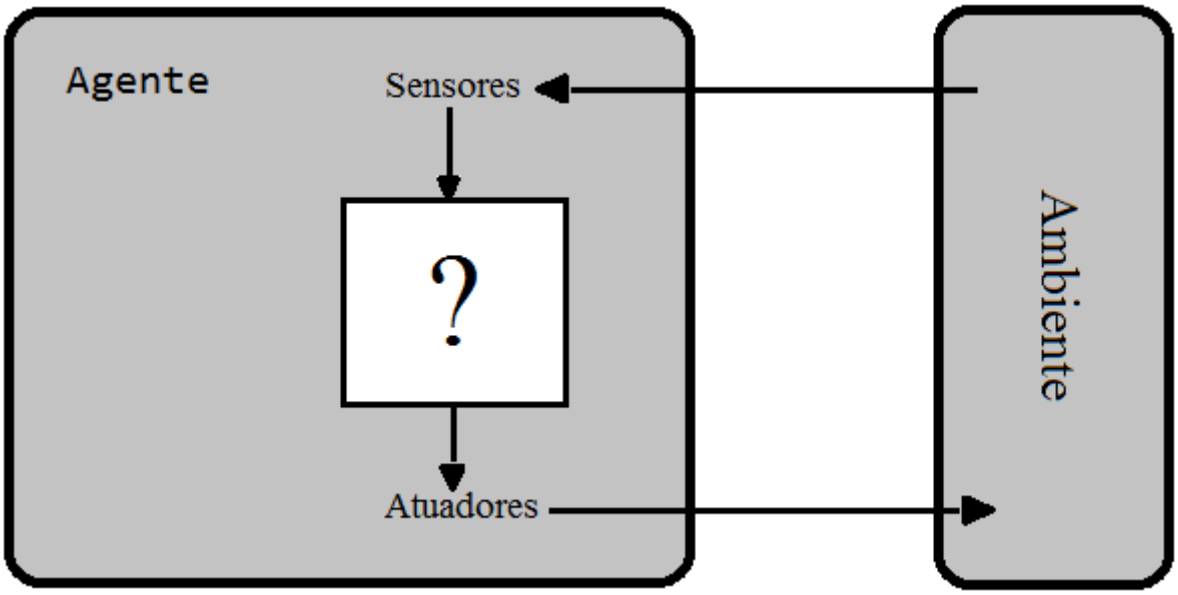
O estudo de agentes inteligentes está integrado aos ambientes e acoplamento entre eles. A observação de que alguns agentes se comportam melhor que outros leva naturalmente à ideia de agente racional, ou seja, uma agente que comporta tão bem quanto possível.

A medida de qualidade do comportamento de um agente depende da natureza do seu ambiente e as propriedades dos mesmos influenciam o projeto de agentes adequados para esse ambiente.

Um **agente** é tudo o que pode ser considerado capaz de perceber seu **ambiente**(meio em que o agente está inserido) por meio de **sensores**(ferramentas que captam as características do ambiente) e de agir sobre esse ambiente por intermédio de **atuadores**(ferramentas do agente que executa ações no ambiente). Em uma analogia, um ser humano tem olhos, ouvidos e outros órgãos como sensores, e tem mãos, pernas, boca e outras partes do corpo que servem como atuadores. Na Figura11 é possível perceber esta arquitetura de funciona-

mento de um agente (SILVA, 2003).

Figura 11 – Modelo gráfico para descrição sumária de um agente.



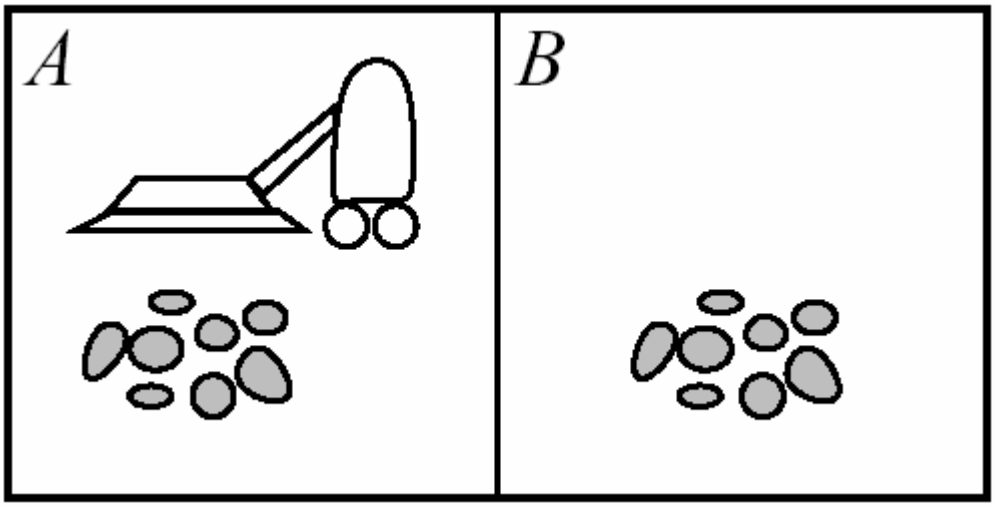
Fonte: [Russel, Norvig, 2004]

Um agente pode perceber de modo geral suas próprias ações mas nem sempre seus efeitos. A **percepção** faz referência as entradas perceptivas do agente em qualquer dado momento. A **sequência de percepções** do agente é a história completa de tudo que o agente já percebeu. Em geral a escolha da ação de um agente em qualquer instante dado pode depender da sequência inteira de percepções observadas até o momento. Afirmamos que o comportamento do agente é descrito pela **função de agente** que mapeia qualquer sequência de percepções específica para uma ação (RUSSEL; NORVIG, 2004).

Internamente, a função de agente, para um agente artificial será implementada por uma programa de agente. É importante manter essas duas ideias distintas, pois a função do agente é uma descrição em termos abstratos enquanto o **programa do agente** é uma implementação concreta, relacionada a arquitetura do agente.

Na ilustração da Figura12 é demonstrado um ambiente para o aspirador de pó onde em dois quadrado A e B ele é alocado para realizar a limpeza. Em uma primeira situação quando o agente aspirador encontra em um local sujo sua ação é *Aspirar*, por sua vez se este ambiente se encontra limpo sua ação é deslocar para outra posição sendo para a *Direita* se está em A, ou para *Esquerda* se está em B. O agente pode também não fazer nada caso o ambiente esteja limpo e o adjacente já tenha sido limpo recentemente. Para ilustrar esta situação o Quadro1 lista uma sequência de percepções aleatória para o ambiente proposto e um algoritmo capaz de realizar as ações do agente para este ambiente, o que nos permite diferenciar uma função de agente de um programa de agente.

Figura 12 – Ambiente de tarefa de um agente ASPIRADOR-DE-PÓ.



Fonte: [Russel, Norvig, 2004]

Quadro 1 – Exemplo de Sequência de percepções do agente ASPIRADOR-DE-PÓ

Sequência de percepções	Ação
(A, Limpo)	Direita
(A, Sujo)	Aspirar
(B, Sujo)	Aspirar
(A, Limpo),(A, Limpo)	Direita
(A, Limpo),(A, Sujo)	Aspirar
...	...
(A, Limpo),(A, Limpo),(A, Sujo)	Aspirar

O Algoritmo1 demonstra as sequências de ações para cada percepção do agente ASPIRADOR DE PÓ:

Algoritmo 1 – Algoritmo para agente ASPIRADOR-DE-PÓ(posicao,estado)

```
função agenteAspirador ([posicao, estado]) retorna uma ação

se estado = Sujo então retorna Aspirar
senao se posição = A então retorna Direita
senao se posição = B então retorna Esquerda
```

5.2 Racionalidade

O conceito de racionalidade está ligado a fazer tudo certo. Em termos conceituais para um agente, toda entrada na tabela corresponde a função de agente ser preenchida de

forma correta. Para cada sequência das percepções possíveis, um agente racional deve selecionar uma ação que venha a maximizar sua medida de desempenho, dada a evidência fornecida pela sequência de percepções e por qualquer conhecimento interno do agente. A definição do que é racional em qualquer instante dado depende de alguns fatores:

- A medida de desempenho que define o critério de sucesso;
- O conhecimento anterior que o agente tem do ambiente;
- As ações que o agente pode executar;
- A sequência de percepções do agente até o momento.

As medidas de desempenho, ou formas de medir o sucesso de um agente não se limita ao comportamento do agente, nem é de fácil aplicação dependendo não só do agente mais do seu ambiente e finalidade. Não existe evidentemente uma medida afixada para todos os agentes, esta deve ser imposta pelo projetista e deve avaliar o resultado realmente desejado para um ambiente e não só o comportamento que se espera de um agente.

5.3 Estrutura de Agentes

Um programa de agente implementa a função de agente que mapeia percepções em ações. Supomos que este programa será executado, possivelmente, em algum tipo de dispositivo de computação com sensores e atuadores físicos. Chamamos esse conjunto de arquitetura: agente = (arquitetura, programa). O programa escolhido deve orientar o agente a funções cabíveis a sua arquitetura, considerando que ela pode ser apenas um computador.

Um **programa de agente** define uma estrutura básica que recebe a percepção atual como entrada dos sensores e retorna uma ação para os atuadores. Neste caso a diferença entre o programa de agente é que ele toma a percepção atual como entrada, e a função de agente, recebe o histórico de percepções completas, enquanto um programa recebe apenas a atual por estar disponível no ambiente para uma sequência seria preciso que ele memorizasse o conjunto de percepções. Existem quatro tipos básicos de agentes que incorporam os princípios subjacentes a quase todos os sistemas inteligentes (RUSSEL; NORVIG, 2004):

- **Agentes reativos simples**: Modelo mais modesto, seleciona ações com base na percepção atual.
- **Agentes reativos baseados em modelo**: Mantém um tipo de estado interno que depende do histórico de percepções e assim reflita pelo menos alguns dos aspectos não observados do estado atual.

- **Agentes baseados em objetivos**: Baseia-se além da descrição do estado atual do ambiente de um agente mas também sobre o objetivo que descreva situações desejáveis para o ambiente.
- **Agentes baseados na utilidade**: Possui uma função de utilidade que mapeia o grau de aceitação associado a cada decisão do agente no ambiente, podendo participar de decisões que alternam o caminho de estados percorrido pelo agente.

5.4 Agente Reativo Simples

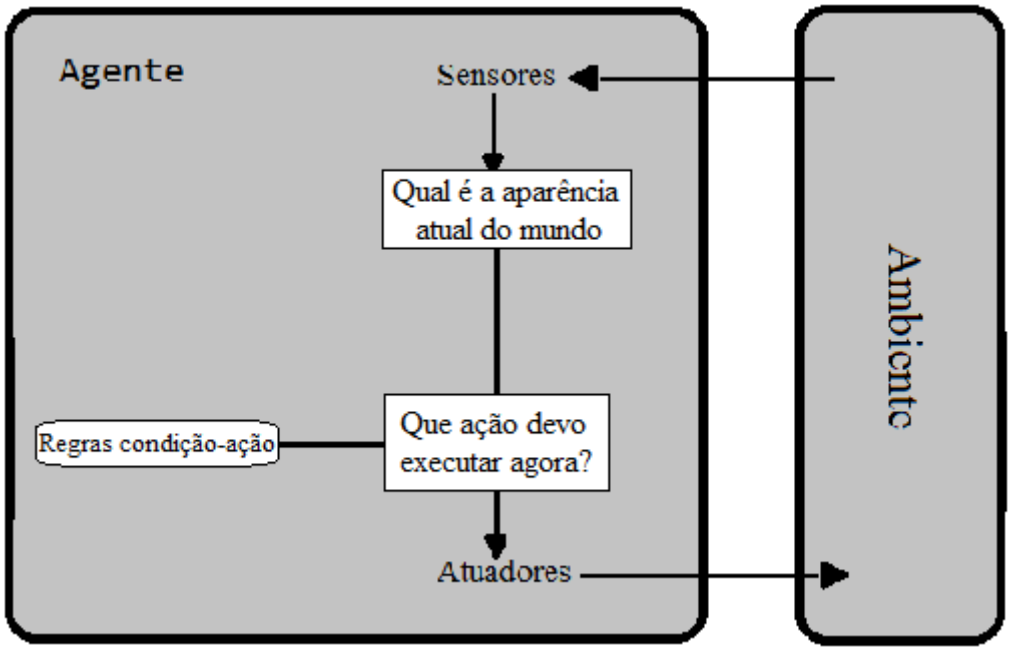
Um agente reativo simples baseia-se na percepção atual e no fato desta posição manipular o agente de acordo com suas regras definidas previamente. O programa aspirador de pó mostrado na Figura12 por exemplo, é bem simplificado em relação ao seu histórico de percepções da tabela correspondente, pois manipula o estado atual, não importando por exemplo em qual quadrado está quando encontra a sujeira. A redução mais óbvia vem de se ignorar o histórico de percepções, o que reduz o número de possibilidades. Este tipo de agente pode facilitar a implementação em ambientes esquematizados, que possuem modelos de controle para situações ambiente, onde suas implicações atuais não influenciaram de forma negativa em ações futuras, por esse motivo será utilizado em nosso estudo de caso, como forma de estruturar o agente para o jogo dos pontinhos.

A conexão estabelecida no programa do agente para a ação desejada é ativada de acordo com a percepção de conexões que podem ser descritas como **regra condição-ação, regras de produção**, ou **regras se-então** escritas para o ambiente proposto.

Uma abordagem mais geral e flexível consistem em primeiro construir um interpretador de uso geral para regras de condição-ação e depois criar conjuntos de regras para ambientes determinados. A estrutura da Figura 13 oferece a estrutura desse programa geral em forma esquemática, mostrando como as regras condição-ação permitem ao agente fazer a conexão entre percepção e ação. Utiliza-se retângulos para denotar o estado interno atual do processo de decisão do agente e elipses para representar as informações suplementares usadas no processo.

Um agente reativo simples é totalmente aplicável a situações de ambientes completamente observáveis por emitir uma decisão correta com base apenas na percepção atual. Uma definição em termos programáveis pode ser analisada onde as especificações de regra cuja condição corresponde ao estado atual de percepção do agente que é denotada pela busca em uma base de dados definida pela condição-ação. É importante relatar que em alguns casos dependendo da forma como se programa e a plataforma que se utiliza para desenvolver um programa para um ambiente de tarefa esta função pode aparecer como um conjunto de regras pré-determinadas na forma de uma matriz, por exemplo, onde o agente faz consultas com determinadas percepções para se obter uma ação (RUSSEL; NORVIG, 2004).

Figura 13 – Descrição gráfica de um agente reativo simples.



Fonte: O autor.

CAPÍTULO 6

LÓGICA LINEAR

Este capítulo tem por objetivo abordar mais um fundamento que envolve este trabalho, completando o aparato teórico com a descrição da lógica linear e sua contribuição para a validação de modelos que será integrado a modelagem proposta com redes de Petri para o cenário do jogo dos pontinhos.

6.1 Contextualização

Criada em 1987 por Jean-Yves Girard, a Lógica linear é um refinamento dos sistemas clássico e intuicionista, unindo algumas características de ambas (GIRARD, 1987). Seu criador a caracterizou como uma lógica de recursos, algo que vem sendo cada vez mais aceito por cientistas da computação. Isso acontece pois as sentenças são tratadas como recursos ou ações e as consequências lógicas são tratadas como transições entre estados.

Para formar uma prova lógica um conjunto de axiomas são associados, formando lemas e construindo teoremas. Um lema provado pode ser usado para ajudar na prova de outras proposições sendo sempre verdadeira de acordo com a lógica clássica onde a verdade é estável e os recursos infinitos e conclusos(PIMENTEL, 2001).

Diferentemente a lógica intuicionista trabalha de outra forma em relação a noção de verdade absoluta, onde, julgamentos sobre declaração baseiam-se na existência de uma prova ou construção de uma afirmação. Ainda assim, a lógica intuicionista é uma lógica de recursos infinitos.

Lógica Linear trabalha com situações que retratam uma lógica consciente de recursos. Na lógica linear, as hipóteses não podem ser copiadas livremente (contração) ou descartados (enfraquecimento), apenas em situações especiais em que um tipo específico de

conectivos será usados, de forma especial para a validação de redes de Petri.

Na área tecnológica, tem-se de tomar decisões. Estas decisões devem levar em consideração o gerenciamento de recursos, uma vez que não existem de recursos ilimitados e a partilha de recursos é tarefa habitual na área da computação, nos sistemas multi programados. Neste escopo, a lógica linear atua lado a lado com o planejamento do sistema, reduzindo este hiato entre o planejamento e a prática.

6.2 Conectivos da Lógica Linear

A lógica linear introduz novos conectivos, que são classificados por grupos que se subdividem entre os multiplicativos, os aditivos e os exponenciais (GIRARD, 1987).

Figura 14 – (a) Resumo de operadores da lógica linear. (b) Gramática de proposições dos operadores

$\&$	e aditivo
$\oplus$	ou aditivo
$\otimes$	e multiplicativo
$\parr$	ou multiplicativo
$\top$	*top*[1]
$\bot$	*bottom*[2]
$\multimap$	implicação linear

(a)

$$A,B,C ::= X \parallel A \multimap B \parallel A \otimes B \parallel A \,\&\, B \parallel A \oplus B \parallel !A$$

(b)

Fonte: (GIRARD, 1987)

Dentre as conjunções o operador *e multiplicativo* representa uma conjunção multiplicativa onde se pode utilizar ambas premissas, e o operador *e aditivo* representa uma conjunção aditiva onde se deve escolher qual premissa utilizar, porém não se usam as duas ao mesmo tempo. Para as disjunções o operador *ou multiplicativo* representa uma disjunção multiplicativa onde se tem a segunda premissa somente se não se tiver a primeira, e o operador *ou aditivo* representa uma disjunção aditiva onde se tem qualquer uma das premissas, porém não as duas juntas, sendo que não há a possibilidade de escolha. É possível perceber a diferença entre elas no resumo de operadores da Figura14(a). Na Figura14(b) é mostrado a gramática de proposições dos operadores da lógica linear.

As proposições da lógica linear, são consideradas como recursos que podem ser consumidos ou produzidos em cada mudança de estado, enquanto, na lógica clássica as proposições podem tomar valores verdadeiro ou falso. As principais diferenças entre a lógica clássica e a lógica linear são a inexistência das regras de enfraquecimento e de contração.

De acordo com a Regra de Enfraquecimento, se A implica B, então A e C implicam B. Considerando um sistema em que A, B e C são recursos, a regra de enfraquecimento mostra-

ria que recursos, neste caso C, poderiam aparecer e desaparecer sem nenhuma influência. Assim, se a partir de um recurso A pode-se produzir um recurso B (consumindo o recurso A), então o aparecimento de um recurso C altera o sistema, o que não é mostrado pela regra de enfraquecimento da lógica clássica.

Segundo a Regra de Contração, se A e A implicam B, então A implica B. No contexto da lógica linear, utilizar essa regra seria o mesmo que aceitar que, havendo a necessidade de dois recursos A para produzir um recurso B, um desses recursos poderia ser dispensado, o que não é verdadeiro neste contexto, pois, se houver apenas um recurso A, então B não poderá mais ser produzido.

Uma definição crucial para a diferenciação das lógicas clássicas e linear, é o fato que uma verdade nunca é estável se tratando de recursos, pois segundo a lógica linear um recurso consumido não pode ser mais válido, ao passo que, em uma lógica clássica um valor é dito estável e pode se manter após sua inferência na produção.

6.3 Regras da Lógica Linear

A lógica linear incentivada pelo sistema de lógica clássica utiliza hipóteses, enquanto, ao mesmo tempo a lógica intuicionista não permite tal recurso. A lógica linear herdou o caráter construtivista da lógica intuicionista, portanto a prova de suas declarações pode ser obtida ou construída. Existem duas premissas, <A> e (A). Uma premissa da forma <A> significa uma representação linear onde se tem apenas uma unidade de recurso A. A segunda forma, (A), oferece uma representação intuicionista, onde se pode utilizar as regras de contração e enfraquecimento, podendo-se gerar tantos recursos quanto desejado. Dá-se a notação A! para um pool infinito de <A>. A suposição <A> pode ser vista como uma unidade de recurso A que se tem à disposição e a suposição [A], ou A!, pode ser vista como uma fonte ilimitada de recursos A.

Na lógica linear pode-se considerar termos como recursos, de forma que se pode ler A implica B como 'consumindo A se produz B'. Uma outra situação é a geração aleatória de produção para um mesmo recurso, onde, consumindo A pode-se produzir A ou B, isto pode ser representado da forma que segue na Equação 6.1

$$\frac{A \vdash B\&C}{A \vdash B}\& \frac{A \vdash B\&C}{A \vdash C}\& \tag{6.1}$$

A mesma suposição pode ser descrita de forma ilimitada, equivalentemente, na lógica linear, o operador *e multiplicativo* expressa tal situação de outra maneira, como na Equação 6.2:

$$\frac{< A > \vdash B \quad < A > \vdash C}{< A >, < A > \vdash B \otimes C} \otimes \tag{6.2}$$

Figura 15 – Regras da lógica linear.

$$\frac{}{\langle A\rangle \vdash A}\,\langle Id\rangle \qquad \frac{}{[A]\vdash A}\,[\mathrm{Id}] \qquad \frac{\Gamma,\Delta\vdash A}{\Delta,\Gamma\vdash A}\,\text{Exchange}$$

$$\frac{\Gamma,[A],[A]\vdash B}{\Gamma,[A]\vdash B}\,\text{Contração} \qquad \frac{\Gamma\vdash B}{\Gamma,[A]\vdash B}\,\text{Enfraquecimento}$$

$$\frac{\Gamma\vdash A}{[\Gamma]\vdash !A}\,!\text{-I} \qquad \frac{\Gamma\vdash !A \quad \Delta,[A]\vdash B}{[\Gamma],\Delta\vdash B}\,!\text{-E}$$

$$\frac{\Gamma,\langle A\rangle\vdash B}{\Gamma\vdash A\multimap B}\,\multimap\text{-I} \qquad \frac{\Gamma\vdash A\multimap B \quad \Delta\vdash A}{\Gamma,\Delta\vdash C}\,\multimap\text{-E}$$

$$\frac{\Gamma\vdash A \quad \Gamma\vdash B}{\Gamma,\Delta\vdash A\otimes B}\,\otimes\text{-I} \qquad \frac{\Gamma\vdash A\otimes B \quad \Delta,\langle A\rangle,\langle B\rangle\vdash C}{\Gamma,\Delta\vdash C}\,\otimes\text{-E}$$

$$\frac{\Gamma\vdash A \quad \Gamma\vdash B}{\Gamma\vdash A\&B}\,\&\text{-I} \qquad \frac{\Gamma\vdash A\&B}{\Gamma\vdash A}\,\&\text{-E1} \quad \frac{\Gamma\vdash A\&B}{\Gamma\vdash B}\,\&\text{-E2}$$

$$\frac{\Gamma\vdash A}{\Gamma\vdash A\oplus B}\,\oplus\text{-I1} \qquad \frac{\Gamma\vdash B}{\Gamma\vdash A\oplus B}\,\oplus\text{-I2} \qquad \frac{\Gamma\vdash A\oplus B \quad \Delta,\langle A\rangle\vdash C \quad \Delta,\langle B\rangle\vdash C}{\Gamma,\Delta\vdash C}\,\oplus\text{-E}$$

Fonte: Soares, 2004

Todas as regras usadas pela lógica linear são apresentadas na Figura15. As características da lógica possibilita uma infinidade de aplicações por conta da sua forma de lidar com as proposições, inclusive:

- Novas formas de analisar problemas, como consumo de recursos, e em aplicações em inteligência artificial, lógicas de programação focadas em estado, interpretação abstrata, etc.

- Novas formas de representar demonstrações que necessitem desta forma de inferência.

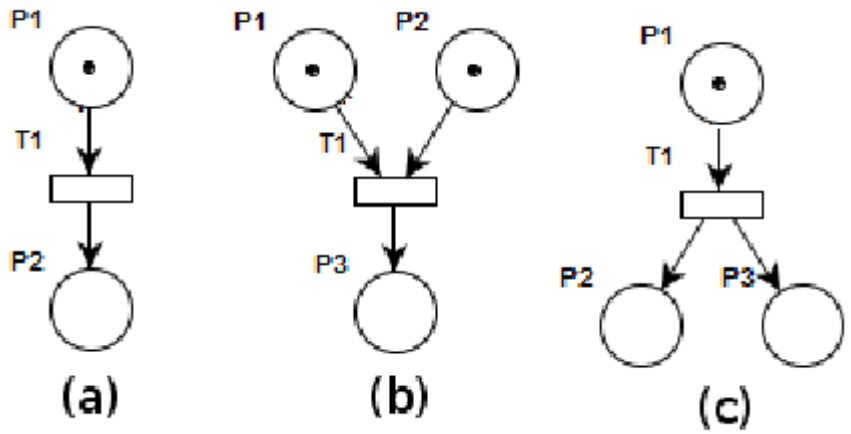
- Reduzir o hiato entre o raciocínio computacional e os recursos e estados do mundo real.

6.4 Tradução de Redes de Petri para Lógica Linear

A lógica linear é uma forma de representar as redes de Petri para um caráter de formalismo lógico, isto implica em formas de dados com mais precisão e qualitativo para validação dos dados (OLIVEIRA; JúLIA; PASSOS, 2011).

As redes de Petri permitem o cálculo de invariantes de lugar e de transição, enquanto a lógica linear trata claramente cenários que envolvem a produção e consumo de recursos (SOARES, 2004). A tradução de uma rede de Petri em fórmulas da lógica linear é feita de forma simples através da aplicação de regras para formação de sequentes com a semântica de transições de uma dada rede de Petri.

Figura 16 – Exemplo de transições para aplicação em fórmulas da lógica linear.



Fonte: O autor.

No exemplo da Figura16, transformando uma rede de Petri para as propriedades de uma lógica linear, temos uma transição como uma expressão da forma M1 implica M2 onde M1 e M2 são marcações. Para as transições da Figura16 pode-se associar as seguintes marcações:

- (a): $t1 = P1 \rightarrow P2$
- (b): $t1 = P1 \otimes P2 \rightarrow P3$
- (c): $t1 = P1 \rightarrow P2 \otimes P3$

Para estas mesmas sequências de marcação, considerando as redes de Petri da Figura16 cada situação apresenta sua marcação com os seguintes sequentes:

- $(a) : P1, P1 \rightarrow P2 \vdash P2$

- $(b) : P1 \bigotimes P2, P1 \bigotimes P2 \rightarrow P3 \vdash P3$
- $(c) : P1, P1 \rightarrow P2 \bigotimes P3 \vdash P2 \bigotimes P3$

6.5 Cálculo de Sequentes para Lógica Linear

O sistema de dedução linear é similar ao sistema de dedução de Gentzen (GOCHET; GRIBOMONT, 1990). Um sequente linear tem a forma $T \vdash \Delta$, onde T e Δ são conjuntos finitos de fórmulas, isto é, $T = T1, T2, ..., Tn$ e $\Delta = \Delta 1, \Delta 2, ..., \Delta n$. O símbolo T é o antecedente da fórmula e o símbolo Δ o consequente.

Um sequente pode ser provado por meio de aplicações sucessivas de regras do cálculo de sequentes. Existe uma equivalência entre a prova de sequentes da lógica linear e o problema de alcançabilidade em uma rede de Petri (PIMENTEL, 2001).

Uma árvore de prova da lógica linear é construída para provar se um dado sequente linear é ou não sintaticamente válido. A árvore de prova é lida de baixo para cima (bottom-up) e termina quando todas as folhas forem sequentes identidade (sequentes do tipo A $\vdash$ A), considerando o caso em que o sequente é válido.

Considerando para este ambiente apenas algumas regras da lógica linear. Assim, somente essas regras serão explicadas e terão suas aplicações exemplificadas. As demais regras do cálculo de sequentes da lógica linear podem ser encontradas em (JULIA; SOARES,).

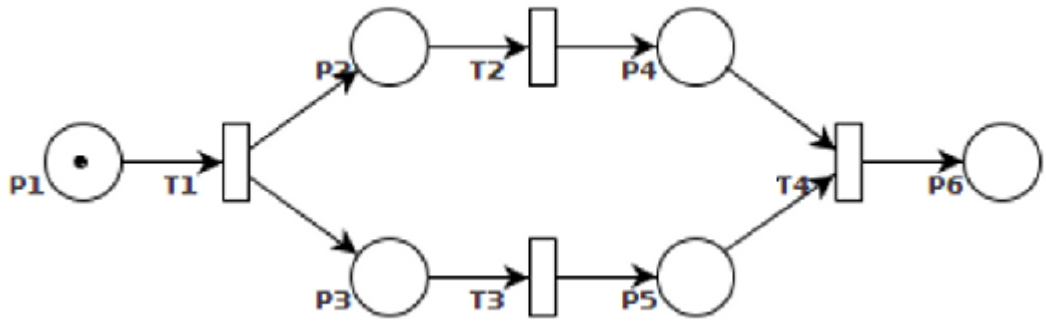
A partir deste ponto iremos representar as regras que serão utilizadas na construção das árvores de prova canônica no contexto das redes de Petri e, consequentemente, no contexto das WorkFlow nets. Vamos considerar que F , G e H são fórmulas e que T e Δ são blocos de fórmulas da lógica linear. As regras são apresentadas a seguir (RIVIERE *et al.*, 2001):

- A regra $\rightarrow L \frac{T \vdash F \quad \Delta, G \vdash H}{T, \Delta, \quad F \rightarrow G \vdash H} \rightarrow L$, expressa o disparo de uma transição e gera dois sequentes, tal que o sequente à direita representa o subsequente restante a ser provado e o sequente à esquerda representa as fichas consumidas pelo disparo da transição.
- A regra $\bigotimes R \frac{T, F, G \vdash H}{T, F \bigotimes G \vdash H} \bigotimes R$ transforma uma marcação em uma lista de átomos.
- A regra $\bigotimes R \frac{T \vdash F \quad \Delta \vdash G}{\Delta, T, T \vdash F \bigotimes G} \bigotimes R$ transforma um sequente do tipo $A, B \vdash A \bigotimes B$ em dois sequentes identidade $A \vdash A e B \vdash B$.

Para exemplificar a construção de uma árvore de prova canônica da lógica linear, utilizaremos a rede de Petri da Figura17. Transformando as transições desta rede de Petri em fórmulas da lógica linear, resulta na Equação 6.3.

$$t1 = P1 \rightarrow P2 \bigotimes P3; t2 = P2 \rightarrow P4; t3 = P3 \rightarrow P5; t4 = P4 \bigotimes P5 \rightarrow P6 \tag{6.3}$$

Figura 17 – Rede de Petri para exemplificar a construção da árvore de prova canônica.



Fonte: O autor.

A marcação inicial desta rede de Petri é apenas P1. Assim, o sequente linear a ser provado é dado por P1, t1 , t2 , t3 , t4 ⊢ P6, onde P1 e P6 são, respectivamente, a marcação inicial e final da rede de Petri da Figura17. Aplicando as regras do cálculo de sequentes a este sequente linear, é possível provar se o mesmo é ou não um sequente sintaticamente válido o que pode ser usado para verificar a corretude de um WorkFlow-net. A árvore de prova é apresentada na Figura 18.

Figura 18 – Arvore de Prova canônica das sequentes da Equação6.3

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{P_4 \vdash P_4 \qquad P_5 \vdash P_5}{P_4, P_5 \vdash P_4 \otimes P_5}\otimes_R \qquad P_6 \vdash P_6}{P_3 \vdash P_3 \qquad P_4, P_5, P_4 \otimes P_5 \multimap P_6 \vdash P_6}\multimap_L}{P_2 \vdash P_2 \qquad P_3, P_4, P_3 \multimap P_5, t_4 \vdash P_6}\multimap_L}{P_2, P_3, P_2 \multimap P_4, t_3, t_4 \vdash P_6}\otimes_R}{P_1 \vdash P_1 \qquad P_2 \otimes P_3, P_2 \multimap P_4, t_3, t_4 \vdash P_6}\multimap_L$$

$$P_1, P_1 \multimap P_2 \otimes P_3, t_2, t_3, t_4 \vdash P_6$$

Fonte: O autor.

Como todos os nós folha da árvore de prova acima são sequentes identidade, ou seja, sequentes do tipo A ⊢ A, tem-se que o sequente linear P1, t1 , t2 , t3 , t4 ⊢ P6 é sintaticamente válido.

6.6 Provando a Corretude das WorkFlow nets

Para provar corretude emprega-se o cálculo de sequente. A partir da construção de árvores de prova canônica, em seguida são utilizadas as sentenças produzidas com a semântica das transições quando uma WorkFlow-net tem corretude (OLIVEIRA; JúLIA; PASSOS, 2011).

Para provar que o critério correto para a corretude do sistema de WorkFlow usado na lógica linear é válido, inicialmente, é necessário representar essa WorkFlow-net por fórmulas da Lógica Linear. Nesta abordagem, uma rede de fluxo de trabalho é representado por um ou mais seqüentes linear.

Assim como em um ambiente de propriedade Workflow net uma sequente para verificação de propriedade *sound* possui apenas um átomo *Start*, que representa a marcação inicial da WorkFlow net. Logo, para a construção dos sequentes da lógica linear, sempre será considerada apenas uma ficha no lugar *Start*. A árvore de prova da lógica linear utilizada no contexto desse trabalho pode terminar de três formas:

- Quando todas as folhas da árvore de prova forem sequentes identidade;
- Quando o átomo *End* for produzido (ou seja, quando o sequente $End \vdash End$ aparecer na árvore de prova);
- Ou quando não houver nenhuma regra do cálculo de sequentes que possa ser aplicada.

Um modelo de análise é aplicado as transições de uma Workflow net, como já foi demonstrado, formando as consequências lógicas que serão inferidas no cálculo de sequente. Então, quando as árvores de prova para todos os cenários da WorkFlow net analisada estão construídas, elas devem ser analisadas seguindo os seguintes passos:

1. Para cada árvore de prova construída irá verificar se:

   (a) Apenas um átomo *End* foi produzido (isto é representado, na árvore de prova, pelo sequente identidade $End \vdash End$). Este fato prova o primeiro requisito para a verificação da propriedade *soundness*, isto é, que apenas uma ficha aparece no lugar *End* para o caso tratado.

   (b) Quando a prova está finalizada, não há nenhum átomo disponível para consumo na árvore de prova. Este fato prova que todos os lugares da WorkFlow net estão vazios para este caso quando sua ficha chega no lugar *End*, ou seja, o segundo requisito para afirmar que uma WorkFlow-net é *sound*.

2. Considerando todos os cenários SC1; SC2; ...; Scn da WorkFlow-net analisada, cada transição $t \epsilon T$ precisa aparecer em um cenário, pelo menos. Isto prova que todas as transições são disparadas, ou seja, nenhuma transição é morta.

Se as condições 1 e 2 acima são satisfeitas, a WorkFlow-net analisada pelas sequentes geradas são *sound*.

CAPÍTULO 7

ESTUDO DE CASO

A partir de modelos gráficos para situações de jogadas elaborados com as redes de Petri, serão testados as possíveis maneiras de abordar o problema, que se resume em desempenhar jogadas de ação, conseguindo pontos, e jogadas de defesa, impedindo que se perca pontos. A lógica linear será muito importante neste ponto, pois poderá possivelmente garantir que a modelagem com redes de Petri é eficaz o bastante, através da prova de corretude, que irá garantir que é possível a partir do formalismo gerado desenvolver um agente com ações satisfatórias para ambientes de jogadas adversárias.

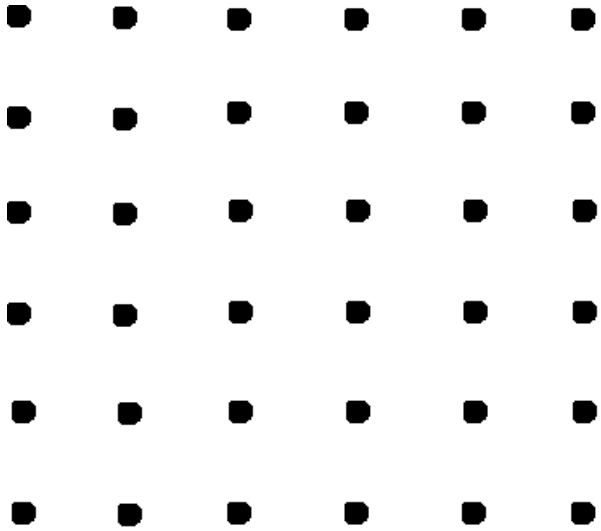
7.1 Jogo dos pontinhos

O jogo usualmente designado por jogo dos pontinhos (SANTOS, 2006) ou *Dots-and-Boxes* (em português, Pontos e Quadrados) é um jogo que pode ser jogado apenas com uma caneta e um papel. Apesar de ser conotado como um jogo para crianças, engane-se o leitor que pense que é um jogo com estratégias simples. De fato, é um jogo muito interessante, sendo mesmo alvo de estudo pelos amantes da matemática e da teoria da computação, isso porque pode envolver estratégias de controle que sejam capazes de criar situações onde um adversário nunca vença um agente que tenha a propriedade de se defender.

Existem formalismos e teorias da computação como, a inteligência artificial por exemplo, que descrevem abordagens para estes problemas de mapeamento e busca de resultados que são capazes de gerar resultados ótimos, que satisfaçam o objetivo do jogo, para estes problemas. A intenção deste caso de uso é demonstrar que através de técnicas de modelagem é possível também gerar boas propriedades para a solução não só deste jogo, mas também de várias situações que envolvam a formação de regras para solução de problemas.

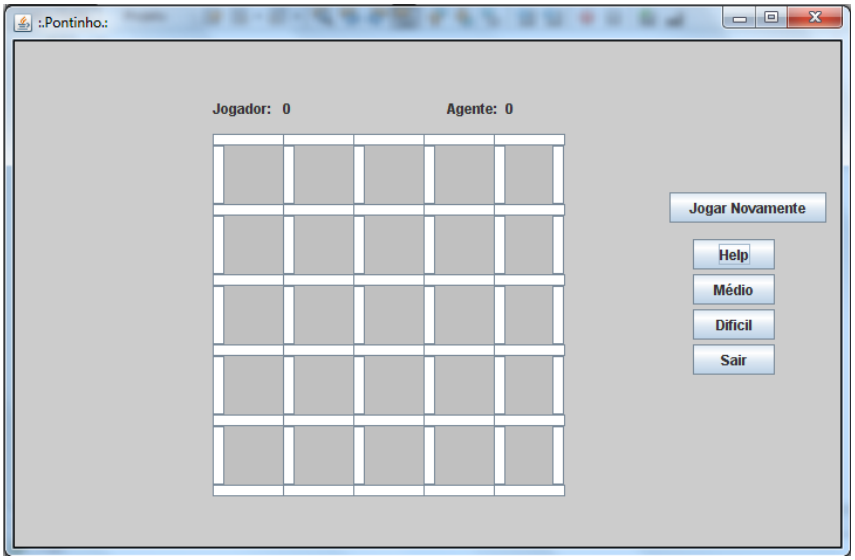
As regras do jogo são fáceis. É dado um conjunto quadrangular ou retangular de pontos alinhados na vertical e na horizontal, neste caso nossa plataforma de desenvolvimento abstraiu este conceito para uma matriz representada na linguagem de programação como um conjunto de pontos que possuem variáveis de apoio para a marcação de arestas, como pode ser mostrado na Figura19 de forma notacional, ou na Figura20 representando nossa plataforma de trabalho.

Figura 19 – Representação simples do ambiente de jogo dos pontinhos



Fonte: O autor.

Figura 20 – Plataforma utilizada no estudo de caso para o ambiente de jogo dos pontinhos

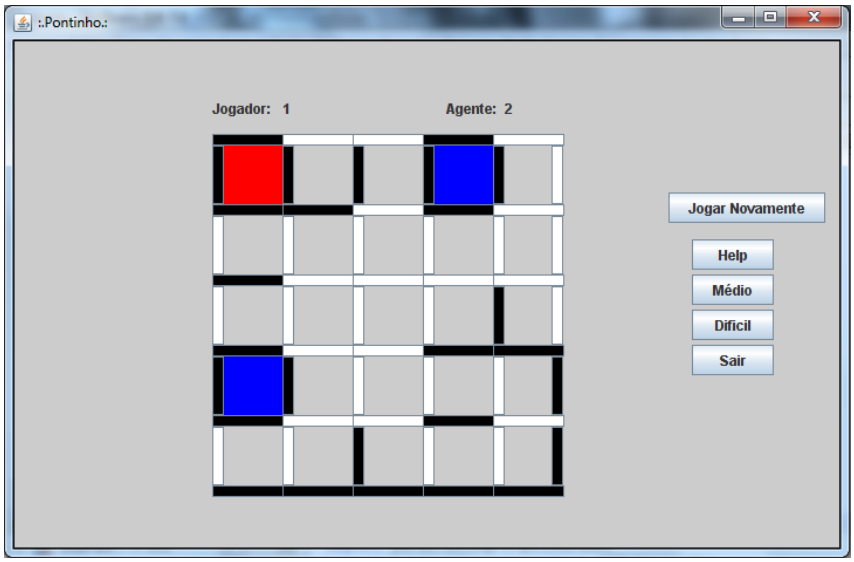


Fonte: O autor.

Cada jogador efetua, um por vez, uma jogada juntando dois pontos adjacentes com uma linha horizontal ou vertical, formando uma aresta. Quando uma jogada adiciona a

quarta aresta a um quadrado o jogador que a efetuou contabiliza um ponto pela marcação deste quadrado, preenchendo este quadrado com sua cor. Sempre que um jogador adiciona a quarta aresta de um quadrado joga novamente. Um jogador não é obrigado a ganhar quadrados mesmo que tal seja possível na sua vez de jogar. O jogo termina quando todas as arestas estiverem desenhadas, ganhando o jogador que possuir mais quadrados marcados com sua cor. A Figura21 mostra como uma marcação de aresta é representada na plataforma de jogo e uma situação onde já foram marcados pontos para ambos os jogadores.

Figura 21 – Plataforma de jogo com uma situação que representa a marcação de arestas e contagem de pontos dos jogadores.



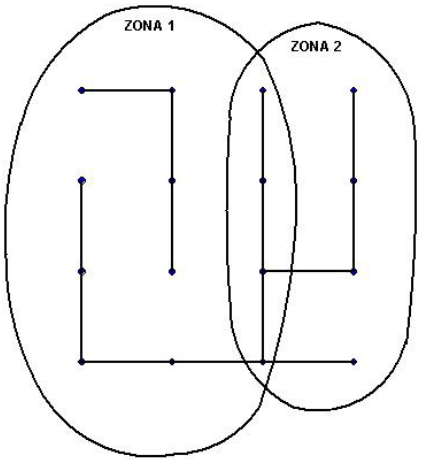
Fonte: O autor.

As estratégias envolvendo este jogo podem parecer fáceis a primeira vista, mas se analisarmos melhor o problema veremos que não se trata de apenas marcar arestas dos quadrados e impedir que o adversário marque pontos. Na realidade quando o jogo envolve jogadores de alto nível de conhecimento lógico/matemático existem percepções que mudam as estratégias locais para ações futuras que lhe trarão benefícios melhores. Veja por exemplo a Figura22 que mostra uma situação de jogo com duas zonas de pontuação.

Em um primeiro momento o jogador ativo pensaria em marcar os três pontos da zona 2 que estão evidenciados como pontos deixados na sua vez de jogar, porém, se olharmos melhor veremos que a zona 1 apresenta um número de pontos a serem marcados bem maior. A melhor jogada possível neste caso seria entregar os pontos da zona 1 fazendo com que o seu adversário deixe para você os pontos da zona 2. Assim, o jogador que marcasse a zona 2 mesmo entregando pontos em situações do presente, acarretaria mais pontos em jogadas futuras, vencendo o jogo, como pode ser percebido na Figura23.

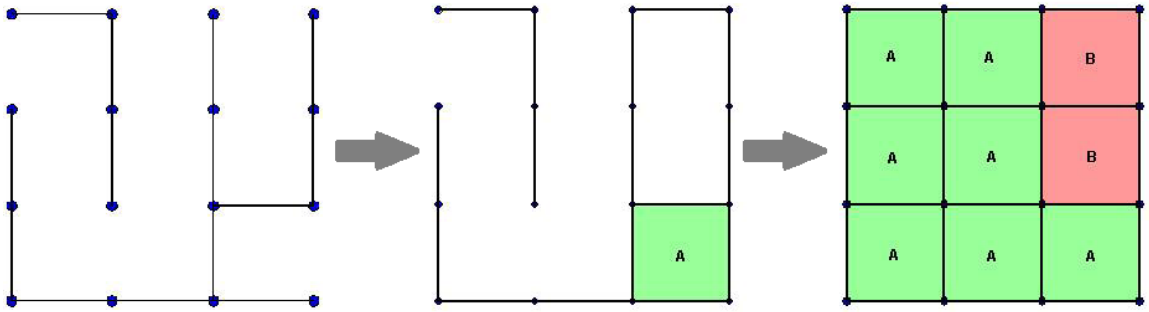
As estratégias demonstradas serão ainda refinadas e complementadas com uma fase de modelagem para este ambiente de jogada. Antes que o projeto seja repassado para pro-

Figura 22 – Marcação de zonas de jogadas do jogo dos pontinhos



Fonte: O autor

Figura 23 – Sequencia de jogadas ideal para vencer situação de jogo envolvendo as zonas da Figura22



Fonte: O autor.

cesso de desenvolvimento é necessário além de estudar e conhecer o jogo, elencar todas as características de jogadas para um formalismo que resultará em uma melhor forma de abordar o problema de adversário do jogo dos pontinhos, através das redes de Petri aplicadas a derivação denominada WorkFlow-nets.

Quadro 2 – Sequência de regras para formação inicial do agente adversário para o Jogo dos Pontinhos

Sequência	Regra
1º	Marcar pontos deixados pelo adversário
2º	Marcar uma aresta na tabela ligando dois pontos
3º	Marcar quadrados com menos de 2 arestas marcadas (Não entregar jogadas)
4º	Quando não houver outra saída, marcar a zona de sequência com menor perca, (Entregar menos jogadas)
...	...

Conforme, apresentado no Quadro2, o jogo dos pontinhos possui regras que podem ser enumeradas e que mantém certa precedência umas em relação as outras. Traduzindo

as informações repassadas a respeito do jogo é possível construir uma tabela de regras inicial para caracterização do agente, onde cada regra listada deve ser cumprida sem deixar de cumprir as regras estabelecidas anteriormente. Quanto maior o número de regras e maior riqueza de detalhes melhor será a especificação obtida.

7.2 Modelo de Especificação do Jogo Baseado em Agente

A execução de um modelo de jogo baseado em agente para um software orientado a objetos pode ser visto como a execução de uma jogada(caso de um processo de negócio). A jogada possui um início e um fim e executará várias operações (métodos de objetos do jogo) seguindo roteiros de execução de tipo seqüencial, alternativo, paralelo, iterativo, etc.

A realização de uma operação durante a execução de uma jogada será promovida por um objeto que fornecerá um método correspondente de um modo semelhante ao de um recurso usado para a execução de uma atividade no caso da execução de um processo de negócio (AALST; HEE, 2002).

Mesmo assim, o modelo de especificação do jogo deverá possuir características próprias à execução da jogada. Em particular, ele deverá permitir a especificação das estruturas de dados e de controle, e ser um modelo suficientemente formal que será posteriormente implementado e transformado em um código executável. A seguir, é apresentado a definição do modelo de especificação do jogo baseado em agente:

Definição: O modelo de especificação do jogo baseado em agente é definido por uma rede de Petri a objeto (SIBERTIN-BLANC, 1985) tal que a rede de Petri subjacente autônoma a partir da qual define-se a estrutura de controle do modelo do jogo dada por um WorkFlow-net. Neste caso, o WorkFlow-net a objetos correspondente pode ser definida pela 9-upla, como demostrado na Equação7.1:

$$No =< Class, P, T, V, Pre, Post, Atc, Ata, Mo > \tag{7.1}$$

Onde:

- *Class* representa o agente que é definido por um conjunto de atributos de dois tipos:

  - atributos que representam dados de entrada do jogo;

  - atributos que representam dados utilizados para caracterizar o roteiro seguido pela jogada do agente, caracterizando-se dessa forma, em dados de saída do teste;

- *P* é um conjunto finito de lugares (lugares de entrada *Start*(Início), lugar de saída *End* (Fim) e lugares envolvidos nos diversos roteiros do WorkFlow-net subjacente cujos tipos são dados por *Class*;

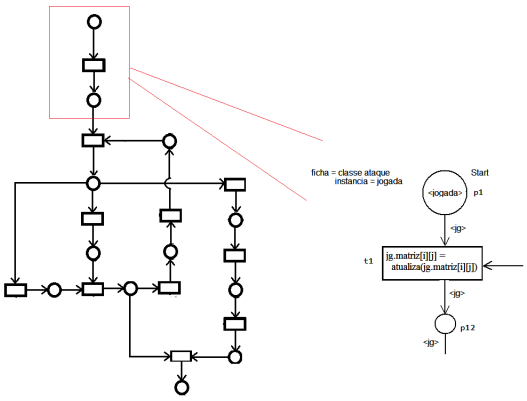
- *T* é um conjunto finito de transições;
- *V* é um conjunto de variáveis, cujos tipos são dados por *Class*;
- *Pre* é a função lugar precedente que a cada arco de entrada de uma transição faz corresponder uma soma formal de n-uplas de elementos V;
- *Post* é a função lugar seguinte que a cada arco de saída de uma transição faz corresponder uma soma formal de n-uplas de elementos V;
- *Atc* é uma aplicação que a cada transição associa uma condição que envolveos atributos das variáveis formais associados aos arcos de entrada;
- *Ata* é uma aplicação que a cada transição associa uma ação/operação em forma de chamada de métodos de um objetivo que envolve os atributos das variáveis formais associadas aos arcos de entrada cujo valor de retorno da chamada (quando existir) é repassado a valores de atributos de variáveis formais associadas aos arcos de saída;
- *Mo* é a marcação inicial que associa ao lugar de início *Start* um objeto de jogo instanciado de *Class*.

O modelo de especificação para o agente que integrará o núcleo do jogo segue as regras determinadas anteriormente nesta seção. As propriedades de WorkFlow-nets a objetos irão compor um formalismo para o ambiente do jogo dos pontinhos, capaz de interagir com os mais variados cenários e situações de jogo. A Figura**??** demonstra a abordagem utilizada para função de adversário complementando as regras demonstradas no Quadro2 o qual apresenta regras iniciais para um agente jogador do jogo dos Pontinhos em uma WorkFlow-nets a objetos.

O ambiente proposto pela WorkFlow-nets a objetos para o ambiente de jogada do agente envolve uma ação onde o mesmo deve avaliar a situação real/atual do jogo, desempenhar jogadas campeãs deixadas pelo seu adversário e posteriormente atacá-lo de forma a não permitir que o adversário faça jogadas ganhadoras. Este esquema pode ser percebido conforme exibido na sequência de disparos da rede.

- A marcação inicial (ficha associada ao lugar *Start*) sensibiliza a transição t1, a qual não possui pré-condição. O disparo de t1 indica que uma jogada foi atualizada e deve-se atualizar a tabela de pontos da matriz. Após o disparo de t1, um novo estado é produzido no lugar, estando a ficha agora em p11.
- Com uma ficha no lugar p11 a transição t2 é sensibilizada. O disparo de t2 indica que o placar do jogo foi acertado. Após o disparo de t2, um novo estado é produzido, permanecendo a ficha em p2.

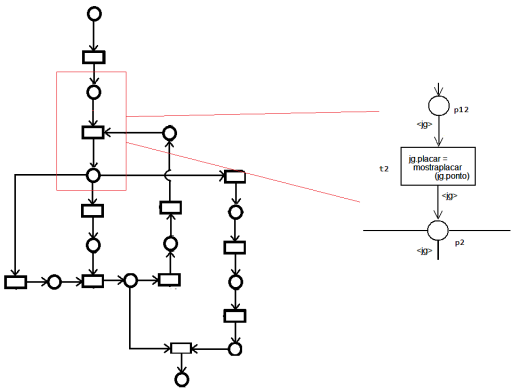
Figura 24 – Disparo 1



Fonte: O autor.

- A ficha assumiu o lugar p2 e a transição t3 é sensibilizada. O disparo de t3 indica que um ponto foi deixado pelo seu adversário e assume-se uma posição na matriz para marcar o ponto. Com o disparo de t3 um valor de aresta é alocado para marcação com a ficha em p3.

Figura 25 – Disparo 2



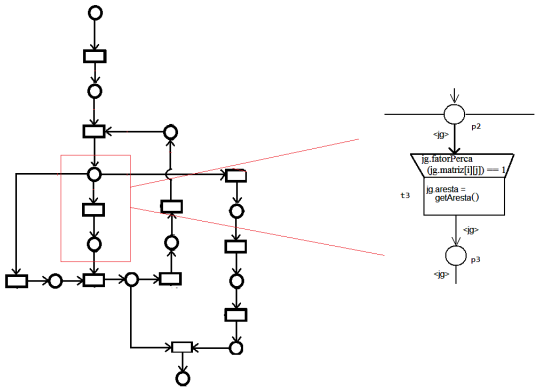
Fonte: O autor.

- A ficha em p3 irá assumir a transição t4. Disparando em t4 a ficha indica a marcação desta aresta. Com o disparo assume-se o lugar p5.

- A ficha que está em p5 sensibiliza t6. Disparando em t6 o ponto é contabilizado na base de dados. Após o disparo o lugar p6 é assumido pela ficha.

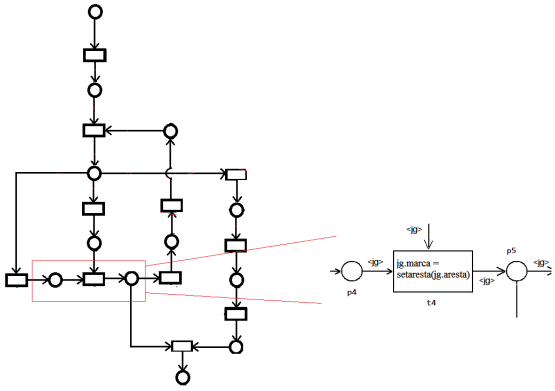
- A ficha em p6 agora sensibiliza t7. Disparando em t7 a função vitória testa se com essa jogada o jogador encerrou o jogo obtendo vitória. Com o disparo a ficha assume o lugar p7.

Figura 26 – Disparo 3



Fonte: O autor.

Figura 27 – Disparo 4



Fonte: O autor.

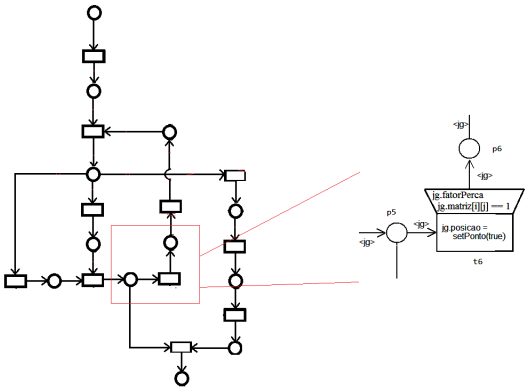
- O mesmo processo descrito acima é realizado em t1 e t2 na rede de Petri. Com o ultimo disparo a ficha assume novamente o lugar p2, pronto para uma nova busca.

- A ficha em p2 sensibiliza t5. Neste caso o disparo de t5 indica que a função que indica a posição para jogada verificou um valor de arestas onde qualquer marcação não indica mudança de estado no jogo. A ficha assume um novo estado chegando a p4.

- A ficha em p4 sensibiliza t4. Disparando em t4 a ficha indica a marcação desta aresta. Com o disparo assume-se o lugar p5.

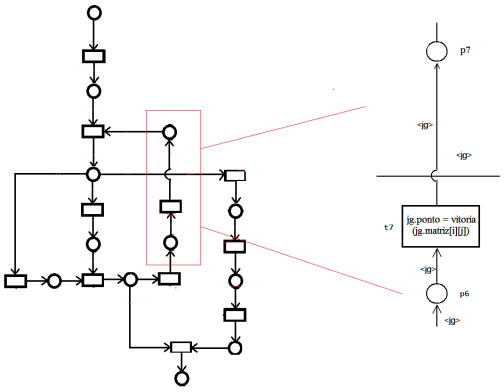
- Agora que a ficha já verificou os pontos deixados e marcou uma aresta no jogo. A ficha em p5 sensibiliza t11. O disparo de t11 indica que a jogada da vez foi concluída chaveando os jogadores para alternar as jogadas. Após o disparo em t11 o estado produzido é FIM, encerrando sua execução.

Figura 28 – Disparo 5



Fonte: O autor.

Figura 29 – Disparo 6



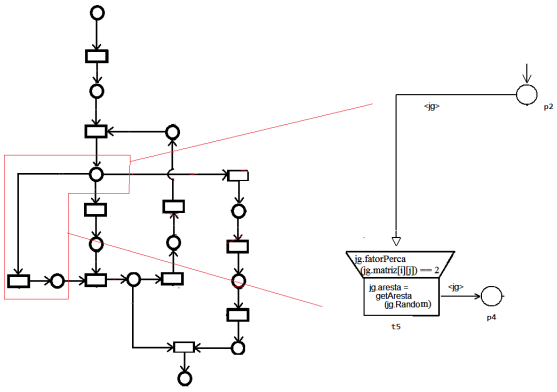
Fonte: O autor.

A sequencia de disparos descritas anteriormente e apresentada na sequencia de figuras apresenta a WorkFlow-net derivada das regras apresentadas pela descrição do jogo. Neste modelo, as transições são associados a métodos (operações) que manipulam atributos fornecidos pela ficha <*jogada*> que se encontra no lugar *Start*. A ficha que representa o objeto <*jogada*> é uma instância da classe *Agente*. Os atributos definidos em <*jogada*> são fornecidos para as transições da rede através da variável de arco <*jg*> (que também é do mesmo tipo da classe <*Agente*>). A variável de arco <*jg*>, dessa forma, fornece os atributos que são manipulados e atualizados pelas transições da rede.

Neste modelo, a classe do agente <*Agente*> é definida por:

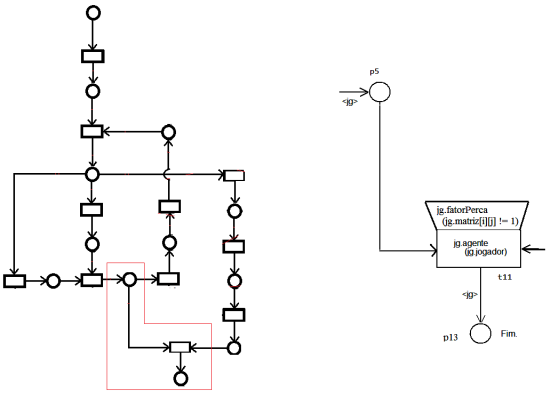
classe <*Agente*>;
Dados de entrada do Agente:
matriz[][]: tipo No;

Figura 30 – Disparo 7



Fonte: O autor.

Figura 31 – Disparo 8



Fonte: O autor.

aresta: integer;

Dados de saída do Agente:
placar[]: integer;
ponto[][]: boolean;
agente: tipo Agente;

Para um exemplo de sensibilização de transição poderíamos atribuir os seguintes atributos ao objeto <*jogada*>:

Dados de entrada do Agente:
matriz[1][1]: No1();
aresta: 0 //referente a uma das arestas da posição (0=acima, 1=direita, 2=esquerda, 3=abaixo);

Dados de saída do Agente:
placar[1]: 3;
ponto[1][1]: false;
agente: jg();

Após a sensibilização da transição t3 a posição *matriz[1][1]* do objeto No1() referente a um ponto do quadro do jogo dos pontinhos testa se a *aresta=0* está marcada ou não. Após a sensibilização da transição a função *jg.marca* recebe a *aresta 0* como parâmetro e muda o valor de *ponto[1][1] = true*. Assim a ficha irá sensibilizar em seguida a transição t6 que irá contabilizar o ponto e atribuir a *placar[1] =+1* em seguida o valor é apresentado na tela e o jogador ativo é chaveado.

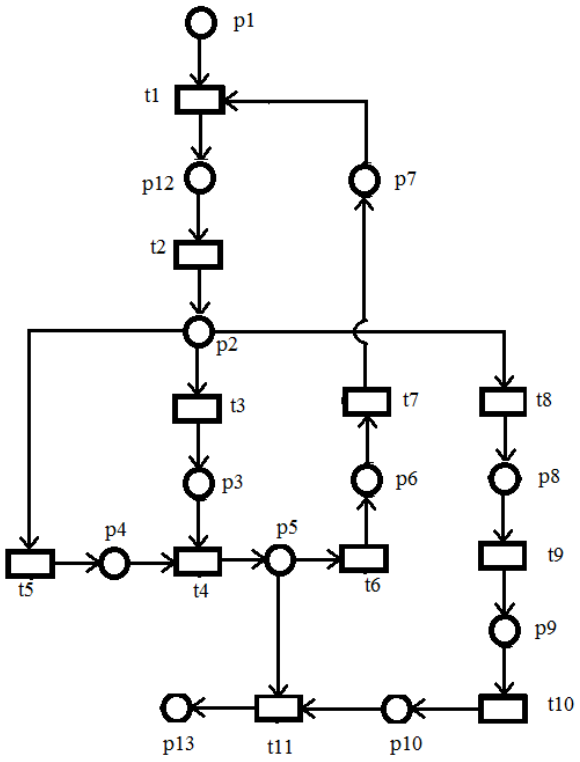
7.3 Verificação do Modelo Utilizando Lógica Linear

Para verificação da corretude das WorkFlow net a objetos utilizaremos a lógica linear, aplicando a prova canônica para as sequentes formadas com as semânticas de transições que são geradas a partir do modelo. Mas para isto, é necessário primeiro, transformar a rede em um grafo de alcançabilidade, como mostrado na Figura32.

Com o grafo de alcançabilidade formado, então podemos aplicar as funções da lógica linear para cada uma das transições, como segue:

- $t1 = P1 \bigotimes P7 \rightarrow P12$
- $t2 = P12 \rightarrow P2$
- $t3 = P2 \rightarrow P3$
- $t4 = P3 \bigotimes P4 \rightarrow P5$
- $t5 = P2 \rightarrow P4$
- $t6 = P5 \rightarrow P6$
- $t7 = P6 \rightarrow P7$
- $t8 = P2 \rightarrow P8$
- $t9 = P8 \rightarrow P9$
- $t10 = P9 \rightarrow P10$
- $t11 = P10 \bigotimes P5 \rightarrow P13$

Figura 32 – Grafo de alcançabilidade para a rede mostrada nos disparos



Fonte: O autor.

A marcação inicial da WorkFlow-net é apenas P1. Assim, o sequente linear a ser provado é dado por P1, t1, t2, t3, t4, t11, P13, onde P1 e P13 são, respectivamente a marcação inicial e final da rede de Petri mostrada na sequência de disparos. Aplicando as regras do cálculo de sequente a este sequente linear, é possível provar se o mesmo é ou não um sequente sintaticamente válido. A árvore de prova é dada a seguir.

Figura 33 – Árvore de prova para a rede de alcançabilidade.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{P_5 \vdash P_5 \qquad P_{10} \vdash P_{10}}{P_5, P_{10} \vdash P_5 \otimes P_{10}}\otimes_R \qquad P_{13} \vdash P_{13}}{P_3 \vdash P_3 \qquad P_4 \vdash P_4 \;\; P_5, P_5 \otimes \boxed{P_{10}} \multimap P_{13} \vdash P_{13}}\multimap_L}{P_3, P_4 \vdash P_3 \otimes P_4}\otimes_R \qquad P_2 \vdash P_2 \;\; P_3, P_3 \multimap \boxed{P_4} \otimes P_5, t_{11} \vdash P_{13}}{P_{12} \vdash P_{12} \qquad P_7 \vdash P_7}\multimap_L}{P_{12}, P_7 \vdash P_{12} \otimes P_7}\otimes_R \qquad P_2, P_2 \multimap P_3, t_4, t_{11} \vdash P_{13}}\multimap_L \qquad P_1 \vdash P_1 \;\; P_{12}, P_{12} \otimes \boxed{P_7} \multimap P_2, t_3, t_4, t_{11} \vdash P_{13}}{P_1, P_1 \multimap P_{12}, t_2, t_3, t_4, t_{11} \vdash P_{13}}\multimap_L
$$

Fonte: O autor.

Como pode ser observado na Figura 33 não é possível construir através do cálculo de sequentes uma prova para o caso da WorkFlow-net apresentada. Isso porque o grafo de alcançabilidade apresenta ciclos em sua construção, fazendo com que os recursos que serão necessários para produzir novos estados não estejam necessariamente presentes.

O fato de alguns recursos, como P4, P7 e P10, serem incertos não garante a continuação da prova e impede a conclusão da árvore em nós folhas. Um estudo interessante em trabalhos futuros seria o estudo de uma aplicação da lógica linear para este caso de uso, ou então, uma outra solução que dê corretude ao sistema. Por este motivo, o modelo descrito para o jogo dos pontinhos será avaliado em testes de usuário, com diferentes tipos de cenários e jogadores.

7.4 Agente Jogador Para Jogo Dos Pontinhos

Após a especificação do jogo concluído e consequentemente os resultados obtidos na verificação do modelo através da aplicação de cálculo de sequentes usando a lógica linear, é o momento de utilizar todas estas informações para a construção do agente inteligente para o jogo dos pontinhos. Nesta fase do projeto a modelagem já atribuiu o formalismo necessário e a equipe de desenvolvimento deve utilizar esse material de apoio para a produção do software.

O primeiro passo nesse momento é entender o sistema que foi descrito, neste caso a abordagem para construção de regras do jogo dos pontinhos, e aplicá-las a linguagem de desenvolvimento a ser utilizada.

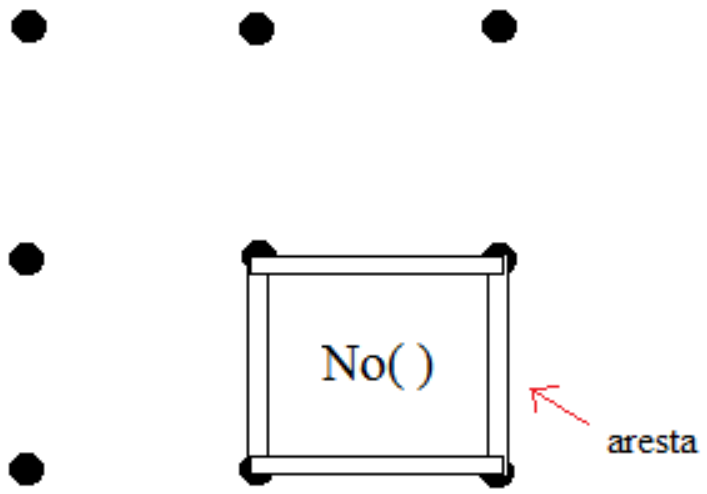
Os dados gerados nas etapas passadas nos mostraram que este jogo é formado por estratégias variantes, ou seja, não existe um meio direto ou pré determinado para garantir que o agente seja sempre o vencedor. Logo para que o agente jogador consiga ter um melhor desempenho em suas jogadas iremos adaptar o modelo a uma estratégia que envolve a defesa como o seu principal ataque. Desta forma iremos concentrar nossos esforços em não perder pontos, ao invés de tentar consegui-los.

Para a formação do agente foram utilizados a linguagem de programação Java, por ser orientada a objetos e apresentar fácil iteração com os elementos de interface, uma vez, que implementando um agente jogador é necessário que se tenha uma plataforma de trabalho iterativo, para testes a apresentação de resultados. O nosso ambiente de trabalho já foi mostrado na Figura20 , esta é a interface de manipulação do usuário para que se interaja com o agente em disputas no jogo.

Na construção do modelo foram utilizadas duas classes principais, além da interface. Para o mapeamento do ambiente, foi criado uma classe *No()* que define todos os atributos de cada posição da matriz. Nesta classe cada instanciação, ou seja, posição da matriz é refe-

renciada como um ponto do gráfico que pode receber quatro marcações, uma vez que cada quadrado do ambiente de jogadas possui quatro arestas a serem marcadas, como mostrado na Figura34.

Figura 34 – Representação gráfica do ambiente para construção da classe No() do Agente-jogador



Fonte: O autor.

Dentre os atributos da classe *No()* estão a referência para as arestas (*acima, abaixo, esquerda e direita*) além dos métodos que retornam ou alteram seu valor que é booleano indicando apenas o grau da marcação.

A classe principal do agente é a classe *Agente()*. Nela estão todas as manipulações e controles para que o agente possa ganhar uma partida de jogo dos pontinhos. Esta classe utiliza instancias da classe *No()* para mapear o seu ambiente, adequando suas reações, sempre que vai desempenhar uma função no jogo.

O método mais expressivo desta classe é a função *fatorPerca*, mostrado no Algoritmo2. Este método tem como atributos um valor inteiro denominado *cont* e recebe como argumento para esta função os valores da matriz que incidem aquela posição e o objeto referente da classe *No*. O resultado deste método é dado quando recebe um nó da matriz, retornando o número de arestas marcadas para aquela região de jogada. Este valor retornado será importante para validar as jogada do Agente-jogador lhe dando a noção de quando uma marcação naquela posição é vantajosa, ou quando lhe oferece riscos.

O resultado do método *fatorPerca* é utilizado para orientar o agente a realizar marcações. Como foi mostrado no Quadro3 para cada retorno é dada uma ação no jogo. Uma análise importante a ser feita é quanto existem posições com valor de retorno igual a 1 ou 2,

Quadro 3 – Estratégias de acordo com o retorno da função fatorPerca

Valor	Regra
0	Jogada boa (sem riscos)
1	Jogada boa (sem riscos)
2	Marcar apenas em ultimo caso(Entrega de pontos)
3	Chamar função *ataca() (ponto deixado)*
4	Região já representa pontos (totalmente marcada)

pois quanto aos outros valores suas ações são bem definidas e não variam de acordo com o cenário geral do jogo.

Sempre que o valor de retorno indica um resultado igual a 1 é importante também analisar o valor da função *fatorPerca* também para o nó vizinho, pois uma aresta sempre incide sobre dois nós ao mesmo tempo, ou seja, a aresta *acima* de um nó é também a aresta *abaixo* de seu nó adjacente e assim por diante.

Além de analisar a questão de incidência de marcações é importante também considerar as consequências das jogadas do agente em cenários de jogos, pois como se trata de um método inteligente, o mesmo deve ser usado buscando sempre obter resultados ótimos que o qualifiquem como uma boa alternativa para a solução deste problema. Esta questão será analisada na fase de testes do agente onde iremos demonstrar para quais situações ele está adequado e como busca resolver estes problemas para cada cenário de jogada.

A outra função importante a se destacar é a função *ataca()* que é responsável por marcar os pontos deixados pelo adversário. Na cadeia de regras do agente jogador, esta é a primeira função a ser chamada, garantindo a sequência de regras determinada para estratégias do jogo. A função *ataca()* é demonstrada no Algoritmo3. Esta função é chamada sempre ao iniciar uma jogada do agente e parte da ultima posição de jogada do adversário e percorre toda a matriz de posições analisando todos os pontos a serem marcados.

Esta função não possui parâmetro de entrada, sendo assim um objeto instanciado da classe Agente apenas invoca este método para um dado momento do jogo fazendo com que ele percorra o ambiente de jogadas marcando e contabilizando os pontos que o jogador adversário lhes atribuiu, ou seja, quando existir posições com três marcações de arestas.

A junção dos elementos descritos acima forma uma abordagem condizente com a modelagem mostrada nas regras do Quadro2, pois utilizam os métodos descritos de forma a encarar o problema assim como foi demonstrado na sequência de passos da rede WorkFlow elaborada. Uma vez que este ambiente foi testado por regras da lógica linear, o resultado esperado para o desenvolvimento deste ambiente de jogada, será também propício a vencer os embates ao qual o agente será submetido.

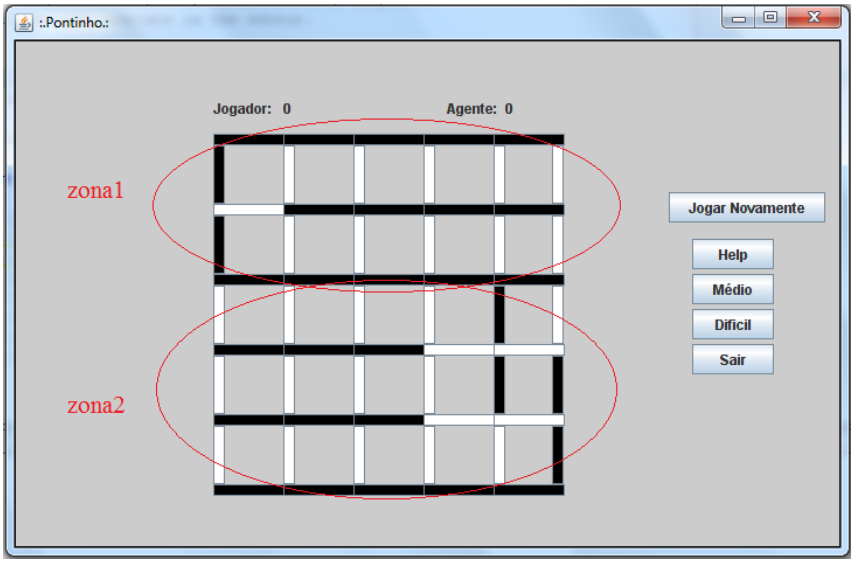
Os algoritmos listado em 2 e 3 formam as principais características do jogo, e portanto são as duas funções principais que são usadas no agente formando a abordagem para

a linguagem Java que descreve as regras do jogo dos pontinhos. É através destas ferramentas de controle que a classe *Agente()* é capaz de definir todas as regras pré-determinadas na fase de modelagem deste jogo.

7.5 Testando o Agente Jogador

Para análise e verificação do modelo o agente jogador foi submetido a algumas situações, em que para alguns cenários de jogos, foi utilizado a plataforma de jogo para medir seu desempenho e avaliar suas reações para ambientes que envolviam, por exemplo, situações de escolha em ambientes de zonas de pontuação, como foi explicado anteriormente. Na Figura35 é demonstrado uma situação de jogo em que algumas marcações feitas por ambos usuários chegou a um ambiente com duas zonas que possuíam números diferentes de pontuações.

Figura 35 – Situação/teste em um cenário*A* de jogo para o Agente jogador.

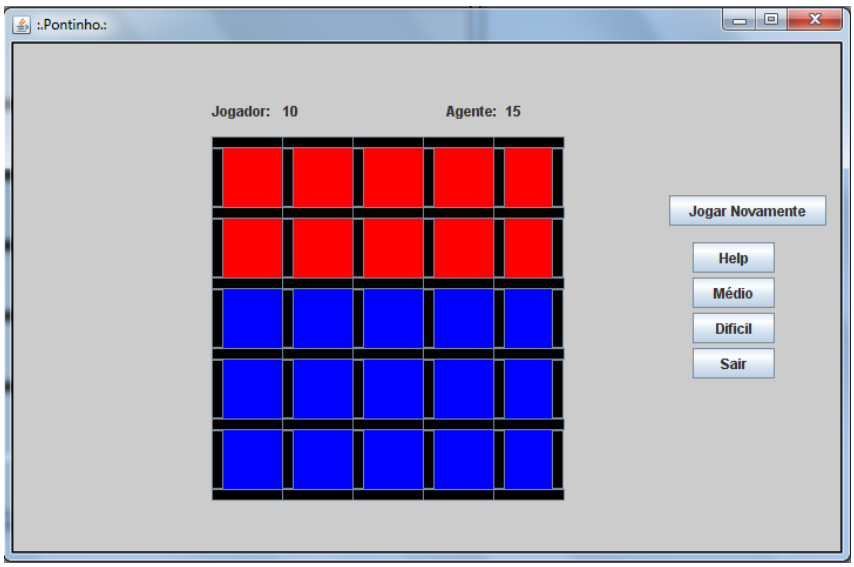


Fonte: O autor

Para esta situação o nosso agente reagiu de forma correta habilitando a zona de menor pontuação para o jogador adversário e realizando a marcação de um número maior de pontos como pode ser demonstrada na Figura36 com o seu resultado após as interações necessárias para concluir o jogo.

Pode-se perceber que suas ações envolveram uma tática de abordagem satisfatória, uma vez que, não entregou pontos ao adversário quando era possível fazê-lo e também em situações de risco, soube agir de forma a pontuar no jogo. É importante destacar que nem sempre o agente pôde sobressair em suas jogadas uma vez que não está apto a aprender e sim a apenas desempenhar as regras pré-estabelecidas na tabela de regras do agente - reativo - simples. Uma característica para este agente é que na maioria das vezes suas premissas

Figura 36 – Resultado das jogadas do agente para o cenário*A*.

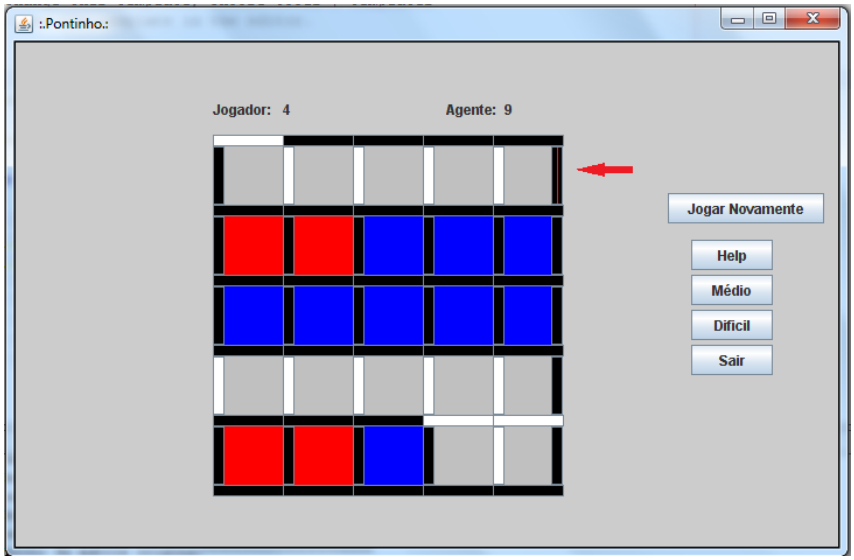


Fonte: O autor

estão corretas por abordar o problema sempre a partir da jogada do adversário, pois, sempre partem dele as jogadas que atribui pontos ao agente.

No exemplo da Figura37 podemos analisar uma boa jogada a ser calculada pelo agente implantado no jogo. Nesta situação a ação ideal seria marcar a aresta indicada no jogo, pois faria com que o adversário marcasse os pontos da zona com menor pontuação. Sendo assim, estas características são imprescindíveis no momento em que se esquematiza a abordagem a ser utilizada na modelagem do agente, pois irá garantir a qualidade de suas jogadas.

Figura 37 – Exemplo de uma boa marcação para um cenário possível do jogo.



Fonte: O autor.

A melhor forma para se ter um resultado favorável para o agente seria então tentar descrever ao máximo todas as situações/cenários possíveis, para que assim o nosso agente jogador tenha um número de informações suficientes para atuar no jogo.

Algoritmo 2 – Função Fatorperca(int matriz[][])

```
public int fator_perca(int mat[][]) {
      int cont = -1;
      if (mat[i][j].getPonto() == false){ //testa se a aresta acima está marcada.

          if (mat[i][j].getAc() == false) {
              if (cont == -1) {
                  cont = 1;
              } else {
                  cont++;
              }

          }

          if (mat[i][j].getAb() == false){ //testa se a aresta abaixo está marcada.
              if (cont == -1) {
                  cont = 1;
              } else {
                  cont++;
              }
          }

          if (mat[i][j].getD() == false){ //testa se a aresta direita está marcada.
              if (cont == -1) {
                  cont = 1;
              } else {
                  cont++;
              }
          }
          if (mat[i][j].getE() == false){ //testa se a aresta esquerda está marcada
              if (cont == -1) {
                  cont = 1;
              } else {
                  cont++;
              }
          }
      }//fim do primeiro if
      return cont;  \\retorna o número de arestas marcadas naquela quadrado.
  }//fim da funcao
```

Algoritmo 3 – Função ataca()

```
public int[] ataca() {

        int vet[] = new int[3];
        vet[0] = -1;
        vet[1] = -1;
        vet[2] = -1;
        boolean o = false;

        for (int i = 0; i < 5; i++) {
            for (int j = 0; j < 5; j++) {
                if (fator_perca(i, j) == 1) {
                    if (mat[i][j].getAc() == false) {
                        vet[0] = i;
                        vet[1] = j;
                        vet[2] = 0;
                        o = true;
                    } else if (mat[i][j].getAb() == false) {
                        vet[0] = i;
                        vet[1] = j;
                        vet[2] = 1;
                        o = true;
                    } else if (mat[i][j].getD() == false) {
                        vet[0] = i;
                        vet[1] = j;
                        vet[2] = 2;
                        o = true;
                    } else if (mat[i][j].getE() == false) {
                        vet[0] = i;
                        vet[1] = j;
                        vet[2] = 3;
                        o = true;
                    }
                    break;
                }//fim do if
            }//fim da coluna
            if (o == true) {
                break;
            }
        }//fim da linha

        return vet;
    }//fim funcao
```

CAPÍTULO 8

CONCLUSÃO

Foi apresentado neste trabalho um método formal de modelagem para agentes inteligentes integrados a jogos de vídeo game. A criação do modelo foi baseado nas WorkFlow-nets a Objetos, uma variação das WorkFlow-nets tradicionais que apresentam propriedades de classes e atributos, e na lógica linear.

No primeiro momento foram utilizados as propriedades das WorkFlow nets para criação de um modelo que se resume, em transcrever as politicas e estratégias de jogo para uma modelo gráfico e matemático para a formalização do desenvolvimento futuro do jogo.

A criação do modelo não reproduz apenas as características do jogo como objeto de apoio, mas da ênfase a uma particularidade da produção de jogos, que se resume em dar aptidão satisfatória no refinamento de informações importantes para o núcleo do jogo. Isso é feito através de especificações com as redes de Petri através das WorkFlow-nets a Objetos para a produção de agentes inteligentes atuantes como adversário em jogos.

Uma abordagem do ambiente de jogadas para um estudo de caso, onde o desenvolvedor é capaz de abstrair de forma mais precisa como enumerar e aplicar regras de controle para o jogo. Sendo assim é possível formular o ataque e defesa do agente jogador com um aproveitamento melhor e mais favorável do que desenvolvimentos convencionais que não levam em conta a precisão de uma modelagem formal. Isso é identificado enquanto se realiza a coleta de dados para a produção de técnicas de abordagem para os ambientes de tarefa do agente produzido.

Além das técnicas descritas, o processo de produção de agentes inteligentes no contexto das WorkFlow-nets apresentou uma segunda fase inserindo ao processo de modelagem, métodos de prova usando a Lógica Linear. Esta relação está ligada a análise quantitativa do jogo onde se espera qualificar de maneira positiva as especificações obtidas com as

WorkFlow-nets, por meio da validação.

Por sua vez, a redução da produção de sequentes válidos para as situações de jogo através da árvores de prova canônica eram um objetivo primário para os sequentes formados pelas transições descritas na rede. De forma, que o resultado da fase de modelagem dos agentes resultasse em uma forma coesa de desenvolver o agente sem tomar um dispendioso tempo, no processo como um todo, que não fosse objetiva.

Em relação ao trabalho proposto por Guilherme em Oliveira, Júlia e Passos (2011) , sua ideia original era apresentar ao contexto de jogos uma fase de modelagem a um contexto inicial de jogos com o objetivo de formalizar a sua criação de maneira geral. Sendo assim, neste projeto nós focamos estes esforços na produção de agentes inteligentes em que suas características necessitam ser representadas por estruturas que sejam capazes de manipular fluxo de dados e de controle que fazem dos agentes adversários aptos a tomar decisões em suas jogadas.

As vantagens da abordagem apresentada para produção de agentes inteligentes no âmbito de jogos são diversas. As WorkFlow-nets foram imprescindíveis enquanto uma abordagem gráfica e de fácil entendimento por ser construída em forma ilustrativa. Enquanto isso, o fato de trabalhar com a lógica linear permite provar o critério de corretude soundness em tempo linear, quando se consideram as estruturas paralelas e sem que seja necessária a construção de um grafo das marcações acessíveis, considerando diretamente a própria estrutura da WorkFlow net ao invés de considerar seu autômato correspondente.

O agente produzido para o jogo dos pontinhos obteve uma abordagem prática por partir de princípios de ataque que priorizava regras em uma tabela enumerada com graus de preferência. Assim, esta abordagem resultou em uma análise que usa a defesa como um ataque natural, fazendo com que o adversário, entregasse pontos por não ter a destreza esquematizada no processo de modelagem garantido pelas regras condição-ação do agente.

Isso pode ser percebido enquanto o agente era submetido a vários ataques pré determinados no estudo de caso com o objetivo de testar sua habilidade para resolução de problemas, que envolvia, não perder pontos e consequentemente fazer com que o adversário atribuísse-os ao agente.

Como trabalhos futuros, seria interessante estender a abordagem utilizada em situações como:

- Aplicação destas técnicas para outras áreas do processo de produção de jogos como por exemplo, jogos que envolvem critérios de tempo, quests, etc;

- Construir agentes inteligentes com este nível de formalismo para áreas diversas que necessitem de rigor matemático e precisão;

- Estender o uso de modelos baseados em rede de Petri para especificar e avaliar arquiteturas de jogadores inteligentes usados em jogos de tabuleiros.
- construir um simulador que permite a partir da especificação das funcionalidades do jogo, gerar de forma automatizada, código para a elaboração de agentes.

REFERÊNCIAS

AALST V. D., W. *Verification of workflow nets*. Cambridge, Massachusetts, 1998. Citado 2 vezes nas páginas 31 e 44.

AALST, W.; HEE, K. *Workflow Management: Models, Methods and Systems*. Massachussets. London England: The MIT Press Cambridge, 2002. Citado 4 vezes nas páginas 32, 33, 46 e 67.

BAZZAN, A. L. C. Coordenação de agentes com o uso de técnicas de teoria dos jogos. *Jornada de Atualização em IA.*, 2001. Citado na página 23.

BEZERRA, E. *Princípio de Análise e Projeto com Sistemas com UML*. 2º. ed. [S.l.: s.n.], 2007. Citado na página 41.

BUCKLAND, M. Al techniques for game programming. *Premier Press ACM - Cincinat/Ohio*, 2002. Citado na página 29.

CARDOSO, J.; VALETE, R. *Redes de Petri*. Florianópolis/SC: [s.n.], 1997. Citado 3 vezes nas páginas 21, 36 e 38.

CHAMPAGNAT, R.; PINGAUD, H.; ALLA. A gas storage example as a bechmark for hybrid modeling. *APII-JESA, Special Issue on Automation of mixed process and hybrid dynamical systems*, v. 32, n. 9-10, p. 1233–1253, 1998. Citado na página 40.

DAVID, R. Discret, continuous and hybrid petri nets. *Springer Seminars in Computers*, 2004. Citado 2 vezes nas páginas 35 e 39.

DELAMARO, M. E.; JINO, M. *Introdução ao Teste de Software*. 1º. ed. [S.l.: s.n.], 2007. Citado na página 40.

GIRARD, J. Y. Linear logic. *Theoretical Computer Science*, 1987. Citado 2 vezes nas páginas 53 e 54.

GOCHET, P.; GRIBOMONT, P. Logique: méthodes pour l informatique fondamentale. *Hermes*, v. 1, 1990. Citado na página 58.

GUEDES, G. T. A. *UML2*: Uma abordagem prática. [S.l.: s.n.], 2011. 32-56 p. ISBN 9788-5752-222812. Citado na página 22.

JULIA, S.; SOARES, M. S. Verification of real time uml specifications through a specialized inference mechanism based on a token player algorithm and the sequent calculus of linear logic. *Procedings of the 15th European Simulation Simposium and Exhibition*, p. 65–70. Citado na página 58.

MURATA, T. Petri nets: properties, analysis and applications. *Procedings of the IEEE*, p. 541–580, 1989. Citado 4 vezes nas páginas 22, 35, 36 e 39.

OLIVEIRA, G. W.; JúLIA, S.; PASSOS, L. S. Game modeling using workflow-nets. *IEEE*, 2011. Citado 4 vezes nas páginas 27, 57, 60 e 84.

PIMENTEL, G. E. *Lógica linear e a especificação de sistemas computacionais*. 3-11 p. Dissertação (Mestrado) — Universidade Federal de Minas Gerais, Belo Horizonte/MG, 2001. Citado 2 vezes nas páginas 53 e 58.

PRESSMAN, R. S. *Engenharia de Software*: Uma abordagem profissional. 7º. ed. São Paulo: [s.n.], 2011. 84-96 p. ISBN 978-85-7936-108-1. Citado 2 vezes nas páginas 21 e 26.

RIVIERE, N. *et al.* Reachability and temtempo conflicts in t-time petri nets. *PNPM Proceedings of the 9th international Workshop on Petri nets and Performance Models*, p. 229, 2001. Citado na página 58.

RUCKER, R. Software engeenering and computer games. *Addison ACM, London*, 2003. Citado na página 29.

RUSSEL, S.; NORVIG, P. *Inteligência Artificial*. Rio de Janeiro, RJ: [s.n.], 2004. Citado 3 vezes nas páginas 48, 50 e 51.

SANTOS, C. P. dos. *Jogos Matemáticos*: Análise de um problema de pontos e quadrados. 2006. 1-5 p. Disponível em: <http://ludicum.org>. Acesso em: 15/dez/2012. Citado 2 vezes nas páginas 23 e 63.

SIBERTIN-BLANC, C. High level petri nets with data structure. *6th European Workshop in Application and Theory of Petri Nets*, Espanha, junho 1985. Citado 4 vezes nas páginas 40, 41, 44 e 67.

SILVA, L. A. M. Estudos de desenvolvimento de sistemas multiagentes usando jade java. *Agent Development Framework UFP*, 2003. Citado na página 48.

SOARES, M. S. Uma abordagem baseada num jogador de redes de petri ptemporal e no cálculo de sequentes da lógica linear para a verificação de cenários de sistemas tempo real especificados através de diagramas dinâmicos da uml. *Faculdade de Computação UFU*, 2004. Citado na página 57.

SOMMERVILLE, I. *Engenharia de Software*. São Paulo: [s.n.], 2011. 45-62 p. Citado na página 26.

TAYLOR, M. J.; GRESTY, D.; BASKETT, M. Computer game flow design. *ACM Computers in Entertainment*, v. 4, n. 1, Janeiro 2006. Citado na página 27.

TAZZA, M. Quantitative analysis of a resource allocation problem: a net theory based proposal. *Concorrency and Nets - Springer, Verlage*, p. 511–532, 1987. Citado na página 40.

VALE, N. L. *Especificação de Testes Funcionais usando Redes de Petri a Objetos para Software Orientado a Objetos*. 34-49 p. Dissertação (Mestrado) — Universidade Federal de Uberlândia, Uberlândia-MG, 2009. Citado 4 vezes nas páginas 34, 36, 37 e 38.

WOODOCK, J.; LARSEN, P. G.; BICARSEGUI, J. Formal methods: Pratce and experience. *ACM Computer Survive*, 2009. Citado na página 29.