UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

CRISTIANE SELEM FERREIRA NEVES

Proposta de um novo método estatístico para geração de conjuntos de teste com uma dada probabilidade de cobertura

Prof. Dr. Éber Assis Schmitz
Orientador

Prof. Dr. Fábio Protti
Co-orientador

Rio de Janeiro, Junho de 2011

Ficha Catalográfica

Neves, Cristiane Selem Ferreira

Proposta de um novo método estatístico para geração de conjuntos de teste com uma dada probabilidade de cobertura / Cristiane Selem Ferreira Neves. – Rio de Janeiro: UFRJ IM, 2011.

59 f.: il.

Dissertação (Mestrado em Informática) – Universidade Federal do Rio de Janeiro. Programa de Pós-Graduação em Informática, Rio de Janeiro, BR–RJ, 2011.

Orientador: Éber Assis Schmitz; Co-orientador: Fábio Protti.

I. Schmitz, Éber Assis. II. Protti, Fábio. III. Proposta de um novo método estatístico para geração de conjuntos de teste com uma dada probabilidade de cobertura.

Proposta de um novo método estatístico para geração de conjuntos de teste com uma dada probabilidade de cobertura

Cristiane Selem Ferreira Neves

Dissertação de Mestrado submetida ao Corpo Docente do Departamento de Ciência da Computação do Instituto de Matemática, e Núcleo de Computação Eletrônica da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários para obtenção do título de Mestre em Informática.

Aprovado por:

_____

Prof. Dr. Éber Assis Schmitz (Orientador)

_____

Prof. Dr. Fábio Protti (Co-orientador)

_____

Profa. Dra. Jonice de Oliveira Sampaio

_____

Prof. Dr. Felipe Maia Galvão França

_____

Prof. Dr. Leonardo Gresta Paulino Murta

Rio de Janeiro, Junho de 2011

Este trabalho é dedicado à memória das minhas avós queridas Maria Marre Neves e Wadad Saliba, que nem sequer viram a minha formatura da graduação, mas estariam muito orgulhosas do sucesso da netinha.

AGRADECIMENTOS

Em primeiro lugar agradeço a Deus, o criador de todas as coisas, razão da existência do mundo e de todos os seres que nele habitam; a Jesus, seu filho único, que morreu por nós na cruz para a nossa salvação e a Maria, que na sua humildade de serva intercede por nós, seus filhos, junto ao Pai, nas nossas orações. Vocês são o meu exemplo, o meu espelho e a razão de tudo que sou até hoje.

Agradeço também aos meus pais, Paulo e Ivone, que sempre me incentivaram desde pequena a conquistar todos os meus objetivos. Obrigada por estarem sempre presentes na minha vida.

Agradeço ao meu noivo Luís Daniel, que nesses sete anos mostrou todo o seu amor, carinho, honestidade e presença nos momentos mais difíceis para mim. Obrigada pelos ótimos conselhos, pela paciência nos meus momentos de tensão e por compreender as minhas ausências por causa do trabalho e do mestrado.

Não posso deixar de agradecer também ao meu gerente na Accenture, Moisés Amaro, que não só escreveu a minha carta de recomendação ao mestrado, como também me liberou para as aulas e reuniões da dissertação durante esses dois anos e meio. Obrigada pela confiança que você sempre depositou no meu trabalho.

Agradeço também a minha família e a todos os amigos e colegas de trabalho que sempre se importaram com a minha carreira e me incentivaram a seguir em frente, mesmo com os obstáculos. São muitos os nomes, não daria para listar todos aqui, mas saiba que vocês tem um lugar reservado no meu coração.

Por fim, agradeço aos meus orientadores Éber e Fábio, por toda a dedicação que depositaram neste trabalho e pelos ótimos ensinamentos. Este sucesso também é de vocês!

RESUMO

O *basis path testing* é uma das mais usadas técnicas para a geração de casos de teste Caixa-Branca. Esta abordagem depende do estado desconhecido de variáveis do programa em tempo de execução, i. e., cada caminho pertencente ao conjunto base pode ser visto como o resultado de um experimento aleatório, associado com uma probabilidade de execução. Podemos definir a cobertura de probabilidade de um conjunto de teste como a soma das probabilidades de execução dos seus membros. Embora executar o programa um número suficientemente grande de vezes possa nos fornecer uma aproximação para esse conjunto, seu custo computacional seria praticamente igual a testar o programa em sua totalidade. Esta dissertação apresenta um método que usa um conjunto pequeno de amostras de execução para selecionar um conjunto pequeno de caminhos de execução, que tem a propriedade de sua probabilidade de cobertura estar acima de um nível de confiança desejado, e então gerar uma especificação, em linguagem natural, do conjunto de casos de teste. Resultados experimentais mostram que ele não é apenas simples de ser aplicado mas também gera um conjunto de casos de teste confiável. O grande benefício, entretanto, é o uso de um conjunto pequeno de casos de teste, que reduz o esforço computacional para garantir que a confiança do programa está acima de um determinado nível.

Palavras-chave: Grafo de fluxo, cobertura de probabilidade, nível de confiança.

Towards a new statistical method for generating test sets with a given coverage probability

ABSTRACT

The basis path testing is one of the most used techniques for generating white box test cases. This approach depends on the unknown state of the program variables at execution time, i.e., each path can be seen as the result of a random experiment, associated with an execution probability. We can define the coverage probability of a test set as the sum of the execution probability of its members. Although running the program a large number of times provide us with an approximation for this set, its computational cost would be in practice equal to that of testing the program itself. This paper presents a method that uses a small set of execution samples to select a small set of execution paths, which has the property of having coverage probability above a required confidence level, and then generate a natural language specification of the test case set. Experimental results show that the method is not only simple to be applied but also generates a reliable test case set. The greatest benefit, however, is the use of a small set of test cases, which reduces the computational effort to guarantee that the program reliability is above a required level of confidence.

keywords: flow graph, coverage probability, confidence level.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AOP	Aspect-Oriented Programming
ACC	AspeCt-oriented C compiler
VV&T	Validação, Verificação e Teste

SUMÁRIO

1 INTRODUÇÃO

O objetivo do teste de software é fornecer aos desenvolvedores informações sobre a qualidade do produto ou serviço sob teste. A qualidade pode estar comprometida se o produto ou serviço apresentar problemas. Conforme orientação da VV&T, existem nomeclaturas diferentes para o momento em que o problema for identificado. Segundo [Pressman 2001], um erro é um problema de qualidade encontrado antes que um software seja entregue aos usuários finais. Um defeito é um problema de qualidade encontrado somente depois que o software foi entregue aos usuários finais.

O teste Caixa-Branca de módulos de software é a atividade de teste onde os casos de teste são derivados a partir da análise do código-fonte do módulo [Janardhanudu 2009, Pressman 2001, Singh e Rakesh 2010]. Uma das mais usadas técnicas para a geração de casos de teste Caixa-Branca é conhecida como *basis path testing*, proposta inicialmente por McCabe [McCabe 1976]. A essência dessa técnica é a representação do código-fonte como um tipo de grafo chamado grafo de controle de fluxo. O grafo de controle de fluxo é um grafo direcionado onde nós representam computações e arestas a transferência do controle. A partir desse grafo, um conjunto de caminhos base é identificado e um caso de teste é gerado para cada um. Isso fornecerá um conjunto mínimo que cobrirá todos os cenários de execução não redundantes para o

módulo, já que elimina a partir do conjunto de teste todos os caminhos provenientes de repetições ou *loops*.

A avaliação dos nós de loop ou decisão depende do estado desconhecido das variáveis em tempo de execução. Isso implica que cada elemento do conjunto base seja associado com uma probabilidade de execução, como uma cadeia de Markov. Como a cobertura obtida a partir da abordagem do conjunto de caminhos de teste base é probabilística, a probabilidade de cobertura para um dado grafo de controle de fluxo pode estar abaixo de um nível de confiança desejado. Em outras palavras, o uso desta técnica pode potencialmente gerar um conjunto de caminhos-base de teste com chance pequena de ser executado na prática.

Quando o objetivo é definir um conjunto de teste tal que a probabilidade de cobertura esteja acima de um nível de confiança desejado, devemos tomar uma abordagem diferente da de [McCabe 1976]. A intenção agora é encontrar o conjunto de casos de teste através de um pequeno conjunto de caminhos de teste que garanta a probabilidade mínima necessária de ser executado. Se a distribuição de probabilidade dos caminhos de execução fosse conhecida, seria um problema simples selecionar um conjunto mínimo que leva ao nível de confiança desejado. Como a distribuição é desconhecida, poderíamos usar, em vez disso, a distribuição de probabilidade dos caminhos obtidos pela execução do programa um número muito grande de vezes, digamos N, que convergeria na distribuição real dos caminhos executados. Mas esta solução requereria um esforço computacional praticamente igual ao do teste do software na sua totalidade.

Esta dissertação apresenta uma proposta para um método eficiente de encontrar o conjunto de casos de teste através da seleção de um conjunto pequeno de caminhos, que satisfaça a probabilidade de cobertura desejada, usando a combinação de amostras de execução de tamanho n ($n << N$) com um algoritmo generativo, de tal

forma que o conjunto pode ser obtido de uma forma eficiente. O método foi testado num conjunto de quatorze programas e os resultados mostraram que o conjunto dos caminhos produzidos é estatisticamente similar aos obtidos pela execução de um amostra grande. Em outras palavras, economizamos tempo de execução porque obtemos um conjunto similar a ser testado com um esforço menor para ser gerado.

A estrutura da dissertação é dada a seguir. O Capítulo 2 apresenta os conceitos mais importantes envolvendo testes Caixa-Branca, testes Qui-Quadrado, testes Kolmogorov-Smirnov e representação de programas por grafos. O Capítulo 3 descreve em detalhes o método, enquanto o Capítulo 4 apresenta os experimentos computacionais, comparados a outros trabalhos relacionados. Conclusões são discutidas no Capítulo 5.

2 FRAMEWORK CONCEITUAL

Este capítulo é uma revisão da literatura dos principais conceitos que são a base para o entendimento deste trabalho. Suas seções estão divididas em Representação de grafo de fluxo de controle, Path based testing e Caminhos probabilísticos.

2.1 Representação do grafo de fluxo de controle

Um módulo de programa M sob teste pode ser representado por um grafo de fluxo de controle. O grafo de fluxo de controle é um grafo direcionado, que possui um único nó de entrada e geralmente vários nós de saída. Cada nó no grafo corresponde a um bloco básico do código no programa e as arestas correspondem a desvios tomados no programa. Um exemplo (veja [McCabe 1976]) é o grafo na Figura 2.1.

O grafo de fluxo de controle pode conter quatro tipos de nós: nós de atividade, nós de junção, nós de decisão e nós de loop. Cada nó de atividade representa uma sequência de sentenças de código-fonte executadas como um bloco e podem ter várias entradas e saídas [Barbosa et al. 2009], veja Figura 2.2. O nó de junção tem um papel sintático de unir vários desvios provenientes de vários lugares diferentes (suas

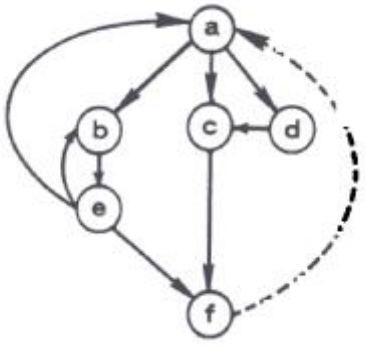

Figura 2.1: Exemplo de grafo de fluxo de controle

entradas) em apenas uma saída, veja Figura 2.3. Nós de decisão têm uma entrada e várias saídas, veja Figura 2.4. Como veremos, cada saída será associada a uma probabilidade.

O nó de loop tem uma entrada e duas saídas, uma direcionada para um nó fora do loop, e outra direcionada para uma nova iteração do loop, veja Figura 2.5. Ele também é associado a probabilidades de saída do loop ou de entrada em uma nova iteração. As estruturas de programação *while-do* e *do-while* associadas a condições booleanas serão tratadas como nós de loop, enquanto estruturas *for-do* serão tratadas como nós de atividade, em função de executarem um número pré-definido de vezes, sem condições booleanas. Se por um acaso uma estrutura *for-do* possuir uma condição booleana, ela deverá ser transformada numa estrutura *while-do* ou *do-while* para a aplicação do nosso método. Como veremos, as probabilidades associadas a nós de loop e nós de decisão serão obtidas por simulação.

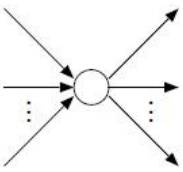

Figura 2.2: Nó de atividade

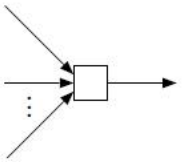

Figura 2.3: Nó de junção

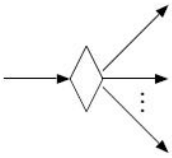

Figura 2.4: Nó de decisão

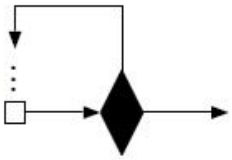

Figura 2.5: Nó de loop

2.2 *Path based testing*

Um módulo de programa M pode ser pensado como uma função $M : D \rightarrow S$, onde D é o domínio de entrada e S é o domínio de saída. Se R é a relação de especificação $\{(d_i, s_i)\}$, onde $d_i \in D$ e $s_i \in S$, dizemos que M está correto se e somente se para cada $d_i \in D, (d_i, M(d_i)) \in R$.

Um conjunto de teste é definido como um subconjunto $T \subseteq D$, ou seja, os elementos que fazem parte do conjunto de teste foram extraídos do domínio de entrada. Um conjunto de teste é dito ideal se para cada $d_i \in D$, $(d_i, M(d_i)) \notin R$ implica $d_i \in T$. Em outras palavras, cada par $(d_i, M(d_i))$ que faz com que o módulo M não esteja correto, pois ele não faz parte da relação de especificação R, indica que a entrada d_i fará parte do conjunto de testes, pois através dela podemos encontrar falhas no programa. Como podemos encontrar no conjunto T elementos que não pertencem a ele e deveriam pertencer e outros que pertencem a ele e não deveriam pertencer, ele se torna um falso positivo.

Entre os conjuntos de teste ideais, temos um interesse especial no conjunto de teste ideal mínimo.

Definição 1 *O conjunto de teste ideal mínimo é aquele que verifica a ocorrência de todos os erros em M utilizando a menor combinação possível de valores de entrada em T.*

Encontrar um conjunto de teste ***ideal*** é um problema NP-difícil [Wong et al. 1995]. Restringimos então nossos esforços em encontrar um conjunto de teste que satisfaça alguns interesses práticos, geralmente baseados na cobertura do código-fonte. O conjunto de teste que é gerado pela análise do código-fonte é chamado Teste Caixa-

Branca ou Teste Estrutural, porque o conhecimento da estrutura interna dos módulos sob teste é usada para desenvolver o conjunto de casos de teste [Janardhanudu 2009, Singh e Rakesh 2010]. Usando este método, o engenheiro de software pode construir casos de teste que [Pressman 2001]:

- garantam que todos os caminhos independentes de um módulo já tenham sido exercitados pelo menos uma vez;
- exercitem todas as decisões lógicas nos lados *verdadeiro* e *falso*;
- executem todos os ciclos nas suas fronteiras e dentro dos seus limites;
- exercitem a estrutura interna dos dados para garantir sua validade.

O *basis path testing*, apresentado por McCabe [McCabe 1976], é uma das técnicas de Teste Caixa-Branca. O programa é associado com o grafo de fluxo de controle, a fim a extrair os caminhos que constituem os casos de teste.

Definição 2 *Um caminho é uma sequência de nós conectados que atravessa o grafo de fluxo de controle do nó inicial até um nó final. Cada caminho corresponde a um elemento do caso de teste.*

Como o número de caminhos é potencialmente infinito, levando a um número infinito de casos de teste, McCabe introduziu no seu trabalho o conceito de independência num grafo fortemente conectado, através da definição de número ciclomático. Um caminho independente é qualquer caminho no programa que introduza pelo menos um novo conjunto de comandos ou uma nova condição. Cada caminho independente deve incluir pelo menos uma aresta que não tenha sido utilizada até o momento de determinar o caminho. O número ciclomático $V(G)$ do grafo de fluxo de controle G é igual ao seu número máximo de caminhos linearmente independentes. Ele pode

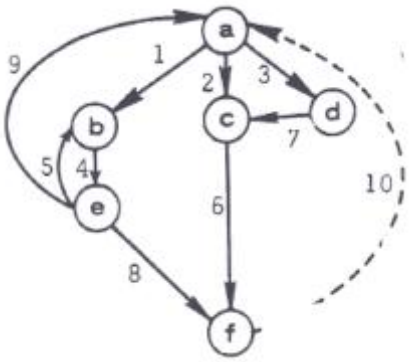

Figura 2.6: Grafo de fluxo de controle rotulado

ser encontrado através do número n de vértices, do número e de arestas, de acordo com a equação:

$$V(G) = e - n + 2 \tag{2.1}$$

Aplicando ao exemplo na Figura 2.1, temos $V(G) = 9 - 6 + 2 = 5$ caminhos linearmente independentes. Para encontrá-los, rotulamos as arestas e atravessamos o grafo. A Figura 2.6 atualiza a Figura 2.1. Os caminhos linearmente independentes (caminhos-base) são *abefa, beb, abea, acfa, adcfa.*

2.3 Caminhos probabilísticos

O conceito de probabilidade pode ser definido através do espaço amostral, que representa a saída de um experimento qualquer em estudo. No nosso caso, o espaço amostral é o conjunto $\mathcal{P}$ de todos os caminhos p_i resultantes da execução do programa sob teste.

$$\mathcal{P} = \{p_1, p_2, \ldots, p_i, \ldots\} \tag{2.2}$$

O conjunto $\mathcal{P}$ pode ser infinito ou muito grande. O ganho do nosso método consistirá obter uma boa aproximação $\mathcal{P}_A$ que não é comparada diretamente a $\mathcal{P}$, mas a uma amostra representativa $\mathcal{P}_L$ de $\mathcal{P}$. Aqui, $\mathcal{P}_L$ denota uma amostra formada pelo conjunto de caminhos obtidos na saída pela execução de um determinado programa com valores de entrada randômicos N vezes, onde N é um valor alto. Em $\mathcal{P}_L$, cada caminho p_i aparecerá com uma certa frequência f_i. Definimos a probabilidade do caminho p_i em $\mathcal{P}_L$ como a frequência relativa $f_L^N(p_i) = f_i/N$. Note que $\mathcal{P}_L$ contém um finito número de caminhos, digamos k. Assumimos sem perda de generalidade uma ordenação $p_1, \ldots, p_k, p_{k+1}, \ldots, p_i, \ldots$ em $\mathcal{P}$ tal que os primeiros k caminhos são precisamente aqueles que ocorrem em $\mathcal{P}_L$, que é, $\mathcal{P}_L = \{p_1, \ldots, p_k\}$ e $\mathcal{P} = \mathcal{P}_L \cup \{p_{k+1}, \ldots, p_i, \ldots\}$. Além disso, definamos $f_L^N(p_i) = 0$ para $i > k$.

O nosso método não introduz o conceito de frequência para cenários de teste. A base prática é originária do método Cleanroom [Selby, Basili e Baker 1987], cuja técnica de testes utiliza uma forma independente e estatística de simulação. O número de testes é feito por amostragem e os cenários são definidos de forma aleatória a cada entrega do produto, a partir de uma distribuição de frequência associada. Essa distribuição é calculada de acordo com o número de falhas identificadas durante os testes. Em outras palavras, o método visa identificar o número mínimo de testes n, que permite estimar a confiabilidade R com um grau de confiança α. Nós estamos dando continuidade a algumas das idéias apresentadas por Cleanroom, com um maior refinamento.

Como escolher N para obter uma amostra representativa de $\mathcal{P}$? Para grandes amostras, podemos usar a tabela de valores críticos do teste de hipótese Kolmogorov-Smirnov [Hoel 1966] com nível de confiança desejado α. O teste Kolmogorov-Smirnov é largamente usado para testes não parametrizados de verificação de igualdade entre duas distribuições de probabilidade, que pode ser aplicado para comparar uma amostra (por exemplo, $\mathcal{P}_L$) com uma distribuição de probabilidade de refer-

ência (aquela associada a $\mathcal{P}$). No nosso caso, a distribuição de probabilidade de referência é obtida através da combinação de todas as tuplas possíveis das variáveis de entrada do programa. Como o número possível de tuplas é infinito, em função das infinitas possibilidades para os valores de entrada das variáveis, não há como obter essa distribuição na prática.

A estatística de Kolmogorov-Smirnov quantifica a distância K_N entre a função de distribuição cumulativa empírica da amostra e a função de distribuição cumulativa empírica da distribuição de referência. É conhecido (veja-se [DeGroot e Schervish 2003, Hoel 1966]) que a distribuição de K_N não depende da função de distribuição cumulativa da distribuição da referência. Em outras palavras, N é independente do número de caminhos independentes do software. Se rodarmos o programa 1000 vezes, poderemos ter no máximo 1000 caminhos diferentes, mas não necessariamente 1000, já que esse número poderá ser menor.

A distribuição nula da estatística é calculada sob a hipótese nula de que a amostra é realizada a partir da distribuição de referência. Para uma distância K_N, as fórmulas abaixo (veja-se [Hoel 1966]) mostram como o valor desejado para N pode ser derivado a partir do nível de confiança desejado α.

$$\alpha = 90\% : \quad 0.05 = \frac{1.22}{\sqrt{N}} \therefore N \geq 596 \tag{2.3}$$

$$\alpha = 95\% : \quad 0.05 = \frac{1.36}{\sqrt{N}} \therefore N \geq 740 \tag{2.4}$$

$$\alpha = 99\% : \quad 0.05 = \frac{1.63}{\sqrt{N}} \therefore N \geq 1063 \tag{2.5}$$

Por exemplo, se $N \geq 740$ então $Pr(K_N = K_{740} \leq 5\%)$ é pelo menos $\alpha = 95\%$. Como veremos, na validação do método usaremos $N = 1000$, que garante $\alpha = 95\%$.

Executar N vezes o módulo M pode ser muito custoso. O ganho do nosso método consistirá em obter uma boa aproximação para $\mathcal{P}_L$, denotada nesta dissertação por $\mathcal{P}_A$, usando um algoritmo generativo que requer um tamanho amostral muito menor, dado por n.

Recapitulando, usamos a seguinte notação:

$\mathcal{P}$ = conjunto de todos os caminhos p_i resultantes da execução do programa sobre teste
$\mathcal{P}_L$ = conjunto de amostras de $\mathcal{P}$ depois de executar o programa um grande número de vezes (N)
$\mathcal{P}_A$ = conjunto aproximado de $\mathcal{P}_L$, obtido pelo nosso método

A avaliação da proximidade entre as frequências de distribuição de $\mathcal{P}_A$ e $\mathcal{P}_L$, também conhecida como o problema do encaixe da distribuição de probabilidade, é central para a validação do nosso método. Para desenvolvê-lo, recorremos ao teste χ^2, baseado no seguinte teorema (veja [Hoel 1966]):

Teorema 3 *Se $f_1, f_2, \cdots, f_k$ e $f_1^*, f_2^*, \cdots, f_k^*$ são frequências observadas e esperadas, respectivamente, para as k possíveis saídas do experimento que é executado N vezes, a distribuição do valor*

$$\chi^2 = \frac{\sum_{i=1}^{k}(f_i^* - f_i)^2}{N} \tag{2.6}$$

é uma variável aleatória que segue a distribuição χ^2 *com* $k-1$ *graus de liberdade.*

O Teste χ^2 consiste em avaliar o valor χ^2 e compará-lo com o valor crítico correspondente ao nível de confiança desejado α. Este valor crítico é obtido de uma tabela já conhecida, a *tabela do Teste* χ^2.

3 O MÉTODO

Neste capítulo, apresentaremos nosso método para obter um conjunto de testes a partir de uma aproximação da distribuição de probabilidade do conjunto de caminhos de execução de um módulo programa M. O ideal é que o método seja aplicado pela mesma pessoa quem desenvolveu o código-fonte, pois ela quem possui um conhecimento completo do programa.

No Passo 1, o código-fonte de M é mapeado em um grafo de fluxo de controle.

No Passo 2, executamos uma versão instrumentada do programa n vezes, cada uma usando como entrada uma amostra aleatória do seu domínio D. A amostra aleatória simula bem os casos típicos de uso em função de sua distribuição de entrada, que tende a ser uniforme em n vezes. Aplicamos então o Algoritmo 1 (ver Apêndice) para obter uma estimativa das probabilidades associadas aos nós de decisão e de loop.

No Passo 3, aplicamos o Algoritmo 2 (ver Apêndice) para produzir um conjunto $\mathcal{P}'_A$ dos caminhos de execução do programa M juntamente com suas probabilidades, usando aquelas associadas aos nós de loop e decisão, obtidas no passo anterior.

A partir de $\mathcal{P}'_A$, através do Algoritmo 3 (ver Apêndice), o conjunto de caminhos desejado $\mathcal{P}_A$ é facilmente derivado. Ele possui a propriedade de que a soma das probabilidades de seus elementos é maior do que um nível de confiança desejado. Note que $\mathcal{P}_A \subseteq \mathcal{P}'_A$.

No Passo 4, o conjunto de testes é obtido através da tradução do conjunto de caminhos $\mathcal{P}_A$. Essa é a saída final do método.

O diagrama de atividades macro do método está sendo exibido na Figura 3.1. Após a apresentação dos passos do método, introduziremos um exemplo para mostrar o funcionamento na prática.

3.1 Passo 1: Geração do grafo de fluxo de controle

O objetivo deste passo é desenhar o grafo de fluxo de controle a partir do código-fonte do módulo M de programa. A saída deste passo é o próprio desenho, que é feito manualmente. Isso significa substituir as estruturas *if* por nós de decisão, e estruturas *while-do* e *do-while* por nós de loop. Lembrando que estruturas *for-do* e ações são substituídas por nós de atividade. Nós de junção são incluídos ao final, quando precisamos fazer várias entradas convergirem em uma única saída.

3.2 Passo 2: Execução do módulo do programa n vezes e obtenção de uma estimativa das probabilidades associadas aos nós de decisão e de loop

Probabilidades devem ser atribuídas aos nós de decisão e de loop. Como temos um conhecimento prévio do código-fonte do programa mas não temos um conhecimento

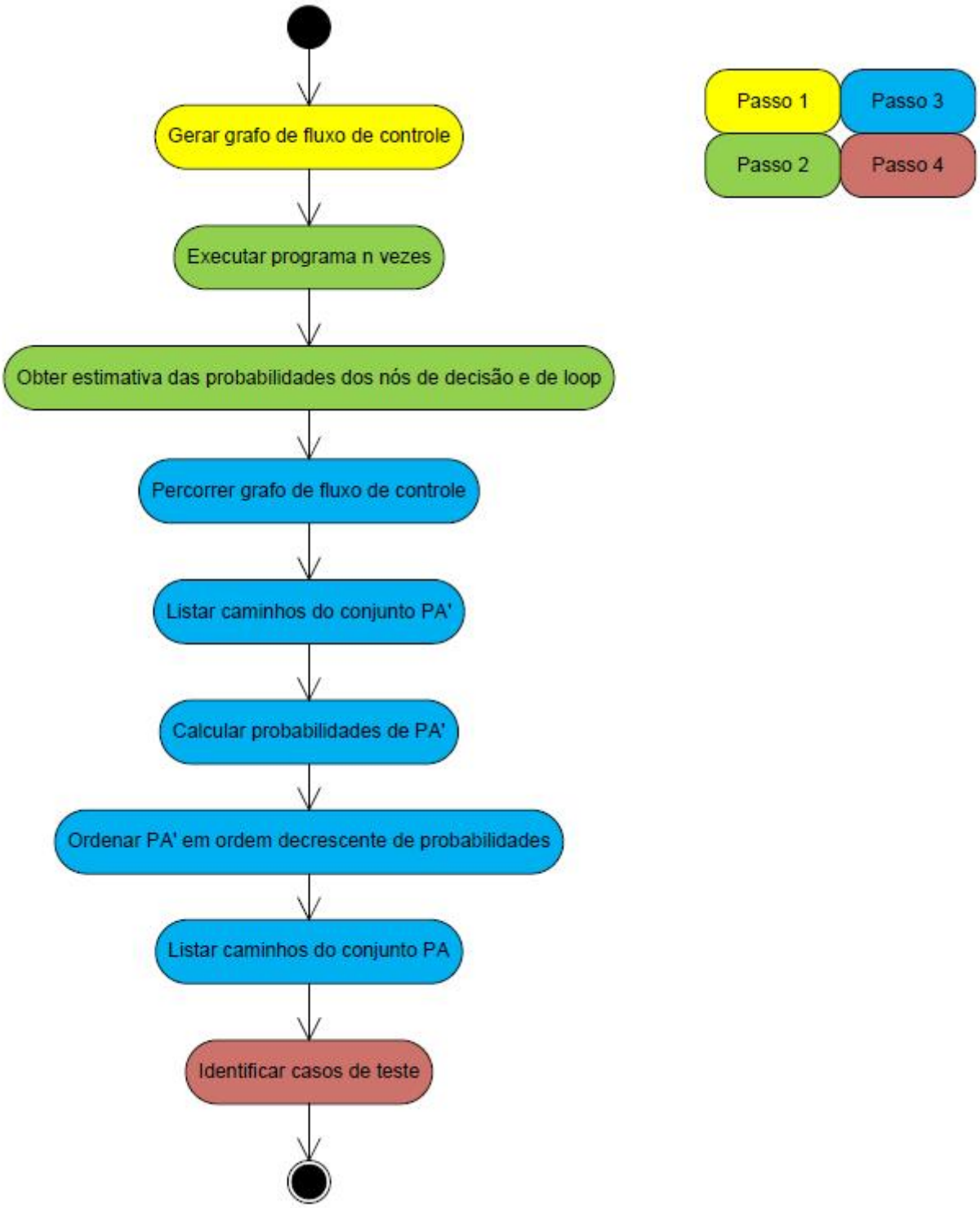

Figura 3.1: Diagrama de atividades com os passos do método

do seu comportamento durante a execução, essas probabilidades são estimadas a partir dos resultados de um número n de execuções do programa, cada uma usando como entrada uma amostra aleatória do domínio do programa. Esses valores nos permitirão calcular a probabilidade de um caminho de execução, através da multiplicação das probabilidades associadas aos nós de decisão e loop no caminho selecionado.

A Figura 3.2 representa o comportamento de um nó de loop, através de um autômato finito. O símbolo l indica que o fluxo de execução passou pela condição de comparação (o próprio nó de loop), o símbolo b indica que o fluxo entrou no corpo do loop, e o símbolo a indica que o fluxo saiu do loop. Logo, o autômato aceita a string $(lb)^*la$. Por exemplo, se a sequência é la, nenhuma iteração foi efetuada, e o autômato atinge o estado T_0. Se a sequência é $lbla$, uma iteração foi efetuada (indicada pelo estado T_1) etc. Em geral, a sequência $(lb)^j la$ significa que j iterações foram executadas em uma particular invocação da estrutura do loop. Como podem existir estruturas de loop aninhadas, a mesma estrutura de loop pode ser invocada várias vezes. O comportamento de um nó de loop pode ser descrito por uma sequência $i_1, i_2, \ldots$ onde i_k é o número de iterações executadas na k-ésima invocação. O Algoritmo 1 estima as probabilidades associadas com os nós de loop e de decisão. Nele, cada t_j (número de vezes que o mecanismo de loop executou j iterações) é calculado sobre todas as invocações.

3.3 Passo 3: Determinação do conjunto de caminhos de execução $\mathcal{P}_A$.

Após atribuir probabilidades aos nós de decisão e de loop, o Algoritmo 2 gera os caminhos de execução do programa M e calcula a probabilidade de cada caminho. Esse conjunto de caminhos temporário é denotado por $\mathcal{P}'_A$. Cada caminho é identifi-

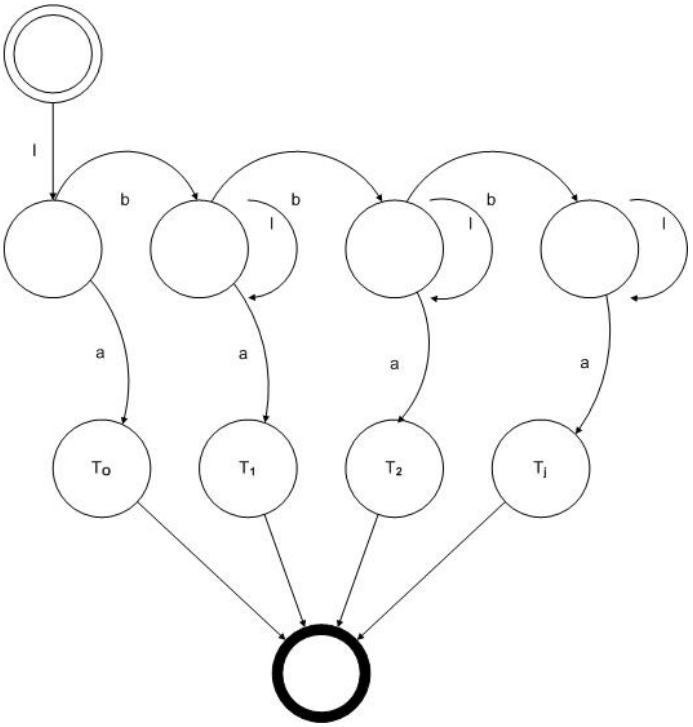

Figura 3.2: Autômato finito representando o comportamento do loop

cado por uma sequência de nós sendo atravessada, por exemplo, os caminhos 1-2-3-8 ou 1-2-3-4-5-6-7-2-3-8 na Figura 3.3.

Finalmente, depois de gerar $\mathcal{P}'_A$, o Algoritmo 3 obtém o conjunto de caminhos de execução $\mathcal{P}_A$, formado pelos caminhos cuja soma das probabilidades é maior que um nível de confiança desejado α. É claro que $\mathcal{P}_A \subseteq \mathcal{P}_{A'}$.

3.4 Passo 4: Especificação do conjunto de teste desejado a partir do conjunto de caminhos $\mathcal{P}_A$.

Cada elemento do conjunto de caminhos será usado para gerar um elemento do conjunto de casos de teste. O problema de obter um conjunto de valores de entrada que force um programa a seguir um caminho de execução específico foi discutido usando várias abordagens [Blanco, Tuya e Adenso-Díaz 2009, Salas e Aichernig 2005]. Embora uma solução computacionalmente amena poderia ser tentada pelo uso de qualquer uma das abordagens citadas acima, recomendamos um procedimento ad-hoc simples, onde o desenvolvedor traduz as transições de nós necessárias em especificações de linguagem natural para os valores de entrada. Essa tradução é feita manualmente.

3.5 Um exemplo prático

Apresentamos um exemplo de aplicação deste método. O programa chamado `c8up` (ver Apêndice) transforma letras minúsculas em letras maiúsculas de uma string aleatória na entrada.

Passo 1. Primeiro, desenhamos um grafo de fluxo de controle a partir do código-

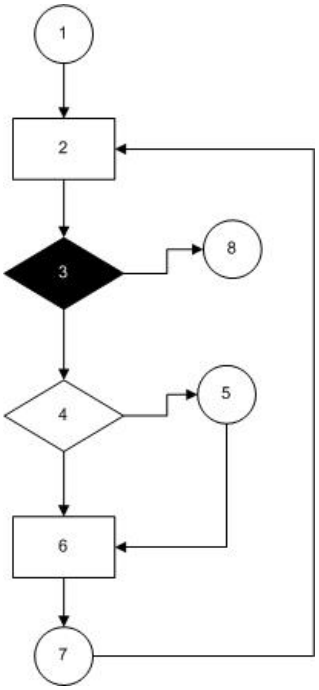

Figura 3.3: Exemplo de grafo do programa c8up

fonte. Ele possui oito nós, como apresentado na Figura 3.4: quatro nós de atividade (1, 5, 7 e 8), um nó de loop (3), um nó de decisão (4) e dois nós de junção (2 e 6).

Passo 2. O programa é então executado $n = 25$ vezes para estimar as probabilidades dos nós de decisão e de loop. Este número é suficiente para gerar a nossa amostra, pois a partir dele a distribuição de probabilidade começa a ficar equivalente ao de número alto de execuções (conforme apresentado no Capítulo 2, no Teste de Kolmogorov-Smirnov). No entanto, o número de execuções necessário para estimar as probabilidades pode variar de acordo com o conjunto de programas usados no experimento. Os resultados são exibidos na Tabela 3.1.

Passo 3. Em seguida, o conjunto de caminhos $\mathcal{P}'_A$ é identificado após a travessia do grafo de fluxo de controle. Ele é formado por sete caminhos. Então, estimamos a probabilidade de cada caminho, de acordo com os resultados na Tabela 3.2.

Tabela 3.1: *Probabilidades para nós de decisão e loop*

	If		While		
	Verdadeiro	Falso	Loop 0x	Loop 1x	Loop 2x
Prob.	0.28	0.72	0	0.52	0.48
Transição	4-5	4-6	3-8	3-4, 3-8	3-4, 3-4, 3-8

p_1:1-2-3-8

p_2:1-2-3-4-5-6-7-2-3-8

p_3:1-2-3-4-6-7-2-3-8

p_4:1-2-3-4-5-6-7-2-3-4-5-6-7-2-3-8

p_5:1-2-3-4-6-7-2-3-4-5-6-7-2-3-8

p_6:1-2-3-4-5-6-7-2-3-4-6-7-2-3-8

p_7:1-2-3-4-6-7-2-3-4-6-7-2-3-8

Tabela 3.2: *Probabilidades dos caminhos do conjunto* $\mathcal{P}'_A$

Caminho	Probabilidade do caminho estimada
p_1:1-2-3-8	0
p_2:1-2-3-4-5-6-7-2-3-8	0.28 * 0.52 = 0.1456
p_3:1-2-3-4-6-7-2-3-8	0.72 * 0.52 = 0.3744
p_4:1-2-3-4-5-6-7-2-3-4-5-6-7-2-3-8	0.28 * 0.28 * 0.48 = 0.0376
p_5:1-2-3-4-6-7-2-3-4-5-6-7-2-3-8	0.72 * 0.28 * 0.48 = 0.0968
p_6:1-2-3-4-5-6-7-2-3-4-6-7-2-3-8	0.28 * 0.72 * 0.48 = 0.0968
p_7:1-2-3-4-6-7-2-3-4-6-7-2-3-8	0.72 * 0.72 * 0.48 = 0.2488

Ordenamos as probabilidades estimadas, como exibido na Tabela 3.3, e incluímos os caminhos no conjunto $\mathcal{P}_A$, de forma que a soma de suas probabilidades associadas alcance o nível de confiança α. Suponha que neste exemplo $\alpha = 90\%$. Logo, o conjunto $\mathcal{P}_A$ é formado pelos caminhos p_3, p_7, p_2, p_5 e p_6, cuja soma das probabilidades associadas é 96.24%.

Passo 4. Finalmente, traduzimos o conjunto de caminhos de execução em um conjunto de casos de teste. O caso de teste é montado a partir das transições identi-

Tabela 3.3: *Conjunto de caminhos ordenado* $\mathcal{P}'_A$

Caminho	Probabilidade do caminho estimada
p_3:1-2-3-4-6-7-2-3-8	0.3744
p_7:1-2-3-4-6-7-2-3-4-6-7-2-3-8	0.2488
p_2:1-2-3-4-5-6-7-2-3-8	0.1456
p_5:1-2-3-4-6-7-2-3-4-5-6-7-2-3-8	0.0968
p_6:1-2-3-4-5-6-7-2-3-4-6-7-2-3-8	0.0968
p_4:1-2-3-4-5-6-7-2-3-4-5-6-7-2-3-8	0.0376
p_1:1-2-3-8	0

ficadas no caminho, em linguagem natural. Para o programa `c8up`, a transformação é exibida na Tabela 3.4. Para facilitar o entendimento, apresentamos as entradas e saídas, porém este passo não é obrigatório.

Tabela 3.4: *Tranformação dos caminhos em caso de teste*

Caminho	Entrada	Saída	Caso de teste
1-2-3-4-6-7-2-3-8	1 caracter fora do intervalo de a até z	Texto não convertido para maiúsculas	Validar um caracter fora do intervalo esperado
1-2-3-4-6-7-2-3-4-6-7-2-3-8	2 caracteres, ambos fora do intervalo de a até z	Texto não convertido para maiúsculas	Validar vários caracteres fora do intervalo esperado
1-2-3-4-5-6-7-2-3-8	1 caracter dentro do intervalo de a até z	Texto convertido para maiúsculas	Validar um caracter dentro do intervalo esperado
1-2-3-4-6-7-2-3-4-5-6-7-2-3-8	2 caracteres, o primeiro fora do intervalo de a até z	Parte do texto convertida para maiúsculas	Validar vários caracteres, o primeiro fora do intervalo esperado
1-2-3-4-5-6-7-2-3-4-6-7-2-3-8	2 caracteres, o primeiro dentro do intervalo de a até z	Parte do texto convertida para maiúsculas	Validar vários caracteres, o primeiro dentro do intervalo esperado

4 VALIDAÇÃO DO MÉTODO

Descrevemos neste capítulo um experimento para validar o método. O objetivo é confirmar que o conjunto de caminhos resultante $\mathcal{P}_A$ juntamente com sua função de distribuição de probabilidades está suficientemente próximo do conjunto real de caminhos $\mathcal{P}$. A Figura 4.1 mostra a ideia da validação. Por um lado, o conjunto $\mathcal{P}_A$ é obtido executando o programa $n = 25$ vezes, calculando $\mathcal{P}_{A'}$ e finalmente selecionando um subconjunto adequado de $\mathcal{P}_{A'}$.

Por outro lado, como pode ser impossível conhecer o conjunto real $\mathcal{P}$, usamos como *proxy* o conjunto $\mathcal{P}_L$, obtido pela execução do programa um número suficientemente grande N de vezes. O número de execuções utilizadas foi $N = 1000$, como recomendado pelas estatísticas de Kolmogorov-Smirnov. Suponha que f^* e f sejam as distribuições de probabilidade de $\mathcal{P}_L$ e $\mathcal{P}_A$, respectivamente. A hipótese nula foi formulada como:

$$H_0 : f^* \neq f$$

O experimento consistiu na aplicação do método a um conjunto de quatorze progra-

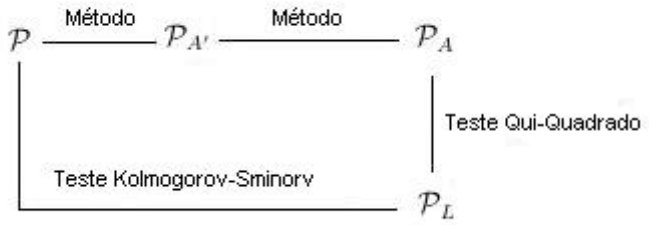

Figura 4.1: Visão simplificada do método e validação

mas pequenos escritos em C/C++. Veja a Tabela 4.1. O fato de serem pequenos não invalida ou deprecia o experimento, já que um programa grande pode ser considerado um conjunto de programas pequenos, pois as boas práticas de programação usam rotinas pequenas. Como o foco do trabalho é Teste Caixa-Branca, trabalhar com pequenas unidades torna a análise do experimento mais fácil.

Tabela 4.1: *Conjunto de programas usados no experimento*

Programa	Descrição
inAddress	Insere endereços numa agenda
c8up	Substitui letras maiúsculas por minúsculas
comb	Calcula combinação de dois números
noLetters	Conta o número de letras num arquivo
upper	Imprime uma frase em letras maiúsculas
factorial	Calcula o fatorial de um número
range	Calcula a amplitude de um intervalo de valores
snacks	Ordena lanches num restaurante fast-food
avgAge	Calcula a idade média de estudantes de um grupo
noVowsCons	Conta vogais e consoantes em uma frase
bisection	Cálculo do zero de uma função transcendente (Método da Bissecção)
binaryNum	Identifica número binário na sua forma decimal e contabiliza o número de 0s e 1s
primeGen	Exibe números primos anteriores a um número
rot13	Implementa cifra de César aplicável aos caracteres alfabéticos com passo 13

Todos os códigos-fonte foram instrumentados segundo a Instrumentação 1 (ver

Apêndice). A instrumentação é feita de forma manual, o que o torna o método semi-automático (existe um grau de automatização). Vale ressaltar que essa instrumentação dos programas é específica para as linguagens C/C++, ou seja, deve ser adaptado conforme a linguagem que seja escolhida para o experimento. Em função disso, optamos por não apresentar a Instrumentação 1 como parte do método, nem junto com os demais algoritmos, porque não pode ser aplicada de forma geral em qualquer linguagem.

Para cada programa, o conjunto de caminhos $\mathcal{P}_L$ juntamente com sua distribuição de probabilidade associada f^* foi gerada. Cada caminho em f^* foi então comparado a cada caminho em f, obtido pelo nosso método. A qualidade do ajuste entre cada caminho de f^* e f foi verificada usando o Teste χ^2 com nível de confiança de 95% (na abordagem moderna, segundo [Zar 1984], o resultado possui credibilidade quando o nível de confiança do experimento é igual ao superior a 95%). A Tabela 4.2 exibe os resultados.

Tabela 4.2: *Resultados experimentais*

Programa	Nós de Atividade	Nós de Junção	Nós de Decisão	Nós de Loop	Total de Arestas	Complexidade Ciclomática	Rejeitou H_0?
inAddress	5	1	2	1	9	2	Sim
c8up	4	2	1	1	9	3	Sim
comb	6	1	2	1	10	2	Não
noLetters	4	2	1	1	9	3	Sim
upper	5	1	1	1	9	3	Sim
factorial	4	1	0	2	8	3	Sim
range	6	4	3	1	17	5	Sim
snacks	7	1	4	1	17	6	Sim
avgAge	7	3	0	3	15	4	Sim
noVowsCons	9	1	4	1	19	6	Sim
bisection	8	2	2	1	15	4	Sim
binaryNum	5	1	2	1	11	4	Sim
primeGen	6	2	2	2	15	5	Sim
rot13	6	1	4	1	16	6	Sim

O tempo dos programas com 25 e 1000 de execuções foi também medido, porém seus resultados não foram evidenciados neste trabalho em função do método não ter foco sobre a parte de otimização de custos. Dessa forma, questões envolvendo tempo e custo ficaram fora do escopo.

Outros programas de complexidade equivalente aos 14 apresentados neste experimento chegaram a ser separados para as simulações. A dissertação originalmente teria 30 programas evidenciados. No entanto, como os resultados dos programas apresentados neste experimento já foram extremamente satisfatórios, não vimos necessidade de realizar novas simulações para a obtenção de conclusões a respeito da confiabilidade do método.

5 DISCUSSÃO DOS RESULTADOS

5.1 Conclusões do experimento

Os resultados mostram que H_0 é rejeitada treze vezes em quatorze, indicando que a probabilidade de obter esses resultados por uma escolha aleatória seria menor que um por cento. `comb` foi o único programa para o qual H_0 não foi rejeitada. Ele possui um caso particular de recursão simultânea, ou seja, ele executa várias vezes operações recursivas em paralelo. Fizemos outro experimento com um programa recursivo, `factorial`, mas como ele calcula apenas uma recursão, o nível de confiança foi satisfatório.

O cenário foi bem representado através das 1000 execuções. As variáveis aleatórias de cada programa possuem uma distribuição de entrada, que tende a ser uniforme após um número muito grande de execuções. Na hora de sortear as variáveis, isso deve ser levado em consideração, pois existe uma tendência do usuário executar determinados cenários com mais frequência e dessa forma, utilizar mais vezes certos valores de entrada que outros. Como os valores de entrada determinam os caminhos, existe uma tendência de certos caminhos aparecerem mais vezes que outros com o uso na prática.

Os resultados preliminares reforçam nossa confiança de que o método é capaz de gerar, com boa precisão, os caminhos mais importantes a serem testados num programa sem precisar passar por um número grande de execuções.

5.2 Dúvidas frequentes

Qual o significado de $\mathcal{P}$?

$\mathcal{P}$ representa o conjunto de percursos feitos pelo programa quando o domínio de entrada D é exaurido, ou seja, quando a combinação entre os possíveis valores de entrada tende a infinito. $\mathcal{P}'_A$ representa a variável aleatória indicativa dos percursos após estimativa de probabilidades de loops e decisões, rodando o programa um número n de vezes, onde n é um valor pequeno. Já $\mathcal{P}_L$ equivale a $\mathcal{P}'_A$, com a diferença de que N é um valor grande. $\mathcal{P}_A$ é uma aproximação de $\mathcal{P}_L$ a partir de $\mathcal{P}'_A$.

O método é trivial?

Não. O método é aparentemente simples, no entanto uma busca nos principais periódicos e conferências dos últimos anos não encontrou nenhuma referência até hoje com esse foco. Se analisarmos os tópicos abordados, veremos que o método envolve conceitos de Engenharia de Software, Grafos, Estatística e até mesmo Compiladores, em trabalhos futuros nesta lista de pesquisa.

A multiplicação das probabilidades é válida?

Sim. A multiplicação é válida porque os caminhos são independentes (veja novamente o exemplo `c8up`). Mesmo quando temos loops aninhados, efetuamos uma

multiplicação de caminhos independentes, pois ao entrarmos na condição interna do loop, já sabemos a probabilidade de passar por ele (de forma amostral).

O fato de os programas do experimento serem pequenos invalida a conclusão?

Não. Um programa grande é um conjunto de programas pequenos, isto é, repleto de funções e procedimentos. O experimento do método foi feito utilizando programas pequenos, mas é viável sua aplicação em programas grandes. A Figura 5.2 mostra um exemplo de programa com três módulos. O módulo M utiliza um domínio de entrada D e chama os módulos M_1 e M_2, que utilizam domínios de entrada D_1 e D_2, respectivamente. Se o método for aplicado ao módulo M_1 isoladamente, e em seguida aplicado ao módulo M_2, também isoladamente, é possível sua aplicação integrada ao módulo M, através da chamada aos dois módulos. Para esta situação, não se recomenda a geração de Stubs para os módulos M_1 e M_2 no teste com M porque M_1 e M_2 serão testados primeiro. A utilização de Stubs é vantajosa para testes *top-down*, quando se quer testar um módulo maior onde os menores ainda não estão prontos, mas neste caso trata-se de um teste *bottom-up*, pois os módulos menores já estão prontos.

O estudo empírico de Binkley [Binkley, Gold e Harman 2007] faz uma análise profunda a respeito da modularização. Segundo os resultados apresentados, um módulo de um programa, em média, deve corresponder a um terço do tamanho do código-fonte original. Ou seja, é mais vantajoso dividir o programa e trabalhar com um número de linhas pequenas.

Por que não usar N amostras diretamente?

Existe uma diferença de tempo entre executar o programa um número pequeno de

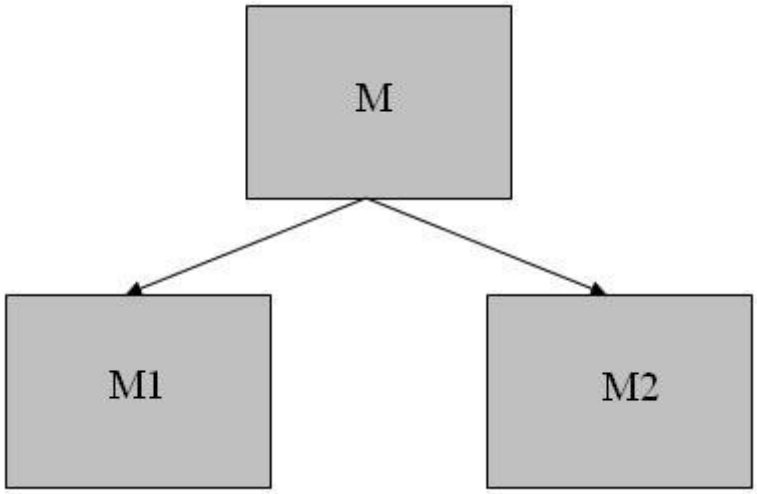

Figura 5.1: Integração entre os módulos

vezes n e um número grande de vezes N, conforme apresentamos no exemplo do capítulo anterior. Quanto maior for a complexidade do código-fonte do programa, essa diferença tende a ser significativa.

Melhora a eficiência de detecção de bugs?

Como os casos de teste mais prováveis estão sendo listados, o desenvolvedor pode fazer mais testes especificamente envolvendo estes caminhos, aumentando a chance de um bug ser detectado. Apenas com os resultados deste trabalho não podemos garantir que melhora a detecção de bugs, porque seria necessário um experimento próprio para isso, porém existe uma forte tendência que sim, em função de os caminhos mais importantes estarem sendo exercitados.

Um software já com os bugs não poderia levar a casos de teste equivocados?

Sim, pois os casos de teste dependem dos valores de entrada e dos caminhos do

programa. Se esses conjuntos forem modificados, os casos de teste serão diferentes.

5.3 Comparação de outros trabalhos relacionados

Nesta seção, apresentaremos alguns trabalhos e faremos um comparativo breve com o nosso, mostrando as diferenças de cada um. Os trabalhos foram selecionados por terem sido publicados em Congressos ou Journals da área de Engenharia de Software (alguns inclusive em Congressos da área de testes) e por discutir o tema de testes baseados em cobertura, com algum tipo de semelhança com o nosso, porém com um enfoque diferente.

O trabalho de Callahan e Schneider [Callahan, Schneider e Easterbrook 1996] analisa os *traces* de execução criados por sentenças de monitoramento durante o Teste Caixa-Branca através de um *model checker*. O *model checker* é usado como um *tableau* semântico para determinar se o trace corresponde ou não a um caminho, de acordo com a árvore computacional do modelo. A diferença deste trabalho para o nosso é que nenhum tipo de análise probabilística de decisões e loops é empregado no trabalho de Callahan and Schneider, enquanto que o nosso emprega.

No trabalho de Whittaker e Poore [Whittaker e Poore 1993], um procedimento para modelagem de software com estados finitos é discutido. Os nós são ações a serem tomadas, e uma transição entre duas ações possui uma frequência associada a elas. Os autores chamam o primeiro passo de fase estrutural, produzida após a análise do sistema, e o segundo passo de fase estatística. A diferença deste trabalho para o nosso é que o trabalho de Whittaker e Poore's é deriva dos Testes Caixa-Preta, no segundo passo, enquanto o nosso é baseado em Testes Caixa-Branca.

O trabalho de Burr e Young [Burr e Young 1998] combina teste baseado em tabelas

e cobertura de código através do *Bellcore's Automatic Efficient Test Case Generator* (AETG), para gerar conjuntos de casos de teste pequenos e eficientes. O método é um mecanismo matemático que cria um conjunto mínimo de vetores de teste que irá exercitar um programa. A diferença deste trabalho para o nosso é que o AETG não usa entradas aleatórias; em vez disso, os valores de entrada são conhecidos, e várias possibilidades são combinadas. O nosso método utiliza entradas aleatórias.

O trabalho de Majumdar e Sen [Majumdar e Sen 2007] apresenta o conceito de hybrid concolic testing, um algoritmo desenvolvido pelos autores que intercala teste aleatório com execução concólica (ou seja, execução simbólica simultânea com execução concreta) para explorar todo o espaço de estados do programa. A proposta é gerar entradas de teste automaticamente através do percurso do código-fonte, assim como o nosso trabalho, porém a diferença é que não é feito análise de probabilidade sobre o espaço de estados, enquanto o nosso trabalho realiza.

O trabalho de Madeyski [Madeyski 2010] apresenta um experimento onde a prática de programação Test-First foi examinada em relação à cobertura de desvio e indicadores de pontuação de mutação. Test-First é uma técnica para testes unitários que os torna mais rigorosos, principalmente em relação à detecção de falhas. Em comparação ao nosso trabalho, nenhum método foi apresentado, apenas detalhes de um experimento envolvendo testes unitários e cobertura de desvio.

O trabalho de Cadar, Dunbar e Engler [Cadar, Dunbar e Engler] apresenta uma ferramenta de execução simbólica para gerar casos de teste com alta cobertura em programas complexos, chamada KLEE. A ferramenta faz análise do código-fonte, porém sem cobertura de probabilidade. No entanto, além de ser um gerador de casos de teste, também foi usada com sucesso como ferramenta de detecção de bugs.

A tabela 5.1 apresenta um resumo da comparação com os trabalhos relacionados.

Tabela 5.1: *Comparativo dos trabalhos relacionados*

Autor(es)	Publicação	Ano	Semelhança	Diferença
Callahan e Schneider	Automated software testing using model-checking	1996	Conceitos de caminho e árvore computacional	Ausência de análise probabilística
Whittaker e Poore	Markov Analysis of Software Specifications	1993	Análise probabilística e máquina de estados	Tipo do teste empregado
Burr e Young	Combinatorial Test Techniques: Table-based Automation, Test Generation and Code Coverage	1998	Geração de Conjuntos de Teste	Ausência de entradas aleatórias
Majumdar e Sen	Hybrid Concolic Testing	2007	Geração de Conjuntos de Teste	Ausência de análise probabilística
Madeyski	The impact of test-first programming on branch coverage and mutation score indicator of unit tests: An experiment	2010	Testes Unitários e Cobertura de Desvio	Ausência de método, somente experimento
Cadar, Dunbar e Engler	KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs	2008	Geração de Conjuntos de Teste e ferramenta de detecção de bugs	Ausência de análise probabilística

6 CONCLUSÃO

6.1 Considerações Finais

Esta dissertação apresenta um método para gerar um conjunto de teste que garante que uma cobertura probabilística alcance um nível de confiança desejado. O método consiste em obter uma aproximação da distribuição de probabilidade do conjunto de caminhos de execução usando uma combinação de um procedimento amostral e um algoritmo generativo baseado na análise probabilística de grafos de fluxo de controle.

A avaliação dos testes confirmou que o método fornece uma boa aproximação quando comparamos $\mathcal{P}_A$ (o conjunto dos caminhos que é a saída do método) e $\mathcal{P}_L$ (uma amostra grande do conjunto real $\mathcal{P}$ de caminhos de execução). O grande benefício do nosso método, entretanto, é o uso de um conjunto pequeno de casos de teste, que reduz o esforço computacional para garantir que a confiança do programa está acima de um determinado nível.

6.2 Trabalhos Futuros

Existem diversos trabalhos que podem ser abertos a partir desta dissertação. Um deles envolve a verificação do comportamento do método com programas maiores. Todos os programas considerados no experimento são pequenos, que não possuem um número muito grande de linhas. O método está preparado para funcionar também com programas maiores, porém nenhum tipo de análise foi feita. Um trabalho futuro seria exercitar o processo com programas maiores para que estes possam também ser analisados da mesma forma.

Outra vertente que pode ser explorada é a automatização da construção do grafo direcionado. Na pesquisa realizada, o digrafo é gerado manualmente, sem o auxílio de nenhuma ferramenta. Existem diversas ferramentas que geram grafos automaticamente no mercado; um sugestão de trabalho seria selecionar as principais e verificar se atendem ao nosso método.

Além da construção do grafo, também é possível automatizar a Instrumentação, utilizando AOP. Atualmente a Instrumentação é feita de forma manual, que consome tempo e pode se tornar inviável dependendo do tamanho do programa. A vantagem da AOP é que a injeção de código é feita na etapa de pré-compilação, sem que seja necessário alterar o código-fonte diretamente para isso. Para essa abordagem, é recomendável utilizar um compilador que ofereça suporte à programação orientada a aspectos, tais como o ACC, bastante conhecido no mercado.

Para assegurar a eficácia da detecção de bugs a partir da utilização do método, seria apropriado a realização de um experimento que comparasse o número de bugs identificados nos programas onde o método foi aplicado com o número de bugs nestes mesmos programas, antes da aplicação do método. Como o foco deste trabalho não foi a detecção de bugs, essas estatísticas não foram coletadas até o momento.

Um outro trabalho interessante envolve a criação de um método para analisar a semântica das condições Booleanas dos loops e transformá-los em blocos seriais. Essa questão envolve principalmente os *while* e *do-while* "disfarçados" de *for*, ou seja, quando se sabe *a priori* o número de passos a serem executados em um comando iterativo.

Finalmente, como o nosso método explora principalmente entradas aleatórias para o cálculo das probabilidades, uma sugestão de trabalho é a criação de um método teórico para associar distribuiçes estatísticas aos valores das variáveis que serão lidas pelo programa, pois isto afeta diretamente a distribuição dos resultados das N rodadas.

REFERÊNCIAS

[Barbosa et al. 2009]BARBOSA, V. C. et al. Structured construction and simulation of nondeterministic stochastic activity networks. *European Journal of Operational Research*, v. 198, n. 1, p. 266–274, October 2009.

[Binkley, Gold e Harman 2007]BINKLEY, D.; GOLD, N.; HARMAN, M. An empirical study of static program slice size. *ACM Transactions on Software Engineering and Methodology*, v. 16, n. 2, April 2007.

[Blanco, Tuya e Adenso-Díaz 2009]BLANCO, R.; TUYA, J.; ADENSO-DÍAZ, B. Automated test data generation using a scatter search approach. *Information and Software Technology*, Lillehammer - Norway, v. 51, n. 4, 2009.

[Burr e Young 1998]BURR, K.; YOUNG, W. Combinatorial test techniques: Table-based automation, test generation and code coverage. In: *7th International Conference on Software Testing, Analysis, and Review*. San Diego, USA: [s.n.], 1998.

[Cadar, Dunbar e Engler]CADAR, C.; DUNBAR, D.; ENGLER, D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: *USENIX Conference on Operating Systems Design and Implementation*. Los Angeles, USA: [s.n.].

[Callahan, Schneider e Easterbrook 1996]CALLAHAN, J.; SCHNEIDER, F.; EASTERBROOK, S. Automated software testing using model-checking. In: *Proceedings 1996 SPIN Workshop*. Vienna, Austria: [s.n.], 1996.

[DeGroot e Schervish 2003]DEGROOT, M. H.; SCHERVISH, M. J. *Probability and Statistics*. New York, USA: Pearson Addison Wesley, 2003.

[Hoel 1966]HOEL, P. G. *Introduction to Mathematical Statistics*. California, USA: John Wiley and Sons, 1966.

[Janardhanudu 2009]JANARDHANUDU, G. *White Box Testing*. [S.l.], set. 2009.

[Madeyski 2010]MADEYSKI, L. The impact of test-first programming on branch coverage and mutation score indicator of unit tests: An experiment. *Information and Software Technology*, v. 52, n. 2, p. 169–184, February 2010.

[Majumdar e Sen 2007]MAJUMDAR, R.; SEN, K. Hybrid concolic testing. In: *International Conference on Software Engineering*. New York, USA: [s.n.], 2007. p. 416–426.

[McCabe 1976]MCCABE, T. J. A complexity measure. *IEEE Transactions on Software Engineering*, v. 2, n. 4, p. 308–320, December 1976.

[Pressman 2001]PRESSMAN, R. S. *Software Engineering - a pratictioner's approach*. New York, USA: McGraw-Hill, 2001.

[Salas e Aichernig 2005]SALAS, P. A. P.; AICHERNIG, B. K. *Automatic TestCase Generation for OCL: a Mutation Approach*. [S.l.], 2005. v. 321.

[Selby, Basili e Baker 1987]SELBY, R. W.; BASILI, V. R.; BAKER, F. T. Cleanroom software development: An empirical evaluation. *IEEE Transactions on Software Engineering*, SE-13, n. 9, p. 1027–1037, September 1987.

[Singh e Rakesh 2010]SINGH, M. K.; RAKESH, L. Mathematical principles in software quality engineering. *International Journal of Computer Science and Information Security*, v. 7, n. 3, p. 178–184, March 2010.

[Whittaker e Poore 1993]WHITTAKER, J. A.; POORE, J. H. Markov analysis of software specifications. *ACM Transactions on Software Engineering and Methodology*, v. 2, n. 1, p. 93–106, January 1993.

[Wong et al. 1995]WONG, W. E. et al. Effect of test set minimization on fault detection effectiveness. In: *17th International Conference on Software Engineering*. Washington, USA: [s.n.], 1995. p. 41–50.

[Zar 1984]ZAR, J. H. *Biostatistical Analysis*. New Jersey, USA: Pearson Education International, 1984.

APÊNDICE A ALGORITMOS

Algoritmo 1: Estimativa das probabilidades dos nós de decisão e de loop

1: **for all** nó do grafo de fluxo de controle **do**
2: **if** o nó é de decisão **then**
3: Conte o número de vezes que o fluxo passou pelos desvios *verdadeiro* e *falso*, respectivamente
4: **end if**
5: **if** o nó é de loop **then**
6: Conte o número de vezes que o loop é executado, e o número de iterações executadas em cada vez
7: **end if**
8: **end for**
9: Execute o programa randomicamente um número de vezes
10: **for all** nó de decisão **do**
11: Seja $\sharp passaVerdadeiro$ o número de vezes que o fluxo passou pelo desvio *verdadeiro*
12: Seja $\sharp passaFalso$ o número de vezes que o fluxo passou pelo desvio *falso*
13: Calcule a probabilidade observada de passar pelo desvio *verdadeiro* usando a fórmula:

$$Prob = \frac{(\sharp passaVerdadeiro)}{(\sharp passaVerdadeiro + \sharp passaFalso)} \tag{A.1}$$

14: Calcule a probabilidade observada de passar pelo desvio *falso* usando a fórmula:

$$Prob = \frac{(\sharp passaFalso)}{(\sharp passaVerdadeiro + \sharp passaFalso)} \tag{A.2}$$

15: **end for**
16: **for all** nó de loop **do**
17: Seja t_j o número de vezes que o fluxo passou pelo estado T_j no autômato da Figura 3.1
18: Seja $T = \sum t_j$
19: Calcule a probabilidade observada de executar j iterações usando a equação:

$$Prob_j = \frac{t_j}{T} \tag{A.3}$$

20: **end for**

Algoritmo 2: Geração do conjunto $\mathcal{P}'_A$

Require: o grafo de fluxo de controle

1: Inclua o nó raiz na lista de caminhos temporária
2: **while** existirem caminhos na lista de caminhos temporária **do**
3: Remova o primeiro caminho da lista (*caminho corrente*)
4: Selecione o último nó do caminho corrente (*nó corrente*)
5: **if** o nó corrente tiver filhos **then**
6: **for all** nó filho do nó corrente **do**
7: Inclua uma cópia do caminho corrente na lista de caminhos temporária
8: Adicione o nó filho ao final da cópia do caminho corrente na lista de caminhos temporária
9: **end for**
10: **else**
11: Inclua o caminho corrente na lista de caminhos final.
12: **end if**
13: **end while**
14: **for all** caminho na lista de caminhos final **do**
15: Calcule a probabilidade observada de obter o caminho, multiplicando as probabilidades de passar pelos nós de decisão e de loop.
16: **end for**

Algoritmo 3: Obtenção do conjunto $\mathcal{P}_A$

Require: o conjunto $\mathcal{P}'_A$; o nível de confiança α

1: Ordene decrescentemente o conjunto $\mathcal{P}'_A$ de acordo com as probabilidades de cada caminho
2: **for all** caminho no conjunto ordenado **do**
3: Adicione o caminho ao conjunto $\mathcal{P}_A$
4: **if** $\sum_{p \in \mathcal{P}'_A} Prob(p) \geq \alpha$ **then**
5: Saia do loop
6: **end if**
7: **end for**

APÊNDICE B INSTRUMENTAÇÕES

Instrumentação 1: Instrumentação para linguagens C/C++ - Parte 1

1: **if** não estiver declarada a biblioteca stdio.h **then**
2: Declarar stdio.h
3: **end if**
4: **if** não estiver declarada a biblioteca stdlib.h **then**
5: Declarar stdlib.h
6: **end if**
7: **if** não estiver declarada a biblioteca string.h **then**
8: Declarar string.h
9: **end if**
10: **if** não estiver declarada a biblioteca time.h **then**
11: Declarar time.h
12: **end if**
13: Declarar estrutura para armazenar a lista de caminhos, que irá conter a string do caminho e o número de vezes em que ele apareceu
14: Declarar vetores para contabilizar o número de vezes passados em cada *if* e *while* nas execuções acumuladas
15: Declarar variável para armazenar o tamanho máximo da string aleatória gerada
16: Declarar variável para armazenar o tamanho máximo da string que armazena o caminho
17: Declarar variável para armazenar quantas vezes o programa será executado
18: Declarar variáveis para contabilizar o número de vezes passados em cada *if* e *while* na execução
19: Declarar variável do tipo da estrutura que armazena a lista de caminhos
20: Declarar método que gera string aleatória
21: Declarar método que concatena o nó por onde o programa acabou de passar na string que armazena o caminho
22: Declarar método que concatena a string que armazena o caminho na lista de caminhos gerados nas execuções
23: Declarar método que busca o caminho na lista de caminhos gerados nas execuções
24: Declarar método que imprime a lista de caminhos gerados nas execuções
25: Declarar método que imprime as probabilidades das decisões e loops
26: Declarar método que chama outros métodos ao final da execuçõ do programa
27: Após a primeira linha do método principal, incluir comando específico para geração de números aleatórios com o passar do tempo
28: Após as declarações locais do método principal, colocar todos os comandos executados dentro de uma estrutura *for* que os executará até o número de vezes máximo definido (20,100,1000 etc)

Instrumentação 1: Instrumentação para linguagens C/C++ - Parte 2

29: Zerar os contadores de *if* e *while* da execução dentro do *for*
30: Alocar memória dinâmica para o caminho e inicializá-lo com `nil` dentro do *for*
31: Substituir as entradas dinâmicas pela chamada ao método que gera números aleatórios dentro do *for*
32: **for all** bloco de comandos executado dentro do *for* **do**
33: Incluir chamada a função de trace de caminho, incrementando sempre uma unidade em relação ao último trace
34: **if** o bloco for de decisão ou loop **then**
35: Incrementar os contadores de *if* e *while* da execução
36: **end if**
37: **end for**
38: Chamar método que acrescenta caminho á lista de caminhos dentro do *for*
39: Liberar a memória do caminho alocada dinamicamente dentro do *for*
40: Chamar método que incrementa os vetores que contabilizam o número de vezes passados em cada decisão e loop de acordo com o número de vezes executado dentro do *for*
41: Chamar método que imprime a lista de caminhos
42: Chamar método que calcula e imprime as probabilidades (de acordo com os vetores e o número máximo de execuções do programa)
43: Acrescentar no corpo do programa, fora do método principal, os métodos declarados inicialmente

APÊNDICE C PROGRAMAS

Programa 1: c8up

```
1 #include<stdlib.h>
2 #include<stdio.h>
3
4 #define TAM_TEXTO 20
5
6 int PaMaius(char *);
7
8 int main ( void  )
9 {
10          char texto[TAM_TEXTO];
11
12          printf("Qual o texto?\n"); fgets(texto, TAM_TEXTO, stdin);
13          printf("Antes: %s\n", texto);
14          PaMaius(texto);
15          printf("Depois: %s\n", texto);
16          exit(0);
17 }
18
19 int PaMaius (char *t)
20 {
21          int i=0;
22
23          while (*(t+i) != '\0')
24          {
25                  if (*(t+i) >= 'a' && *(t+i) <= 'z')
26                          *(t+i) = 'A' + *(t+i) - 'a';
27                  i++;
28           }
29
30          return 0;
31 }
```