Caminhos Não Executáveis no Teste de Integração: Caracterização, Previsão e Determinação

Silvia Regina Vergilio (DINF-UFPR)
José Carlos Maldonado (ICMSC-USP)
Mario Jino (DCA-FEE-UNICAMP)
Centro Politécnico - Jardim das Américas
81531-970 - C.P. 19081 - Curitiba - PR
e-mail: silvia@inf.ufpr.br

Resumo

Extensões de critérios estruturais para realizar o teste de integração têm sido propostas com o objetivo de revelar erros de interface e oferecer medidas de cobertura. Esses tipos de erros são em geral mais custosos pois, podem causar mudanças em outros programas. Um problema com esses critérios é que eles podem requerer a execução de caminhos não executáveis. A existência de caminhos não executáveis dificulta a tarefa de geração de dados de teste para satisfazer um dado critério. Este trabalho apresenta os principais resultados obtidos estudando-se a executabilidade de caminhos requeridos para realizar o teste de integração de programas extraídos de um "benchmark". Esses resultados mostram que a executabilidade de caminhos num programa pode ser dependente do contexto de outros programas. Além disso, o trabalho mostra como certas facilidades utilizadas para tratamento de não executabilidade no teste de unidade, podem ser usadas para caracterizar, prever e determinar caminhos não executáveis no teste de integração.

Abstract

Criteria for structural testing have been extended to integration testing to detect interface errors and offer coverage metrics. These kinds of errors are usually costly because they can cause changes in other programs. One problem with these criteria is that they may choose infeasible paths. Infeasible paths transform the test data generation into a difficult task to satisfy a given criterion. This paper presents the main results obtained from the study of infeasible paths, required for integration testing of programs extracted from a benchmark. The results show that path infeasibility in a program can be dependent on other programs' context. It is also shown how certain facilities to deal with infeasible paths of the unit testing can be used to classify, predict and determine infeasible paths in the integration testing.

Palavras Chave: teste de integração, critérios de teste, caminhos não executáveis.

1 Introdução

Testar um software consiste em exercitá-lo com o objetivo de encontrar erros. Técnicas de projeto de casos de teste têm a finalidade de oferecer uma maneira sistemática para projetar casos de teste efetivos, ou seja, casos de teste com alta probabilidade de revelar

erros ainda não descobertos. Segundo Pressman [16], uma estratégia para teste de software deve integrar técnicas de projeto de casos de teste numa série de passos: teste de unidade - teste de cada unidade implementada em seu código fonte; teste de integração - teste baseado no projeto e na arquitetura do software: teste de validação - validação dos requisitos de software estabelecidos na fase de análise: teste de sistema - teste do software com outros elementos do sistema.

Neste trabalho dá-se ênfase à técnica estrutural comumente utilizada para o teste de unidade. Critérios de teste estrutural exigem que elementos do grafo de fluxo de controle do programa em teste sejam exercitados (esses elementos podem ser arcos, nós, laços, caminhos, etc.). Os critérios mais conhecidos são critérios baseados em fluxo de controle [6], critérios baseados em complexidade [11] e critérios baseados em fluxo de dados [9],[17],[12],[13],[18]. Critérios estruturais têm sido estendidos para realizar o teste de integração [2],[10],[8]. Segundo Harrold [8], 40% dos erros encontrados em um programa são erros de integração. Esses erros são mais custosos pois são detectados nas últimas fases do teste e podem causar mudanças em sub-programas testados anteriormente. A extensão de técnicas estruturais permite revelar tipos de erros diferentes dos revelados por técnicas funcionais e também permite a quantificação do teste de integração através de medidas de coberturas.

Um problema com os critérios de teste estrutural, tanto no teste de unidade quanto no teste de integração, é que eles podem requerer a execução de caminhos não executáveis. Um caminho é executável se existir um conjunto de valores para as variáveis de entrada, variáveis globais e parâmetros do programa que causam a execução do caminho; se o conjunto não existir, o caminho é dito não executável. Determinar se um dado caminho é ou não executável é uma tarefa difícil pois, não existe algoritmo que dado um caminho qualquer através de um programa, determine sua executabilidade, sendo esta uma questão indecidível [3], [24]. Vários estudos considerando conjuntos de programas comprovam que a maioria deles contém caminhos não executáveis. Weyuker [24] realizou um estudo com um conjunto de 15 programas e encontrou uma média de 50% de elementos não executáveis. A mesma porcentagem também havia sido encontrada por Vergilio, Maldonado e Jino em [19] [20].

Myers [15] considera um critério de teste estrutural um ítem a ser satisfeito para se considerar a atividade de teste encerrada. A existência de caminhos não executáveis traz muitos problemas para o teste estrutural, pois dado um critério C, nem sempre é possível gerar um conjunto de casos de teste T, tal que T seja C-adequado, ou seja exercitar todos os elementos requeridos por C e alcançar uma medida de cobertura igual a 100%. Não é possível nem mesmo determinar se T existe [3]. Isso dificulta muito a automatização da atividade de teste, principalmente a tarefa de geração de dados de teste.

A problemática de caminhos não executáveis tem sido abordada predominantemente no teste de unidade, podendo ser identificadas três linhas de pesquisas: caracterizar [5] - mostrar como os caminhos não executáveis são gerados: prever [14] - estabelecer uma métrica para prever a probabilidade de um dado caminho ser não executável: determinar [3] - fornecer heurísticas para determinar não executabilidade.

Em [19] aspectos de caracterização, previsão e determinação foram explorados para o teste de unidades, no contexto de uma ferramenta de testes denominada POKE-TOOL [1], que apoia a utilização dos critérios Potenciais Usos [13], e utilizando-se um conjunto de programas ("benchmark"). O número de potenciais-du-caminhos não executáveis en-

contrados nesses programas foi maior que o esperado (1338 (56.5%) de um total de 2371). Além disso, apenas três dos 29 programas não apresentaram caminhos não executáveis. Um outro fato importante que serviu como motivação para este trabalho e demonstra a importância do teste de integração é que nos programas testados (programas bem formulados que dão ênfase em modularidade), de um total de 615 comandos. 197(33%) são chamadas de sub-programas ou comandos de "return".

O objetivo deste trabalho é, portanto, explorar a problemática de executabilidade de caminhos no teste de integração, discutindo e estendendo os conceitos utilizados no teste de unidade. Para isso, o "benchmark" aplicado e os estudos realizados anteriormente [13], [19] foram retomados considerando esse novo contexto. Na Seção 2, aspectos de executabilidade dependente do contexto de outras rotinas durante o teste de unidade, mas relacionados com o teste de integração, são discutidos. Na Seção 3, são apresentados os aspectos de caracterização, previsão e determinação de caminhos não executáveis no teste de integração. Na Seção 4 estão as conclusões e perspectivas de continuidade deste trabalho.

2 Caminhos Não Executáveis e o Teste de Unidade

Um experimento foi conduzido pelos autores [13],[19] com o objetivo de estudar a ocorrência e influência de caminhos não executáveis em programas reais. Para isso, foi utilizada a ferramenta POKE-TOOL [1] que apoia a utilização dos critérios Potenciais Usos. O experimento consistiu em realizar o teste de unidades para um conjunto de 29 programas ("benchmark"), utilizados por Weyuker [23] para avaliar sua família de critérios baseados em fluxo de dados [17].

Os Critérios Potenciais Usos [13] são critérios baseados em fluxo de dados e estão baseados no conceito de potencial uso. Executam elementos do grafo de fluxo de controle do programa que exercitam definições de variáveis. São requeridos: associações (i,j,x) ou (i,(j,k),x) estabelecidas quando há uma definição da variável x no nó i e existe no grafo um caminho livre de definição com relação a x de i para j ou (j,k); e potenciais-du-caminhos a partir do nó i, caminhos livres de laço e livres de definição com relação à variável x definida em i). Observe que para se requerer um elemento não é necessário a ocorrência de um uso explícito da variável x; daí o termo "potencial uso".

Durante a condução do "benchmark", a geração de casos de teste e a determinação dos elementos não executáveis foram realizadas manualmente. Foram construídos programas "drivers" para todos os programas testados. Esta decisão foi tomada porque o objetivo era testar separadamente cada unidade, visto que o contexto no qual a rotina é chamada pode variar. Nenhum caminho não executável dependente do contexto no qual a rotina é chamada foi caracterizado. Por outro lado, nenhum "stub" foi criado e as sub-rotinas reais foram utilizadas. Isso porque, as rotinas em teste em nenhum caso chamariam rotinas diferentes das rotinas reais utilizadas e porque estas rotinas estavam disponíveis. A executabilidade dependente do contexto de rotinas chamadas pela rotina em teste foi caracterizada como uma causa de não executabilidade dos caminhos não executáveis encontrados nos programas do "benchmark" [19].

Então, entre os caminhos requeridos por um critério para um dado programa, podem existir caminhos que são não executáveis por causa do contexto de outros programas. A executabilidade dos caminhos pode ser dependente do contexto no qual o programa em

teste é chamado. Ou pode ser dependente do contexto de programas subordinados. Frankl [3] sugere que o teste de unidade seja feito independentemente do contexto de qualquer outro programa.

A seguir serão apresentados exemplos dessas dependências. Esses exemplos foram obtidos através de novos estudos com as rotinas do "benchmark", considerando aspectos de dependência de contexto na executabilidade dos caminhos.

2.1 Dependência do Contexto no qual o Programa é Chamado

Para testar a rotina *dodash* (Figura 2.1) foi criado um programa "driver" (Figura 2.2). Também foram gerados casos de teste e todos os potenciais du-caminhos requeridos puderam ser executados (Tabela 2.1); ou seja não foram encontrados caminhos não executáveis. A rotina *dodash* é chamada por mais de uma rotina (Figuras 2.3 e 2.4). Essas rotinas são *getccl* e *makeset*, que também fazem parte do conjunto de rotinas chamadas, respectivamente, por *makepat* e *translit* . Alguns caminhos de *dodash* tornam-se não executáveis de acordo com o contexto no qual a rotina é chamada.

Suponha que *dodash* seja chamada por *makeset*, que por sua vez foi chamada por *translit* (comandos em negrito na Figura 2.4). Na rotina *makeset*, j = 0 sempre. Em dodash o parâmetro j não será redefinido no caminho 1 2 3 5 7. Portanto, a condição (*j<=0) || (src[*i+1]==ENDSTR) do "if" no nó 7 será satisfeita. O caminho 1 2 3 5 7 8 é executado e o caminho 1 2 3 5 7 9 é não executável. Assim sendo, os potenciais du-caminhos requeridos que contiverem o caminho 1 2 3 5 7 9 também serão não executáveis. Esses caminhos estão assinalados com um * na Tabela 2.1, e tinham sido executados quando *dodash* foi chamada pelo programa "driver". Por outro lado, considerando a rotina *getccl*, nota-se que o laço onde *getccl* é chamada pode ser executado várias vezes, a rotina pode incrementar j, que passará a ter um valor maior que zero. Além disto, basta tomar o cuidado para que src[*i+1] não seja ENDSTR, ou seja que ele tenha pelo menos i+2 caracteres e o caminho 1 2 3 5 7 9 poderá ser executado.

2.2 Dependência dos Módulos Subordinados

A rotina *translit*, Figura 2.4, exemplifica esse caso. O nó de um grafo de fluxo de controle associado a um programa de nome P será denotado neste trabalho pelo número do nó mais a primeira letra de P. A função *error* chamada no nó 2t (o número 2, mais a primeira letra do nome translit), contém um comando "exit" que termina a execução do programa. Portanto, todos os caminhos de translit que passam pelo nó 2t serão não executáveis. Nesse caso os caminhos poderiam ser executados se um "stub" diferente de *error* fosse criado.

Em [19], um subcaminho tal como o dado por 2t, foi caracterizado como um padrão de não executabilidade, ou seja, um padrão é dado por uma sequência de nós não executável. Qualquer caminho que contiver um padrão de não executabilidade será não executável. A seguir será dada uma classificação para os padrões de não executabilidade de um dado programa. Alguns padrões são obtidos durante o teste de unidade mas considerando as dependências da unidade em teste com relação ao contexto de outros programas.

2.3 Relação entre Padrões de Não Executabilidade e o Teste de Integração

Padrões de não executabilidade **intrínsecos** à unidade em teste são independentes do contexto de outros programas, tanto do contexto no qual a rotina é chamada quanto do contexto dos módulos subordinados. Na rotina *translit* tem-se o padrão de não executabilidade intrínseco dado por 22t 23t ... 27t 28t 29t. Em 23t (c!=ENDFILE) portanto a condição no nó 28t é satisfeita e o próximo nó executado será 17t.

Padrões de não executabilidade **extrínsecos** à unidade em teste são padrões dependentes do contexto de outros programas. Podem ser **dependentes do módulo superior**, por exemplo o padrão 1 2 3 5 7 9 na rotina *dodash* ou **dependentes do módulo subordinado**, por exemplo o padrão 7t 8t na rotina *translit*.

A determinação de padrões de não executabilidade é uma tarefa útil, pois permite que vários caminhos não executáveis sejam determinados rapidamente. Além disso, eles revelam características do programa em teste que poderão ser utilizadas no teste de integração e no teste de regressão.

Os padrões intrínsecos por serem independentes de contexto, podem ser utilizados no teste de integração da unidade com qualquer outro programa. Os padrões extrínsecos dependentes do contexto de um módulo superior, poderão ser utilizados na integração da unidade com esses módulos, numa estratégia de teste "top-down". Por exemplo, o padrão 1 2 3 5 7 9 da rotina *dodash* com o módulo *makeset*. Os padrões extrínsecos dependentes do contexto de rotinas subordinadas poderão ser utilizados numa estratégia de teste "bottom-up". Por exemplo o padrão 6t 7t, poderia ser utilizado no teste de integração das rotinas *translit* e *error*.

Os padrões de não executabilidade também poderão auxiliar a depuração e manutenção de programas, durante o teste de regressão. Os padrões intrínsecos à um programa, por exemplo, poderão ser utilizados mesmo que mudanças sejam efetuadas em módulos superiores e subordinados e só serão modificados se o programa se modificar.

Tabela 2.1. Potenciais Du-caminhos Requeridos para a Rotina Dodash

1) 1 2 19	
•2) 1 2 3 5 7 9 14 15 16 17 18 2	12) 10 11 13 15 16 17 18 2 3 5 6 17
•3) 1 2 3 5 7 9 10 11 13 15 16 17 18 2	13) 10 11 13 15 16 17 18 2 3 4 18
•4) 1 2 3 5 7 9 10 11 12 11	14) 10 11 12
5) 1 2 3 5 7 8 16 17 18 2	15) 12 11 13 15 16 17 18 2 19
6) 1 2 3 5 6 17 18 2	16) 12 11 13 15 16 17 18 2 3 5 7 9 14 15
7) 1 2 3 4 18 2	17) 12 11 13 15 16 17 18 2 3 5 7 9 14
8) 10 11 13 15 16 17 18 2 19	18) 12 11 13 15 16 17 18 2 3 5 7 8 16
9) 10 11 13 15 16 17 18 2 3 5 7 9 14 15	19) 12 11 13 15 16 17 18 2 3 5 6 17
10) 10 11 13 15 16 17 18 2 3 5 7 9 10	20) 12 11 13 15 16 17 18 2 3 4 18
11) 10 11 13 15 16 17 18 2 3 5 7 8 16	21) 12 11 12

```
void dodash(int delim, char *src, int i,
char *dest, int *j, int maxset)
/* Expande o conjunto src[i] ate' encontrar delim,
produzindo dest[j] */
1   {int k;
2    while((src[*i]!=delim)&&(src[*i]!=ENDSTR))
3    {if (src[*i]==ESCAPE)
4       addstr(esc(src,i),dest,j,maxset);
5    else if (src[*i]!=DASH)
6       addstr(src[*i],dest,j,maxset);
7       else if ((*j<0)||(src[*i+1]=ENDSTR))
8          addstr(src[*i],destj,maxset);
9       else if(isalnum(src[*i-1])&&(isalnum(src[*i+1]
              && (src[*i-1]<=src[*i=1])))
10          {
10 11 12       for(k=src[*i-1];k<=src[*i+1];k++)
12                *i=*i+1;
13             addstr(k,dest,maxset);
13          }
14          else
15             addstr(DASH,dest,j,maxset);
15 16 17 /* end if */
18    *i=*i+1;
19    }
20  }
```

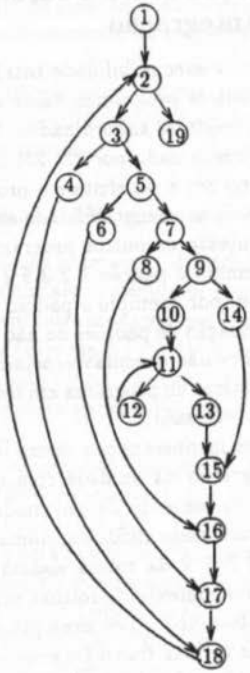

Figura 2.1: Rotina Dodash

```
void main(argc, argv)
int argc;
char *argv[];
{
   int i,j;
   char str[MAXSTR], dest[MAXSTR];
   if (argc <= 3)
   {
     printf("Erro numero de argumentos");
     exit(1);
   }
   strcpy(str,argv[1]);
   i = atoi(argv[2]);
   j = atoi(argv[3]);
   dodash(ENDSTR,str,&i,dest,&j,MAXSTR);
   dest[j] = ENDSTR;
   puts(dest);
}
```

Figura 2.2: Programa "driver" construído para a Rotina Dodash

```
int getccl(char *arg, int *i, char *pat, int *j)
/* Expande classe de char de arg[i] para pat[j] */
{ int k,jstart;
  *i = *i + 1;
  if (arg[*i] == NEGATE)
  { addstr(NCCL,pat,j,MAXPAT);
    *i = *i + 1; }
  else
    addstr(CCL,pat,j,MAXPAT);
  jstart = *j;
  addstr('0',pat,j,MAXPAT);
  dodash(CCLEND,arg,i,pat,j,MAXPAT);
  k = *j-jstart-1;
  pat[jstart] = k;
  return(arg[*i] == CCLEND);
}

void stclose(char *pat, int *j, int lastj)
/* Insere closure em pat[j] */
{ int jp,jt;
  for (jp=*j-1; jp>=lastj; jp- -)
  { jt = jp+CLOSIZE;
    addstr(pat[jp],pat,&jt,MAXPAT); }
  *j = *j+CLOSIZE;
  pat[lastj] = CLOSURE;
}
```

```
int makepat(char *arg, int start, int delim, char *pat)
/* Inicializa o vetor pat, via addstr */
{ int i,j,lastj,lj,done;
  j = 0; i = start; lastj = 0; done = 0;
  while((!done)&&(arg[i]!=delim)&&(arg[i]!=ENDSTR))
  { lj = j;
    if (arg[i] == ANY) /* "?" */
      addstr(ANY,pat,&j,MAXPAT);
    else if ((arg[i] == BOL) && (i == start))
      addstr(BOL,pat,&j,MAXPAT);
    else if ((arg[i] == EOL) && (arg[i+1] == delim))
      addstr(EOL,pat,&j,MAXPAT);
    else if (arg[i] == CCL)
      done = getccl(arg,&i,pat,&j);
    else if ((arg[i]==CLOSURE)&&(i>start))
    { lj = lastj;
      if ((pat[lj]==BOL)||(pat[lj]==EOL)||
          (pat[lj] == CLOSURE))
        done = 1;
      else
        /* incluir " * " */
        stclose(pat,&j,lastj); }
    else
    { addstr(LITCHAR,pat,&j,MAXPAT);
      addstr(esc(arg,&i),pat,&j,MAXPAT);
    }
    lastj = lj;
    if (!done)
      i++;
  }
  if ((done)||(arg[i] != delim))
    return -1;
  else if (!addstr(ENDSTR,pat,&j,MAXPAT))
    return -1;
  else
    return (i);
}
```

Figura 2.3: Rotina Makepat

```
int index(char *s; int c)
/* Retorna a posicao de c na string s */
{ int i=0;
  while ((s[i] != c) && (s[i] != ENDSTR))
    i++;
  if (s[i] == ENDSTR)
    return -1;
  else
    return(i);
}


int xindex(char* inset; int c; int allbut; int lastto)
/* Inverte o valor da funcao index, se allbut e' 1. */
{ if (c==ENDFILE)
    return -1;
  else if (!allbut)
       return(index(inset,c));
     else if (index(inset,c) ¿= 0)
          return -1;
        else
          return(++lastto);
}


int makeset(char *inset; int k; char *outset;
int maxset)
1m   { int j=0;
1m    dodash(ENDSTR,inset,&k,outset,&j,maxset);
1m    return(addstr(ENDSTR,outset,&j,maxset));
1m   }
```

```
void translit()
1t  { char fromset[MAXSTR-1], toset[MAXSTR-1],
        arg[MAXSTR-1];
1t    int i,c,lastto, allbut, squash;
1t    if (!getarg(1,arg))
2t      error ("usage: translit from to");
3t    allbut = (arg[0] == NEGATE);
3t    if (allbut)
4t      i = 1;
5t    else
5t      i = 0;
6t    if (!makeset(arg,i,fromset,MAXSTR))
7t      error("translit: from set too large");
8t    if (!getarg(2,arg))
9t      toset[0] = ENDSTR;
10t   else if (!makeset(arg,0,toset,MAXSTR))
11t        error("translit: to set too large");
12t     else if ((strlen(fromset)) < (strlen(toset)))
13t        error("translit: from shorter than to");
14t
15t   /* end if */
16t
17t   lastto = strlen(toset) - 1;
17t   squash = ((strlen(fromset)>lastto+1)||(allbut));
17t   do {
18t     i = xindex(fromset,c=getchar(),allbut,lastto);
18t     if ((squash) && (i>=lastto) && (lastto+1>0))
19t     { putchar(toset[lastto]);
19t       do
20t       i = xindex(fromset,c=getchar(),allbut,lastto);
20t       while (i>=lastto);
20t     }
21t
22t     if (c != ENDFILE)
23t     { if ((i>=0) && (lastto+1>0))
24t         putchar(toset[i]);
25t       else if (i==-1)
26t            putchar(c);
27t     }
28t   }while (c!=ENDFILE)
29t }
```

Figura 2.4: Rotina Translit

3 Caminhos Não Executáveis no Teste de Integração

Esta seção tem como objetivo apresentar a problemática de caminhos não executáveis no teste de integração. Primeiramente será apresentada uma proposta de extensão do teste estrutural para apoiar o teste de integração [10]. Depois, serão apresentados aspectos de não executabilidade de caminhos interprocedurais.

3.1 Aplicação de Critérios Estruturais no Teste de Integração

A técnica de teste estrutural comumente utilizada para realizar o teste de unidade tem sido estendida para realizar o teste de integração [2],[10],[8]. A maioria dos critérios propostos procura testar as interfaces entre os módulos sendo de especial interesse os comandos de chamadas de sub-programas.

Em [10] propõem-se critérios baseados em grafos de chamadas como extensão para os critérios baseados em fluxo de controle, com o objetivo de testar as relações entre os módulos, isto é, testar os comandos de chamadas de funções. Também foram propostos critérios baseados em fluxo de dados interprocedural para testar as interfaces, isto é, testar os valores de parâmetros e variáveis globais do programa.

As interfaces entre os módulos são determinadas pelos dados que são usados em ambos os procedimentos e que podem afetar a comunicação entre os módulos: ou seja, por parâmetros e por variáveis globais. Para teste de integração, realiza-se uma análise de fluxo de dados interprocedural e selecionam-se aqueles caminhos (ou associações) que influenciam diretamente a comunicação entre os procedimentos:

— Para as variáveis x usadas como entradas em um sub-programa, são considerados os caminhos compostos de sub-caminhos de uma definição de x precedendo a chamada desse sub-programa até o ponto dessa chamada, concatenados com sub-caminhos do nó de entrada do sub-programa chamado para uma referência a x.

Exemplo1: Considere os programas *compress* e *putrep* (Figura 3.1). A variável n definida no nó 1c, é uma variável de entrada para o programa *putrep* chamado em 5c. Essa variável é referenciada em *putrep* no nó 1p. Portanto a associação (1c,1p,n) e o caminho 1c 2c 3c 4c 5c 1p serão requeridos. Note que o caminho de 1c a 1p é livre de definição com relação a variável n.

— Para as variáveis x que são usadas como saídas, são considerados os caminhos compostos de sub-caminhos de uma definição de x no sub-programa chamado até o ponto de return desse sub-programa concatenados com sub-caminhos do comando de chamada para uma referência a x no programa que chama o sub-programa.

Exemplo2: Considere ainda os programas da Figura 3.1. A variável n definida no nó 2p é uma variável de saída para *putrep* e referenciada no nó 4c de *compress*. Portanto a associação (2p,4c,n) e o caminho 2p 3p 4p 5p 7c 14c 15c 2c 3c 4c serão requeridos. Note que o caminho de 2p a 4c é livre de definição com relação a n.

Os problemas relativos à não executabilidade encontrados no teste de unidade também são encontrados no teste de integração. Uma das associações (caminhos) dos exemplos acima é não executável. Qual ? A resposta será dada a seguir.

3.2 Caminhos Interprocedurais Não Executáveis

O caminho 2p 3p 4p 5p 7c 14c 15c 2c 3c 4c e a associação (2p,4c,n) são não executáveis, pois, em 7c, (c=ENDFILE), em 15c (lastc=c=ENDFILE) e o caminho 2c 16c será executado pois a condição do "while" em 2c não será mais satisfeita. Para cobrir a associação (2p,4c,n) é necessário executar o caminho 2p 3p 5p 7c 15c 2c 3c 4c, sendo que os laços dos nós 3p e 2c poderão ser executados zero ou mais vezes através de caminhos que não redefinam n. Mas todos os caminhos assim obtidos são não executáveis.

Define-se que um caminho através de vários programas (interprocedural) é não executável se não existir atribuição de valores para as variáveis de entrada, variáveis globais e parâmetros de todos os programas envolvidos, que executa o caminho dado. Uma associação interprocedural será não executável se não existir caminho executável que a cobre.

3.3 Caracterização

Como visto na Seção 2.3, um caminho interprocedural pode ser não executável por conter um padrão de não executabilidade intrínseco a algum dos programas envolvidos.

Exemplo1: O caminho 2p 3p 4p 5p 7c 14c 15c 2c 3c 4c da seção anterior, envolvendo os programas *compress* e *putrep*, é não executável porque contém o padrão de não

executabilidade 7c 14c 15c 2c 3c intrínseco à rotina *compress*.

Outros caminhos interprocedurais são não executáveis por conterem padrões extrínsecos a um programa. Nesse caso, o programa real cujo contexto originou a não executabilidade já deverá ter substituído o "driver" ou "stub", no caso de não executabilidade dependente do programa superior e subordinado, respectivamente.

Exemplo2: Seja o caminho interprocedural 1m 1d 2d 3d 5d 7d 9d 10d 11d 12d, envolvendo as rotinas *dodash* e *makeset*. Em 1m existe uma definição de j que será exercitada em 12d. Esse caminho é não executável pois 1d 2d 3d 5d 7d 9d é um padrão de não executabilidade extrínseco para *dodash*, dependente do módulo superior *makeset*.

Exemplo3: Considere a rotina *translit*, o parâmetro real fromset é um parâmetro para a rotina *makeset* no nó 6t, correspondente ao parâmetro formal outset definido em 1m. Em 16t fromset é utilizado e caminhos livres de definição de 1m para 6t e de 6t para 16t devem ser requeridos. Um desses caminhos é 1m 6t 7t 8t 9t 16t, que é não executvel por conter o padrão extrínseco 7t 8t à *translit*, dependente do módulo subordinado *error*.

As mesmas causas que geraram esses padrões de não executabilidade encontrados durante o teste de unidade também são causas de não executabilidade para os caminhos interprocedurais não executáveis do teste de integração. Em [19] as causas de não executabilidade encontradas nos programas do "benchmark" são apresentadas em categorias. Entre esses causas tem-se: 1) laços que serão executados pelo menos uma vez; 2) testes com variáveis exigindo que elas tenham valores diferentes dos que lhe foram atribuídos (muitas vezes essas variáveis controlam laços); 3) condições dependentes cuja a satisfação de uma implique a satisfação (ou não) da outra. Para exemplificar os casos 2) e 3) considere respectivamente os exemplos 2 e 1 dados acima.

Essas causas poderão gerar também um padrão de não executabilidade interprocedural. Considere o novo programa *makeset* dado pela Figura 3.2. Note que para esse novo programa o padrão não executável extrínseco à *dodash* 1d 2d 3d 5d 7d 9d, dependente do contexto de *makeset*, não será estabelecido durante o teste de unidade. Mas o padrão interprocedural 2m ... 1d 2d 3d 5d 7d 9d é estabelecido.

Em resumo, as mesmas causas de não executabilidade encontradas em caminhos durante o teste de unidade são encontradas para os caminhos interprocedurais. É como se uma extensão do código fosse realizada e um só programa fosse considerado. Assim sendo, é de se esperar que as mesmas técnicas utilizadas para previsão e determinação de não executabilidade utilizadas no teste de unidade, possam ser utilizadas também no teste de integração.

3.4 Previsão e Determinação

Em [14] foi proposta a utilização do número de predicados de um caminho como uma heurística para prever a probabilidade do caminho ser ou não executável. Malevris et al propõe e justifica estatisticamente, utilizando um conjunto de rotinas FORTRAN, que quanto maior o número de predicados de um caminho maior a dificuldade para que estes predicados sejam consistentes entre si e portanto maior a probabilidade do caminho ser não executável.

Em [19] e [21], são apresentados resultados de um estudo análogo para as rotinas que compõem o "benchmark", que comprovam as afirmações de Malevris no contexto de critérios baseados em Fluxo de Dados. Um estratégia baseada nesses resultados foi

proposta para gerar dados de teste para satisfazer os critérios baseados em fluxo de dados [22].

Nota-se que muitos dos predicados ao longo de um caminho interprocedural não executável não podem ser satisfeitos, devido a inconsitências. Por isso, um estudo sobre a influência do número de predicados na executabilidade de um caminho merece ser feito, como tentativa de prever quais caminhos têm maior probabilidade de serem não executáveis e quais devem ser selecionados para executar associações interprocedurais.

A heurística de Frankl [3] é utilizada para determinar associações não executáveis e utiliza técnicas de fluxo de dados e de execução simbólica. Em [19] podem ser encontrados aspectos de implementação e extensões para essa heurística. Uma associação interprocedural não executável será não executável se todos os caminhos que a cobrem forem não executáveis. Aplicando-se a heurística de Frankl, no teste de integração, primeiramente, são determinados os caminhos candidatos a cobrir a associação dada – caminhos interprocedurais livres de definição com relação às variáveis da associação – e depois, os caminhos candidatos não executáveis são eliminados. Se todos os caminhos candidatos forem eliminados a associação será não executável. Para eliminar os caminhos candidatos não executáveis os laços que são executados pelo menos uma vez são analisados para ver se existem caminhos livres de definição com relação às variáveis da associação através desses laços, que redefinam as variáveis do predicado do laço.

Um exemplo de aplicação dessa heurística está na Figura 3.3. Esta figura contém o grafodef da rotina getlis. O grafodef é o grafo de fluxo de controle da rotina que contém as variáveis definidas em cada nó do grafo. No nó 1, a variável global nlines é definida. Essa definição poderá ser utilizada na rotina que chama *getlis*, e uma associação interprocedural é estabelecida. No nó 2 existe um comando "while" cuja condição (!done) é satisfeita e esse laço será executado pelo menos uma vez; portanto os caminhos candidatos que contiverem o subcaminho 1 2 9 podem ser eliminados. Restam os caminhos que percorrem o laço. Mas entre esses, não existe nenhum caminho livre de definição com relação a nlines que redefine a variável done, pois nlines é definida no nó 3. Portanto a associação é não executável.

```
void compress()

1c { int n,lastc, c:
1c     n = 1;
1c     lastc = getchar();
2c     while(lastc!=ENDFILE)
3c     { if ((c=getchar())==ENDFILE)
4c         if ((N>1)||(lastc==WARNING))
5c           putrep(n,lastc);
6c         else
6c           putchar(lastc);
7c
8c       else
8c       {if (c==lastc)
9c          n++;
10c       else
10c          if ((n>1)||(lastc==Warning))
11c          { putrep(n,lastc);
11c            n = 1;
11c          }
12c         else
12c          putchar(lastc);
13c      }
14c
15c      laastc=c;
15c    }
16c }
```

```
void putrep(n,c)
int n, c;
1p { int i;
1p while((n>=THRESH)——((c==WARNING)&&(n>0)))
2p { putchar(WARNIG);
2p     putchar(min(n,MAXREP)-1+'A');
2p     putchar(c);
2p     n= n - MAXREP;
2p }
3p for (i=n;i>0;j-)
4p     putchar(c);
5p }
```

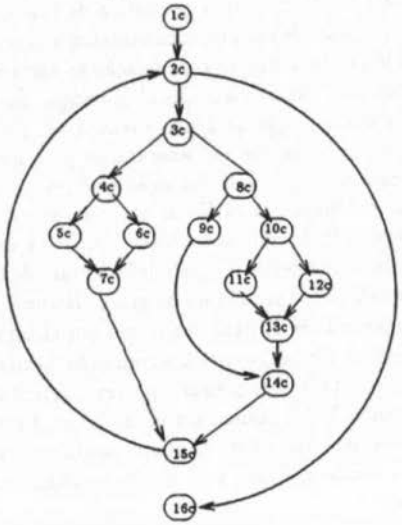

Figura 3.1: Rotina Compress

```
int makeset(char *inset, int k, int flag, char *outset;
int maxset)
1m  { int j;
1m    if (flag)
2m      j = 0;
3m    else
3m      j = 1;
4m    dodash(ENDSTR,inset,&k,outset,&j,maxset);
4m    return(addstr(ENDSTR,outset&j,maxset));
4m  }
```

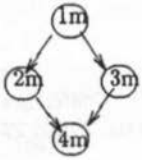

Figura 3.2: Rotina Makeset modificada

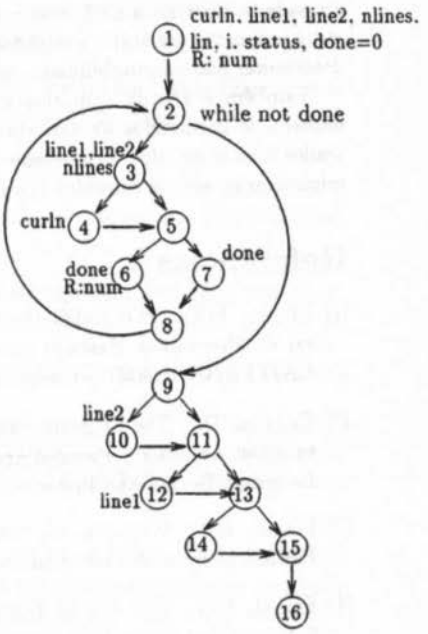

Figura 3.3: Rotina Getlis

4 Conclusões

Esse trabalho procurou mostrar os principais resultados de um estudo com um conjunto de rotinas ("benchmark") sobre a problemática de caminhos não executáveis no teste de integração. Mostrou-se através de exemplos, como o contexto de outros programas pode gerar padrões de não executabilidade e consequentemente, caminhos não executáveis no teste de unidade. Esses padrões foram classificados em intrínsecos e extrínsecos à unidade em teste.

O uso de padrões de não executabilidade permite a determinação de muitos caminhos não executáveis de uma vez: além disso, os padrões revelam características de programa que podem ser utilizadas no teste de regressão durante as atividades de depuração e manuntenção de programas. Além disso, poderão ser utilizados no teste de integração pois, muitos dos caminhos interprocedurais não executáveis contêm os padrões determinados durante o teste de unidade: outros são não executáveis devido a a padrões de não executabilidade interprocedurais, estabelecidos no teste de integração.

As causas de não executabilidade que geraram esses padrões são as mesmas que foram determinadas em [19] para o teste de unidade. Assim sendo, pode-se concluir que as mesmas facilidades e abordagens para tratamento de não executabilidade no nível de

unidade (a heurística de Frankl e extensões, eliminação de padrões, execução simbólica, etc.) propostas em [19] e ilustradas nesse trabalho, podem ser facilmente aplicadas para determinar não executabilidade, no nível de integração.

Também é grande a motivação para num trabalho futuro estudar a influência do número de predicados na executabilidade de um caminho interprocedural e então, baseados nesses estudos, estabelecer um método de geração de dados de teste que vise a minimizar os efeitos causados por caminhos não executáveis no teste de integração.

Referências

[1] Chaim, M.L., POKE-TOOL - Uma Ferramenta para Suporte ao Teste Estrutural de Programas Baseado em Análise de Fluxo de Dados. *Tese de Mestrado*, DCA/FEE/UNICAMP - Campinas, SP, Abril 1991.

[2] Callahan D., "The Program Summary Graph and Flow-Sensitive Interprocedural Data Flow Analysis", Proceedings of the SIGPLAN'88, Conference on Programming Language Design and Implementation. Atlanta, Georgia, Junho 22-24, 1988.

[3] Frankl, F.G.; Weyuker, E., "Data Flow Testing in The Presence of Unexecutable Paths", in Proc Workshop on Software Testing, Banff, Canada, Julho 4-13, 1986.

[4] Frankl, F.G., The Use of Data Flow Information for Selection and Evaluation of Software Test Data, PhD Dissertation, New York, Oct., 1987.

[5] Hedley, D.; Hennell, A., "The Causes and Effects of Infeasible Paths in Computer Programs", Proc. 8th ICSE, London, UK, pp 259-266, 1985.

[6] Huang, J.C.; "An Approach to Program Testing", ACM Computer Surveys, Vol. 7, No 3, pp. 114-128, Set., 1975.

[7] Kernighan, B.W.; Plauger, P.J., Software Tools in Pascal. Addison-Wesley Publishing Company, Reading, Massachusetts, 1981.

[8] Harrold, M.J.; Soffa, M.L., "Selecting and Using Data for Integration Testing", IEEE Software, pp 58-65, Março 1991.

[9] Laski, J.W.; Korel, B., "A Data Flow Oriented Program Testing Strategy", IEEE Trans.on Software Eng., 9(3), 347-354, Maio, 1983.

[10] Linnenkugel, V.; Müllerburg, M., "Test Data Selection Criteria for (Software) Integration Testing", Proceedings of The First International Conference on Systems Integration. Morristown, New Jersey, Abril 23-26. 1990.

[11] McCabe, T.; "A Software Complexity Measure", IEEE Trans.on Software Eng., 2(6), 308-320, Dez., 1976.

[12] Maldonado, J.C; Chaim, M.L.; Jino, M., "Seleção de Casos de Teste Baseada nos Critérios Potenciais Usos", II Simpósio Brasileiro de Engenharia de Software. Canela, RS, Out., 1988.

[13] Maldonado, J.C., Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software. *Tese de Doutorado*, DCA/FEE/ UNICAMP - Campinas. SP, Julho, 1991.

[14] Malevris, N.; Yates, D.F.; Veevers, A., "Predictive Metric for Likely Feasibility of Program Paths", Information and Software Technology, 32(2), Março, 1990.

[15] Myers, G.J., The Art of Software Testing, Wiley, 1979.

[16] Pressman, R.B., Software Engineering: a Practitioner's Approach, Third Edition, New York, McGraw-Hill, 1992.

[17] Rapps, S.; Weyuker, E.J., "Data Flow Analysis Techniques for Test Data Selection". *in* Proc. Int. Conf. Software Engineering, Tokyo, Set., 1982.

[18] Ural, U.; Yang, B., "A Structural Test Selection Criterion", Information Processing Letters, Julho., 1988.

[19] Vergilio, S.R., Caminhos Não Executáveis: Caracterização, Previsão e Determinação para Suporte ao Teste de Programas. *Tese de Mestrado*, DCA/FEE/UNICAMP, Campinas, SP, Jan. 1992.

[20] Vergilio, S.R.; Maldonado, J.C.; Jino, M. "Caminhos Não Executáveis na Automação das Atividades de Teste", VI Simpósio Brasileiro de Engenharia de Software, Gramado, RS, Nov. 1992.

[21] Vergilio, S.R.; Maldonado, J.C; Jino, M., "Influência do Número de Predicados na Executabilidade de Um Caminho no Contexto de Teste Baseado em Fluxo de Dados", XIX Conferencia Latinoamericana de Informatica, Buenos Aires, Argentina, Março, 1993.

[22] Vergilio, S.R.; Maldonado, J.C; Jino, M., "Uma estratégia de Geração de Dados de Teste", VII Simpósio Brasileiro de Engenharia de Software, Rio de Janeiro, RJ, Out. 1993.

[23] Weyuker, E.J., "The Cost of Data Flow Testing: An Empirical Study", IEEE Trans. on Software Eng. SE-16(2), 12-18, Fev., 1988.

[24] Weyuker, E. J. "More Experience with Data Flow Testing", IEEE Trans. on Software Eng., Vol. 19, 912-919, Set., 1993.