O **CFG (Control Flow Graph)** é um modelo de representação gráfica que descreve todos os caminhos possíveis de execução de um programa ou sistema. Ele é amplamente usado em análise estática, compiladores e verificação formal, incluindo execução simbólica.

Estrutura do CFG:

- **Nodos (ou vértices)**: Representam instruções, blocos de código ou estados do programa.
    - Exemplo: Um bloco de código como `if (x > 0) { ... }` é um único nodo no gráfico.
- **Arestas**: Representam as possíveis transições entre os nodos, com base no fluxo de controle.
    - Exemplo: A transição de um `if` para o bloco `then` ou `else`.

Como o CFG funciona:

1. **Divisão do código em blocos básicos**:
    - Um bloco básico é uma sequência linear de instruções sem bifurcações ou saltos.
    - Cada bloco é representado como um nodo no CFG.
2. **Conexão dos blocos**:
    - As arestas conectam os blocos, representando o fluxo de controle entre eles.
    - Por exemplo, após um bloco com um `if`, haverá arestas para os blocos correspondentes aos caminhos `then` e `else`.
3. **Entrada e saída**:
    - O CFG tem um nodo inicial (ponto de entrada do programa) e normalmente um ou mais nodos finais (onde o programa termina).

Exemplo de CFG:

Para o seguinte código:

```C/C++
if (x > 0) {
    y = 1;
} else {
    y = -1;
}
z = y + 2;
```

O CFG seria:

- Nodo 1: `if (x > 0)`
- Nodo 2: `y = 1` (bloco `then`)
- Nodo 3: `y = -1` (bloco `else`)
- Nodo 4: `z = y + 2`
- Arestas conectam `Nodo 1 -> Nodo 2`, `Nodo 1 -> Nodo 3`, e ambos `Nodo 2` e `Nodo 3 -> Nodo 4`.

Observações:

- **Exploração de caminhos**: A execução simbólica utiliza o CFG para explorar diferentes caminhos de execução do programa.
- **Detecção de loops**: Loops e bifurcações podem ser identificados no CFG para uma análise mais eficiente.
- **Verificação de invariantes**: O CFG ajuda a identificar onde no fluxo de controle os invariantes precisam ser verificados.
- **Cobertura de código**: É usado para garantir que todos os caminhos relevantes foram analisados.

Aplicações do CFG:

- **Compiladores**: Para otimização de código, análise de dependências e eliminação de blocos mortos.
- **Análise estática**: Para encontrar vulnerabilidades, como caminhos que levam a falhas.
- **Verificação de propriedades**: Para verificar invariantes e segurança em software crítico, como contratos inteligentes.