


Testes Baseados em Máquinas de Estados Finitos

Prof. Dr. Valério Gutemberg

 Discente: Alessandro, Guilherme Cadete, Moisés, Rafael Augusto, Samuel Lucas

 Disciplina de Testes de Software

 Curso de Sistemas para Internet

Introdução

O teste de software é uma atividade essencial para assegurar a qualidade de um sistema, identificando falhas o mais cedo possível no ciclo de desenvolvimento. Como testar todas as possíveis execuções de um software é inviável, técnicas e critérios específicos são necessários para selecionar casos de teste com alta probabilidade de revelar falhas, garantindo a viabilidade prática dos testes.

Introdução

Uma técnica amplamente adotada na indústria é o Teste Baseado em Modelos, que utiliza modelos comportamentais derivados dos requisitos funcionais do software para orientar o processo de teste. Entre as diversas abordagens, as Máquinas de Estados Finitos(Autômatos Finitos) se destacam por sua eficácia em modelar sistemas reativos e controlados por eventos, além de sua ampla aplicabilidade em diferentes tipos de sistemas.



O que é uma Máquinas de Estados Finitos (FSM)?

- **FSM:** É um modelo matemático composto por um número finito de estados, transições entre esses estados, e ações que podem ocorrer em cada estado. Significa, em resumo, que é uma máquina que só pode ter um estado por vez e precisa de uma forma de transicionar entre estados.



Como Funcionam os Testes Baseados em FSMs?

- **Modelagem do Sistema:** sob teste como uma FSM, definindo os estados, transições, eventos, entradas e saídas.
- **Definição dos Testes:** Com a FSM definida, elaboram-se casos de teste para cobrir diversos cenários e transições entre estados, verificando se o sistema responde corretamente a entradas e eventos em cada estado.
- **Execução dos Testes:** Os testes são então executados, e o comportamento real do sistema é comparado com o comportamento esperado baseado na FSM.
- **Análise dos Resultados:** Caso haja divergências entre o comportamento esperado e o observado, o sistema é analisado para identificar e corrigir possíveis falhas.



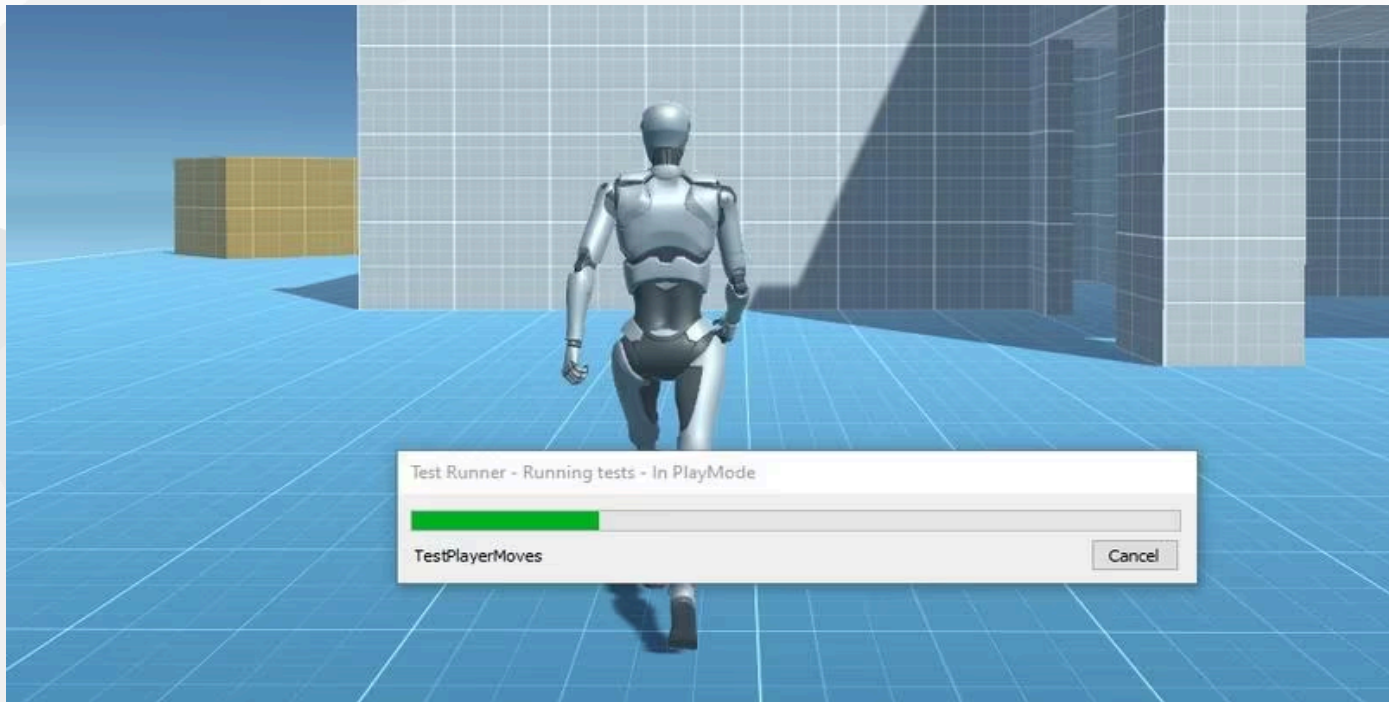
Finite State Machine em Jogos Eletrônicos

As FSMs desempenham um papel crucial no desenvolvimento e teste de jogos eletrônicos. Elas não apenas ajudam a estruturar o comportamento dos personagens e a lógica do jogo, mas também oferecem uma base sólida para realizar testes eficientes e abrangentes. Este enfoque nas MEFs permite uma abordagem sistemática para verificar se os diferentes estados e transições estão funcionando corretamente, garantindo uma experiência de jogo robusta e livre de falhas.

Teste de Comportamento de Personagens

Nos jogos eletrônicos, personagens podem ter diversos estados, como "parado", "andando", "correndo", "pulando" e "atacando". Utilizar FSMs para modelar esses estados possibilita a criação de casos de teste focados em transições entre esses estados. Por exemplo, os testes podem verificar se a transição de "parado" para "andando" ocorre corretamente quando uma tecla é pressionada. Este tipo de teste ajuda a garantir que o comportamento do personagem esteja conforme o esperado em diferentes cenários de jogo.

Figura 1 - Exemplo de Teste de Comportamento de Personagens



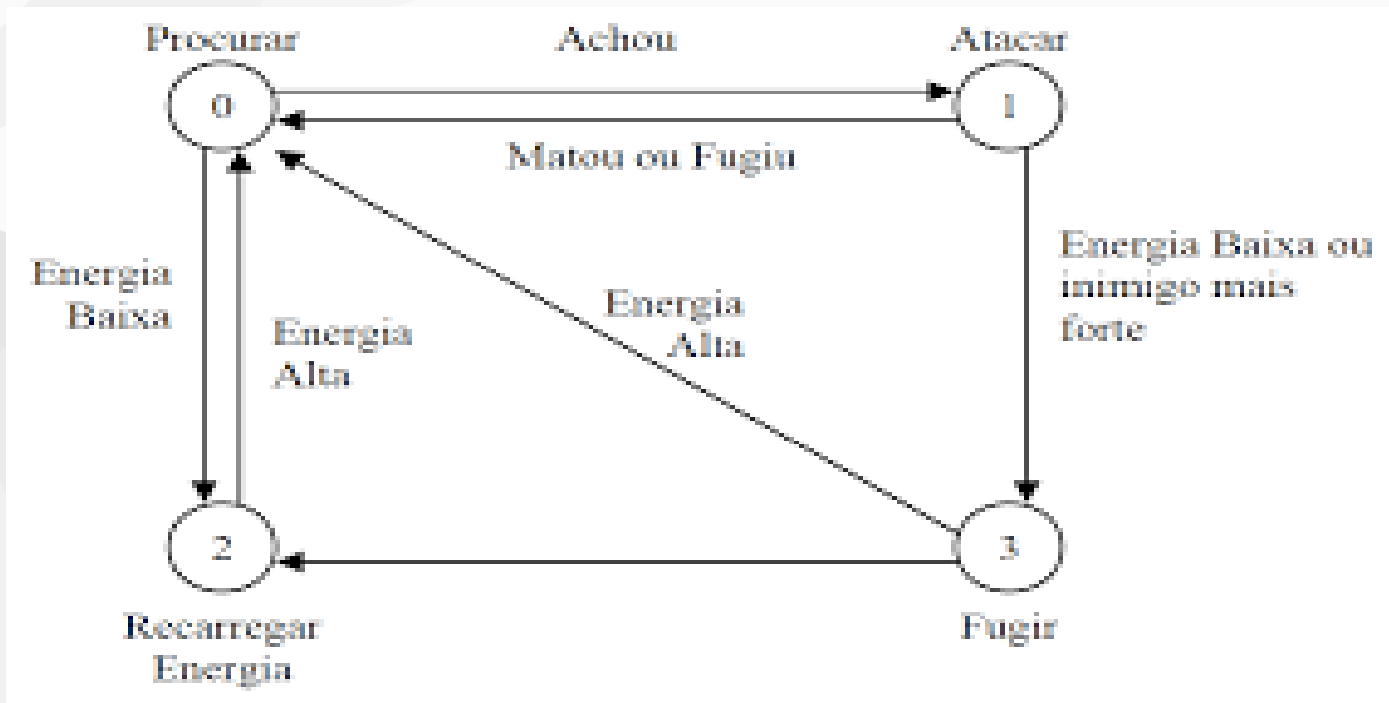
Fonte: Unity, 2022



Teste da Inteligência Artificial (IA)

A IA dos inimigos e NPCs em jogos pode ser complexa, com estados como "patrulhando", "perseguindo", "atacando" e "fugindo". Testar essas transições usando FSMs permite verificar se a IA responde adequadamente às mudanças nas condições de jogo, como a aproximação do jogador ou alterações na saúde do inimigo. Os testes podem incluir simulações de diferentes cenários para assegurar que a IA se comporte de maneira previsível e eficiente em diversas situações.

Figura 2 - Exemplo de teste de IA



Fonte: Augusto Baffa, 2016

Teste do Fluxo de Jogo

No gerenciamento do fluxo geral do jogo, estados como "início", "jogando", "pausado", "game over" e "vencedor" podem ser modelados usando FSMs. Testar essas transições ajuda a garantir que o jogo passe corretamente de um estado para outro e que as regras e condições associadas a cada estado sejam seguidas rigorosamente. Por exemplo, os testes podem verificar se o jogo corretamente entra no estado "game over" quando o jogador perde, e se ele pode ser reiniciado sem problemas.



Benefícios e Razões para Usar Testes Baseados em FSMs

- **Cobertura Completa:** Testes baseados em FSMs garantem a exploração de todos os cenários e transições do sistema, assegurando que todas as funcionalidades sejam testadas e que o sistema funcione corretamente em todos os possíveis estados.
- **Detecção Precoce de Defeitos:** A abordagem sistemática das FSMs facilita a identificação de falhas nas fases iniciais do desenvolvimento, permitindo a correção de problemas antes que se tornem mais complexos e difíceis de resolver.

Benefícios e Razões para Usar Testes Baseados em FSMs

- **Melhoria da Qualidade:** Ao testar todas as transições e estados, a metodologia baseada em FSMs contribui para a criação de um software mais confiável e robusto, melhorando a qualidade geral do produto.
- **Eficiência e Automação:** A estrutura modular das FSMs simplifica a criação e execução de casos de teste, economizando tempo e recursos. Além disso, facilita a automação dos testes, permitindo verificações contínuas e repetidas à medida que o sistema evolui.

Figura 3 - Exemplo de código simples

```
codigo > portao.py > ...
1 class FSM:
2     def __init__(self, estados, acoes, transicoes, estado_inicial):
3         self.estados = estados
4         self.acoes = acoes
5         self.transicoes = transicoes
6         self.estado_atual = estado_inicial
7
8     def transition(self, estado, acao):
9         if (estado, acao) in self.transicoes:
10             return self.transicoes[(estado, acao)]
11         return estado
12
13
14 fechado = 'Fechado'
15 aberto = 'Aberto'
16 trancado = 'Trancado'
17 em_uso = 'Em Uso'
18
19 eventos = ['inserir_chave', 'girar_chave', 'abrir_portao', 'fechar_portao', 'trancar_portao', 'destrancar_portao']
20
21 estado_inicial = fechado
22
23 def inserir_chave(estado):
24     if estado == fechado:
25         return 'Chave Inserida'
26     return estado
27
```

Fonte: Autoria própria, 2024

Figura 4 - Exemplo de código simples

```
codigo > portao.py > ...
28 def girar_chave(estado):
31     elif estado == aberto:
32         return trancado
33     elif estado == trancado:
34         return aberto
35     return estado
36
37 def abrir_portao(estado):
38     if estado == aberto:
39         return em_uso
40     return estado
41
42 def fechar_portao(estado):
43     if estado == em_uso:
44         return aberto
45     return estado
46
47 def trancar_portao(estado):
48     if estado == aberto:
49         return trancado
50     return estado
51
52 def destrancar_portao(estado):
53     if estado == trancado:
54         return aberto
55     return estado
56
57 estados = {fechado, aberto, trancado, em_uso, 'Chave Inserida'}
58 acoes = {inserir_chave, girar_chave, abrir_portao, fechar_portao, trancar_portao, destrancar_portao}
59
```

Fonte: Autoria própria, 2024

Figura 5 - Exemplo de código simples

```
codigo > portao.py > ...
60 transicoes = {
61     (fechado, inserir_chave): 'Chave Inserida',
62     ('Chave Inserida', girar_chave): aberto,
63     (aberto, girar_chave): trancado,
64     (trancado, girar_chave): aberto,
65     (aberto, abrir_portao): em_uso,
66     (em_uso, fechar_portao): aberto,
67     (aberto, trancar_portao): trancado,
68     (trancado, destrancar_portao): aberto,
69 }
70
71 modelo = FSM(estados, acoes, transicoes, estado_inicial)
72
73 def simular_fsm(modelo, eventos):
74     estado_atual = modelo.estado_atual
75     print(f"Estado inicial: {estado_atual}")
76     for evento in eventos:
77         acao = globals()[evento]
78         estado_novo = modelo.transition(estado_atual, acao)
79         print(f"Evento: {evento} | Estado atual: {estado_atual} -> Novo estado: {estado_novo}")
80         estado_atual = estado_novo
81
82 simular_fsm(modelo, ['inserir_chave', 'girar_chave', 'abrir_portao', 'fechar_portao', 'trancar_portao', 'destrancar_portao'])
83
```

Fonte: Autoria própria, 2024

Figura 6 - Exemplo de teste

```
codigo > teste.py > ...
1 from portao import FSM, inserir_chave, girar_chave, abrir_portao, fechar_portao, trancar_portao, destrancar_portao
2 import portao
3
4 estado_inicial = portao.fechado
5
6 def teste_fsm():
7     estado = estado_inicial
8     estado = inserir_chave(estado)
9     assert estado == 'Chave Inserida', f"Erro: O portão deveria estar em 'Chave Inserida' ao inserir a chave, mas está {estado}."
10
11     estado = girar_chave(estado)
12     assert estado == portao.aberto, f"Erro: O portão deveria estar aberto após girar a chave, mas está {estado}."
13
14     estado = abrir_portao(estado)
15     assert estado == portao.em_uso, f"Erro: O portão deveria estar em uso após abrir o portão, mas está {estado}."
16
17     estado = fechar_portao(estado)
18     assert estado == portao.aberto, f"Erro: O portão deveria estar aberto após fechar o portão, mas está {estado}."
19
20     estado = girar_chave(estado)
21     assert estado == portao.trancado, f"Erro: O portão deveria estar trancado após girar a chave, mas está {estado}."
22
23     estado = destrancar_portao(estado)
24     assert estado == portao.aberto, f"Erro: O portão deveria estar aberto após destrancar, mas está {estado}."
25
26     estado = girar_chave(estado)
27     assert estado == portao.trancado, f"Erro: O portão deveria estar trancado após girar a chave novamente, mas está {estado}."
28
29     estado = estado_inicial
30     estado = abrir_portao(estado)
31     assert estado == portao.fechado, f"Erro: O portão deveria permanecer fechado ao tentar abri-lo sem inserir a chave, mas está {estado}."
```

Fonte: Autoria própria, 2024

Figura 7 - Exemplo de teste

```
codigo > teste.py > teste_fsm
6 def teste_fsm():
33     estado = estado_inicial
34     estado = inserir_chave(estado)
35     assert estado == 'Chave Inserida', f"Erro: O portão deveria estar em 'Chave Inserida', mas está {estado}."
36
37     estado = girar_chave(estado)
38     assert estado == portao.aberto, f"Erro: O portão deveria estar aberto após girar a chave, mas está {estado}."
39
40     estado = girar_chave(estado)
41     assert estado == portao.trancado, f"Erro: O portão deveria estar trancado após girar a chave novamente, mas está {estado}."
42
43     estado = destrancar_portao(estado)
44     assert estado == portao.aberto, f"Erro: O portão deveria estar aberto após destrancar o portão, mas está {estado}."
45
46     estado = estado_inicial
47     estado = inserir_chave(estado)
48     estado = girar_chave(estado)
49     estado = girar_chave(estado)
50
51     estado = abrir_portao(estado)
52     assert estado == portao.trancado, f"Erro: O portão deveria permanecer trancado ao tentar abri-lo enquanto trancado, mas está {estado}."
53
54     print("Todos os testes foram concluídos com sucesso!")
55
56     teste_fsm()
57
```

Fonte: Autoria própria, 2024



Explicação

- Nos códigos mostrados, simulamos um FSM (Finite State Machine) para simular a abertura e fechamento de um portão.
- **aberto** e **fechado** são os estados que o portão pode assumir. Nunca pode estar aberto e fechado ao mesmo tempo.
- **inserir_chave**, **abrir_portao** e **fechar_portao** são transições de estado.



Tecnologias de FSM

PyModel (python)

A biblioteca que utilizamos é uma alternativa simples para montar testes e criar máquinas de estado finito: precisamos apenas informar os estados, as transições e as ações(que executam a transição).

Outras alternativas são o [TestFSM \(com Rust\)](#) e o [GraphWalker](#), ambos para teste



Conclusão

e. A FSM é uma poderosa ferramenta para modelar sistemas que têm um conjunto finito de estados e um conjunto definido de transições entre esses estados. Exemplos de usos são semáforos, portões e situações parecidas



Referências

- PUC. Máquinas de Estados em Jogos Eletrônicos. Disponível em:
https://www.maxwell.vrac.puc-rio.br/4711/4711_3.PDF. Acesso em: 29 ago. 2024.
- FANTINATO, Marcelo. Teste de Software Baseado em Máquina de Estado Finito - Uma Revisão. Disponível em:
https://sedici.unlp.edu.ar/bitstream/handle/10915/23121/Art_C165.pdf;jsessionid=72A87F4CF39ABC95A8403629E01AB409?sequence=1. Acesso em: 29 ago. 2024.

Referências

- UNITY, Automated Tests Unity Test Framework. Disponível em: <https://unity.com/pt/how-to/automated-tests-unity-test-framework>. Acesso em: 29 ago. 2024.
- BAFFA, Augusto. INF 1771 – Inteligência Artificial. Rio de Janeiro: Vision Lab, 2016. 24 slides, color, 254 × 190 mm.



INSTITUTO FEDERAL
Rio Grande do Norte

Obrigado!

Dúvidas?