PUC

Série: Monografias em Ciência da Computação
Nº 18/77

GERAÇÃO DE DADOS DE TESTE BASEADA NOS ASPECTOS ESTRUTURAIS DE PROGRAMAS

por

Antonio Morais da Silveira
Carlos José Pereira Lucena

Departamento de Informática

Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente 225 — ZC 19
Rio de Janeiro — Brasil

Série: Monografias em Ciência da Computação

Nº 18/77

Editor da Série: Michael F. Challis novembro,1977

GERAÇÃO DE DADOS DE TESTE BASEADA NOS ASPECTOS ESTRUTURAIS DE PROGRAMAS

por

Antonio Morais da Silveira

Carlos José Pereira Lucena

* Este trabalho foi patrocinado em parte pela FINEP.

Para obter cópias dirija-se a:

Rosane T.L. Castilho
Chefe, Set. Doc. e Informação
Depto. de Informática - PUC/RJ.
Rua Marquês de São Vicente, 209 - Gávea
20.000-Rio de Janeiro-RJ-BRASIL.

ABSTRACT

In this paper we discuss the difficulties met in the process of generating test data and current research in this area.

One methodology of test data generation is proposed based on the structural aspects of programs. This methodology shows that the problems of nontraversable paths and arrays are solved when we symbolically execute a program by forward substitution. With the proposed method the detection of nontraversable paths occurs when we meet a pair of inconsistent restrictions during a symbolic execution of a program path.

KEYWORDS

Constraints, symbolic execution nontraversable path, test data generation, digraph, loop, backward substitution, forward substitution.

RESUMO

Neste trabalho falamos das dificuldades encontradas no processo de geração de dados de teste e as pesquisas desenvolvidas na área.

É proposta uma metodologia de geração de dados de teste baseada nos aspectos estruturais de programas, a qual mostra que os problemas de caminhos não atravessáveis e de arrays são solucionados quando executamos simbolicamente um programa por substituição forward. Com o método proposto, a deteção de caminhos não atravessáveis ocorre no momento em que encontramos um par de restrições inconsistente durante a execução simbólica de um caminho do programa.

PALAVRAS CHAVE

Restrições, execução simbólica, caminho não atravessável, geração de dados de teste, dígrafo, loop, substituição backward, substituição forward.

SUMÁRIO

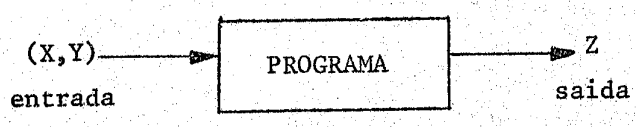
1. INTRODUÇÃO

Um dos grandes problemas associados ao processo de desenvolvimento de software, é o de determinar até que ponto um determinado programa é confiável. Cerca de 50% dos custos totais envolvidos no desenvolvimento de software [1] são relativos aos processos de teste e verificação. Tal fato ilustra o interesse de se pesquisar métodos voltados para o problema de teste e verificação de programas.

Vários trabalhos já foram desenvolvidos na área de teste e, continua-se a pesquisar novos métodos em busca de um aperfeiçoamento cada vez maior das soluções já existentes. Vale salientar que todas as metodologias de teste se prestam à descoberta da presença de erros mas nunca da ausencia dos mesmos [1]. Das pesquisas já realizadas na área de teste, é possível destacar duas linhas principais de trabalho: uma que busca a solução do problema a partir da análise dos aspectos estruturais de um programa(ex: [3] e [6]) e, outra que tenta projetar um plano de teste a partir da especificação do programa (ex.: [4]).

Uma estratégia ideal para o teste de programas, seria a execução do programa para todos os possiveis valores de suas variáveis de entrada.Infelizmente, isto é impraticável uma vez que o número de casos de teste necessários, mesmo para programas muito simples é proibitivamente grande, como podemos observar no exemplo a seguir.

Considere-se que um programa a ser testado, tem duas variáveis de entrada e uma variável de saida, conforme ilustra o gráfico abaixo. Se assumirmos que X e Y são variáveis inteiras e



que o programa será processado em um computador com registradores de 32 bits, teriamos então $2^{32} \times 2^{32} = 2^{64}$ atributos possiveis para o par de entrada (X,Y) e, consequentemente 2^{64} casos de teste. Supondo-se que este programa é relativamente pequeno e que leva em média apenas um milisegundo para cada execução completa, precisariamos de mais de 50 bilhões de anos para completar o teste.

Dada a impraticabilidade da solução ideal, o que se tem feito é procurar soluções realísticas para este problema, as quais produzem como resultado um subconjunto do conjunto dos possiveis atributos que podem ser associados às variáveis de entrada do programa.

Antes de passarmos à discussão das soluções baseadas nos aspectos estruturais dos programas, vamos apresentar a terminologia a ser usada neste trabalho.

1.1. TERMINOLOGIA USADA

A fim de facilitar a compreensão do presente trabalho, vamos fixar a terminologia a ser usada durante o mesmo, para a identificação de partes das estruturas representativas de programas (seus fluxogramas).

- Caminho em um programa, é definido como qualquer trajetória através do mesmo que tenha inicio na entrada da estrutura do programa e termine em uma saida possível.
- Ramo em uma estrutura de programa, é definido como qualquer ligação entre dois nós consecutivos da mesma.

Na estrutura da FIG. 1.1, o conjunto {a,b,c,d,e,f} , é o conjunto de ramos da estrutura, enquanto o conjunto {abf, abcdef} é o conjunto de caminhos através da mesma.

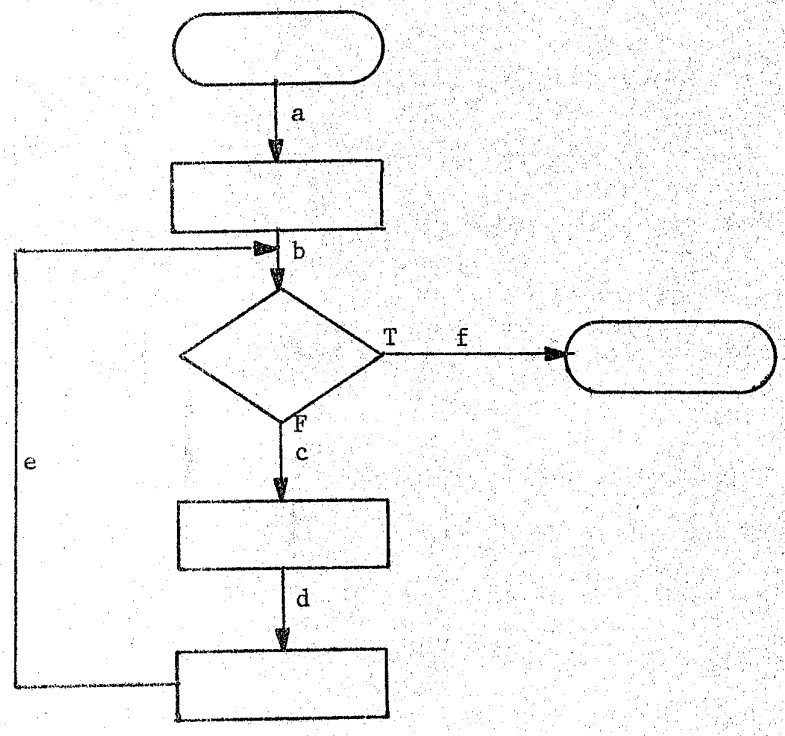


FIG. 1.1

1.2. ESTRATÉGIAS DE TESTE BASEADAS NA ANÁLISE DA ESTRUTURA DO PROGRAMA

Uma vantagem das metodologias baseadas nos aspectos estruturais de programas além da aplicabilidade manual imediata é a de que elas possibilitam a automatização com relativa facilidade. Nas discussões das estratégias de teste baseadas nos aspectos estruturais de programas, pesquisadores têm dividido suas opiniões a cerca de que aspectos estruturais devem ser levados em consideração em uma dada metodologia de geração de dados de teste. Em termos gerais, metodologias de geração de dados de teste encontradas na literatura, consideram importante testar os seguintes aspectos associados às estruturas de programas:

- todos os caminhos possíveis do programa.
- ser capaz de percorrer, pelo menos uma vez, cada ramo da estrutura do programa.
- ser capaz de executar pelo menos uma vez cada comando do programa.

A importância dos aspectos acima mencionados, com relação a eficácia no processo de teste está expressa em ordem decrescente. Em outras palavras, uma metodologia de teste que executa todos os caminhos possíveis atraves de um programa, obrigatóriamente percorrerá pelo menos uma vez cada ramo da estrutura do programa e, consequentemente executará pelo menos uma vez cada comando do programa. O recíproco não é verdadeiro, uma vez que executar todos os comandos de um programa, não implica em percorrer pelo menos uma vez cada ramo da estrutura do programa (conforme podemos constatar no exemplo da FIG. 1.2) nem em executar todos os caminhos possíveis através do programa.

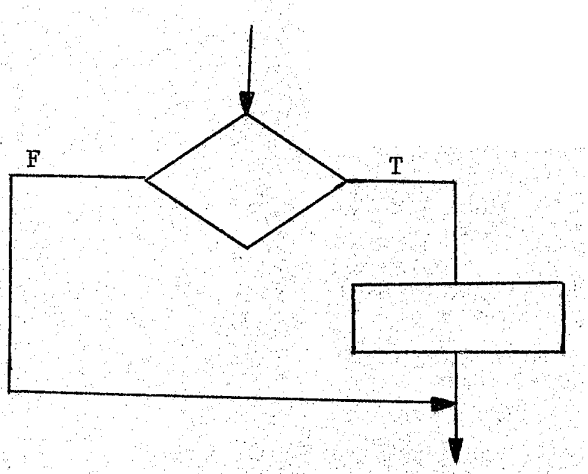


FIG. 1.2

Um enfoque ideal dentro da análise de aspectos estruturais, seria adotar soluções que testassem todos os caminhos possíveis através do programa. Infelizmente isto às vêzes não é possível, em virtude de problemas tais como:

- caminhos não atravessáveis no programa

- numero de caminhos possíveis exageradamente grande, etc.

Como ilustração dessas situações, considere o exemplo da FIG. 1.3. Se não considerarmos o loop, representado pela linha tracejada ou, iterando-o apenas uma vez, teremos 9 caminhos possíveis de execução na estrutura. Se, considerarmos a iteração do loop 10 vêzes, teremos 9^{10} ($\approx 10^{10}$) caminhos possíveis de execução, o que ilustra a impraticabilidade de todas as metodologias que procuram esta característica.

Dada a impraticabilidade de se tentar testar todos os caminhos possíveis através da estrutura do programa, o que a maioria dos pesquisadores tem feito, é procurar formular metodologias que se preocupam apenas com execução de cada ramo da estrutura do programa pelo menos uma vez. Vários resultados já foram obtidos neste sentido, mas infelizmente, fatores tais como:

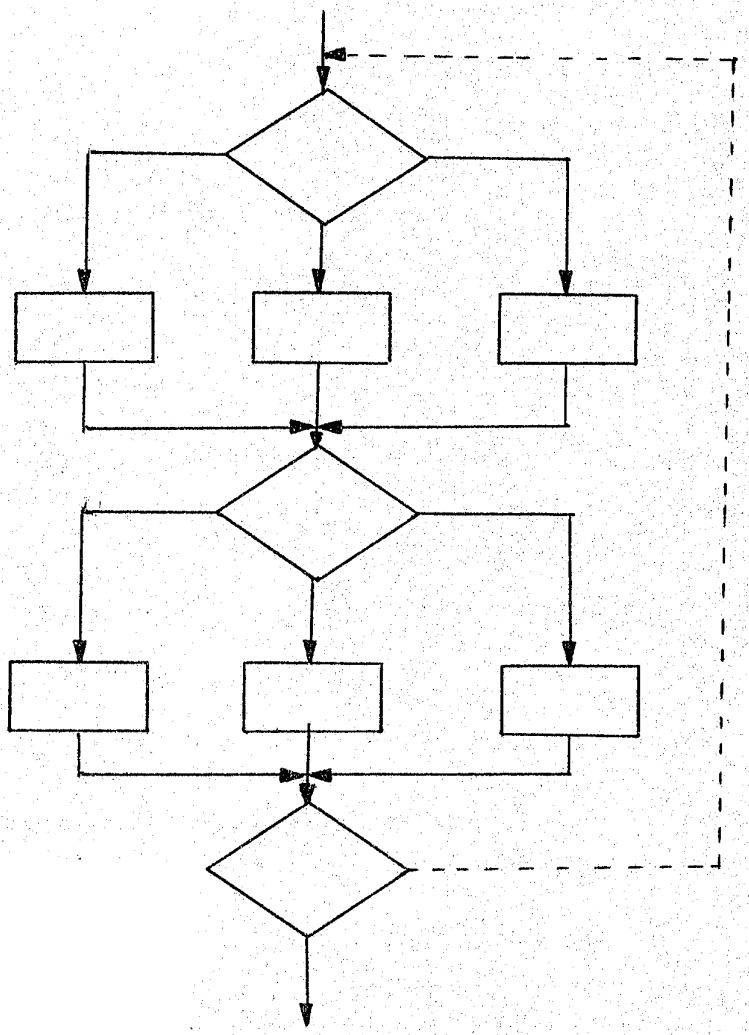


FIG. 1.3

- numero de caminhos envolvidos em grandes programas
- caminhos não atravessáveis em programas
- loop's em programas
- variáveis subscritas
- estruturas de blocos e chamadas de procedures ou subrotinas.

têm contribuido para limitar a abrangência das metodologias disponíveis a apenas subconjuntos específicos de programas.

1.3. DEFINIÇÃO GERAL DO MÉTODO PROPOSTO

O método proposto, baseia-se nos aspectos estruturais de programas. O que faremos é analisar estes aspectos e usar o resultado desta análise como subsídio para auxílio na geração de dados de teste. Este método pode ser definido da seguinte forma:

1) Dado um programa codificado em uma linguagem de programação, associa-se a ele um grafo dirigido. Cada nó do grafo, pode representar um único comando, ou parte de um comando (ver seção 2). Como a maioria dos programas lança mão de iterações, o grafo dirigido obtido é quase sempre de natureza cíclica. A este grafo dirigido cíclico associamos um dígrafo acíclico equivalente, a fim de que tenhamos uma visualização melhor da estrutura do programa, principalmente no que diz respeito aos caminhos possíveis sobre a estrutura.

2) De posse da estrutura acíclica do programa, o usuário então escolhe qual caminho do programa deseja testar, sendo esta escolha baseada no critério que ele julgar conveniente.

3) O caminho escolhido é então executado simbólicamente por substituição FORWARD (da entrada para a saida). A medida em que se vai executando o programa, gera-se um conjunto de restrições (expressas em termos das variáveis de entrada), que devem ser satisfeitas para que o caminho seja exercitado. A execução simbólica para quando se deteta alguma inconsistência num par de restrições, o que ocorre quando executamos simbólicamente um caminho não atravessável. Caso o caminho seja atravessável , no final da execução do mesmo teremos um conjunto de restrições (desigualdades) que solucionadas nos fornecerão um conjunto de dados de teste que executarão o caminho.

Comparando este método com os existentes, constata-se que o mesmo tambem oferece condições para execução de cada ramo da estrutura do programa pelo menos uma vez.

Uma vez que o tipo de substituição usado na execução simbólica é o FORWARD, torna-se possível submeter ao método proposto, programas que manipulam arrays, característica esta desejável nas metodologias de teste.

A metodologia proposta além de ser operacional manualmente pode tambem ser mecanizada. A mesma tem a vantagem de deixar a critério do usuário a escolha do caminho a ser testado, tarefa esta que é facilitada pela melhor visualização da estrutura do programa, fornecida pelo dígrafo acíclico correspondente. A maior vantagem da metodologia sobre as demais metodologias disponíveis, é a de se poder detetar caminhos não atravessáveis antes de que todo o caminho seja executado simbolicamente. A deteção ocorre no momento em que aparece uma combinação impossível de restrições durante a execução simbólica de um caminho.

2. REPRESENTAÇÃO GRÁFICA DE PROGRAMAS

Problemas relacionados à estrutura de programas, são problemas que podem ser vantajosamente tratados através da aplicação de resultados existentes na teoria dos grafos.

Um dos aspectos importantes da análise de um programa é o estudo de sua sequência de execução. A seguir apresentamos a representação gráfica de alguns comandos FORTRAN, os quais serão numerados para que possam ser referenciados; a cada comando corresponderá um nó e nós auxiliares serão introduzidos sempre que necessário.

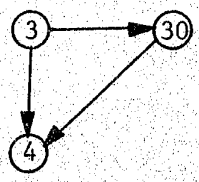
a) 3 A=B

4 B=X

b) 3 IF(A.NE.B) A=B

4 X=A+B

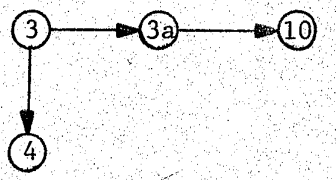
3 a é um nó auxiliar e corresponde a A=B



c) 3 IF(A.NE.B) GO TO 10

4 X=A+B

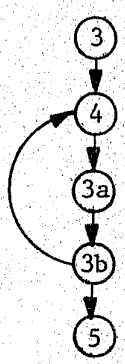
uma outra solução, seria ligar 3 a 10 diretamente sem considerar o nó auxiliar 3a



d) 3 DO 4 I=1,N

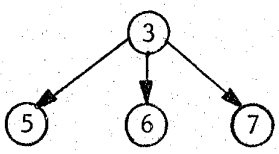
4 A(I)=0

5 ---------



O nó auxiliar 3a corresponde à incrementação de I e o 3b à comparação de I com N.

e) 3 GO TO(5, 6, 6, 7),I



2.1. VANTAGENS OBTIDAS COM ESTA REPRESENTAÇÃO

Ao representarmos um programa por seu dígrafo, uma compilação inicial do programa é realizada pois, erros como os descritos abaixo, podem ser detetados através do dígrafo.

1. Descontinuidade - Olhando-se o grafo, podemos verificar se todos os comandos pertencem a algum caminho sobre a estrutura do programa.

2. Transferências para comandos de declaração - como os comandos de declaração não são representados no dígrafo, esse tipo de erro pode ser detetado pela observação do dígrafo (verificação: existe algum arco com extremidade terminal em nó não existente?).

3. Transferência para o interior do domínio de um DO.

Exemplo 2.1 - Consideremos o subprograma da FIG. 2.1 o qual determina o menor elemento de um vetor

- SUBPROGRAMA -

```
1       FUNCTION MENOR(VETOR,N)
2       INTEGER VETOR(N)
3       IF(N.LE.0) STOP
4       MENOR=VETOR(1)
5       DO 10 I=1,N
6          IF(VETOR(I).LT.MENOR)MENOR=VETOR(I)
7  10   CONTINUE
8       RETURN
9       END
```

- DÍGRAFO CORRESPONDENTE -

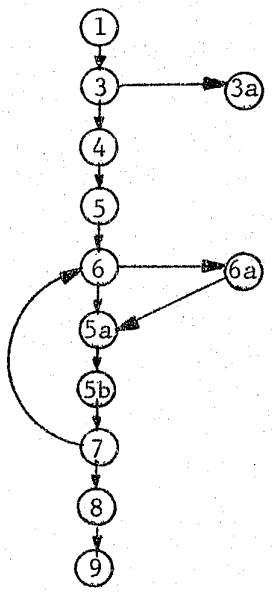


FIG. 2.1

3. EXECUÇÃO SIMBÓLICA

Executar simbólicamente um programa, significa atribuir às suas variáveis de entrada, expressões ao invés de valores durante a execução de um determinado programa.

Antes de prosseguirmos citando os tipos de execução simbólica usados em teste de programas, vamos mostrar algumas características deste tipo de execução.

Quando as variáveis de entrada de um programa são definidas como constantes numéricas no interior de um programa, a execução simbólica do mesmo deve coincidir com o resultado da execução normal do programa. Como exemplo, considere que queremos executar simbólicamente o seguinte programa, o qual calcula o perímetro e a área do triângulo de lados 20, 30 e 50.

```
1   A=20.
2   B=30.
3   C=50.
4   PERIM=A+B+C
5   S=PERIM/2
6   AREA=SQRT(S(S-A)(S-B)(S-C))
```

Executando-se simbólicamente este programa teremos:

```
1   A=20.
2   B=30.
3   C=50.
4   PERIM=100.
5   S=50.
6   AREA=0.
```

que, conforme notamos coincide com o valor que obteriamos caso o programa fosse executado normalmente.

Mas, a vantagem maior da execução, são os resultados que obtemos quando as entradas do programa são simbólicas. Por exemplo, considere que queremos analisar a estrutura de programa da FIG. 3.1 usando execução simbólica.

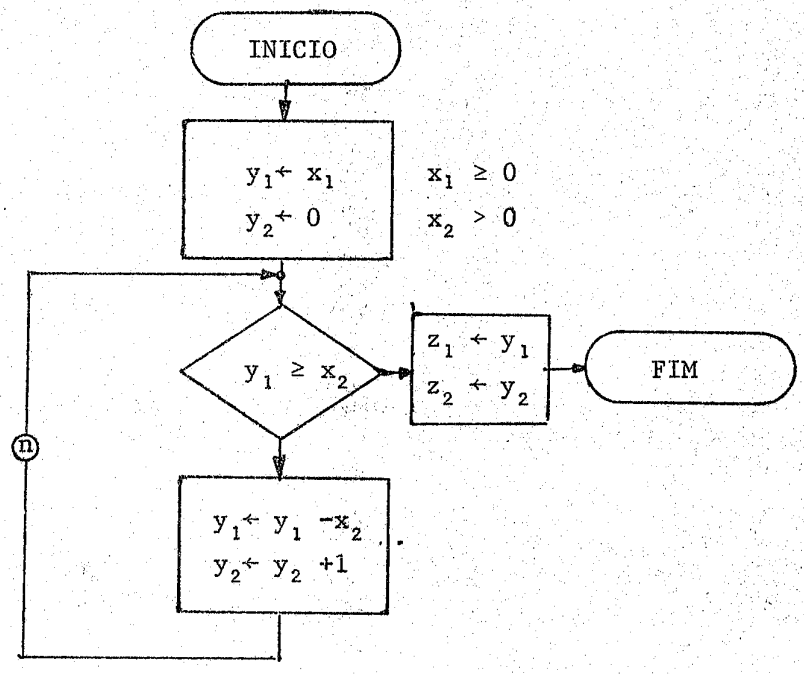


FIG. 3.1

Executando-se simbólicamente esta estrutura, vemos que os valores das variáveis y_1 e y_2 antes de entrarem no loop n são:

$$y_1(0) = x_1$$

$$y_2(0) = 0$$

e a condição inicial de parada do programa será:

$$x_1 < x_2$$

dando os seguintes valores para z_1 e z_2:

$$z_1 = x_1$$

$$z_2 = 0$$

Suponhamos agora que o loop n é executado. Na primeira iteração teremos

$$y_1(1) = y_1(0) - x_2 = x_1 - x_2$$

$$y_2(1) = y_2(0) + 1 = 1$$

Na segunda iteração

$$y_1(2) = y_1(1) - x_2 = x_1 - x_2 - x_2 = x_1 - 2x_2$$

$$y_2(2) = y_2(1) + 1 = 1 + 1 = 2$$

Na terceira iteração

$$y_1(3) = y_1(2) - x_2 = x_1 - 2x_2 - x_2 = x_1 - 3x_2$$

$$y_2(3) = y_2(2) + 1 = 2 + 1 = 3$$

Se observarmos o comportamento das variáveis do interior do loop com respeito às iterações, poderemos concluir que para uma iteração genérica n do loop, teremos as seguintes expressões para y_1 e y_2

$$y_1(n) = x_1 - nx_2$$

$$y_2(n) = n$$

as quais podemos chamar de formulas fechadas das variáveis no interior do loop.

A condição de parada deste programa será portanto

$$x_1 - nx_2 < x_2 \Rightarrow x_1 < (n+1)x_2$$

ou $$n > \left\lfloor \frac{x_1}{x_2} \right\rfloor - 1$$

o valor final de n será

$$nf = \min_n \left(n > \left\lfloor \frac{x_1}{x_2} \right\rfloor - 1\right) = \left\lfloor \frac{x_1}{x_2} \right\rfloor$$

e, consequentemente

$$y_1(nf) = x - \left\lfloor \frac{x_1}{x_2} \right\rfloor \cdot x_2$$

$$y_2(nf) = \left\lfloor \frac{x_1}{x_2} \right\rfloor$$

resultando em

$$z_1 = x_1 - \left\lfloor \frac{x_1}{x_2} \right\rfloor \cdot x_2$$

$$z_1 = \left\lfloor \frac{x_1}{x_2} \right\rfloor$$

Portanto, as expressões encontradas são os valores simbólicos finais de z_1 e z_2 resultantes da execução simbólica deste programa.

Como neste trabalho estamos interessados na execução simbólica de caminhos de programa, vamos ver agora os diferentes modos de executarmos simbolicamente um caminho.

A execução simbólica de um caminho pode ser realizada de duas maneiras: por substituição BACKWARD (do fim para o início do caminho) ou por substituição FORWARD (do inicio para o fim do caminho).

3.1. SUBSTITUIÇÃO BACKWARD

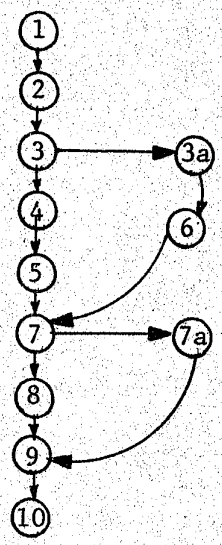
Com este tipo de substituição, o caminho a ser executado simbolicamente é percorrido da saida para a entrada do caminho, gerando restrições as quais devem ser satisfeitas para que o caminho seja executado. Como ilustração, consideremos a execução simbólica do subprograma da FIG. 3.2 por substituição backward.

SUBPROGRAMA

```
1       SUBROUTINE SUB(J,K)
2       J = J+1
3       IF(J.GT.K) GO TO 10
4          J = K-J
5          GO TO 20
6   10  J = J-K
7   20  IF(J.GT.-1) GO TO 30
8          J = -J
9   30  RETURN
10      END
```

(a)

REPRESENTAÇÃO GRÁFICA



(b)

FIG. 3.2

Vamos considerar que queremos executar o caminho 1,2,3,4,5,7,8, 9,10.

8		
↓		
7	not(J>-1)	
↓		
4	not(J>-1)	
↓		
3	not(K-J>-1)	not(J>K)
↓		
2	not(K-J>-1)	not(J+1>K)
↓		
1	not(K-J>-1)	not(J+1>K)

Portanto, teremos que satisfazer a seguinte restrição para que o caminho seja executado:

not(K-J>-1)

and not(J+1>K)

O mesmo raciocínio poderá ser empregado para execução dos outros caminhos.

Como podemos observar no exemplo da FIG. 3.2, quando executamos simbólicamente um caminho por substituição BACKWARD, comandos de atribui - ção que não afetam qualquer expressão condicional, não serão executados simbólicamente e, nenhuma memória é requerida para armazenar os valores simbólicos das variáveis.

3.2. SUBSTITUIÇÃO FORWARD

Na substituição FORWARD o caminho a ser executado simbólicamente é percorrido da entrada para a saída do caminho, gerando restrições que devem ser satisfeitas para que o caminho seja executado. Vamos ilustrar essa técnica, usando o mesmo exemplo da substituição BACKWARD (FIG.3.2).

Vamos considerar também que queremos executar o caminho 1,2,3,4, 5,7,8,9,10 e, que I1 e I2 denotem os valores de entrada dos parâmetros J e K do subprograma.

```
1
↓
2     J = I1+1
↓
3     I1+1 ≤ I
↓
4     J = I2 - (I1+1)
↓
5
↓
7     I2-I1-1 ≤ -1
↓
8     J = -(I2-I1-1)
↓
9
```

Portanto, teremos que satisfazer o seguinte par de restrições para que o caminho seja executado:

$I1+1 \leq I$

$I2-I1 \leq -1$

Novamente aqui, o mesmo raciocínio poderá ser empregado para exe

cução dos outros caminhos.

Como podemos observar, neste caso ao contrário do que foi visto para a substituição BACKWARD, todos os comandos sobre o caminho são executados simbólicamente e é necessário armazenar os valores simbólicos das variáveis, uma vez que os mesmos poderão ser necessários nas expressões condicionais.

3.3. CONSIDERAÇÕES SOBRE EXECUÇÃO SIMBÓLICA

Conforme foi visto, a execução simbólica pode ser feita através de dois tipos de substituição. No primeiro, restrições ao caminho são geradas de uma maneira "bottom-up", enquanto no segundo as mesmas são geradas de uma maneira "top-down".

Embora a substituição backward pareça ser menos complexa operacionalmente que a substituição forward, a última permite a deteção de caminhos não atravessáveis mais cedo (no momento em que ocorrer um par de restrições não consistente sobre o caminho) e, tem vantagem sobre a substituição backward na manipulação de arrays.

Na execução simbólica, loops não podem ser processados sem que o número de iterações seja precisamente conhecido. Se o número de iterações depender de algum valor de input do programa, é necessário deduzirmos uma forma fechada do valor para cada variável do interior do loop [5].

Uma outra maneira de solucionarmos este problema, é executar simbolicamente o "loop" k vezes onde, K pode ser especificado pelo usuário ou escolhido pelo sistema de geração de dados de teste quando automatizado.

Na seção seguinte, descreveremos o método proposto e, sua aplicação através de exemplos.

4. DESCRIÇÃO E APLICAÇÃO DO MÉTODO

Nesta seção descrevemos o método, através de exemplos explicativos, levando em consideração os aspectos mencionados na definição geral do método enunciada na seção 1.3. Essencialmente, o método proposto consta de 5 passos, os quais enunciamos abaixo:

PASSO 1 - Dado um programa, a ele é associado um dígrafo, conforme especificado na seção 2.

PASSO 2 - Associa-se ao dígrafo do programa, um digrafo acíclico equivalente.

PASSO 3 - O usuário escolhe qual caminho do programa deseja testar.

PASSO 4 - O caminho escolhido é executado simbólicamente usando substituição FORWARD verificando-se, durante a execução, se o caminho é atravessável.

PASSO 5 - O conjunto de restrições para que o caminho seja executado é solucionado, ocasionando um conjunto de dados de teste que executarão o caminho em questão.

A seguir aplicamos o método a exemplos:

Exemplo 4.1

Inicialmente aplicaremos o método ao exemplo da seção 3.1.

Dado o subprograma abaixo, encontrar um conjunto de dados de teste que execute pelo menos uma vez cada ramo de sua estrutura.

```
 1        SUBROUTINE(J,K)
 2        J=J+1
 3        IF(J.GT.K) GO TO 10
 4           J=K-J
 5           GO TO 20
 6   10   J=J-K
 7   20   IF(J.GT.-1) GO TO 30
 8           J=-J
 9   30   RETURN
10        END
```

PASSO1 - Representação gráfica do programa (FIG. 4.1)

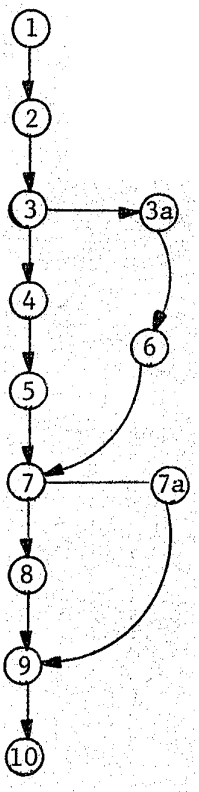


FIG. 4.1

PASSO2 - É desnecessário visto o dígrafo ser acíclico

PASSO 3 - Suponha que escolhemos o caminho 1,2,3,3a,6,7,8,9,10.

PASSO 4 - Execução simbólica do caminho

1

2 J=I1+1

3 I1+1 > I2

3a

6 J=I1+1 - I2

7 I1+1-I2 $\leq$ -1

Como já temos um par de restrições,antes de continuarmos a execução simbólica, verificaremos se elas são consistentes.

Resolvendo o sistema de inequações, verificamos que o par de i

I1+1 > I2

I1+1-I2 $\leq$ -1

nequações é inconsistente e, consequentemente o caminho não é atravessá - vel, o que dispensa a continuação da execução simbólica.

Procedendo análogamente, teremos os seguintes resultados:

- Caminho 1,2,3,4,5,7,7a,9,10

I1+1 $\leq$ I2

I2-I1-1 > -1

I1=0 e I2=1 satisfazem ao conjunto de inequações e consequentemente o caminho é atravessável para este par de valores de entrada.

- Caminho 1,2,3,3a,6,7,7a,9,10

I1+1 > I2

I1+1-I2 > -1 ou I1+1 > I2

I1=2 e I2=1 satisfazem à inequação e, consequentemente o cami - nho é atravessável para estes valores de input.

- Caminho 1,2,3,4,5,7,8,9,10

$I1+1 \leq I2$

$I2-I1 \leq -1$

Conjunto de inequações inconsistente e, consequentemente o caminho não é atravessável.

Exemplo 4.2

Bubble Sort é uma técnica através da qual uma sequência de números armazenados em um array são trocados de posição sempre que dois números adjacentes não estiverem em ordem crescente. O array é pesquisado até que todos os possíveis pares de elementos do array tenham sido comparados e possivelmente permutados.

```
C**************
C*BUBBLE SORT *
C**************
1       SUBROUTINE SORT(N,IX)
2       DIMENSION IX(20)
3       K=N-1
4   6   IF(K.LT.1)GO TO 10
5         J=1
6   7     IF(J.GT.K)GO TO 8
7           IF(IX(J).LE.IX(J+1))GO TO 9
8             M=IX(J)
9             IX(J)=IX(J+1)
10            IX(J+1)=M
11  9       J=J+1
12          GO TO 7
13  8     K=K-1
14        GO TO 6
15  10  RETURN
16      END
```

Representação gráfica do subprograma (FIG. 4.2)

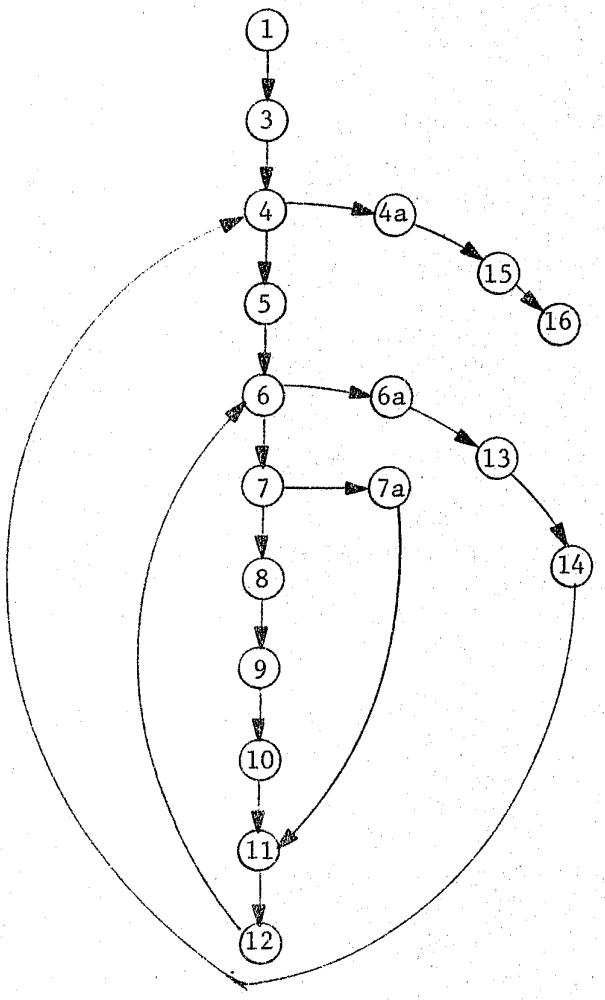


FIG. 4.2

Como o dígrafo representativo do subprograma é cíclico, vamos associar ao mesmo, um dígito acíclico equivalente, a fim de que fique mais clara a estrutura do programa.

Para traçar um dígrafo acíclico equivalente a um dígrafo cíclico, poderemos usar o algoritmo de condensação de dígrafo o qual consiste em substituir certos subgrafos por pontos ou melhor, condensando o dígrafo com respeito às suas componentes [8]. Se Si e Sj são nós do dígrafo con-

densado, a presença de uma linha de Si para Sj garantirá que cada par de pontos dentro de Si ou de Sj são mútuamente atingíveis e que cada ponto de Sj é atingido de qualquer ponto de Si.

Para ilustrar a condensação com respeito às suas componentes fortes, vamos considerar o dígrafo D da FIG. 4.3. Podemos ver que os pontos de componentes forte do dígrafo são $S_1 = \{v_1, v_2, v_3\}$ e $S_2 = \{v_4, v_5\}$ e, consequentemente a condensação D^* de D é o dígrafo contendo os pontos S_1 e S_2:

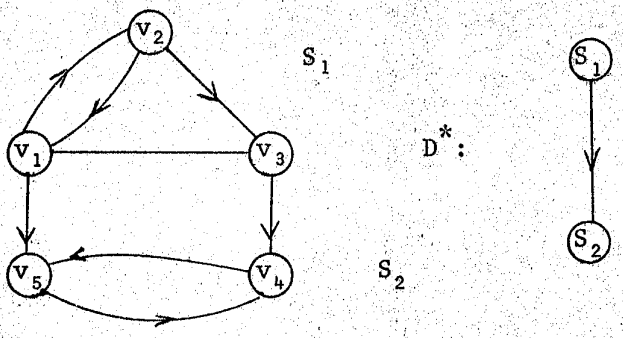


FIG. 4.3

TEOREMA: Se em um dígrafo D, v_1 e v_2 são componentes fortes de S_1 e S_2 respectivamente, então existirá um caminho de v_1 para v_2 em D, se e somente se existir um caminho de S_1 para S_2 em D^*.

Para o exemplo 4.2 teremos então o dígrafo acíclico da FIG. 4.4.

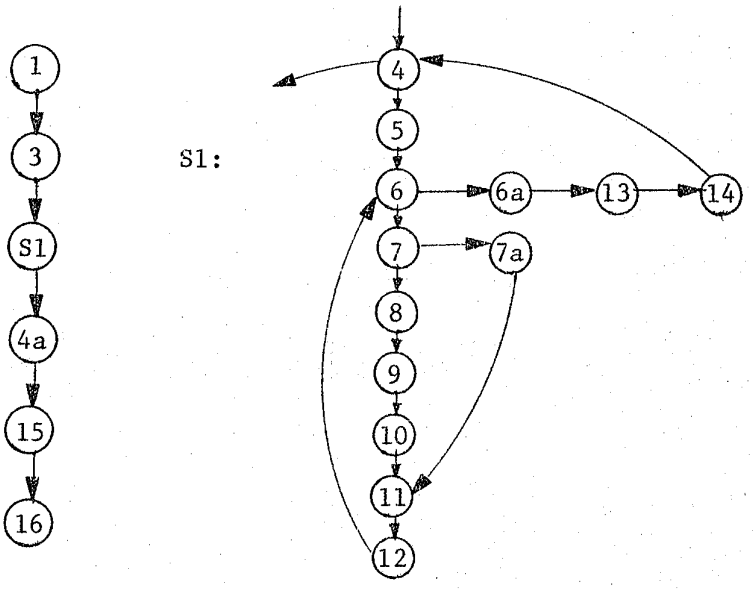


FIG. 4.4

Suponhamos agora que queremos determinar um conjunto de dados de entrada que execute o caminho 1,3,4,5,6,6a,13,14,4,4a,15,16.

Executando-se simbolicamente o caminho, encontramos o primeiro par de inequações que devem ser satisfeitas para que o caminho seja executado até o comando 6a.

$$N1-1 \geq 1$$

$$1 > N1-1$$

Como as inequações são inconsistentes, o caminho em questão não é atravessável e, a execução simbólica deve parar.

Agora, queremos um conjunto de dados de entrada que execute o caminho 1,3,4,5,6,7,8,9,10,11,12,6,6a,13,14,4,4a,15,16.

Executando simbólicamente o caminho teremos:

1		
3	K=N1-1	
4	$N1-1 \geq 1$	(a)
5	J=1	$N1-1 \geq 1$
6	$1 \leq N1-1$	(b)

Como as inequações (a) e (b) são consistentes, a execução simbólica continua.

7 IX1(1) > IX1(2) (c)

Novamente aqui (c) é consistente com N1-1 ≥ 1

8 M=IX1(1)

9 IX1(1) = IX1(2)

10 IX1(2) = IX1(1)

11 J=2

12

6 2 > N1-1 (d)

6a

Novamente aqui, (d) é consistente com N1-1 ≥ 1 e, temos o novo par de inequações

N1-1 = 1

IX1(1) > IX1(2)

13 K=N1-2

14

4 N1-2 < 1 (e)

Como (e) é consistente com N1-1 = 1, a execução simbólica continua

4a

15

Logo, para que o caminho seja executado, as inequações abaixo devem ser satisfeitas

N1-1 = 1 N1=2

ou

IX1(1) > IX1(2) IX1(1) > IX1(2)

O mesmo raciocínio pode ser empregado para os outros caminhos e, caso os mesmos sejam atravessáveis, no final da execução simbólica de cada caminho, teremos sempre um conjunto de inequações que, uma vez solucionadas nos fornecerão um conjunto de dados de teste que executarão o caminho.

5. CONCLUSÕES

O método de geração de dados de teste apresentado mostra a vantagem do uso da execução simbólica na solução de problemas encontrados no processo de geração de dados de teste. Problemas tais como os de caminhos não atravessáveis são facilmente detetados e, a manipulação de arrays é possível inclusive quando os subscritos não são conhecidos durante a execução simbólica [5].

O método proposto deixa a critério do usuário a escolha do caminho a ser testado, o que é muito importante visto o numero de caminhos possíveis em determinados programas ser muito grande.

Pelos exemplos mostrados, podemos ter a impressão de que o problema de geração de dados de teste é simples e de solução direta mas, na realidade, gerar dados de teste é uma tarefa muito difícil e a obtenção de uma solução geral é praticamente impossível, dada a necessidade da mesma solucionar muitos problemas complexos (alguns sem solução) [1],[3]. Gerar dados de teste para um caminho de programa específico, já em si um problema sem solução geral pois, para isto seria necessário um método capaz de resolver qualquer sistema de inequações, problema este não solucionável[3]. Em muitos casos, o sistema de inequações resultante da execução de um caminho é linear e, as técnicas de programação linear podem ser empregadas para solucioná-lo. Se isso não acontece, ferramentas mais poderosas tais como método do gradiente conjugado podem ser aplicadas para solucionar as

desigualdades (requer interação humana) [3],[5].

Conforme visto na seção 3, em execução simbólica são atribuidas expressões às variáveis ao invés de valores durante a execução de um caminho. Sistemas automativos de geração de dados de teste usando execução simbólica já foram desenvolvidos. O sistema SELECT gera dados de teste e cria uma representação simbólica das variáveis de saída para programas escritos em um conjunto do LISP [3],[5],[10] . O sistema EFFIGY desenvolvido por King aceita programas escritos em um subconjunto do PL/1, permitindo apenas valores inteiros [6]. Lori Clarke desenvolveu também um sistema que tenta gerar dados de testes para programas escritos em Fortran ANSI. Em [3] o autor mostra como são representadas simbólicamente as variáveis de saída dos programas, em termos de suas variáveis de entrada.

No momento, o interesse pelas ferramentas automáticas de teste de programas está restringido práticamente a aplicações espaciais e militares. Contudo, o uso de tais ferramentas pode, com o tempo, tornar-se predominante, principalmente em aplicações onde a confiabilidade seja de interesse primário.

É verdade que os métodos de teste de programas podem ser usados apenas para detetar a presença de erros mas não sua ausência, conforme observado por Dijkstra. Contudo, teste de programas é uma técnica usada por todos e, desde que não existe ainda um método que possa ser usado para mostrar que um programa está livre de erros [1], tem-se dado importância a qualquer esforço no sentido de melhorar nossa capacidade de descobrir erros através de teste de programas.

REFERENCIAS BIBLIOGRÁFICAS

[1] HUANG, J.C. - "An Approach to Program Testing", ACM Computing Surveys Vol. 7; Nº 3, Setembro de 1975.

[2] GABOW, H.N:, MAHESHWARI, S.N. e OSTERWEIL, L.J. - "On Two Problems in the Generation of Program Test Paths", IEEE Transactions on Software Engineering, Vol. SE-2, Nº 3, Setembro de 1976.

[3] CLARKE, L.A. - "A System to Generate Test Data and Symbolically Execute Programs", IEEE Transactions on Software Engineering, Vol. SE-2, Nº 3, Setembro de 1976.

[4] GOODENOUGH, J.B. & GERHART, S.L. - "Toward a Theory of Test Data Selection", IEEE Transactions on Software Engineering, Vol. SE-1, Nº 2, Junho de 1975.

[5] RAMAMOORTHY, C.V., HO, S-BF e CHEN, W.T. - "On the Automated Genera - tion of Program Test Data", IEEE Transactions on Software Engineering, Vol. SE-2, Nº 4, dezembro de 1976.

[6] KING, J.C. - "Symbolic Execution and Program Testing", Communication of the ACM, Vol. 19, Nº 7, Julho de 1976.

[7] KRAUSE, K.W., SMITH, R.W. e GOODWIN, M.A. - "Optimal Software Test Planning Through Automated Network Analysis", Proc. 1973 IEEE Symposium on Computer Software Reliability, New York, abril-maio de 1973.

[8] HARARY, F., NORMAN, R.Z. e CARTWRIGHT, D. - "Structural Models", John Wiley & Sons, Inc., 1965.

[9] FURTADO, A.L. - "Teoria dos Grafos Algoritmos", LTC Editora S.A.1973.

[10] HOWDEN, W.E. - "Reliability of the Path Analysis Testing Strategy" , IEEE Transactions on Software Engineering, Vol. SE-2, Nº 3, setembro de 1976.