Máquinas de estados finitos

Autor: Wesley Matteus Araújo dos Santos

Introdução

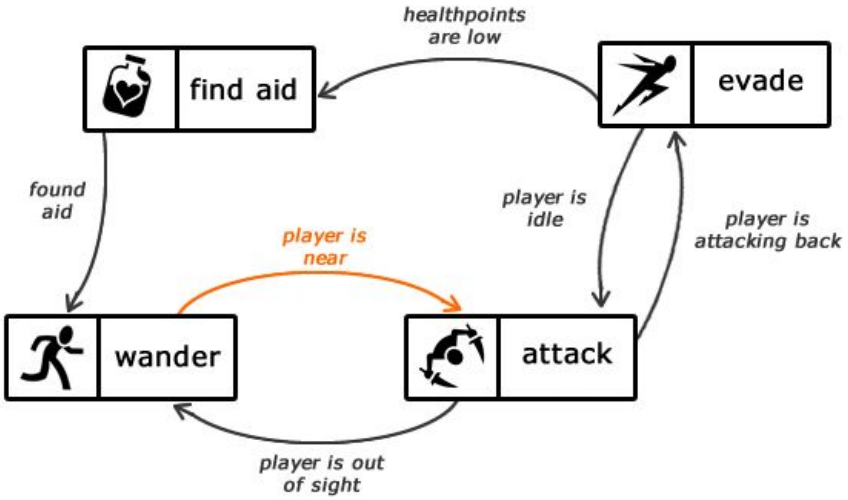
Com o objetivo de entender mais sobre esse conceito importantíssimo na computação foram escolhidas duas abordagens para o seu aprendizado que irão se complementar, uma **informal** e uma mais **formal**.

Abordagem Informal

***Se faça uma pergunta*:** *como será que era simulada a inteligência artificial dos inimigos em um jogo virtual antes dos avanços modernos em redes neurais e aprendizado de máquina?* Bem, se você já jogou algum desses jogos sabe que os inimigos normalmente respondem a ações dos jogadores ou dos seus próprios status (saúde baixa, por exemplo).

Considerado isso, como nós programadores simulamos esse tipo de comportamento condicionado? Isso mesmo! Com expressões condicionais que respondem ao estado atual desse inimigo. Para deixar mais claro o que está sendo dito, considere a seguinte imagem:

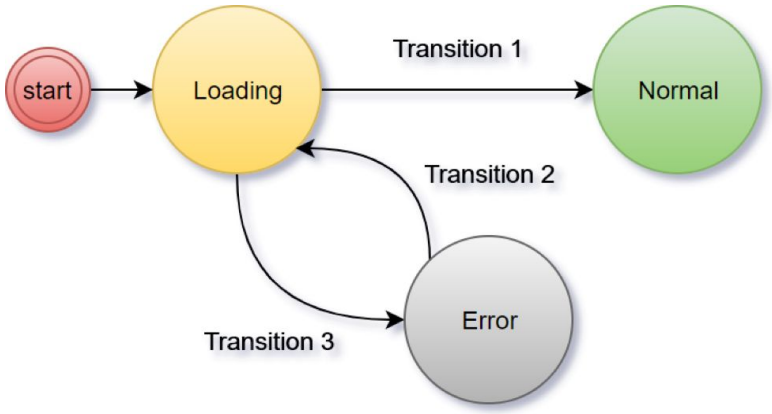
Fonte: Envato tuts +



FSM representando o cérebro de um inimigo.

De forma concisa, esse diagrama nos diz que: o inimigo **vagueia** pelo mapa até que **o jogador esteja próximo,** se isso acontecer então ele passa ao **estado de ataque** e provavelmente tentar diminuir a distância entre eles, se o jogador sair da vista do inimigo então ele volta a **vagar.** No caso do jogador atacar de volta ele deve **evadir** as investidas e voltar a atacar **se o jogador não estiver reagindo.** Caso ele esteja evadindo e seus pontos de vida estejam baixos então é hora de ele tentar encontrar alguma poção de cura ou algo equivalente e voltar a **vagar** pelo mapa.

Com isso dito, esse é um exemplo simples de como um inimigo de um desses jogos poderia se comportar, entretanto esse estilo de diagrama de transições e estados é muito mais generalista e pode ajudar a simular praticamente o comportamento de qualquer sistema computacional. Não acredita? Pois veja o próximo diagrama, ele provavelmente soa bem familiar a você:



FSM of a running task

Suspeito que esses estados sejam bem costumeiros ao leitor, não importa o sistema, sempre há uma chance de erro na execução de uma tarefa (seja ela carregar um arquivo, receber dados do usuário por uma caixa de texto, realizar um cálculo, entre outras). Podemos chegar a um extremo de enxergarmos e modelarmos todas as nossas interações a partir desses diagramas e essa abstração é usada por nós a todo momento quando pensamos em um problema (mesmo sem percebermos). Dito isso, iremos mostrar um pouco da história desse conceito e tentaremos formalizar mais nossa ideia de FSM (Finite State Machine).

Fonte: **https://gamedevelopment.tutsplus.com/tutorials/finite-state-machines-theory-and-implementation--gamedev-11867**

Um pouco de história

As primeiras pessoas a considerar o conceito de máquina de estados finitos incluíram um time de biólogos, psicólogos, matemáticos, engenheiros e alguns dos primeiros cientistas da computação. Eles todos compartilhavam um interesse em comum: modelar o processo de pensamento humano, no cérebro ou em um computador. Warren McCulloch e Walter Pitts, dois neurofisiologistas, foram os primeiros a apresentar a descrição de um autômato finito em 1943. Seu artigo, intitulado, “Um cálculo lógico imanente à atividade nervosa”, fez contribuições significativas para o estudo da teoria das redes neurais, teoria dos autômatos e teoria da computação. Mais tarde, dois cientistas da computação, G.H Mealy e E.F. Moore, generalizaram a teoria para máquinas muito mais poderosas em diferentes artigos, publicados em 1955-56.

Fonte: **https://cs.stanford.edu/people/eroberts/courses/soco/projects/2004-05/automata-theory/basics.html**

Abordagem mais formal

As FSMs são máquinas abstratas, consistindo em um conjunto de estados (**conjunto Q**), conjunto de eventos de entrada (**conjunto I**), um conjunto de eventos de saída (**conjunto Z**) e uma função de transição de estado. A função de transição de estado pega o estado atual e um evento de entrada e retorna o novo conjunto de eventos de saída e o próximo estado. Portanto, pode ser vista como uma função que mapeia uma sequência ordenada de eventos de entrada em uma sequência correspondente, ou conjunto, de eventos de saída.

Função de transição de estado: I → Z

Máquinas de estados finitos são modelos de computação ideais para uma pequena quantidade de memória e não mantêm memória do histórico de todos os estados que já se passaram. Esse modelo matemático de uma máquina pode atingir apenas um número finito de estados e transições entre esses estados. Sua principal aplicação é na análise matemática de problemas. Essas máquinas também são usadas para propósitos além dos cálculos gerais, como o reconhecimento de linguagens regulares.

Para entender conceitualmente uma máquina de estados finitos, considere uma analogia com um elevador:

Um elevador é um mecanismo que **não lembra todas as solicitações de serviço anteriores**, mas o piso atual, a direção do movimento (para cima ou para baixo) e a coleção de solicitações de serviços ainda não atendidas. Portanto, a qualquer momento, um elevador operado seria definido pelos seguintes termos matemáticos:

- **Estados:** conjunto finito de estados para refletir o histórico passado das solicitações dos clientes.
- **Entradas:** conjunto finito de entradas, dependendo do número de andares que o elevador pode acessar. Podemos usar o conjunto I, cujo tamanho é o número de andares no edifício.
- **Saídas:** conjunto finito de saídas, dependendo da necessidade do elevador subir ou descer, de acordo com as necessidades dos clientes.

Fonte: **https://cs.stanford.edu/people/eroberts/courses/soco/projects/2004-05/automata-theory/basics.html**

Cenário da implementação com SystemVerilog:

Dado que já vimos uma breve introdução sobre o conceito e além disso sabemos um pouco da história e temos em mãos um formalismo mais rigoroso acerca do modelo, chegou a hora de vermos como esses autômatos podem ser implementados na nossa ferramenta de descrição de circuitos, o SystemVerilog, mas primeiro iremos entender um pouco melhor a implementação em alto nível.

As máquinas de estados finitos podem ser implementadas com circuitos sequenciais[1] síncronos[2] que podem assumir $\mathbf{2^k}$ estados únicos, dado que possuem k registradores de um bit (flip-flops, latches, entre outros). Uma FSM tem ***M*** entradas, ***N*** saídas e ***k*** bits de estado. Elas também recebem um ***clock*** e, opcionalmente, um sinal de ***reset***.

Elas consistem de dois blocos de *lógica combinacional*, que são responsáveis por gerenciar a lógica de transição para o próximo estado e a lógica de saída do circuito, além de um registrador responsável por guardar o estado atual.

É importante ressaltar também que existem duas classes genéricas de máquinas de estados finitos, caracterizadas pelas suas especificações funcionais: máquina de Moore e máquina de Mealy. Na máquina de Moore a saída depende apenas do estado atual da máquina enquanto que a de mealy também irá depender das entradas atuais do circuito, dessa forma respondendo mais rapidamente a mudanças nas entradas. No

entanto, para a disciplina não será de importância que tipo será implementado no laboratório.

[1] É um tipo de circuito que depende não somente dos valores presentes nos sinais de entrada, mas também na sequência de entradas anteriores.

[2] Um circuito síncrono é um circuito digital em que mudanças no estado dos elementos de memória são sincronizadas por um sinal de clock.

Implementação propriamente dita

Antes de partirmos para uma implementação mais específica, podemos definir uma genérica que pode ser vista em:

```
module mod_name (input logic clock, reset, … ,
    output logic … );

    enum logic [ … : 0] {q0, q1, ... } state;

    always_ff @(posedge clock)
        if (reset) begin
            state <= q0;
            . . .
        end
        else
        unique case (state)
            q0: if (…) state <= q1;
            q1: ...
            . . .
        endcase
    always_comb …
endmodule
```

Como vocês sabem, o bloco **always_comb** é usado para guardarmos a lógica combinacional dos nossos circuitos enquanto que o **always_ff** guarda a lógica sequencial sincronizada com o clock que definirmos. Portanto, nas nossas FSM's vamos precisar de ambos (é muito normal termos dois always_comb representando os dois blocos combinacionais de entrada e saída citados anteriormente).

No nosso código representamos os estados através de uma variável do tipo **enum** (no qual podemos dar nomes aos nossos estados e usá-los posteriormente). Ao definir o

intervalo de [... : 0] se lembre de quantos estados você irá precisar e de quantos bits você vai precisar para representar um estado unicamente. (Exemplo: no caso de serem necessários só 4 estados são suficientes dois bits, isto é, [1 : 0]).

Por fim, caso o **reset** esteja ligado, o circuito deve se manter em um estado pré-estabelecido do problema e se não for o caso deverão ser testadas as condições sobre o estado atual e possivelmente alguma entrada para determinar se uma transição irá ocorrer.

Explicado o código, é hora de modelarmos um problema com uma máquina de estados no estilo que pode ser cobrado na disciplina, portanto fique atento ao que é necessário, esses são passos que você deve seguir ao desenvolver:

1° Leia o problema com atenção:

Implemente o circuito controlador de uma máquina de ar-condicionado.

Entradas

Entrada	Onde?
clock	LED[7]
reset	SWI[7]
diminuir	SWI[0]
aumentar	SWI[1]

- O ***reset*** deve ser assíncrono.
- O ***clock*** deve ter frequência de 1 Hz.
- A ***diminuir*** representa o comando para diminuir a temperatura desejada em 1 grau Celsius.
- A ***aumentar*** representa o comando para aumentar a temperatura desejada em 1 grau Celsius.

Saídas

Saída	Onde?
real	LED[6:4]
desejo	LED[2:0]
pingando	LED[3]

- O real representa a temperatura atual do ambiente.
- O desejo representa a temperatura desejada para o ambiente.
- O pingando indica que o ar-condicionado está gotejando atualmente.
- Tanto real quanto desejo representam valores entre 20 e 27 graus Celsius (inclusive).
    - Por exemplo, se real tem valor 3, o ambiente está a 23 graus Celsius.

Descrição

- No reset, a temperatura desejada e temperatura real são colocadas imediatamente para 20 graus Celsius.
- Através das chaves diminuir e aumentar, o usuário diminui ou aumenta a temperatura desejada em 1 grau Celsius a cada ciclo de *clock*.
- As temperaturas real e desejada não podem ser menores que 20 graus Celsius, nem maiores que 27 graus Celsius.
- Caso nenhuma das chaves (diminuir e aumentar) estejam acionadas, a temperatura desejada permanece igual. O mesmo vale se as duas chaves forem acionadas ao mesmo tempo.
- A temperatura real tende a se igualar à temperatura desejada, mas só consegue aumentar ou diminuir em 1 grau Celsius a cada 2 ciclos de *clock*.
- O ar-condicionado começa a pingar 10 ciclos de *clock* após o reset.
- Para interromper o gotejamento, o usuário deve ter a temperatura real em 27 graus Celsius por, pelo menos, 4 ciclos de *clock*. Depois desse intervalo, o ar-condicionado irá parar de pingar.

2° Isole as características mais importantes

Essa é uma questão extensa e com muitos requisitos, mas vamos ensiná-lo a dividi-la em partes menores e mais simples a partir de perguntas simples:

1. Que estados são realmente necessários nesse sistema?

    Nós temos um controlador de ar-condicionado que aceita uma temperatura desejada e atualiza a temperatura real do ambiente, também sabemos que com o

tempo o ar-condicionado começa a pingar e tem condições de transição para interromper o gotejamento. Pelo visto, com apenas dois estados podemos modelar o problema com sucesso: **PINGANDO** e **FUNCIONAMENTO NORMAL**.

2. Que informações você precisa manter durante a execução do programa?

   Pelo problema é necessário saber qual o valor das temperaturas real e desejada e se nesse instante essas temperaturas estão diminuindo, aumentando ou nenhum dos casos. Além disso é necessário contar os ciclos de clock passados desde o último reset ou desde que a tempera real se tornou 27°C (tanto para definir se está ocorrendo gotejamento, quanto para saber se o gotejamento deve parar). Também precisamos definir o clock de 1 Hz (é explicado na seção de dúvidas recorrentes). E por último também é importante manter informações dos índices dos componentes de entrada (SWITCHES) e de saída (LED), com isso o código da definição das variáveis fica assim:

```
// Índices usados no módulo
parameter LED_CLK = 7;
parameter SWI_RESET = 7;
parameter SWI_DIMINUIR = 0;
parameter SWI_AUMENTAR = 1;

// Flags booleanas
logic reset, clk_1;
logic diminuir, aumentar, gotejou;
logic temp_real_aumenta, temp_real_diminui;

// Ciclos de clock após o reset ou desde que
// a temperatura real se tornou 27° C
logic [3:0] ciclos = 0;

// Bits de temperatura
parameter NBITS_TEMP = 3;

// Temperaturas do ar-condicionado
logic [NBITS_TEMP-1:0] temp_real, temp_desejada;

enum logic [1:0] {
```

```
    FUNCIONAMENTO_NORMAL,
    PINGANDO
  } estado_atual;

// Gera clock de 1Hz
 always_ff @(posedge clk_2) begin
    clk_1 <= ~clk_1;
 end
```

3. Definir o bloco combinacional de entrada.

```
always_comb begin
    reset <= SWI[SWI_RESET];
    diminuir <= SWI[SWI_DIMINUIR];
    aumentar <= SWI[SWI_AUMENTAR];
 end
```

4. Definir a parte sequencial e as condições de transição

```
always_ff @(posedge clk_1 or posedge reset) begin
    if (reset) begin
      // Em caso de reset as temperaturas voltam a 20° C
      // e o funcionamento é normal
      estado_atual <= FUNCIONAMENTO_NORMAL;
      temp_real <= 0;
      temp_desejada <= 0;
      gotejou <= 0;
    end
    else begin

      // Testa de antemão se no ciclo passado houve aumento
      // em temperatura desejada e incrementa ou decrementa a real
      if (temp_real_aumenta) begin
        temp_real <= temp_real + 1;
        temp_real_aumenta <= 0;
      end else if (temp_real_diminui) begin
        temp_real <= temp_real - 1;
```

```
      temp_real_diminui <= 0;
    end

    // Caso aumentar ou diminuir sejam verdadeiros individualmente
    if (aumentar != diminuir) begin
      if (aumentar && temp_desejada < 7 && temp_real < 7) begin
        temp_desejada <= temp_desejada + 1;
        temp_real_aumenta <= 1;
      end else if (diminuir && temp_desejada > 0 && temp_real > 0) begin
        temp_desejada <= temp_desejada - 1;
        temp_real_diminui <= 1;
      end
    end

    unique case (estado_atual)
      FUNCIONAMENTO_NORMAL: begin
        ciclos <= ciclos + 1;
        if (ciclos == 10 && !gotejou) begin
          estado_atual <= PINGANDO;
          ciclos <= 0;
          gotejou <= 1;
        end
      end
      PINGANDO: begin
        if (temp_real == 7)
          ciclos <= ciclos + 1;
        if (ciclos == 4)
          estado_atual <= FUNCIONAMENTO_NORMAL;
      end
    endcase
  end
end
```

5. Definir o bloco de saída

```
always_comb begin
  LED[LED_CLK] <= clk_1;
  LED[6:4] <= temp_real;
  LED[2:0] <= temp_desejada;
  LED[3] <= (estado_atual == PINGANDO);
```

```
end
```

Algumas dúvidas recorrentes:

Como assim um clock de 1 Hz?

O clock no laboratório por padrão é de 2 Hz, ou seja, tem um período de 0.5s. E o desejado é um período de 1s, para isso, precisamos de um divisor de frequência e podemos usar um flip-flop com esse fim, o código é o seguinte:

```
// Gera clock de 1Hz
always_ff @(posedge clk_2) begin
  clk_1 <= ~clk_1;
end
```

O que significa um reset síncrono e assíncrono?

Quando falamos de algo síncrono ou assíncrono no laboratório, falamos de algo que é sincronizado a partir do clock ou não. No caso, um reset assíncrono é aquele que independe da transição de subida ou de descida do flip-flop, por isso expressamos ele dessa forma no código:

```
always_ff @(posedge clk_1 or posedge reset)
```

Ou seja, caso o reset seja ligado, independente do valor de clk_1, o estado do circuito passará a ser o definido dentro da expressão condicional do **reset**.

Considerações finais:

A apresentação dada sobre o assunto está longe de ser a mais completa, poderíamos aqui ainda tentar modelar nossas máquinas de estados diretamente em circuitos digitais, mas acho melhor não, já que não é o foco da disciplina, todavia se quiser aprofundar seu conhecimento, nas fontes existem alguns links que podem ajudar nessa tarefa.

E por fim, como em qualquer coisa que deve ser aprendida, a prática é necessária, por isso é de extrema importância que você faça mais exercícios como o demonstrado acima. E eles podem ser encontrados no **tamburetei**, projeto da OpenDevUFCG:

https://github.com/OpenDevUFCG/Tamburetei/tree/master/loac/leites

Mais fontes:

https://en.wikipedia.org/wiki/Finite-state_machine
https://github.com/OpenDevUFCG/Tamburetei/tree/master/loac/leites
http://labarc.ufcg.edu.br/loac/uploads/OAC/sv_tutorial3_pt.odp
http://labarc.ufcg.edu.br/loac/index.php?n=OAC.Loac
https://en.wikipedia.org/wiki/Sequential_logic