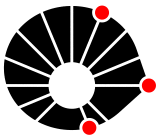
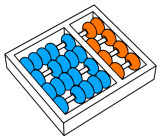




Simulações de Máquinas de Estado *Fuzzy*

R. C. Sofer *E. Martins*

Relatório Técnico - IC-PFG-25-50
Projeto Final de Graduação
2025 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Simulações de Máquinas de Estado *Fuzzy*

Riccardo C. Sofer* Eliane Martins†

Resumo

Máquinas de estado *fuzzy* expandem as máquinas de estado tradicionais, incorporando incertezas em seus modelos. Isso facilita a modelagem de sistemas complexos e torna-as atrativas para várias áreas. No entanto, sua adoção é limitada por duas dificuldades. A primeira refere-se à forma de calcular o valor de pertinência das variáveis de entrada, que são usadas para formar sentenças em lógica *fuzzy* nas transições. A segunda está relacionada à escolha da função de transferência de pertinência entre estados, a qual leva em conta a pertinência do estado fonte e da transição. Ademais, o estudo da simulação das máquinas *fuzzy* é limitado, de modo que certos tópicos, como a validação do modelo, foram pouco desenvolvidos.

Nesse contexto, este trabalho busca aprofundar os estudos na área da simulação como um meio de mitigar essas dificuldades. Isso é feito por meio do desenvolvimento de um método de comparar opções de funções e validar o modelo com o uso de entradas controladas.

1 Introdução

As máquinas de estado *fuzzy* (abreviadas como *FuSMs*, do inglês *Fuzzy finite State Machines*) são modelos matemáticos capazes de representar sistemas e objetos. Uma expansão das máquinas de estado convencionais, as FuSMs incorporam diversas noções da lógica *fuzzy*, descrita originalmente por L. A. Zadeh em 1965 [18]. Notavelmente, permitem que incertezas sejam incorporadas aos modelos, o que permite representar sistemas complexos mais facilmente. Essa categoria de máquina possui diversas aplicações nas áreas do estudo de comportamentos [10], jogos eletrônicos [12] e redes neurais [2].

As FuSM apresentam algumas características notáveis que as tornam ferramentas muito poderosas. Seus estados possuem **graus de atividade**, descritos como graus de **pertinência** em relação ao conjunto dos estados ativos. Ademais, os eventos são tratados como **condições** sobre variáveis, definidas por meio de **sentenças *fuzzy***. Essas sentenças convertem valores das variáveis em graus de verdade por meio de **funções de pertinência**. Quando um evento possui grau de verdade positivo, ele ativa transições, o que resulta na transferência da pertinência de um estado para outro. O quanto é transferido é calculado pela **função de transferência** do modelo, a qual considera o grau de verdade do evento e a pertinência do estado fonte.

Das características das FuSMs surgem dois grandes desafios que limitam sua adoção. O primeiro diz respeito à escolha das funções de pertinência das sentenças *fuzzy* dos eventos, enquanto o segundo está relacionado à escolha da função de transferência do modelo. Para ambos os casos, existem diversas opções de funções, mas não há uma estratégia consolidada para escolhê-las. Ademais, as dificuldades dependem diretamente do modelo em questão, pois afetam seu comportamento. Logo, precisam estar alinhadas com seus requisitos e objetivos, o que se torna outro empecilho.

*Instituto de Computação, UNICAMP, 13083-852 Campinas, SP
†Instituto de Computação, UNICAMP, 13083-852 Campinas, SP

Outro obstáculo no uso das FuSMs é que existem poucos trabalhos que estudam a fundo a simulação dessas máquinas. Disso resulta que não há métodos claros de simular um modelo e validá-lo, ao ponto que essas área são amplamente estudadas nas máquinas de estado convencionais. Sem esses métodos, usar modelos em cenários reais torna-se mais difícil.

Considerando as dificuldades, dois trabalhos se destacam. O trabalho de G. L. Santos, publicado em 2004 [5], estuda a aplicação de FuSMs em jogos virtuais. Nele, o autor apresenta um processo de simulação, desde a construção do modelo, sua codificação e sua execução em tempo real em um cenário simulado fiel a uma aplicação real. Já o trabalho de Todinca et al., publicado em 2018 [17], busca estudar o impacto de diferentes funções de transferência no comportamento do modelo, usando de simulações com entradas controladas para realizar análises. Apesar de distintos, estes estudos avançaram sobre estes tópicos propondo soluções interessantes.

Nosso trabalho tem como enfoque um aprofundamento na simulação de FuSMs, buscando usá-la para mitigar as dificuldades apresentadas. Baseando-se nos trabalhos de G. L. Santos e Tondica et al., buscamos criar uma metodologia que simula o modelo para diferentes funções de transferência e pertinência e que analisa os resultados para definir quais valores são mais adequados ao caso de uso. Aqui, construímos um sistema capaz de realizar esses experimentos e estudamos sua aplicação no modelo de agente de jogos virtuais proposto por Santos, levando em conta seus requisitos e limitações.

1.1 Objetivos

O principal objetivo deste trabalho é estudar a simulação de FuSMs e algumas das dificuldades no seu uso, como a seleção de funções de pertinência de variáveis *fuzzy* e funções de transferência de pertinência entre estados. Usando simulações em um modelo parametrizado, podemos comparar várias opções de valores de parâmetros e analisar critérios como cobertura de estados, grau de atividade dos estados e seleção de estados ativos. Esses dados permitem a escolha fundamentada dos parâmetros a fim de melhor satisfazer um caso de uso.

1.2 Metodologia

A metodologia usada para simular FuSMs consiste em quatro passos fundamentais:

1. Construção de um modelo parametrizado de FuSM baseado em outros trabalhos da literatura, de forma a poder simulá-lo com diferentes combinações de entrada.
2. Codificação do modelo usando a linguagem C++, utilizando-se como base o código de G. L. Santos [5] e aplicando-se a biblioteca *fuzzylite* [14].
3. Construção de um orquestrador de simulações em Python capaz de simular o modelo sob diferentes parâmetros e construir gráficos e relatórios de sua evolução para o usuário.
4. Análise dos resultados para averiguar quais valores dos parâmetros são mais adequados para o caso de uso em questão.

1.3 Organização do Texto

O texto está organizado em seis seções, incluindo a introdução. A seção 2 apresenta uma fundamentação teórica dos conceitos utilizados no texto. A seção 3 apresenta a teoria por trás das máquinas de estado *fuzzy*. A seção 4 trata da simulação das máquinas, abordando pontos como o modelo a ser simulado, a codificação desse modelo e a construção de um sistema de orquestração para realizar várias simulações. A seção 5 expõe os resultados práticos obtidos e reflete sobre eles.

Por fim, a seção 6 apresenta uma conclusão do trabalho e sugestões de próximos passos, seguida das referências bibliográficas utilizadas.

2 Fundamentação Teórica

Nesta seção, trataremos de vários assuntos pertinentes ao universo das máquinas de estado *fuzzy* que serão de suma importância para o restante do trabalho. Primeiro, apresentamos formalmente a lógica *fuzzy* e suas propriedades. Em seguida, apresentamos as máquinas de estados finitos em mais detalhes.

2.1 Fundamentos da Lógica *Fuzzy*

A lógica *fuzzy* é um tipo de lógica multivalorada concebida com o intuito de representar as imprecisões do pensamento humano em proposições matemáticas. Ela representa a realidade de maneira nuançada, enquanto a lógica *crisp* (do inglês, nítida) representa a verdade de modo absoluto. Isso faz com que esse tipo de lógica seja muito utilizado em várias aplicações que precisam incorporar essas imprecisões em seus sistemas.

Quando tratamos de lógicas em um contexto computacional, é comum atribuirmos o valor 1 para proposições verdadeiras e 0 para proposições falsas. A lógica *fuzzy* subverte isso, tratando com proposições cujas verdades estão no intervalo real $[0, 1]$. Isso mostra uma outra propriedade importante da lógica *fuzzy*: ela contém a lógica *crisp* como um caso particular.

Para entendermos a lógica *fuzzy* precisamos apresentar alguns conceitos importantes, como os conjuntos *fuzzy*, as variáveis linguísticas e o processo de *fuzzificação*.

2.1.1 Conjuntos *Fuzzy*

Conjuntos *crisp* possuem a característica intrínseca de que elementos de um universo de discurso pertencem ou não a eles — não há meio termo. Assim como a verdade na lógica *crisp*, a pertinência (pertencimento) a um conjunto é absoluta.

Os conjuntos *fuzzy*, de modo contrário, permitem pertinência parcial a um conjunto. A cada elemento x do espaço de discurso X é atribuído um valor no intervalo $[0, 1]$, o qual recebe nome de **pertinência** ou **verdade**. Esse mapeamento é a chamada função de pertinência (do inglês, *membership function*) do conjunto *fuzzy* [18]. A função define unicamente o conjunto, podendo ser usada para representá-lo. De maneira mais enxuta, podemos representar um conjunto *fuzzy* S de elementos dentro de um universo X qualquer pela seguinte notação:

$$S = \{(x, \delta_S(x)) \mid x \in X\} \tag{1}$$

$\delta_S(x) : X \to [0, 1]$ é a função de pertinência característica de X. Note que, para conjuntos *crisp*, as funções de pertinência possuem valores exatamente iguais a 0 ou 1. Para conjuntos *fuzzy*, é comum usarmos funções de pertinência baseadas em funções reais que respeitam a condição $\delta_S(x)) \leq 1$ para todo x. Funções triangulares, trapezoidais e gaussianas são comumente usadas nesse contexto.

No geral, para conjuntos *fuzzy* sobre uma dimensão, que serão o foco deste trabalho, as funções de pertinência podem ser representadas de maneira gráfica, com o eixo das abscissas representando o universo de discurso X. A figura 1 apresenta alguns exemplos de funções de pertinência características.

No geral, as propriedades e operações dos conjuntos *fuzzy* são as mesmas que os conjuntos *crisp*. Notavelmente, as operações de interseção e união são generalizadas:

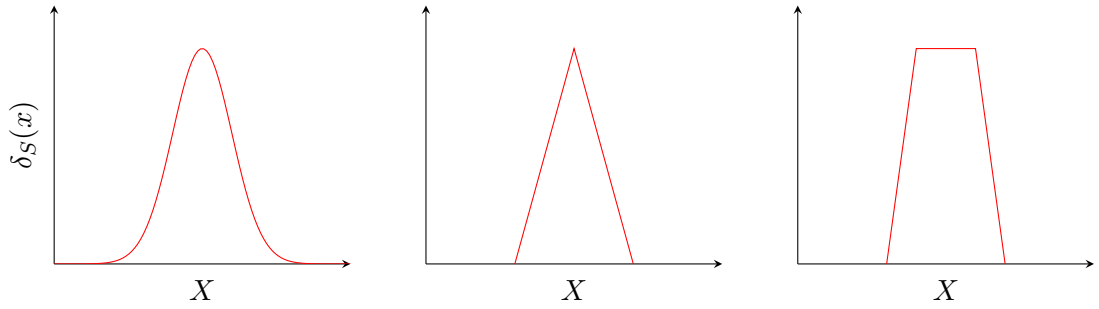


Figura 1: Exemplos de funções de pertinência para diferentes conjuntos *fuzzy*.

- **Interseção**: $A \cap B = T\{\delta_A(x), \delta_B(x)\}$
- **União**: $A \cup B = S\{\delta_A(x), \delta_B(x)\}$

T é uma norma-T e S é uma norma-S, as quais são famílias de funções que obedecem a uma série de propriedades específicas [7]. Logo, existem diversos operadores de interseção e união para conjuntos *fuzzy*. Normalmente, para o contexto de conjuntos *fuzzy*, aplica-se o **mínimo** como interseção padrão e o **máximo** como união padrão. Esse é o comportamento visto para conjuntos *crisp*.

2.1.2 Variáveis Linguísticas

Variáveis linguísticas traduzem a imprecisão da linguagem humana para uma notação matemática. Por esse motivo, são a base de muitas áreas da lógica *fuzzy*, permitindo conectar o universo de fala com números e conjuntos *fuzzy*, dos quais operações e transformações podem ser derivadas.

Pode-se definir variáveis linguísticas por meio de **sentenças** do formato "X é $ADJETIVO$", em que X é a variável linguística em questão e $ADJETIVO$ é o **termo** que caracteriza a variável [19]. Cada sentença possui um conjunto *fuzzy* associado, cuja função de pertinência define a verdade da sentença para os diferentes elementos do universo de fala. Chamamos essas sentenças de **sentenças *fuzzy***.

Por exemplo, tomemos a variável linguística *Temperatura*. Alguns termos que descrevem a temperatura são *quente*, *adequada* e *fria*. Logo, podemos escrever três sentenças a partir disso:

$$\begin{aligned} \delta_1 &= \text{TEMPERATURA é FRIA.} \\ \delta_2 &= \text{TEMPERATURA é ADEQUADA.} \\ \delta_3 &= \text{TEMPERATURA é QUENTE.} \end{aligned} \tag{2}$$

Para cada temperatura em graus Celsius x, teremos uma verdade associada para cada uma dessas sentenças. Desse modo, podemos avaliar a verdade de x em cada sentença por meio da notação $\delta_i(x)$ para $i = 1, 2, 3$. Logo, podemos ver que $\delta_i(x)$ configura uma função de pertinência para a sentença i. Com isso, podemos representar cada variável de modo gráfico e, naturalmente, extrair valores de verdade para cada sentença para uma dada temperatura.

Para esta subseção e a próxima, usamos como exemplo das funções de pertinência de cada sentença o que é mostrado na figura 2.

Tendo-se as funções de pertinência para cada variável, é possível realizar a *fuzzificação* de um valor *crisp*.

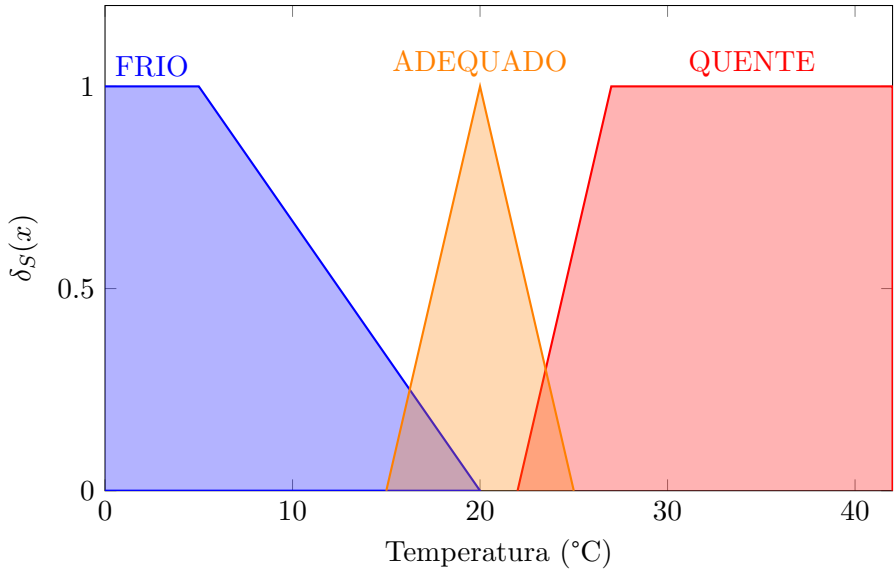


Figura 2: Funções de pertinência para variáveis linguísticas referentes a temperatura.

2.1.3 *Fuzzificação*

O processo de *fuzzificação* consiste em transformar um valor *crisp* em um valor *fuzzy* através da avaliação das funções de pertinência. Para cada sentença com função de pertinência $\delta_i(x)$ no universo X, realiza-se a avaliação da função para obter o valor de verdade para x. Com isso, obtém-se um vetor com os valores de pertinência para cada uma das sentenças avaliadas [9].

Para as funções de pertinência da figura 2, se formos *fuzzificar* o valor $x = 17$, obtemos o vetor F_{17} da equação 3, o qual pode ser obtido seguindo a figura 3.

$$F_{17} = [\delta_1(17), \delta_2(17), \delta_3(17)] = [0.2, 0.4, 0] \tag{3}$$

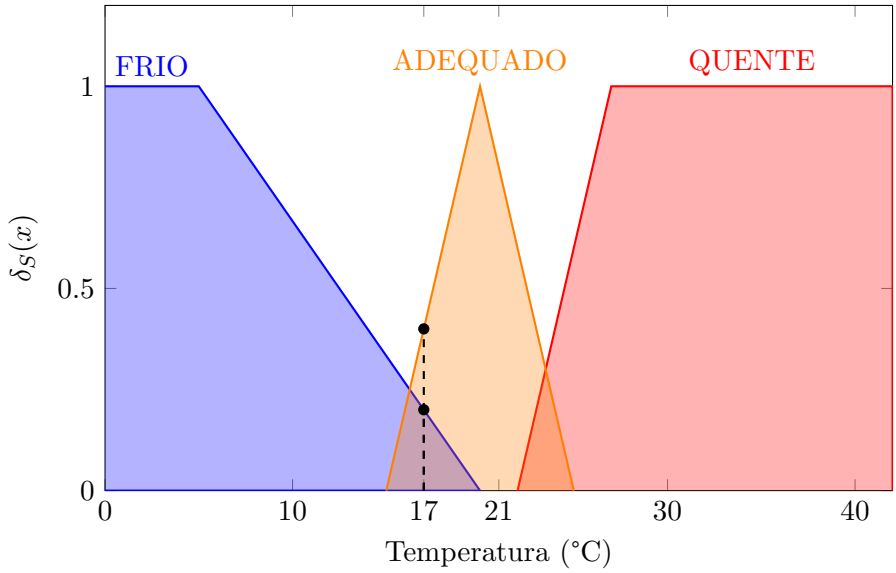


Figura 3: Exemplo de *fuzzificação* para o valor $x = 17$.

Assim, a *fuzzificação* permite passarmos do universo *crisp* ao universo *fuzzy* de maneira simples e eficiente, o que será útil ao tratarmos de várias categorias de máquinas de estado *fuzzy*.

2.2 Máquinas de Estados Finitos

Máquinas de Estados Finitos (abreviadas como MEFs ou *FSMs*, do inglês *Finite State Machines*), também referidas como *autômatos determinísticos finitos*, são um aspecto essencial da teoria da computação e apresentam diversos usos práticos, como a representação de sistemas e comportamentos. Neste contexto, é comum usar o termo **modelo** para as FSMs e suas variantes *fuzzy*.

FSMs são compostas por **estados** (representando, por exemplo, estados de um sistema) e por **transições** entre esses estados. Pode-se ver uma FSM como um **grafo direcionado**, no qual os nós representam os estados, e as arestas direcionadas, as transições. O sistema possui um estado chamado **ativo** ou **corrente**, o qual indica o estado atual. Para FSMs, apenas um estado pode estar ativo por vez. A máquina é capaz de receber sinais chamados de **entradas** ou **eventos**. Quando isso ocorre, a máquina verifica se uma dessas entradas ativa uma transição que sai do estado atual. Em caso positivo, a máquina realiza a transição e o estado ativo muda. Caso contrário, o estado corrente permanece o mesmo. Esse processo é detalhado formalmente na subseção 2.2.1.

Formalmente, pode-se modelar FSMs como uma 6-tupla $M = (Q, \Sigma, \delta, Z, q_0, F)$[16], em que:

- Q é o conjunto finito de **estados**.
- Σ é o conjunto finito de **entradas** (chamado **alfabeto de entrada**).
- $\delta : Q \times \Sigma \to Q$ é a **função de transição**, a qual define qual será o próximo estado, dado um estado atual e uma entrada recebida.
- Z é o conjunto finito de **saídas** (chamado **alfabeto de saída**).
- $q_0 \in Q$ é o **estado inicial** da máquina.
- $F \subseteq Q$ é o conjunto dos estados finais, o qual é muito usado em máquinas de reconhecimento de linguagens. Para este trabalho, este conjunto será sempre vazio.

Um ponto importante das FSMs são suas saídas, contidas no alfabeto Z. No geral, saídas são dados, os quais podem ser simples valores, como números ou caracteres, ou algo mais sofisticado, como um sinal que, por sua vez, aciona uma ação em outro sistema. Com isso as FSMs podem ser usadas para modelar diversos sistemas reais.

Na literatura, costuma-se fazer a distinção entre dois tipos de FSMs no que diz respeito às suas saídas. Nas **máquinas de Moore**, as saídas são devolvidas nos estados, enquanto nas **máquinas de Mealy** elas são devolvidas nas transições [6]. Porém, em diversos contextos, é comum ter uma abordagem híbrida, na qual as saídas podem ser devolvidas tanto em estados quanto em transições. Essa será a abordagem adotada neste trabalho.

2.2.1 Representação

FSMs podem ser representadas de forma gráfica, adotando o estilo de representação comumente usado para grafos em geral. Costuma-se incluir na representação anotações e marcações para indicar aspectos relevantes, como as entradas que ativam as transições correspondentes. Ademais, costuma-se marcar o estado inicial com uma seta. Exemplos estão dispostos nas figuras 4 e 5.

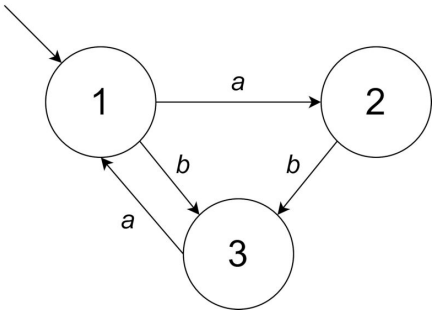


Figura 4: FSM com três estados e duas entradas possíveis. As anotações nas arestas indicam as entradas que as ativam.

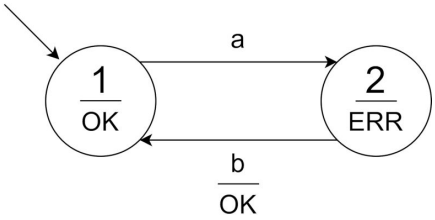


Figura 5: FSM com dois estados, duas entradas e duas saídas (OK e ERR), as quais podem ocorrer tanto em estados quanto transições.

2.2.2 Evolução e Simulação

Como previamente estabelecido, as FSMs podem alterar seu estado corrente à medida que recebem diferentes entradas. Chamamos esse processo de **execução** da máquina, o qual consiste em uma série de **iterações** nas quais a máquina recebe uma entrada, avalia-a e modifica ou não seu estado corrente. No caso das máquinas de estado *fuzzy*, que veremos mais adiante, esse processo chama-se **evolução**.

Formalmente, uma iteração i decorre por meio dos seguintes passos:

1. A máquina apresenta estado corrente q_i (para a primeira iteração, o estado inicial é q_0 por definição).
2. A máquina recebe uma entrada $a \in \Sigma$.
3. Para cada transição que sai de q_i, avalia-se se alguma delas é ativada por a. Se sim, realizamos a transição e alteramos o estado corrente para o novo estado q_{i+1}. Caso contrário, a máquina permanece no estado em que estava.

É importante ressaltar que, para uma máquina **determinística**, no máximo uma transição poderá ser executada a cada iteração; ou seja, não há ambiguidade.

Para este trabalho, trataremos também do conceito de **simulação**. Entende-se como simulação, para os propósitos deste trabalho, a aplicação de entradas virtuais à máquina para estudar sua evolução. O modelo representa o comportamento do sistema, de modo que a simulação permite avaliar tal comportamento **antes mesmo do sistema ser implementado**. É possível simular o modelo apenas conhecendo sua estrutura e as entradas pertinentes. A simulação permite ver se todos os estados são visitados, se todas as transições são exercitadas e como ocorre a progressão de estados, o que pode ser usado para ajustar o modelo, caso necessário [1].

3 Máquinas de Estado *Fuzzy*

Uma generalização das FSMs tradicionais são as máquinas de estados finitos *fuzzy* (abreviadas como FuSMs, do inglês *Fuzzy finite State Machines*. Elas apresentam a característica importante de ter estados com **atividade parcial**, os quais possuem um **grau de pertinência** no intervalo $[0, 1]$ [15]. É importante ressaltar que esta é uma característica de **cada estado** e não do conjunto de estados, de modo que não temos um conjunto *fuzzy* de estados, mas sim um **conjunto de estados *fuzzy***.

Estados *fuzzy* resultam em um aspecto de suma importância para as FuSMs, que é o fato de que vários estados podem estar ativos ao mesmo tempo. Isto é, vários estados podem ter pertinência positiva ao mesmo tempo. Logo, há um **conjunto de estados correntes**, ao invés do único estado ativo das FSMs. Essa característica será amplamente abordada neste trabalho.

Existem diversos modelos de FuSMs [8][15][4], cada um deles apresentando características e funcionamento diferentes. Aqui, focaremos em um único modelo que une ideias de diferentes autores a fim de trazer resultados mais interessantes para a simulação.

3.1 Um Modelo Híbrido de FuSM

Nesta seção, construímos o modelo de simulação, incorporando ideias e conceitos de diferentes autores. De modo básico, temos estados com graus de pertinência e a possibilidade de vários estados com pertinência positiva ao mesmo tempo. Ampliaremos esse modelo introduzindo conceitos como **eventos *fuzzy***, **transições múltiplas**, **funções de transferência** e **sorteio de estados para a realização de ações**.

A primeira adição de nosso modelo é a introdução de **entradas *fuzzy***, chamadas também de **eventos condicionais**, o que deriva do modelo de D. Brubaker [3]. Entradas também apresentam graus de pertinência no intervalo $[0, 1]$, o que indica sua intensidade. No geral, os eventos são apresentados na forma de **sentenças *fuzzy*** atreladas a alguma variável de contexto da máquina. A avaliação dessas sentenças, por meio da *fuzzificação*, permite-nos obter um grau de pertinência para o evento em questão.

No nosso modelo, usaremos funções de pertinência de mesmo formato para todas as sentenças, apenas com escalas e deslocamentos distintos. Por exemplo, podemos ter funções *triangulares* ou *exponenciais*. Vamos nos referir a esse formato e às funções de pertinência no geral como δ.

Nosso modelo também pode ter várias transições que são ativadas ao mesmo tempo, inclusive para um mesmo estado. Isso ocorre porque, para um dado valor de uma variável *crisp*, podemos ter várias sentenças *fuzzy* com pertinência positiva. Ademais, quando uma entrada é recebida pela máquina, **todas as transições ativadas** pela entrada que saem de **todos os estados com pertinência positiva** precisam ser avaliadas. Isso faz com que, no pior caso, todas as transições sejam avaliadas, o que gera grandes problemas de desempenho. Logo, é comum introduzirmos um **limiar mínimo** de pertinência de eventos para ativar transições.

Para a evolução da FuSM, precisamos também definir como ocorrem as mudanças de estado. Isso é feito por meio do uso de uma **função de transferência de pertinência**, o que deriva, novamente, das ideias de Brubaker. Quando uma transição é realizada, parte da pertinência do estado de origem é transferida para o estado de destino. O quanto é transferido é definido, justamente, pela função de transferência. Sendo μ_s, μ_d, μ_e as pertinências da fonte s, do destino d e da entrada e recebida, respectivamente, temos as novas pertinências de s e d na equação a seguir:

$$\mu_s = \mu_s - (\mu_s \cdot \mu_e) \tag{4a}$$
$$\mu_d = \mu_d + (\mu_s \cdot \mu_e) \tag{4b}$$

Aqui, expandimos essa noção para uma família de funções baseadas na função F_1 descrita por Doosfatemeh e Kremer [4]. Assim, qualquer função $F(\mu_s, \mu_e)$ que respeite as seguintes propriedades pode ser uma função de transferência:

- $0 \leq F(\mu_s, \mu_e) \leq 1$
- $F(0,0) = 0$ e $F(1,1) = 1$

Com isso, várias funções, como a média aritmética e a média geométrica, podem ser usadas como funções de transferência. Neste trabalho, usaremos F para nos referirmos à função de transferência em geral.

O último aspecto de nosso modelo é que teremos apenas um **único estado capaz de realizar ações**, apesar de termos vários estados ativos. Nosso modelo, conforme apresentaremos em maiores detalhes mais adiante, é utilizado para modelar um agente em jogos virtuais. Nessa aplicação, assim como em várias outras, apenas é possível realizar uma ação ou ter uma saída por vez, visto que não é possível (ou desejável) que o agente realize várias ações ou tenha vários comportamentos ao mesmo tempo. Desse modo, adotamos a metodologia proposta por G. L. Santos [5] para lidar com essa limitação: um **sorteio do estado que realizará a ação**.

O sorteio ocorre da seguinte maneira: ao final de cada iteração da máquina, todas as pertinências dos estados são tratadas como probabilidades (visto que são valores entre 0 e 1), e uma escolha aleatória, com base nesses pesos, é feita para escolher qual será o próximo estado a realizar sua ação. Apenas as ações e saídas relacionadas ao estado escolhido podem ser realizadas. Para que o sorteio seja possível, introduzimos a limitação de que a soma de todas as pertinências dos estados é 1, característica que é mantida ao longo da evolução já que as transições conservam pertinência. Chamamos o estado selecionado para realizar a ação de **estado sorteado**.

É importante ressaltar que, apesar de termos apenas um estado sorteado capaz de realizar ações, todos os estados ativos (com pertinência positiva) podem, ainda assim, transferir verdade. Logo, como as probabilidades de escolha dos estados mudam de maneira dinâmica, isso gera uma evolução muito menos previsível e interessante do que a FSM convencional.

Uma consequência do sorteio aleatório de estados é que, de uma iteração para outra, podemos ir de um estado sorteado para outro, mesmo que não haja uma transição entre eles. Porém, isso não é um problema, pois apenas podemos eleger como sorteado um estado ativo. Ademais, as transferências de pertinência ainda respeitam as transições. Trata-se apenas de um modo de eliminar a multiplicidade de ações e não impacta a evolução da máquina diretamente.

4 Simulando uma FuSM

Uma vez que definimos a FuSM, podemos partir para a simulação. Esta seção está dividida em quatro partes que cobrem todos os preparativos da simulação: a apresentação do modelo a ser simulado, a definição dos objetivos de simulação, a codificação do modelo e a construção do ambiente de simulação.

4.1 O Modelo de Simulação

Para este trabalho, simulamos um modelo de um agente de jogos virtuais descrito inicialmente por G. L. Santos [5]. Esse modelo representa um agente inimigo do jogador, tendo como objetivo a

sua derrota. Santos define que o agente é composto por um sistema de várias camadas que se comunicam. Dessas, duas camadas são chave para esse trabalho. A **camada de tomada de decisões** é a camada responsável por receber dados do motor gráfico do jogo e, com isso, produzir novos dados para que uma decisão seja tomada. Ela se comunica diretamente com a **camada da máquina de estados**, a qual usa os dados recebidos para calcular novas iterações e eleger um novo estado para realizar uma ação. A ação é efetivamente realizada por outras camadas, as quais se comunicam com o motor gráfico para gerar resultados visíveis ao jogador. Esse processo está esquematizado na figura 6.

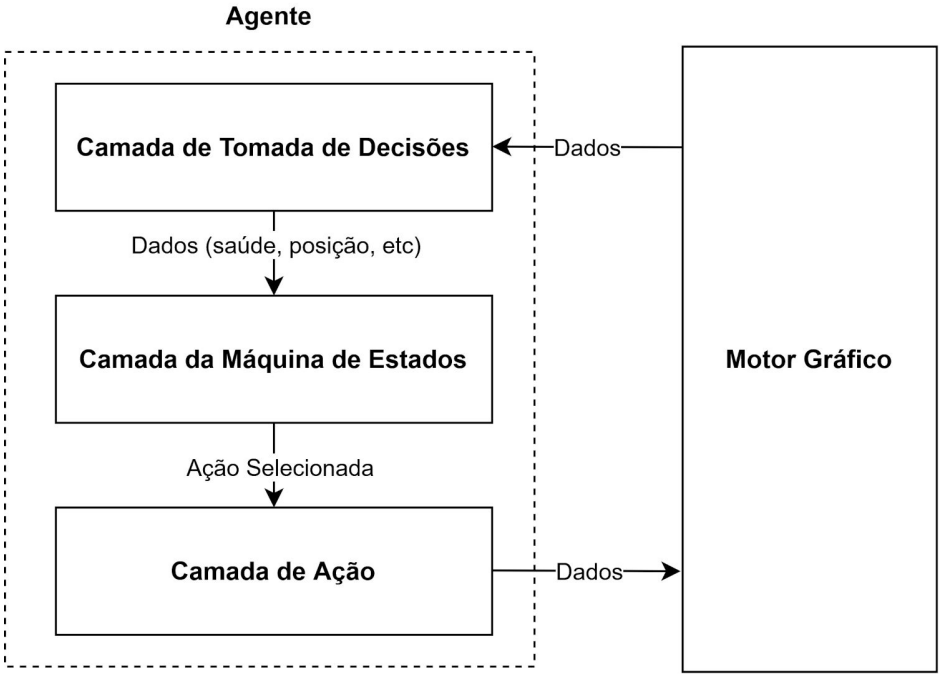


Figura 6: Camadas de operação do agente e as trocas de informação entre elas e o motor gráfico.

Nosso foco está na camada da máquina de estados, a qual é composta por uma FuSM. Para este trabalho, usamos uma versão modificada do modelo de Santos. Nosso modelo apresenta quatro estados:

1. **Vagando**: O agente fica vagando lentamente e não realiza outras ações.
2. **Perseguindo o Inimigo**: O agente detectou o oponente e está se movendo em direção a ele, recebendo ataques de longo alcance do inimigo nesse processo.
3. **Atacando o Inimigo**: O agente está atacando o inimigo com ataques de curto alcance e sendo contra-atacado no processo.
4. **Descansando**: O agente permanece parado para recuperar saúde perdida em batalha.

O funcionamento do modelo segue o que foi descrito por Santos em seu artigo. De modo simples, o agente vaga até que o inimigo entre na distância de perseguição. O inimigo tenta se aproximar do oponente e, caso fique perto o suficiente, ataca-o. Caso sua saúde caia a um nível crítico, o agente foge e se recupera. Uma vez recuperado, o ciclo recomeça.

Além dos quatro estados, o modelo apresenta duas variáveis de contexto, recebidas pela camada de tomada de decisões. Essas variáveis são a **distância** entre o agente e seu inimigo (o jogador) e a

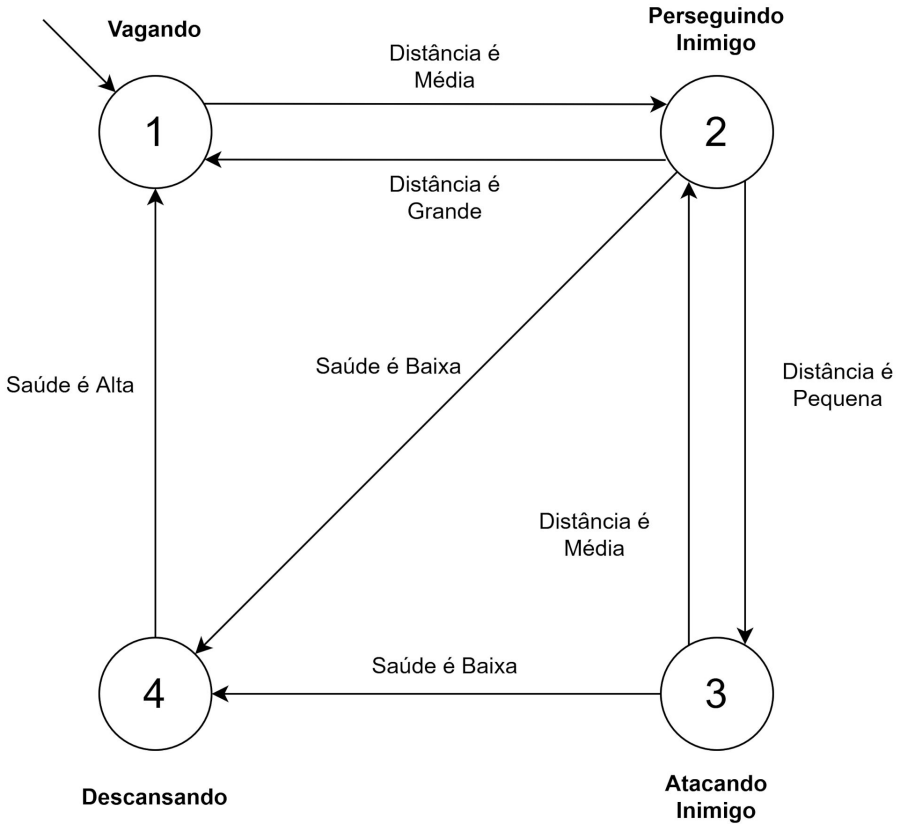


Figura 7: Modelo de agente de jogos virtuais modificado para o modelo de FuSM proposto neste trabalho.

saúde do agente, representando sua integridade física. No modelo, essas variáveis são usadas para gerar eventos condicionais, sendo representadas como sentenças *fuzzy*. Para cada variável, podemos avaliá-la como **grande/alta**, **média** ou **pequena/baixa**.

Todos esses detalhes permitem a construção da máquina, a qual está disposta na figura 7.

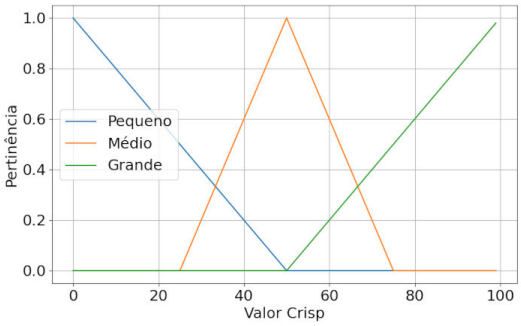
Notavelmente, existem dois aspectos que foram deixados de fora da explicação do modelo. O primeiro deles é a **função de pertinência** δ, a qual define o formato geral das funções de pertinência dos eventos condicionais (como funções triangulares, cossenoidais ou exponenciais). O segundo é a função de transferência de verdade entre os estados, F. Elegemos esses aspectos como **parâmetros** variáveis do modelo, de modo que possamos escolher diversas funções para cada um deles. Neste trabalho, usamos os seguintes valores possíveis para cada:

- **Funções de Pertinência** δ
  - **Funções triangulares**: figura 8a.
  - **Funções exponenciais**: figura 8b.
  - **Funções cossenoidais**: figura 8c.
  - **Função razão (*fuzzyEval*)** [5]: figura 8d.
- **Função de Transferência**: $F(\mu_s, \mu_e)$
  - **Média Aritmética**:$\frac{(\mu_s+\mu_e)}{2}$
  - **Máximo**: $max\{\mu_s, \mu_e\}$

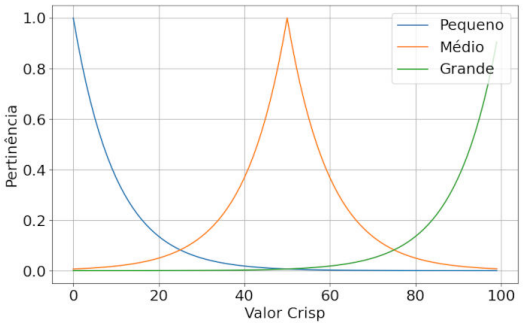
- **Mínimo**: $min\{\mu_s, \mu_e\}$
- **Média Geométrica**: $\sqrt{\mu_s \cdot \mu_e}$
- **Diferença Limitada**: $max\{0, \mu_s + \mu_e - 1\}$
- **Produto**: $\mu_s \cdot \mu_e$

Para as funções de pertinência com formatos triangular, exponencial e cossenoidal, utilizamos as mesmas funções para cada variável de contexto, apenas com escalas diferentes. As funções usadas estão dispostas nas figuras 8a, 8b, 8c. A exceção para isso é a função *fuzzyEval*, usada por Santos em sua implementação. Essa função não usa variáveis linguísticas, mas sim valores alvo *crisp* específicos de cada transição. Ela realiza uma razão entre o valor atual da variável de contexto e o valor alvo. Essa razão tem como denominador sempre a variável de maior valor. A equação que representa essa função está na equação 5, e seu formato pode ser visto na figura 8d.

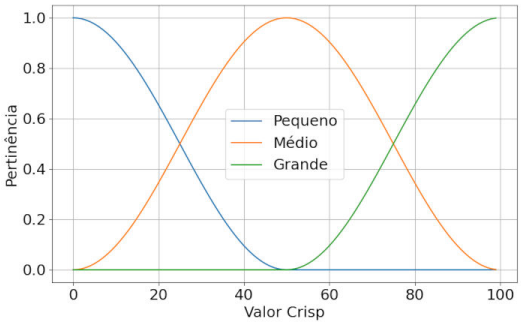
$$fuzzyEval(value, target) = \begin{cases} \frac{\text{value}}{\text{target}}, & \text{se } value < target \\ \frac{\text{target}}{\text{value}}, & \text{caso contrário.} \end{cases} \tag{5}$$



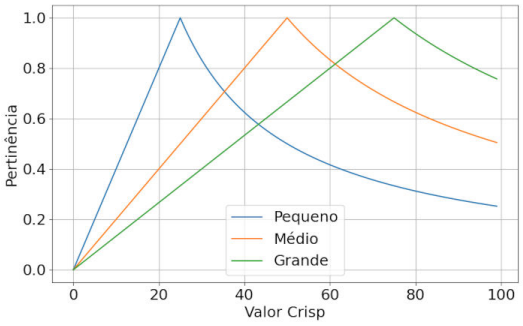
(a) Funções de pertinência de formato triangular.



(b) Funções de pertinência de formato exponencial.



(c) Funções de pertinência de formato cossenoidal.



(d) *fuzzyEval* para alvos pequeno (25), médio (50) e grande (75).

Figura 8: Opções de função de pertinência para as variáveis linguísticas.

Com isso, temos o modelo bem definido para que possamos simulá-lo.

4.2 Objetivos da Simulação

Um aspecto importante da simulação a ser definido são seus objetivos. Isso permite estabelecer quais são as métricas e aspectos a serem observados durante a simulação. No caso, queremos

analisar a evolução da máquina ao longo da simulação em face de diferentes parâmetros (a função de pertinência δ e a função de transferência de pertinência F). Isso, por sua vez, permite-nos escolher a melhor opção, dados os critérios desejados, para um caso de uso específico.

A fim de alcançar nosso objetivo, vamos observar os seguintes aspectos da máquina:

- **Variação dos graus de pertinência dos estados ao longo do tempo**. Isso permite observar as tendências de transferência de pertinência entre estados, além das probabilidades de escolha do sorteio. Além de uma análise visual, coletamos a média das pertinências para análises mais precisas.
- **Variação do estado sorteado ao longo do tempo**. Isso permite ver como o sorteio foi afetado conforme a máquina evoluiu. Ademais, permite-nos analisar as ações da máquina ao longo da simulação.
- **Cobertura de estados**. É importante saber se todos os estados foram visitados durante a simulação. Caso isso não ocorra, é possível que uma combinação de parâmetros não ative certas transições e limite a evolução do modelo.

Com isso, podemos realizar análises substanciadas sobre o comportamento da máquina em face de diferentes parâmetros.

4.3 Codificando o Modelo

A fim de simular o modelo, precisamos primeiro codificá-lo em uma linguagem de programação que possa ser executada. Esta subseção trata do código usado para simular o modelo, o qual está dividido em dois módulos, conforme mostra a figura 9:

- O **módulo de execução** lê entradas e configurações a fim de comandar a simulação, executando as iterações e coletando métricas para serem salvas em um arquivo de saída.
- O **módulo da FuSM** implementa a máquina de estados *fuzzy* por meio de uma classe cujos parâmetros podem ser alterados, permitindo utilizá-la para várias simulações.

O código desta subseção e das demais está disponível em `https://github.com/ra247362/PFG`.

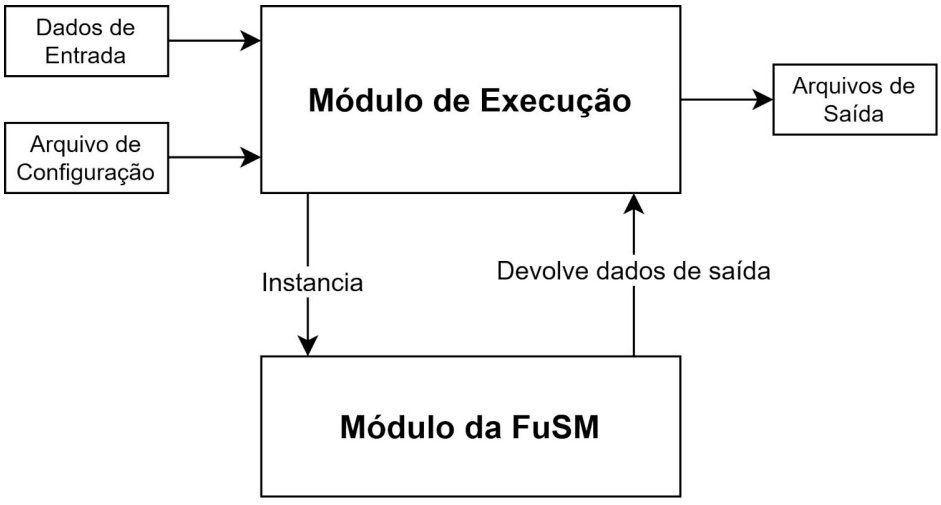


Figura 9: Diagrama mostrando os módulos do código do modelo.

4.3.1 Módulo de Execução

O módulo de execução, que é a entrada do programa, realiza a simulação em si. Ele recebe como entrada um arquivo de configuração no formato JSON (como visto na figura 10) passado como argumento. Esse arquivo dita vários aspectos da simulação, como quais parâmetros serão usados, quais os valores iniciais das variáveis de contexto, entre outros. Essas configurações são usadas para instanciar uma FuSM que respeite essa configuração e realize, assim, a simulação necessária.

O módulo de execução usa o arquivo de configuração para ler também um possível arquivo de entrada com valores de distância e saúde para cada iteração. Isso permite que tenhamos uma progressão definida dessas variáveis de contexto, o que nos permite alterar apenas as funções de transferência e pertinência e realizar comparações mais interessantes.

Para cada iteração até que o número máximo seja atingido, este módulo chama o módulo de FuSM e, após a evolução ser calculada, coleta as seguintes estatísticas:

- Pertinência de cada estado.
- Distância
- Saúde
- Pertinência da distância para cada sentença *fuzzy*.
- Pertinência da saúde para cada sentença *fuzzy*.
- Estado sorteado.

Esses valores são compilados em arquivos no formato CSV para serem analisados.

```
{
    "name": "PROD - RAMPS",
    "transference_function": "PROD",
    "membership_function": "RAMPS",
    "iteraction_count": 150,
    "display_output": false,
    "csv_output": "./PFG/output/csvs/PROD - RAMPS - State Memberships.csv",
    "distance_output": "./PFG/output/csvs/PROD - RAMPS - Distance.csv",
    "health_output": "./PFG/output/csvs/PROD - RAMPS - Health.csv",
    "active_state_output": "./PFG/output/csvs/PROD - RAMPS - Active States.csv",
    "progression_output": "./PFG/output/csvs/PROD - RAMPS - Progression.csv",
    "distance_membership_output": "./PFG/output/csvs/PROD - RAMPS - Distance Membership.csv",
    "health_membership_output": "./PFG/output/csvs/PROD - RAMPS - Health Membership.csv",
    "initial_distance": 75,
    "initial_health": 100,
    "randomize": false,
    "input_file": "./PFG/input/generated.csv"
}
```

Figura 10: Exemplo de arquivo de configuração lido pela FuSM.

4.3.2 Módulo da FuSM

O código utilizado neste módulo foi criado com base, novamente, no trabalho de Santos. Aqui, a máquina é implementada, assim como toda a lógica necessária para executar uma iteração da máquina, como transições, cálculos de pertinência e transferência de verdade. Aspectos do código foram mantidos, enquanto outros foram revisados e expandidos.

Notavelmente, um aspecto mantido foi o uso da linguagem de macros em C++ (definições reutilizáveis de código) para a definição de FSMs concebida originalmente por Rabin em 2002 [13]. Essa linguagem permite escrever de modo conciso máquinas de estados finitos e pode ser facilmente

expandida para definir máquinas de estado *fuzzy*. Ela oferece macros para definir estados, tratar eventos e realizar transições. A interface dessa linguagem está exibida na tabela 1.

Macro	**Explicação**
EVENT_ENTER	Gatilho que é ativado quando a máquina entra em um estado.
EVENT_UPDATE	Gatilho que é ativado quando a máquina permanece em um estado.
EVENT_EXIT	Gatilho que é ativado quando a máquina deixa um estado.
BeginStateMachine e EndStateMachine	Delimitam a estrutura da máquina em si.
State(a)	Condicional que é ativada quando se tem o estado a.
OnEvent(a)	Condicional que é ativada quando o evento a é recebido.
OnEnter, OnUpdate, OnExit	Instâncias de OnEvent para a sendo EVENT_ENTER, EVENT_UPDATE e EVENT_EXIT, respectivamente.

Tabela 1: Linguagem de macros de Rabin.

A linguagem de macros baseia-se na definição de FSM da *Unified Modeling Language* (UML), em que saídas e ações podem ser realizadas quando se entra, quando se permanece e quando se saí de um estado [11]. O momento de se realizar as ações é indicado pelos eventos *OnEnter*, *OnUpdate* e *OnExit*. Esses eventos são chamados de **gatilhos** (do inglês *trigger*) pela UML, e são distintos dos eventos condicionais. Neste trabalho, as ações principais são realizadas quando o evento *OnUpdate* é recebido, e os demais eventos são usados para realizar ações auxiliares (como obter a identidade do inimigo mais próximo).

Santos define sua FuSM na classe *CFuSM*, o que nos dá uma vasta gama de ferramentas para definir a máquina. Os atributos da nossa versão e os métodos implementados estão dispostos nas figuras 11 e 12, respectivamente.

```
// Configuration
std::string m_transfer;
std::string m_membershipFunction;
bool m_randomize;

// General members
int m_numberOfStates;
std::vector<double> m_degreeOfMembership;
int m_activeState;
int m_enemyDist;
int m_health;

// Membership values
std::map<std::string, double> m_membershipDistance;
std::map<std::string, double> m_membershipHealth;

// Pointers
Player *m_enemy;
```

Figura 11: Atributos da classe CFuSM

Outro ponto crucial do código é como definimos e tratamos os parâmetros da FuSM, F e δ (ver seção 4.1). Cada um deles é implementado em uma combinação de atributos, métodos e arquivos

```
// Methods
std::string process();
std::string getVariableMembership(double number, double maximumValue);
bool runMachine(int event, int state);
void setState(int sourceState, int destState, double factor);
int getRandomStateBasedOnDegreeOfMembership();

// Membership function method
double membershipFunction(double number, std::string transition, double maximumValue);

// Transfer method
double transferFunction(double sourceStateMembership, double eventMembership);

// Simulated agent actions
void wander();
void runToEnemy(Player *enemy);
void findSafeRestingPosition();
void rest();
void attackEnemy(Player *enemy);
Player *getClosestEnemy();
```

Figura 12: Métodos da classe CFuSM.

externos de maior complexidade. Isso permite escalar a implementação facilmente caso mais funções precisem ser adicionadas, além de melhorar a manutenibilidade do código.

Para F, todas as funções de transferência são implementadas no arquivo *TransferFunctions.cpp*. O atributo *m_transfer* indica qual função a máquina deve usar, e o método *transferenceFunction* analisa esse valor e chama a implementação correta para a coleta do valor da transferência.

A transferência de pertinência entre estados é realizada de fato no método *setState*, o qual recebe a pertinência do estado fonte e a verdade do evento condicional (calculado usando δ). Aqui, apenas chamamos a função F com as entradas corretas, obtemos o valor resultante, subtraímos da pertinência do estado fonte e somamos na pertinência do estado destino.

O parâmetro δ requer maior cuidado. Para ele, temos todas as implementações das funções de pertinência no arquivo *MembershipFunctions.cpp*, e o atributo *m_membershipFunction* indica qual função a máquina deve utilizar. Como temos funções que usam lógica *fuzzy* (funções triangular, exponencial e cossenoidal) e uma função que usa valores numéricos (função razão), foi necessário implementar uma lógica com maior nuance para calcular a pertinência.

Primeiro, atribuiu-se a cada transição da máquina um valor numérico (para a razão) e um termo (para as demais funções). Os valores numéricos utilizados foram retirados do trabalho de Santos, a fim de manter sua intenção original e prover comparativos interessantes. Para os termos, foram usados aqueles definidos na definição do modelo da figura 7.

Com a distinção entre tipos de funções, usamos o método *membershipFunction* para receber o valor *crisp*, a transição que estamos analisando (a fim de coletar o termo ou valor associado) e o valor máximo da variável analisada (para alterar a escala das funções). Esse método também aplica um limiar mínimo de 0.6 de pertinência, zerando valores abaixo desse valor. Isso segue as observações de Santos, que argumenta que valores pequenos de pertinência podem afetar o desempenho do modelo.

As funções *fuzzy* foram implementadas usando a biblioteca de lógica *fuzzy* *fuzzyLite* [14]. Ela permite criar variáveis linguísticas com vários termos e avaliar a pertinência de cada um deles de maneira simples, conforme mostra a figura 13. Usamos funções de pertinência já implementadas na *fuzzyLite* para compor nossas funções de pertinência.

Um outro ponto relevante da implementação das funções de pertinência no arquivo *MembershipFunctions.cpp* é que cada variavel apresenta termos específicos ao seu contexto. Por exemplo, o

```cpp
double membership::triangular(double maximumValue, double number, std::string term)
{
    fl::InputVariable variable; // Define crisp variable to be fuzzified
    std::string translatedTerm = membership::TERM_TRANSLATIONS.at(term);
    variable.setName("Variable");
    variable.setRange(0.0, maximumValue); // Define range of the variable (X-axis)

    variable.addTerm(new fl::Ramp("SMALL", maximumValue / 2, 0.0));
    variable.addTerm(new fl::Triangle("MEDIUM", maximumValue / 4, maximumValue / 2, 3 * maximumValue / 4));
    variable.addTerm(new fl::Ramp("LARGE", maximumValue / 2, maximumValue));

    return variable.getTerm(translatedTerm)->membership(number);
}
```

Figura 13: Exemplo de implementação de funções de pertinência triangulares com a biblioteca *fuzzyLite*.

termo *baixo* se aplica apenas à saúde, não à distância. No entanto, todos os termos apresentam alguma noção comum que nos permite mapeá-los para termos genéricos, como *grande*, *médio* e *pequeno*. Isso faz com que possamos reutilizar as mesmas funções para as duas variáveis. Os mapeamentos podem ser vistos na tabela 2.

Transição	**Identificador**	**Valor Numérico**	**Termo Distância**	**Termo Saúde**	**Termo Genérico**
$1 \rightarrow 2$	WANDER_TO_CHASE	90	Média	-	Médio
$2 \rightarrow 1$	CHASE_TO_WANDER	110	Grande	-	Grande
$2 \rightarrow 3$	CHASE_TO_ATTACK	25	Pequena	-	Pequeno
$3 \rightarrow 2$	ATTACK_TO_CHASE	35	Média	-	Médio
$2 \rightarrow 4$	CHASE_TO_REST	25	-	Baixa	Pequeno
$3 \rightarrow 4$	ATTACK_TO_REST	25	-	Baixa	Pequeno
$4 \rightarrow 1$	REST_TO_WANDER	100	-	Alta	Grande

Tabela 2: Mapeamentos de transições, seus identificadores e seus termos.

Outro atributo importante é o **m_randomize**, que define se a máquina irá ou não alterar os valores das variáveis de contexto. Se for falso, a máquina lê todos os valores de distância e saúde do arquivo de entrada e não os modifica. Porém, caso seja verdadeiro, a máquina irá usar valores iniciais de distância e saúde e vai alterá-los a cada iteração com base na ação sorteada. No geral, a variação consiste na adição ou subtração de um valor aleatório, e a quantidade variada depende da especificação do estado sorteado. No código do modelo, utilizamos diversos métodos descritos por Santos para criar essas ações:

- O método *wander* representa o estado *Vagando* e tende a diminuir lentamente a distância.
- O método *runToEnemy* representa o estado *Perseguindo Inimigo*, tende a diminuir a distância (mas pode aumentá-la, indicando fuga do inimigo) e reduz um pouco a saúde (indicando ataque à distância do inimigo).
- O método *attackEnemy* representa o estado *Atacando Inimigo*. Nele, a distância tende a aumentar um pouco (indicando fuga) e a saúde diminui (indicando o contrataque do inimigo).
- O método *findSafeRestingPosition* é acionado ao entrar no estado *Descansando* (por meio da macro com gatilho *OnEnter*), indicando a fuga do agente até um local seguro para descansar. Aumenta muito a distância.

- O método *rest* representa o estado *Descansando* e aumenta a saúde.

No trabalho de Santos, esses métodos eram implementados com base no sistema do jogo em si. Como não dispomos desse sistema, a simulação dessas ações foi o caminho preferido. Isso permite que não precisemos ter o sistema implementado para simular o modelo, tornando-o mais flexível.

O principal método da *CFuSM* é o método *runMachine*, o qual recebe um estado e um gatilho do método *process*. Ele usa a linguagem de macros para construir a seguinte lógica para cada estado:

- Caso o evento seja *OnEnter* ou *OnExit*, ele realiza a ação associada a este gatilho para o estado. No geral, essa ação é uma atribuição de variável, mas pode ser uma ação do agente de fato (como a chamada de *findSafeRestingPosition* para o estado *Descansando* em *OnEnter*).
- Caso o evento seja *OnUpdate*, a ação associada ao estado é realizada apenas se o estado estiver marcado como sorteado segundo o atributo *m_activeState*. Depois disso, todas as transições que saem do estado são avaliadas usando o método *membershipFunction* para obter a verdade de cada evento condicional. Para cada transição, caso a pertinência seja positiva, o método *setState* é chamado para realizar a transferência de pertinência entre o estado fonte e o estado destino.

Por fim, o método *process* chama *runMachine* para todos os estados ativos com o gatilho *OnUpdate* e, em seguida, realiza o sorteio do novo estado sorteado pelo método *getRandomStateBasedOnDegreeOfMembership*. Se o estado sorteado mudou de i para j com o sorteio, *process* chama *runMachine* com o gatilho *OnExit* para i e chama com o *OnEnter* para j. Com isso, concluímos uma iteração da máquina.

4.4 Construindo o Ambiente de Orquestração de Simulações

O código do modelo é capaz de realizar uma simulação para um conjunto definido de parâmetros lidos do arquivo de configurações. A fim de automatizar a criação de arquivos de configuração, a geração de arquivos com valores de distância e saúde, e a construção de gráficos e relatórios de métricas para análise, foi criada uma série de *scripts* em Python que formam o **ambiente de orquestração de simulações**.

O ambiente de orquestração é formado por quatro módulos: o orquestrador, o gerador de entradas, o executor e o analisador. O fluxo de iteração do sistema pode ser visto na figura 14.

Com base no diagrama, podemos explicar cada um dos 11 passos que compõem sua execução:

1. O usuário ativa o módulo orquestrador, o que inicia o processo de simulações.
2. O orquestrador lê as configurações globais das simulações de um arquivo de configuração global no formato JSON. Essas configurações incluem a lista dos parâmetros a serem simulados, um valor que indica se as entradas (variáveis de contexto) serão geradas ou aleatorizadas, o número de iterações, entre outros.
3. Para o caso em que as entradas são geradas, o orquestrador ativa o módulo de geração de entradas. Esse módulo cria arquivos CSV com a distância até o inimigo e a saúde que a FuSM deve ter para cada iteração.
4. O módulo de geração cria os dados em ciclos, seguindo a estratégia de Todinca et al [17]. Em cada ciclo, a variável vai de seu valor máximo até o mínimo e depois retorna ao máximo, variando uma unidade por iteração. Todos os dados são salvos em arquivos CSV.
5. O orquestrador cria um arquivo de configuração em JSON para cada possível combinação de parâmetros F e δ das opções definidas no arquivo global. Neles, o orquestrador também

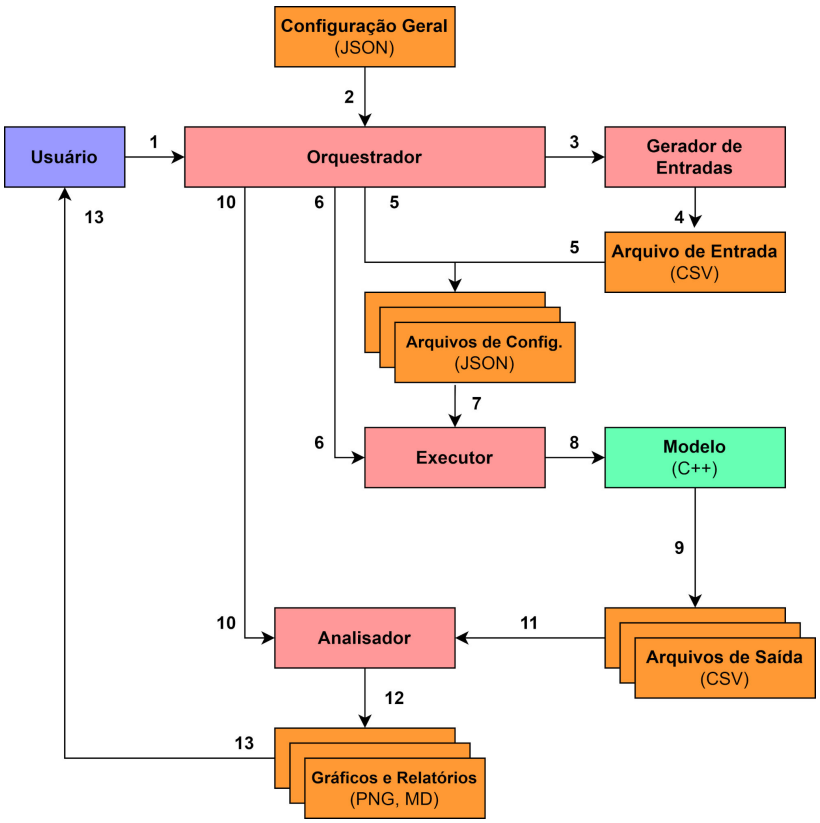


Figura 14: Diagrama do orquestrador de simulações.

insere o caminho do arquivo de entrada produzido pelo gerador.

6. O orquestrador ativa o módulo executor, responsável por chamar o executável do modelo com as configurações corretas.
7. O executor lê os arquivos de configurações, um por vez.
8. O executor chama o executável do modelo com um caminho de arquivo de configuração. Esse passo é repetido para cada arquivo de configuração possível.
9. O executável do modelo gera uma série de arquivos de saída no formato CSV com dados como pertinência dos estados e os estados sorteados.
10. O orquestrador chama o módulo analisador para começar a análise dos resultados.
11. O analisador lê todos os arquivos de saída. Para cada combinação de parâmetros, ele gera um relatório com estatísticas, como a pertinência média dos estados e o estado que mais foi sorteado, e gráficos mostrando a evolução de vários aspectos da máquina.
12. O analisar salva os gráficos em formato PNG e os relatórios em formato MarkDown (MD).
13. Os resultados finais são acessados pelo usuário para que as análises sejam realizadas.

Caso a configuração geral indique que os experimentos devem usar dados de distância e saúde criados pelo próprio modelo, a diferença no processo é que os passos relacionados ao gerador são omitidos. Fora isso, o processo permanece o mesmo.

Os *scripts*, como mostra o diagrama, geram diversos arquivos. O que se segue é uma lista de todos os arquivos gerados pelos diferentes módulos.

- Orquestrador
  - Para cada combinação possível de F e δ, um arquivo de configuração JSON é gerado.
- Gerador de Entradas
  - *generated.csv*
- Analisador
  - Relatório em Markdown com os seguintes dados:
    - Média e desvio padrão da pertinência de cada estado.
    - Valores máximos, médios e mínimos para distância e saúde.
    - Média, moda e desvio padrão dos estados sorteados.
    - Quantidade de vezes que cada estado foi sorteado.
  - Gráficos para as seguintes séries de dados:
    - Estados sorteados.
    - Distância.
    - Saúde.
    - Pertinência de Estados
    - Pertinências da distância para cada sentença *fuzzy*.
    - Pertinências da saúde para cada sentença *fuzzy*.

5 Resultados e Discussões

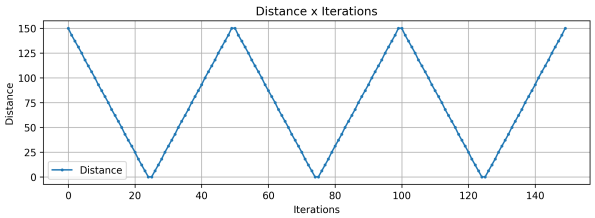
Nesta seção, apresentamos os principais resultados obtidos e discutimos sobre eles. Para isso, explicamos quais foram as configurações escolhidas e as justificativas envolvidas. Por uma questão de brevidade, nem todas as opções para F e δ são apresentadas neste relatório. Porém, todos os resultados podem ser encontrados no diretório *output* do repositório de código do projeto, disponível em `https://github.com/ra247362/PFG`

5.1 Configurações Selecionadas

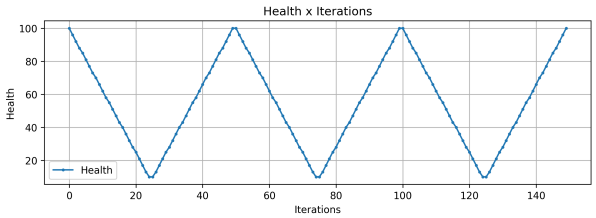
Para os resultados analisados neste relatório, optamos por gerar dados de distância e de saúde para cada iteração, o que permite ter uma progressão idêntica dessas variáveis para todas as combinações de parâmetros. Isso está de acordo com o método proposto por Todinca et al. [17], e permite uma melhor análise da influência de cada parâmetro. Além disso, utilizamos uma semente com valor 2025 para o gerador de números aleatórios para garantir a replicabilidade dos experimentos. Essas estratégias resultam em experimentos controlados que levam a análises mais detalhadas.

Os valores gerados estão dispostos nas figuras 15a e 15b. No total, foram realizadas 150 iterações. Cada variável faz três ciclos de variação de seu máximo até o mínimo e voltando. Para a distância, os valores máximo e mínimo são 150 e 0, respectivamente. Para saúde, são 100 e 10, respectivamente.

Um ponto importante a se notar é que a saúde mínima é 10 e não 0. Isso está de acordo com a especificação do modelo. Caso a saúde chegue a 0, o agente é derrotado, o que deve abortar a simulação. O valor 10 permite simular, pelos dados gerados, o agente fugindo do oponente quando a saúde fica crítica, trazendo resultados mais interessantes para a simulação.



(a) Progressão da distância.

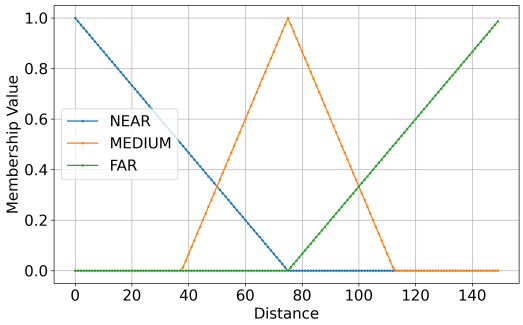


(b) Progressão da saúde.

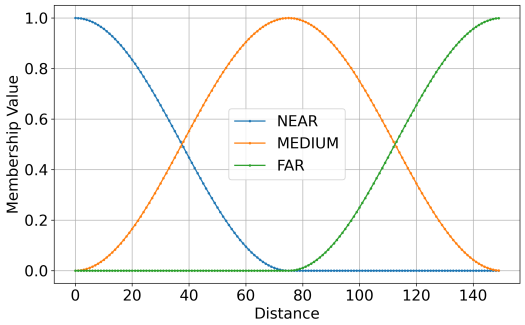
Figura 15: Progressão das variáveis de contexto para os experimentos.

5.2 Resultados

Para este relatório, apresentamos apenas os resultados referentes às funções F **produto** e **diferença limitada** e às funções de pertinência δ **triangular** e **cossenoidal**. Os valores de pertinência para cada distância e saúde podem ser vistos nas figuras 16 e 17.

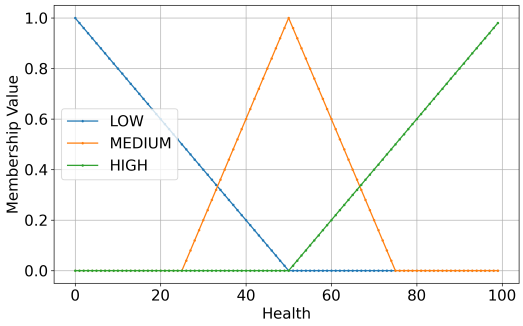


(a) δ triangular.

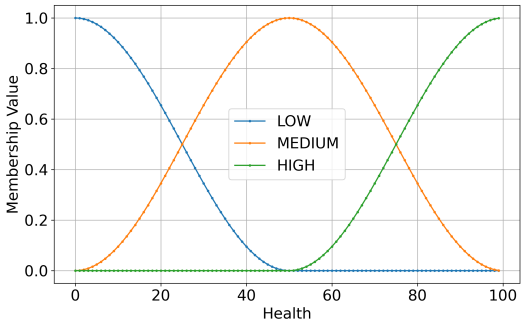


(b) δ cossenoidal.

Figura 16: Evolução dos graus de pertinência da distância para cada termo de cada δ.



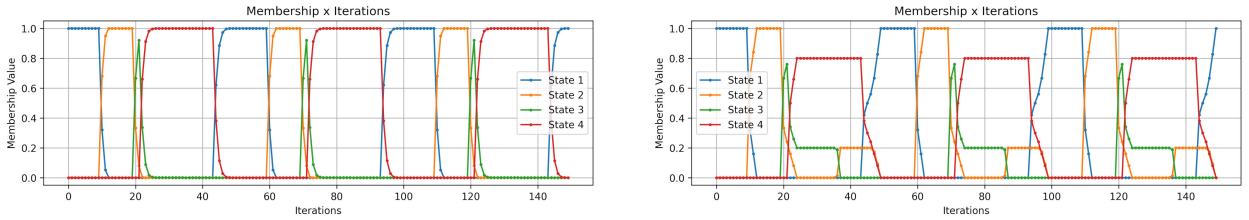
(a) δ triangular.



(b) δ cossenoidal.

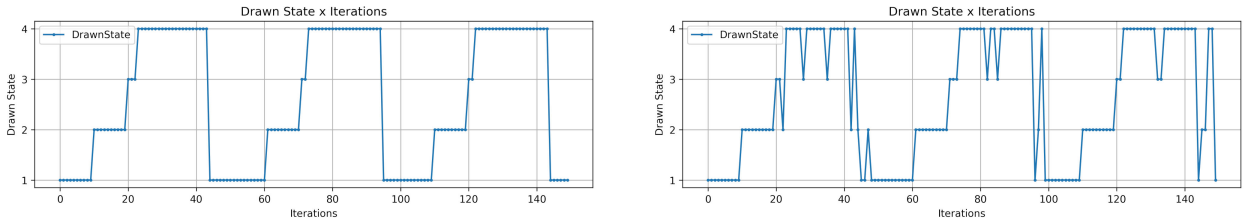
Figura 17: Evolução dos graus de pertinência da saúde para cada termo de cada δ.

Primeiro, analisamos a influência da escolha de F na evolução do modelo. Para isso, fixamos δ como **triangular** e comparamos a progressão dos graus de pertinência de cada estado na figura 18 e a sequência de estados sorteados na figura 19.



(a) Produto x Triangular. (b) Diferença Limitada x Triangular.

Figura 18: Evolução dos graus de pertinência de cada estado para diferentes opções de F.



(a) Produto x Triangular. (b) Diferença Limitada x Triangular.

Figura 19: Estados sorteados para cada iteração para diferentes valores de F.

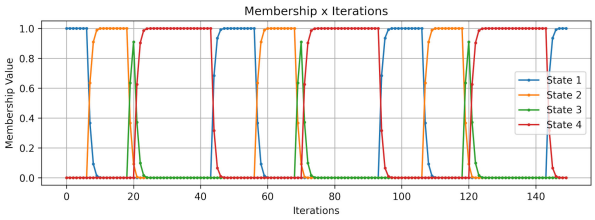
Existem vários pontos a serem analisados. Primeiro, nota-se que as progressões de pertinência divergem consideravelmente. O produto seguiu uma progressão clara de pertinências, com cada estado tendo dominância (sendo o estado com maior pertinência) um depois do outro. No geral, observou-se apenas um estado ativo, com exceção das iterações em que o estado dominante muda. A diferença limitada, por outro lado, apresenta um comportamento com muito mais incertezas, visto que temos vários estados ativos por muito mais iterações, como ocorre entre as iterações 25 e 30 e entre 120 e 150 da figura 18b.

A presença de vários estados ativos também leva a divergências na sequência de estados sorteados. Para o produto, ela segue exatamente a ordem dos estados (1, 2, 3 e 4), permanecendo mais nos estados 1 e 4 do que nos demais. A diferença limitada apresenta um comportamento muito mais imprevisível. Apesar de ainda sortear predominantemente os estados 1 e 4, as sequências de sorteios desses estados são interrompidas por sorteios dos estados 2 e 3, como ocorre entre as iterações 20 e 40. De qualquer modo, para ambas as funções, todos os estados receberam algum valor de pertinência e foram sorteados no mínimo uma vez.

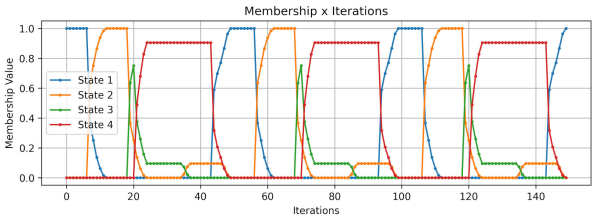
Conhecendo a progressão da máquina para diferentes valores de F, podemos variar δ e entender o que muda. A figura 20 mostra a evolução da pertinência dos estados para δ cossenoidal, e a figura 21 exibe como fica a sequência de estados sorteados para esse formato de função.

À primeira vista, pouco muda quando trocamos δ do formato triangular para o formato cossenoidal. No entanto, existem alguns pontos a serem observados. Para o produto, o formato das curvas de progressão de pertinência dos estados permanece, no geral, o mesmo. Porém, ao se analisar as primeiras 20 iterações pela figura 22, notamos que a função δ cossenoidal tende a adiantar a progressão, deslocando o gráfico como um todo para a esquerda. Podemos interpretar isso como uma maior **reatividade** do modelo à variações de distância e saúde.

No caso da diferença limitada, as distinções estão mais claras. Primeiro, nota-se que o formato das curvas de pertinência dos experimentos é parecido, mas a quantidade de pertinência para cada estado é diferente. No geral, o uso de funções cossenoidais faz com que alguns estados acumulem mais pertinência em relação aos outros. Como mostra a figura 23, para as últimas 20 iterações da

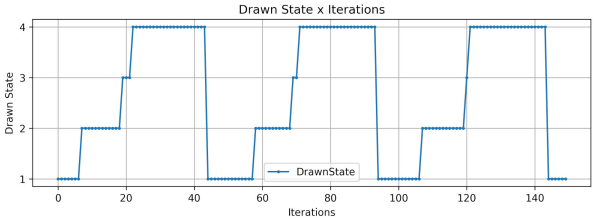


(a) Produto x Cossenoidal.

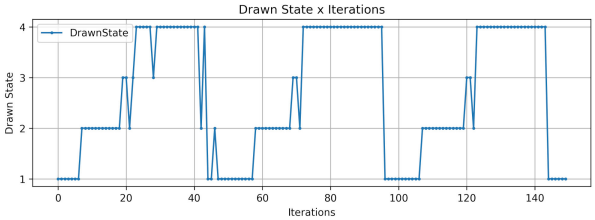


(b) Diferença Limitada x Cossenoidal.

Figura 20: Evolução dos graus de pertinência de cada estado para δ cossenoidal.



(a) Produto x Cossenoidal.

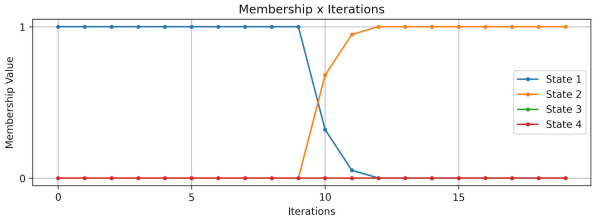


(b) Diferença Limitada x Cossenoidal.

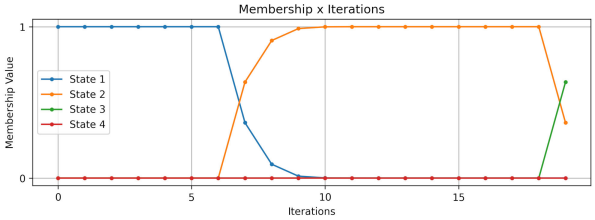
Figura 21: Estados sorteados para cada iteração para δ cossenoidal.

simulação, o estado 4 apresenta cerca de 0,9 de pertinência para δ cossenoidal, enquanto apresentava 0,8 quando δ era triangular. Uma consequência disso é que a maior incerteza anteriormente vista com a diferença limitada é reduzida, o que gera sorteios de estados com sequências contínuas maiores, como vemos na figura 21b.

As análises feitas até aqui podem ser aprofundadas utilizando os dados presentes nos relatórios em formato Markdown de cada experimento. Desses dados, duas categorias são mais interessantes: as estatísticas referentes aos sorteios e as estatísticas referentes aos graus de pertinência dos estados.

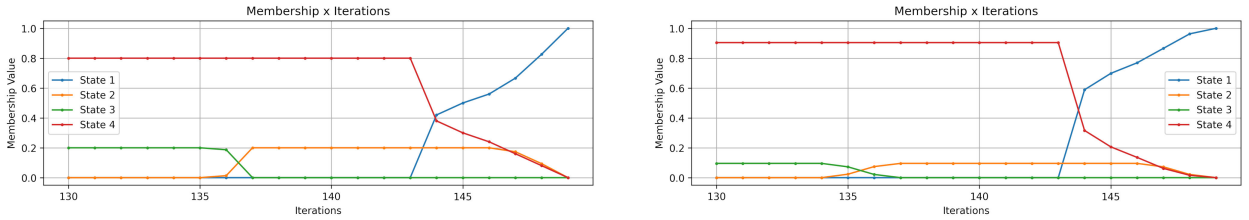


(a) Produto x Triangular.



(b) Produto x Cossenoidal.

Figura 22: Primeiras 20 iterações para os dois experimentos com o produto como F.



(a) Diferença Limitada x Triangular.

(b) Diferença Limitada x Cossenoidal.

Figura 23: Primeiras 20 iterações para os dois experimentos com o produto como F.

$F(\mu_s, \mu_e)$	δ	N^1_{sorteio}	N^2_{sorteio}	N^3_{sorteio}	N^4_{sorteio}
Produto $\mu_s \cdot \mu_e$	**Triangular**	48	30	7	65
	Cossenoidal	40	36	6	68
Diferença Limitada $\max\{0, \mu_s + \mu_e - 1\}$	**Triangular**	39	37	13	61
	Cossenoidal	37	41	8	64

Tabela 3: Quantidade de vezes que cada estado foi sorteado (N^i_{sorteio} para cada estado $i \in \{1, 2, 3, 4\}$) para as diferentes combinações de parâmetros.

$F(\mu_s, \mu_e)$	δ	$\mu^1_{\text{pertinência}}$	$\mu^2_{\text{pertinência}}$	$\mu^3_{\text{pertinência}}$	$\mu^4_{\text{pertinência}}$
Produto $\mu_s \cdot \mu_e$	**Triangular**	0,317	0,201	0,041	0,441
	Cossenoidal	0,262	0,240	0,041	0,458
Diferença Limitada $\max\{0, \mu_s + \mu_e - 1\}$	**Triangular**	0,289	0,252	0,092	0,366
	Cossenoidal	0,254	0,263	0,066	0,416

Tabela 4: Média de pertinência para cada estado ($\mu^i_{\text{pertinência}}$ para cada estado $i \in \{1, 2, 3, 4\}$) para as diferentes combinações de parâmetros.

$F(\mu_s, \mu_e)$	δ	$\sigma^1_{\text{pertinência}}$	$\sigma^2_{\text{pertinência}}$	$\sigma^3_{\text{pertinência}}$	$\sigma^4_{\text{pertinência}}$
Produto $\mu_s \cdot \mu_e$	**Triangular**	0,451	0,386	0,163	0,485
	Cossenoidal	0,425	0,412	0,161	0,486
Diferença Limitada $\max\{0, \mu_s + \mu_e - 1\}$	**Triangular**	0,420	0,364	0,161	0,377
	Cossenoidal	0,402	0,385	0,147	0,430

Tabela 5: Desvio padrão da pertinência para cada estado ($\sigma^i_{\text{pertinência}}$ para cada estado $i \in \{1, 2, 3, 4\}$) para as diferentes combinações de parâmetros.

A tabela 3 mostra a quantidade de vezes que cada estado foi sorteado para cada combinação

de parâmetros. Dela, podemos ver alguns pontos interessantes:

- Para um mesmo F, a função de base δ triangular resulta em mais sorteios para os estados 1 e 3 quando comparada à δ cossenoidal. Por outro lado, esta apresenta mais sorteios para os estados 2 e 4.
- Para um mesmo δ, o produto oferece mais sorteios para os estados 1 e 4, enquanto a diferença limitada fornece mais sorteios para os estados 2 e 3, comparativamente.
- Independentemente da combinação, há uma prevalência de sorteios para o estado 4. Os estados 1 e 2 também são amplamente sorteados. Porém, o estado 3 é sorteado, em média, em menos de 10% das iterações.

Os dados dos sorteios estão diretamente relacionados aos dados de pertinência dos estados. A tabela 4 mostra as médias dos graus de pertinência dos estados, levando em conta todas as iterações. Podemos notar que há uma relação direta entre as médias e os estados com maior número de sorteios. Para todas as combinações, o estado 4 apresentou a maior média, e o estado 3, a menor. No caso dos estados 1 e 2, quando a média da pertinência de 1 foi maior, ele também foi mais sorteado do que 2.

Para todos os casos, o desvio padrão da pertinência de todos os estados foi alto, geralmente tendo valor absoluto maior que a própria média. Isso indica dados com grande variação, o que é explicado pelo fato que os estados costumam aumentar significativamente e depois diminuir rapidamente ao longo dos experimentos. Nota-se, porém, que os desvios padrão são menores quando se usa a diferença limitada do que quando se utiliza o produto. Podemos relacionar a isso ao fato que as pertinências no caso da diferença limitada não se anulam em vários casos, de modo que a variação total da pertinência de cada estado ao longo do experimento é menor.

Um ponto interessante é que a média de pertinência dos estados, quando multiplicada pelo número de iterações, resulta em valores próximos ao número de vezes que o estado foi sorteado. Por exemplo, para o caso do produto com funções triangulares, para o estado 4 temos $150 \cdot 0.441 = 66.15$, o que é próximo de 66, o número de sorteios obtido empiricamente. Isso ocorre porque usamos as pertinências como probabilidades e, por consequência, a média exprime uma "probabilidade global"de escolha do estado. No entanto, a média não considera as limitações locais do modelo, como transferências de pertinências, que afetam a evolução e as probabilidades para cada iteração.

Um último aspecto a ser citado é que todos os estados apresentaram pertinência positiva em ao menos uma iteração e foram sorteados pelo menos uma vez. Essa informação é útil na validação do modelo, pois mostra que todos os estados têm condições de receber pertinência e realizar suas ações. Caso isso não ocorresse, é possível que alguma transição não estivesse sendo ativada do modo pretendido, e o modelo não seria um agente interessante no contexto de jogos.

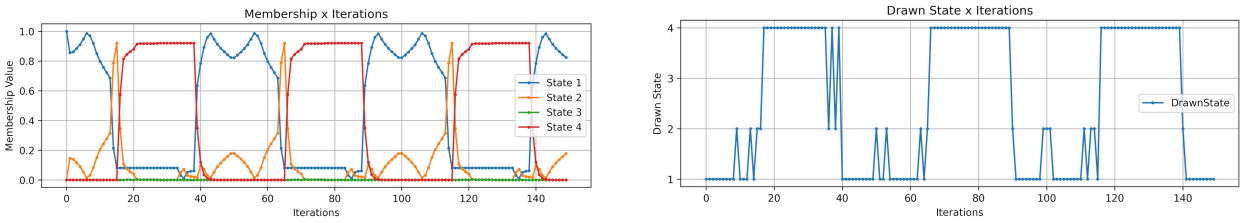
Todas as análises nos permitem ver diferentes casos de uso para os diferentes valores de parâmetros:

- Usar o produto permite ter um agente mais previsível. Caso ele precise ser reativo, podemos usar δ cossenoidal. Caso contrário, podemos usar as funções triangulares.
- No caso de se desejar um agente mais imprevisível, a diferença limitada é recomendada, pois mantém diversos estados ativos para várias iterações, possibilitando sorteios de vários estados. No caso em que a imprevisibilidade precise ser atenuada, pode-se aplicar δ cossenoidal.

Com isso, podemos ver que podemos usar o mesmo modelo para representar agentes distintos. No contexto de um jogo virtual, isso traria mais imprevisibilidade ao comportamento dos inimigos e um maior desafio ao jogador. Por outro lado, diferentes tipos de oponentes poderiam usar o mesmo modelo (implementado, assim, apenas uma vez) e variar apenas os parâmetros, facilitando o desenvolvimento desses agentes.

5.3 Comparativo com os Parâmetros Usados por Santos

Nesta subseção, apresentamos os resultados dos experimentos quando os valores de parâmetros usados por G. L. Santos em seu trabalho, que são o **produto** como F e a **razão** como δ. Isso permite observar como o uso de funções numéricas como pertinência diverge do uso de funções *fuzzy* e mostrar como os resultados deste trabalho e os de Santos podem ser comparados. Os resultados estão dispostos na figura 24.



(a) Progressão dos graus de pertinência.

(b) Estados sorteados.

Figura 24: Resultados quando se usa a razão como δ e o produto como F.

Comparando os resultados, notamos diferenças consideráveis nos estados ativos. Para δ razão, tivemos mais de um estado ativo pela maioria da simulação, o que não ocorria quando aplicamos δ triangular ou cossenoidal. Consequentemente, a sequência dos estados sorteados é mais imprevisível, de modo que há vários momentos em que uma longa sequência de sorteios repetidos é interrompida pela escolha de outro estado ativo não-dominante (como entre as iterações 0 e 20).

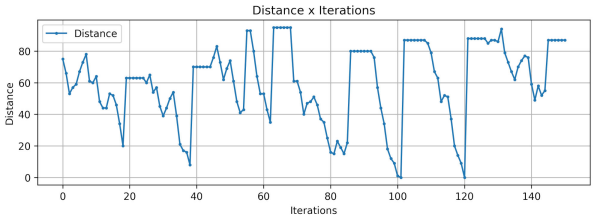
Porém, um ponto importante é que o estado 3 ficou com pertinência nula ao longo da simulação. Logo, nunca foi sorteado. Podemos analisar isso com base nos pesos numéricos associados a cada transição. A única transição que leva ao estado 3 requer que a distância seja aproximadamente 25. Para os nossos dados, a saúde nesse ponto é cerca de 20, o que ativa também as transições que levam ao estado 4. Assim, toda a pertinência que iria para o estado 3 acaba sendo redirecionada para o estado 4.

Assim, para os dados gerados, a combinação de parâmetros proposta por Santos não gerou um agente capaz de chegar em todos os estados. No entanto, a razão introduziu maior incerteza nas ações do modelo, podendo ser útil para gerar mais nuance quando se utiliza o produto como função de transferência.

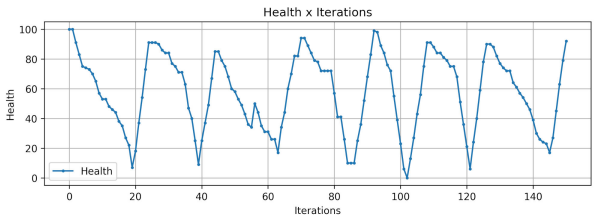
5.4 Modo de Operação com Variação Aleatória dos Dados

Apesar de o foco do trabalho ser a análise de simulações com dados gerados deterministicamente, trazemos nesta subseção um resultado obtido quando o modelo calcula seus próprios dados com base no estado sorteado. Essa análise também deriva do trabalho de Santos. Porém, enquanto Santos simula o modelo com o sistema do jogo plenamente implementado, nós usamos ações simuladas, as quais tentam variar a distância e a saúde de modo aleatório, mas que faça sentido dada a ação realizada.

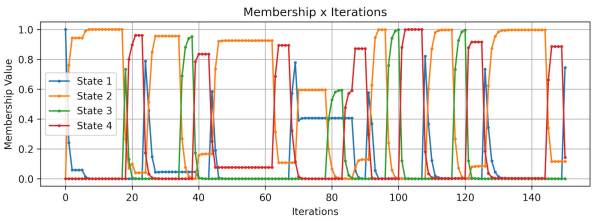
Para esse modo de operação, analisaremos os dados obtidos ao usar F **produto** e δ **triangular** e utilizaremos como distância e saúde iniciais os valores 75 e 100, respectivamente. Para as demais iterações, os valores das variáveis são alterados com base na ação sorteada, o que envolve, no geral, a soma ou subtração de um número aleatório. Os dados de distância e saúde gerados pela execução podem ser vistos nas figuras 25a e 25b, respectivamente. Já a evolução da pertinência e os resultados dos sorteios estão dispostos nas figuras 25c e 25d, respectivamente.



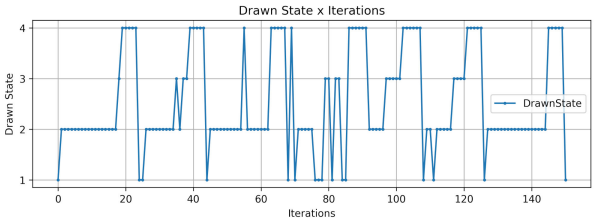
(a) Distância.



(b) Saúde.



(c) Evolução das pertinências.



(d) Estados sorteados.

Figura 25: Resultados da simulação com dados gerados aleatoriamente pela máquina para F produto e δ triangular.

N^1_{sorteio}	N^2_{sorteio}	N^3_{sorteio}	N^4_{sorteio}
16	79	17	39

Tabela 6: Quantidade de vezes que cada estado foi sorteado na geração aleatória.

$\mu^1_{\text{pertinência}}$	$\mu^2_{\text{pertinência}}$	$\mu^3_{\text{pertinência}}$	$\mu^4_{\text{pertinência}}$
0,112	0,545	0,099	0,244

Tabela 7: Médias de pertinência para os estados na geração aleatória.

$\sigma^1_{\text{pertinência}}$	$\sigma^2_{\text{pertinência}}$	$\sigma^3_{\text{pertinência}}$	$\sigma^4_{\text{pertinência}}$
0,21	0,423	0,263	0,374

Tabela 8: Desvios padrão da pertinência para os estados na geração aleatória.

Os dados apresentados permitem notar alguns pontos importantes sobre o experimento:

- Há uma prevalência do estado 2 em contraste com a prevalência do estado 4 nos experimentos anteriores. Isso pode ser explicado pelo modo em que a ação do estado 4 é realizada. Nela, a cura do agente implica na obtenção de uma quantidade de saúde suficiente para que, em poucas iterações, tenha saúde alta e tenda a transferir pertinência ao estado 1. Como o estado 4 perde prevalência, o estado 2 ganha espaço.
- Em vários pontos temos mais de um estado ativo por várias iterações seguidas. Por consequência, vemos que há vários pontos em que há alternância rápida entre estados sorteados.

Isso gera um comportamento mais imprevisível, o que não pôde ser visto quando usamos dados determinísticos.

- Mesmo com os dados aleatórios, o estado 3 continua sendo o estado com a menor média de pertinência. Porém, o estado 1 apresentou baixa pertinência média também, e foi, neste caso, o estado menos sorteado.

Analisar a evolução do modelo sob entradas menos previsíveis permite analisar como o modelo reage a entradas erráticas. Isso nos permite estudar casos que os dados determinísticos não mostram, enriquecendo os resultados. Apesar disso, esse modo de operação dificulta a análise da influência direta de parâmetro. Como os dados são majoritariamente aleatórios e dependem diretamente da combinação dos parâmetros em questão, analisar o impacto direto de cada opção de parâmetro torna-se mais difícil. Mesmo assim, esse modo ainda traz pontos interessantes e pode auxiliar na escolha dos valores dos parâmetros.

5.5 Casos Pouco Interessantes

Dentre todas as combinações de opções de parâmetros analisadas, algumas trouxeram resultados pouco interessantes no contexto de FuSMs. Notavelmente, a função de transferência **máximo** fez com que a máquina atuasse como uma FSM. Isso ocorre pois as pertinências dos estados são sempre 0 ou 1, de modo que se tornam valores *crisp* novamente. A evolução da pertinência pode ser vista na figura 26.

Apesar de ser uma ponte direta entre FSMs e FuSMs, combinações que usam o máximo não oferecem mais do que uma FSM no contexto de agentes virtuais. Logo, elas foram desconsideradas para análises mais profundas neste relatório.

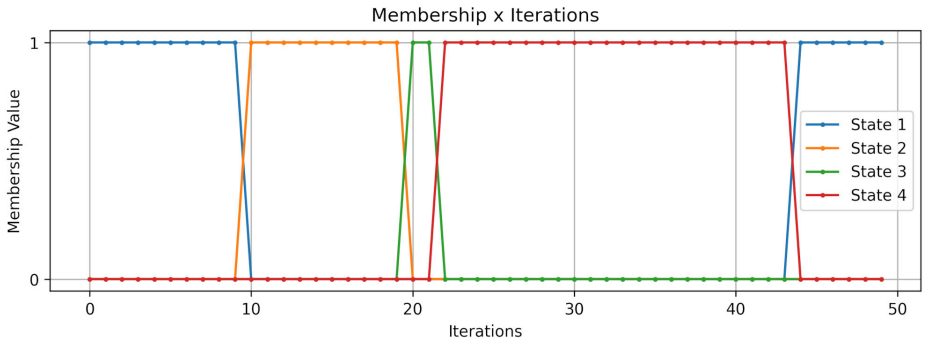


Figura 26: Evolução dos graus de pertinência dos estados para F máximo e δ triangular para as primeiras 50 iterações da simulação.

6 Conclusões e Trabalhos Futuros

6.1 O Que Foi Feito

O trabalho abordou vários aspectos da simulação das máquinas de estado *fuzzy*. Primeiro, vimos o que são FuSMs e suas aplicações na modelagem de sistemas, e como podem ser usadas para trazer incertezas aos modelos. Mostramos que, apesar dos casos de uso interessantes, dois grandes desafios impactam a adoção das FuSMs: a escolha da função de transferência do modelo, F, e a escolha das funções de pertinência das variáveis de contexto, representadas por δ.

Nesse contexto, o trabalho estudou a fundo como a simulação pode ser utilizada para mitigar esses desafios. Para isso, introduzimos formalmente as FuSMs, exibindo características como a multiplicidade de estados ativos e o processo de evolução, o qual consiste na transferência de pertinência entre estados com base nas funções F e δ a cada iteração. Como existem diversas opções para essas funções, propusemos a simulação como um modo de auxiliar na escolha, facilitando a adoção das FuSMs.

A simulação foi construída com base no trabalho de G. L. Santos, utilizando um modelo de agente de jogos virtuais para a simulação. O modelo foi codificado de modo a ser testado com várias opções de F e δ, e um sistema auxiliar foi construído para gerir as diferentes combinações e extrair dados para a análise. Pelos dados, foi possível ver como cada parâmetro altera a evolução do modelo, tornando-o mais ou menos reativo a alterações das variáveis de contexto ou introduzindo maior ou menor incerteza às ações realizadas. Com isso, é possível escolher a combinação que melhor atenda aos requisitos do modelo.

Assim, podemos afirmar que o trabalho alcançou os objetivos iniciais, facilitando a escolha de F e δ. O método proposto oferece uma solução simples, customizável e expansível para a simulação, tornando a adoção de máquinas de estado *fuzzy* mais acessível para situações reais.

6.2 Trabalhos Futuros

Para trabalhos futuros, existem alguns pontos que podem ser aprimorados ou explorados. Notavelmente, pode-se ampliar o estudo da simulação das ações dos agentes, as quais são usadas quando o modelo calcula variações de distância e saúde. No nosso caso, utilizamos funções com cálculos simples e aleatórios. Um refinamento dessas funções poderia gerar resultados mais interessantes e fiéis ao caso de uso, proporcionando maior embasamento para a tomada de decisões a respeito do modelo.

Referências

[1] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools.* Addison-Wesley Longman Publishing Co., Inc., USA, 1999.

[2] A. Blanco, M. Delgado, and M.C. Pegalajar. Fuzzy automaton induction using neural network. *International Journal of Approximate Reasoning*, 27(1):1–26, 2001.

[3] David Brubaker. A fuzzy state machine model. *The Huntington Technical Brief*, 3(36), 03 1993. [Online em `https://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/fuzzy/doc/notes/brub/brub9.txt`; Acessado 05/10/2025.].

[4] Mansoor Doostfatemeh and Stefan C. Kremer. New directions in fuzzy automata. *International Journal of Approximate Reasoning*, 38(2):175–214, 2005.

[5] Gilliard Lopes dos Santos. Máquinas de estados hierárquicas em jogos eletrônicos. Master's thesis, Pontíficia Universidade Católica do Rio de Janeiro - PUC-RIO, 2004.

[6] Sarah L. Harris and David Harris. 3 - sequential logic design. In Sarah L. Harris and David Harris, editors, *Digital Design and Computer Architecture*, pages 106–169. Morgan Kaufmann, 2022.

[7] George J. Klir and Bo Yuan. *Fuzzy sets and fuzzy logic: theory and applications.* Prentice-Hall, Inc., USA, 1994.

[8] Zhihui Li, Ping Li, and Yongming Li. The relationships among several types of fuzzy automata. *Information Sciences*, 176(15):2208–2226, 2006.

[9] Yue Ma. Chapter eight - motion stability enhanced controller design. In Yue Ma, editor, *Dynamics and Advanced Motion Control of Off-Road UGVs*, Emerging Methodologies and Applications in Modelling, Identification and Control, pages 191–218. Academic Press, 2020.

[10] Gadelhag Mohmed, Ahmad Lotfi, and Amir Pourabdollah. Enhanced fuzzy finite state machine for human activity modelling and recognition. *Journal of Ambient Intelligence and Humanized Computing*, 11(12):6077–6091, Dec 2020.

[11] Object Management Group (OMG). OMG Unified Modeling Language (OMG UML). Technical report, Object Management Group (OMG), Dec 2017. Version 2.5.1. Disponível em `https://www.omg.org/spec/UML/`. Acessado em 01/11/2025.

[12] Michele Pirovano. The use of fuzzy logic for artificial intelligence in games. *University of Milano, Milano*, 2012.

[13] Steve Rabin. Implementing a state machine language. *AI Game Programming Wisdom*, pages 314–320, 2002.

[14] Juan Rada-Vilela. fuzzylite: a fuzzy logic control library, 2017.

[15] L.M. Reyneri. An introduction to fuzzy state automata. In *Lecture Notes in Computer Science*, volume 1240, pages 273–283, 06 1997.

[16] M. Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2012.

[17] Doru Todinca, Ioana Sora, Daniel-Eugen Butoianu, and Radu-Emil Precup. A novel method to compute the membership value of the states of fuzzy automata. In *2018 IEEE 12th International Symposium on Applied Computational Intelligence and Informatics (SACI)*, pages 000107–000112, 2018.

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

[19] L.A. Zadeh. The concept of a linguistic variable and its application to approximate reasoning—i. *Information Sciences*, 8(3):199–249, 1975.