VISUALIZANDO O FLUXO DE CONTROLE DE PROGRAMAS

Ana M. Price, Fabio Garcia, Carla Purper

Pós-Graduação em Ciência da Computação
Universidade Federal do Rio Grande do Sul

R E S U M O

O fluxo de controle de programas representado através de grafos dirigidos tem sido considerado como uma ferramenta útil a várias etapas do desenvolvimento de software: projeto, teste, depuração, manutenção e análise de complexidade. Este artigo apresenta um conjunto de algorítmos para projetar na tela grafos de fluxo de controle gerados automàticamente a partir de codigo fonte. São apresentados exemplos de grafos gerados por uma implementação dos algorítmos desenvolvidos.

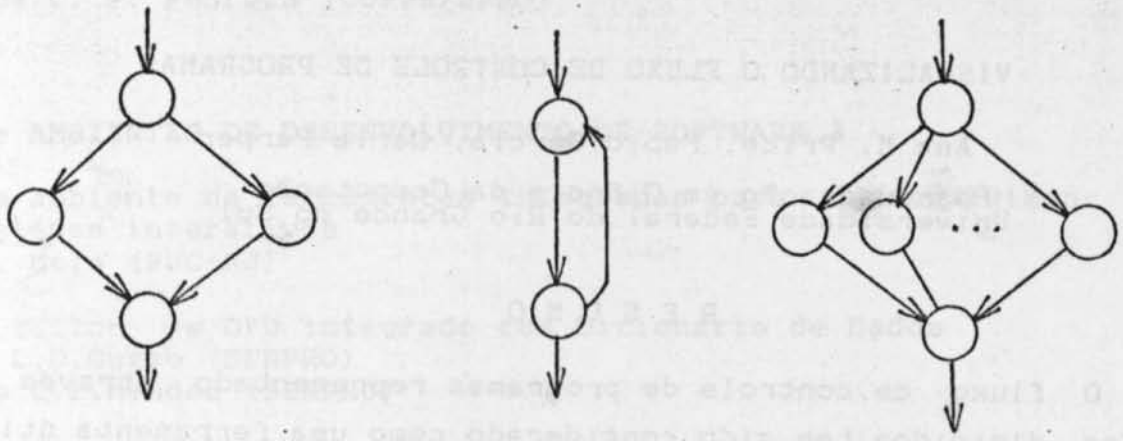
1. INTRODUÇÃO

A representação do fluxo de controle de programas através de grafos dirigidos tem provado sua utilidade em vários contextos do processo de desenvolvimento do software [2,4,6,7], particularmente,

i) na compreensão da estrutura e fluxo de execução de programas. Considerando-se que os comandos de controle de linguagens de programação podem ser representados através de estruturas gráficas padronizadas (Figura 1) torna-se fácil a identificação destas estruturas num grafo que represente um programa completo. O fluxo de execução do programa é imediatamente visualizado através dos arcos dirigidos indicados no grafo.

ii) identificação de caminhos de teste. Com base no grafo de controle de um programa pode-se calcular o número de caminhos de teste necessários para, por exemplo, realizar um teste de cobertura de arco [7].

iii) análise de complexidade. Várias métricas de complexidade de programas, tais como, Numero Ciclomático [5], Scope Ratio [3] e a Métrica de Chen [1], baseiam-se no grafo de fluxo de controle de programa.



IF-THEN-ELSE REPEAT-UNTIL CASE

Figura 1

De acordo com o exposto acima, acredita-se que ferramentas de desenvolvimento de software providas de facilidades para gerar automaticamente grafos de controle a partir de especificações de programas contribuem para a construção de software mais confiável e de melhor qualidade. Os algorítmos aqui descritos fazem parte de uma ferramenta de apoio ao teste que está sendo desenvolvida pelos autores deste artigo. Os grafos de controle, gerados automàticamente a partir da análise de programas-fonte, servem de base para a determinação de caminhos de teste, os quais são posteriormente executados simbòlicamente visando a seleção de dados para teste. A ferramenta pretende, além de projetar o grafo completo do programa em análise, mostrar também subgrafos representando caminhos de execução específicos quando de sua interpretação simbólica.

A projeção na tela de um grafo de fluxo gerado automàticamente é uma tarefa razoàvelmente complexa devido à dificuldade de distribuir nodos e traçar arcos, tal que, o grafo gerado fique balanceado, compreensível e agradável ao usuário. O algorítmo aqui descrito tem apresentado um ótimo desempenho para programas estruturados, gerando sistemàticamente grafos corretos.

2. REPRESENTANDO O FLUXO DE CONTROLE DE PROGRAMAS.

A programação estruturada, com suas regras de paragrafação, pretende tornar mais clara e legível a estrutura de controle do programa. De certo modo, este objetivo é atingido. Porém, quando o fluxo de controle de um programa é representado através de um grafo dirigido, não há dúvida de que este esquema de representação é mais vantajoso do que a paragrafação, pois propicia melhor visualização da estrutura e fluxo de execução do programa. Por outro lado, um grafo de fluxo não explicita qualquer código, logo, sòmente é útil quando associado ao código fonte a partir do qual foi gerado.

Num grafo de fluxo, os nodos representam sequências de comandos consecutivos cuja execução só pode ser realizada a partir do início da sequência, sendo processada até o final sem possibilidade de parada ou desvio, exceto ao final da mesma. Os arcos representam desvios, condicionais ou incondicionais, do fluxo de execução do programa.

Geralmente, representa-se um grafo de fluxo através de uma matriz binária de adjacência M[nXn], onde n é igual ao número de nodos e M(i,j)=1 se existe um arco dirigido do nodo i ao nodo j. Se não existe tal arco, M(i,j)=0. A vantagem desta representação é a sua natureza compacta e a possibilidade de obter-se informações adicionais sobre o fluxo de controle do programa através de operações sobre a matriz. Por exemplo, descobrir se algum segmento de programa nunca será executado, ou obter o número de caminhos possíveis a partir de um dado nodo [6].

A computação da matriz de adjacência pode ser obtida através de um analizador que identifica os comandos de controle do programa, agrupando comandos sequenciais consecutivos num mesmo nodo, e representando os desvios de fluxo de execução através de ligações (M(i,j)=1) entre os pares (i,j) de nodos envolvidos.

Segue a descrição de um algorítmo para a geração de matrizes de adjacência a partir de blocos básicos (procedures ou programas sem subrotinas) escritos em Pascal. A fim de não extender demasiadamente este artigo, apenas foi desenvolvida a análise para alguns comandos de controle. As estruturas de dados referenciadas

pelo algorítmo são as seguintes:

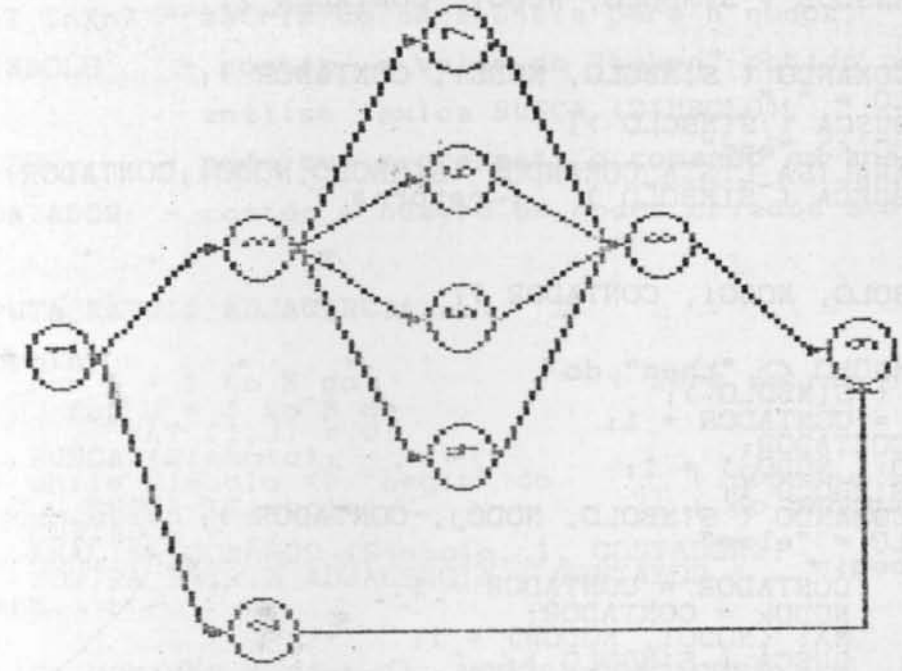
- MAT [nXn] - matriz de adjacência para n nodos,
- SIMBOLO - contém o valor do "token" obtido pela rotina de análise léxica BUSCA (SIMBOLO),
- NODOx - nodo que representa o comando em análise,
- CONTADOR - contém o número de nodos criados até então.

```
COMPUTA_MATRIZ_ADJACENCIA;
   begin
     for I = 1 to N do                    ( zera matriz de adjacência)
       for J = 1 to N do
         MAT [I,JJ = 0;
     BUSCA (Símbolo);
     while Símbolo <> "begin" do          ( procura inicio do corpo
       BUSCA (Símbolo);                   ( do programa  )
     CONTADOR = 1;
     ANALISA_COMANDO (Símbolo, 1, CONTADOR);
     MOSTRA_MATRIZ_ADJACENCIA ( CONTADOR )
   end.

ANALISA_COMANDO ( Símbolo, NODOi, CONTADOR );
     case Símbolo of
        "begin" : ANALISA_BEGIN (Símbolo, NODOi, CONTADOR);
        "if"    : ANALISA_IF    (Símbolo, NODOi, CONTADOR);
        "while" : ANALISA_WHILE (Símbolo, NODOi, CONTADOR);
        "repeat": ...
        "for"   : ...
        "case"  : ...
        otherwise : { comando simples, procura fim de comando }
          while Símbolo <> ( ";" | "end" | "else" ) do
            BUSCA (Símbolo);

ANALISA_WHILE ( SIMBOLO, NODOi, CONTADOR );
     begin
        while SIMBOLO <> "do" do
          BUSCA ( SIMBOLO );
        CONTADOR = CONTADOR + 1;
        NODOj = CONTADOR;
        MAT [NODOi, NODOj] = 1;
        BUSCA ( SIMBOLO );
        ANALISA_COMANDO ( SIMBOLO, NODOj, CONTADOR );
        MAT [NODOj, NODOi] = 1;
     end;

ANALISA_BEGIN ( SIMBOLO, NODOi, CONTADOR );
     begin
        BUSCA ( Símbolo );
        ANALISA_LISTA_COMANDOS ( SIMBOLO, NODOi, CONTADOR );
     end;
```

```
ANALISA_LISTA_COMANDOS ( SIMBOLO, NODOi, CONTADOR );

     begin
        ANALISA_COMANDO ( SIMBOLO, NODOi, CONTADOR );
        if SIMBOLO = ";"
           then BUSCA ( SIMBOLO );
        if SIMBOLO <> "end"
           then ANALISA_LISTA_COMANDOS (SIMBOLO,NODOi,CONTADOR)
           else BUSCA ( SIMBOLO )    ( retorna )
     end;

ANALISA_IF ( SIMBOLO, NODOi, CONTADOR );

     begin
        while SIMBOLO <> "then" do
           BUSCA ( SIMBOLO );
        CONTADOR = CONTADOR + 1;
        NODOj = CONTADOR;
        MAT [NODOi, NODOj] = 1;
        BUSCA ( SIMBOLO );
        ANALISA_COMANDO ( SIMBOLO, NODOj, CONTADOR );
        if SIMBOLO = "else"
           then begin
                   CONTADOR = CONTADOR + 1;
                   NODOk = CONTADOR;
                   MAT [NODOi, NODOk] = 1;
                   BUSCA ( SIMBOLO );
                   ANALISA_COMANDO (SIMBOLO, NODOk, CONTADOR);
                   CONTADOR = CONTADOR + 1;
                   NODOl = CONTADOR;
                   MAT [NODOk, NODOl] = 1;
                end
           else begin
                   CONTADOR = CONTADOR + 1;
                   NODOl = CONTADOR;
                   MAT [NODOi, NODOl] = 1;
                end;
        MAT [NODOj, NODOl] = 1;
        NODOi = NODOl;
     end;
```

3. ALGORITMO DE VISUALIZAÇÃO DO GRAFO DE CONTROLE

A projeção do grafo de fluxo de um programa, a partir da matriz de adjacência associada, não é um problema trivial pois requer um processamento complexo para dispor nodos e arcos sem que ocorram intersecções entre eles, particularmente, quando o programa contém comandos não estruturados do tipo GOTO. Mesmo considerando-se programas estruturados, o posicionamento de nodos pode ser dificultado em certas situações, tais como as descritas a seguir. Problemas de sobreposição de nodos podem ocorrer, por exemplo, quando o programa se desenvolve através de vários subgrafos (caso de comando CASE), tendo, cada um destes, grande densidade de nodos. Intersecções de arcos com nodos podem ocorrer, por exemplo, na junção de dois subgrafos, quando um destes é muito mais denso e longo do que o outro. A Figura 2 ilustra um



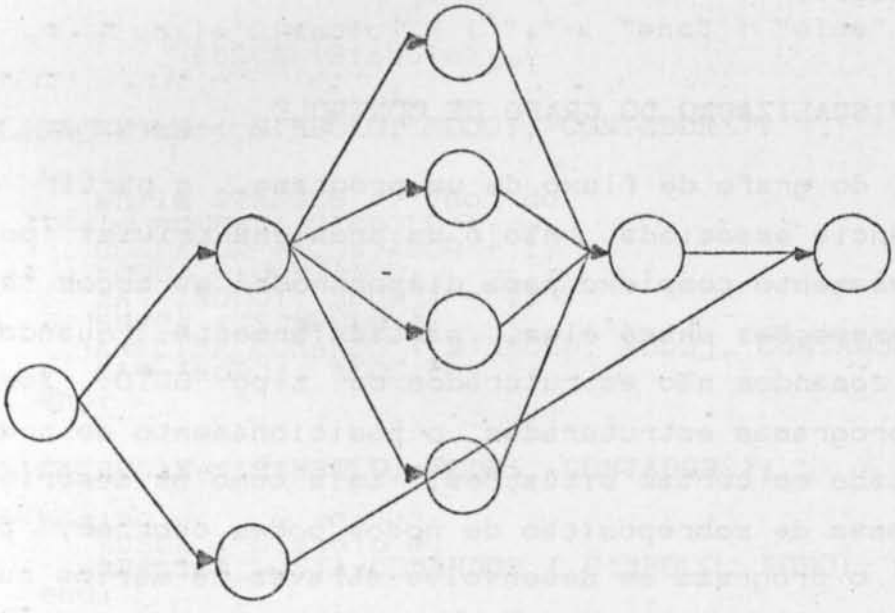


FIGURA 2

exemplo de intersecção de arco e nodo e a solução deste problema através do algorítmo desenvolvido.

O algorítmo de projeção de grafos parte da matriz de adjacência que representa o grafo de fluxo do programa em análise. A cada nodo é atribuido um intervalo em X de modo que o intervalo de um nodo-pai deve conter os intervalos de seus descendentes, não podendo ocorrer quaisquer sobreposições de intervalos. O cálculo do tamanho de cada intervalo é feito a partir do nodo inicial para o nodo final, segundo a numeração crescente dos nodos. Para cada nodo-pai verifica-se o número de nodos-filhos atribuindo-se a cada um deles, um intervalo padrão. Se a soma dos intervalos padrões alocados aos nodos-filhos for maior que o intervalo do nodo-pai, atualiza-se o intervalo do nodo-pai para que se iguale a soma dos intervalos de seus filhos. Ao atualizar-se o intervalo do nodo-pai deve-se também atualizar os intervalos de seus antecedentes, de forma que o deslocamento gerado pela inclusão de nodos-filhos e seus intervalos seja absorvido por todo o grafo, mantendo-se assim uma disposição balanceada de nodos. Após todos os intervalos terem sido computados, calcula-se o ponto central de cada intervalo, armazenando-o na tabela de nodos (TAB-NODOS) como sendo a coordenada X relativa ao nodo. A coordenada Y relativa a cada nodo é calculada da seguinte forma: cada nodo-filho recebe a coordenada Y de seu pai acrescentada de um deslocamento padrão.

Após o cálculo de posicionamente de nodos, é computado o posicionamento dos arcos. Para cada par de nodos pai/filho calcula-se, baseado nas informações de posicionamento contidas na TAB_NODOS, os pontos inicial e final do arco que os une. Se o nodo-filho estiver há um nível abaixo do pai, o arco que os une é direto. Caso o nodo-filho esteja a dois ou mais níveis abaixo do nodo-pai, ou acima (significando um retorno de controle para trás), é necessario calcular o caminho do arco de modo que este não corte outros arcos e nodos. Os pontos calculados são armazenados numa outra estrutura chamada TAB_ARCOS. Tendo-se as coordenadas relativas dos nodos e pontos iniciais e finais dos arcos, pode-se, então, realizar a visualização pròpriamente dita do grafo.

Um problema adicional a ser resolvido na visualização ocorre quando a tela não comporta o grafo por inteiro. Para solucioná-lo utilizou-se a técnica de janelas, que consiste em definir uma janela de seleção em coordenadas do plano cartesiano, a qual percorre as estruturas TAB_NODOS e TAB_ARCOS selecionando os elementos a serem visualizados, e uma janela de exibição, definida em coordenadas de tela do equipamento. Através de uma conversão de coordenadas entre as janelas (seleção e exibição), consegue-se visualizar os elementos selecionados pela janela de seleção. Quando o grafo é mostrado na tela, ficam habilitadas as teclas de movimento do cursor que deslocam o grafo permitindo ao usuário a visualização de todos os nodos. Internamente ao programa, essa teclas implicam na movimentação da janela de seleção sobre as estruturas.

Exemplos de grafos gerados podem ser observados ao final deste relatório.

4. CONCLUSOES

Ferramentas de desenvolvimento de software providas de facilidades para gerar automàticamente grafos de fluxo a partir de especificações de programas facilitam o entendimento do programa, auxiliam em manutenções futuras, e promovem o teste sistemático e a análise de complexidade de programas.

A representação de grafos de fluxo através de matrizes de adjacência, além de ser compacta, pode derivar informações adicionais sobre o fluxo de controle do programa, como por exemplo, a identificação de segmentos que jamais serão executados e a obtenção do número de caminhos de teste a partir de um determinado nodo. Um algorítmo para a geração de matrizes de adjacência a partir de programas/procedures Pascal encontra-se descrito na Secção 2 deste relatório.

A projeção na tela de um grafo de fluxo a partir da matriz de adjacência associada requer um processamento iterativo para realizar o posicionamento de nodos sem que hajam intersecções entre nodos e arcos. O algorítmo desenvolvido tem resultado em grafos corretos, balanceados e compreensíveis ao usuário. A im-

plementação foi realizada em Turbo Pascal, e conta com aproximadamente 1500 linhas de código fonte.

AGRADECIMENTOS

Os autores agradecem ao Prof. Anatólio Laschuk por suas sugestões quanto ao desenvolvimento do algorítmo de posicionamento de nodos, e ao CNPq, pelo incentivo financeiro recebido.

5. REFERENCIAS BIBLIOGRAFICAS

[1] E. Chen, Program Complexity and Programmer Productivity, IEEE Trans. on Software Eng., Vol SE-4, May 1978, pp.187-94.

[2] L. Clarke, Automatic Test Data Selection Techniques, in Infotech State of The Art Report, Software Testing, Vol. 2, Infotech International, England, 1979.

[3] W. Harrison & K. Magel, A Complexity Measure based on Nesting Level, ACM SIGPLAN Notices, March 1981, pp. 63-74.

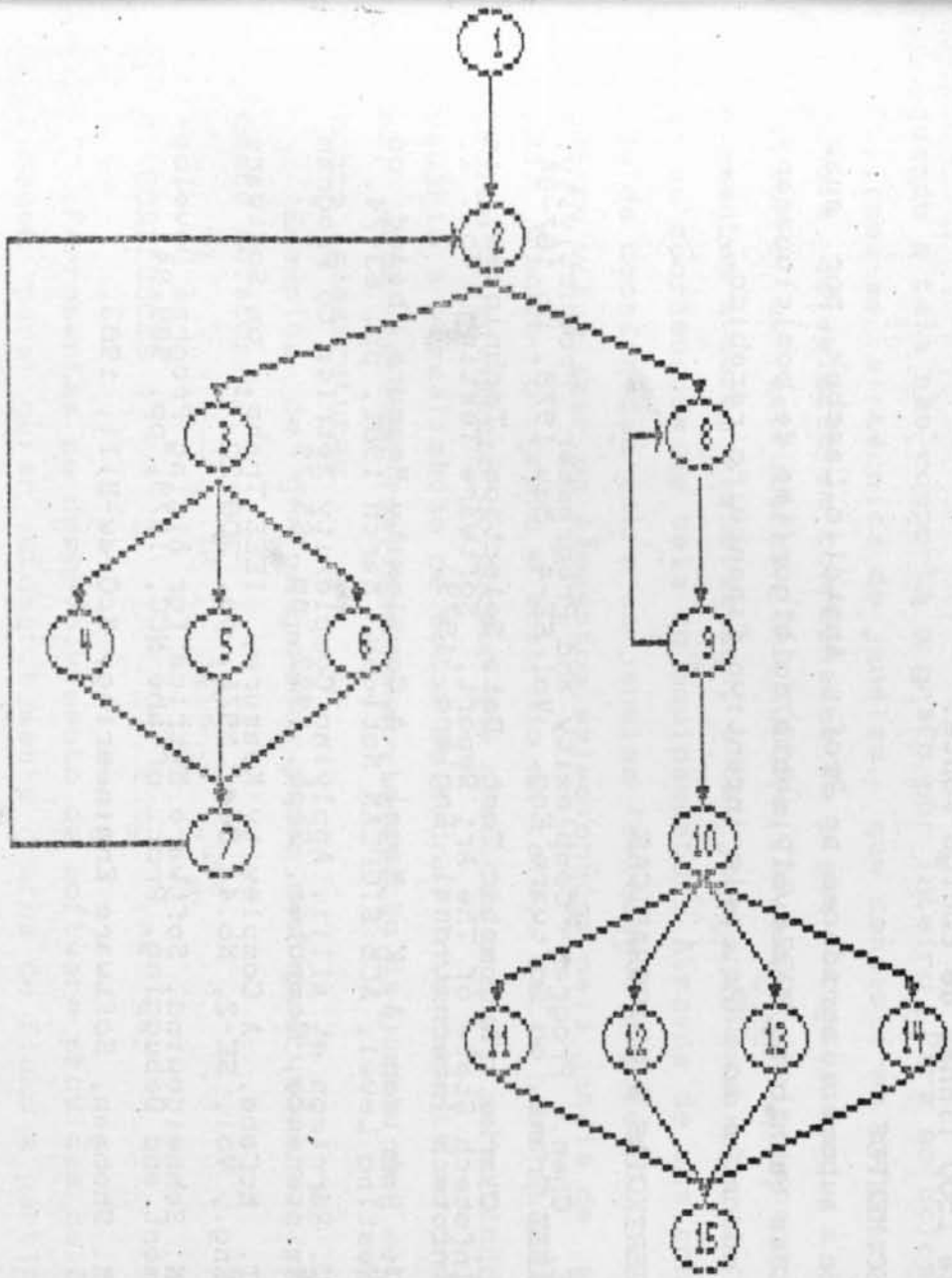
[4] W. Harrison et allii, Applying Complexity Metrics to Program Maintenance, Computer, Sept. 1982, pp. 65-79.

[5] T. McCabe, A Complexity Measure, IEEE Trans. on Software Eng., Vol. SE-2, No.4, Dec. 1976, pp. 308-20.

[6] N. Schneidewind, Software Metrics for Aiding Program Development and Debugging, Proc. of the NCC, 1979, pp. 989-94.

[7] M. Shooman, Software Engineering, McGraw-Hill, 1983.

```
BEGIN
  WHILE  <condição> DO
    CASE <expressão> OF
      <const-1>: <comando>;
      <const-2>: <comando>;
      <const-3>: <comando>;

    END;
  REPEAT
      <comando>
  UNTIL <condição>;
  CASE  <expressão> OF

      <const-1> : <comando>;
      <const-2> : <comando>;
      <const-3> : <comando>;
      <const-4> : <comando>;

  END
END
```



```
BEGIN
  WHILE <condição> DO
    IF <condição>
      THEN <comando>
      ELSE <comando>;

  WHILE <condição> DO
      CASE <expressão> OF
       <const-1>: <comando>;
       <const-2>: <comando>;
       <const-3>: <comando>;

      END;
   <comando>
END
```