UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO

DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO INDUSTRIAL

Uma Ferramenta de Apoio ao Teste de Regressão

Autor: **Ivan Granja**

Orientador: **Prof. Dr. Mario Jino**

Dissertação submetida à Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas, como requisito parcial para obtenção do Título de Mestre em Engenharia Elétrica.

17 de Dezembro de 1997.

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

Granja, Ivan
G766f Uma ferramenta de apoio ao teste de regressão. / Ivan Granja.-- Campinas, SP: [s.n.], 1997.

Orientador: Mario Jino
Dissertação (mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Engenharia de Software. 2. Software - Manutenção. 3. Software - Validação. I. Jino, Mario. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Agradecimentos

Gostaria de agradecer, inicialmente, ao meu orientador Prof. Dr. Mario Jino pela oportunidade, pelas incontáveis contribuições e por acreditar no meu potencial.

À Profa. Angela de Mendonça Engelbrecht, cuja amizade, críticas e sugestões fizeram este trabalho agradável e produtivo.

Aos meus colegas do Grupo de Teste de Software do Departamento de Engenharia de Computação e Automação Industrial da Faculdade de Engenharia Elétrica da Unicamp: Adalberto N. Crespo, Daniela Soares Cruzes, Edmundo Sérgio Spoto, Inês A.G. Boaventura, Letícia Mara Peres, Marcos L. Chaim, Paulo Marcos Siqueira Bueno, Plínio Roberto Souza Vilela e Sílvia Regina Vergilio, pelas sugestões, contribuições, críticas e pelo companheirismo. Aos meus colegas do Instituto de Informática da Pontifícia Universidade Católica de Campinas, Nélson de Carvalho Mendes e Maria Cristina L.F.M. Aranha, cuja interação, por motivo óbvios, foi além do Grupo de Testes.

A todos os demais colegas do Instituto de Informática da Pontifícia Universidade Católica de Campinas, que sempre acreditaram no meu trabalho. Em especial, para os Professores José Oscar Fontanini de Carvalho, Ricardo Pannain e José Estevão Picarelli.

Aos meus amigos, cuja presença, companheirismo e amizade sempre foram estimulantes.

A todos os funcionários do Instituto de Informática pelo apoio logístico e pela torcida.

Aos bolsistas de Iniciação Científica Karlo Augusto Pedro Franco Corrêa, Eduardo Barbosa Vitor e Márcio Faber, cuja dedicação e esforço contribuíram para este trabalho. Ao Programa PIBIC/CNPq e à CEAP da Puc-Campinas, que financiaram esses projetos.

Ao Instituto de Informática da Pontifícia Universidade Católica de Campinas pelo apoio financeiro dado a esse trabalho.

Dedico esse trabalho a minha família que sempre me apoiou. Aos meus pais, ***Francisco*** *e* ***Maria Cândida****, minhas irmãs* ***Silvia*** *e* ***Lúcia****, meu cunhado* ***Milton*** *e minhas sobrinhas* ***Lívia*** *e* ***Thaís****, com amor, respeito e admiração.*

Uma dedicatória especial à minha namorada ***Luciana****, cuja paciência, carinho e apoio fizeram possível este trabalho. Para você* ***Lú****, dedico todo o amor que houver no mundo.*

Resumo

Este trabalho apresenta as principais características de uma ferramenta de apoio ao teste de regressão estrutural baseado na família de Critérios Potenciais-Usos e uma estratégia para sua aplicação.

A ferramenta, denominada **RePoKe-Tool** (**Re**gression Testing support for **Po**tencial-Uses **Cri**teria **Tool**), pode ser aplicada em unidades (e.g., funções, procedimentos) que foram testadas através da ferramenta de teste POKE-TOOL e sofreram manutenção corretiva (teste de regressão corretivo) ou então adaptativa ou perfectiva (teste de regressão progressivo), utiliza estratégia seletiva para identificar apenas os elementos requeridos para o teste estrutural (caminhos e associações) que foram inseridos ou modificados após essa manutenção e sugere um subconjunto dos casos de teste originais que, potencialmente, cobre esses elementos. Além disso, configura e atualiza os arquivos necessários para que a POKE-TOOL possa realizar o teste de regressão.

O trabalho apresenta também estudos de casos que mostram a viabilidade da aplicação de uma estratégia de seleção de casos de testes de regressão baseada em conceitos de teste funcional, com objetivo de atingir um bom índice de cobertura de elementos requeridos selecionados para o teste de regressão estrutural. Os resultados obtidos através desses estudos de casos contribuíram para a definição de um Guia de Referência para Programadores de Manutenção, cujo objetivo é diminuir os altos custos envolvidos com o Teste de Regressão, sem relevar para segundo plano o principal objetivo de qualquer atividade de teste: a revelação de defeitos no software.

Abstract

This work presents the most relevant concepts and characteristics of a regression testing tool, based upon Potential Uses Criteria and a strategy to be applied.

RePoKe-Tool (**Re**gression Testing support for **Po**tential-Uses **Cri**teria **Tool**) can be used in units (e.g., functions and procedures). These units, previously tested by POKE-TOOL testing tool, have been modified either by means of a corrective maintenance (which implies in corrective regression testing) or by means of an adaptive or perfective maintenance (which implies in progressive regression testing). This regression testing tool uses a selective strategy to identify modified and new required elements for white-box testing (paths and associations) and select a subset of original test cases (used in original testing procedures) which potentially may cover these elements. Furthermore, the POKE-TOOL environment is reconfigured to avoid regression testing procedures.

This work presents, additionally, a study of cases that show a selection strategy based upon black-box testing concepts that obtain an expressive cover of white-box required elements. The results help the definition of a guideline to support maintenance programming activities, whose objectives are to save cost and, above all, to reveal faults in modified software.

Índice

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivação

Às portas do terceiro milênio o conceito de qualidade vem ganhando cada vez mais importância. Oferecer produtos e serviços com preços competitivos e de grande aceitação no mercado deixou de ser apenas um detalhe para tornar-se o grande fator diferencial em um mercado consumidor cada dia mais exigente, qualificado e globalizado. Naturalmente, a Engenharia de Software não está alheia a essas profundas transformações.

Dentre as características do software citadas por Pressman [Pre92] como fundamentais em qualquer modelo de qualidade de software estão a testabilidade, manutenibilidade e confiabilidade, que medem a facilidade de testar (descobrir defeitos), manter (modificar a configuração) e a confiança que se deposita em um produto de software, respectivamente. Os novos modelos de qualidade de software (e.g., ISO 12119, CMM, SPICE) vêm sendo constantemente melhorados e aperfeiçoados e já contemplam preocupações com a qualidade do processo de desenvolvimento e evolução de software e não apenas com a avaliação do produto final. As atividades da Engenharia de Software vêm sendo novamente estudadas sob a ótica dessa nova realidade vigente.

Além disso, a cada dia novos produtos de software são oferecidos para dar suporte a áreas onde a vida humana é essencial, fato que exige que esses produtos sejam desenvolvidos (e conseqüentemente testados e mantidos) com alto nível de qualidade e confiabilidade.

Estudos comprovam que, ainda nos dias de hoje, gasta-se cerca de 60% do esforço de desenvolvimento de software com atividades de teste e que, durante o ciclo de vida de um software, cerca de 50% do tempo é gasto em atividades relacionadas à manutenção e evolução de software. Apesar de serem caras e demandarem tempo, teste e manutenção são atividades imprescindíveis da Engenharia de Software uma vez que é impossível construir um software que não possua defeitos ou então que não evoluirá durante seu ciclo de vida [Pre92].

Considerando-se esse quadro, onde as atividades de teste e de manutenção de software são consideradas fundamentais, o foco principal deste trabalho será a atividade de Teste de Regressão que, superficialmente, pode ser definido como "testes que ocorrem após um software ter sofrido processo de manutenção". Tal atividade, que possui características muito peculiares, está diretamente relacionada às atividades de teste e de manutenção e, devido a isso, também pode ser classificada como atividade fundamental da Engenharia de Software.

Por se tratar de uma área que vem despertando o interesse da comunidade de pesquisa em Engenharia de Software apenas nos últimos dez anos, o teste de regressão ainda é uma área carente de ferramentas e métricas que possam dar amplo suporte à

sua condução em um ambiente de desenvolvimento de software industrial. A grande maioria do mercado produtor de software ainda desconhece ou não aplica essa atividade a contento.

Esse trabalho apresentará uma ferramenta de apoio ao Teste de Regressão Estrutural Baseado em Análise de Fluxo de Dados, denominada **RePoKe-Tool** (***Regression**Ttesting Support for **Potential** Uses **Criteria** Tool*). Como é característico das ferramentas de apoio ao Teste de Regressão, a ferramenta trabalhará em conjunto com uma ferramenta de teste denominada **POKE-TOOL** (***Potential**-Uses **Criteria** **TOOL** for program testing*) [Cha91], que implementa os Critérios Estruturais Baseados em Análise de Fluxo de Dados, da família **Potenciais Usos** [Mal91]. Utilizando-se a ferramenta desenvolvida neste trabalho, serão apresentados estudos de casos que comprovam a eficiência de uma metodologia de seleção de casos de teste de regressão baseada em técnicas funcionais (ou caixa preta) que garantem uma boa cobertura dos requisitos exigidos pelo teste estrutural (ou caixa branca).

A seguir, serão discutidos os principais objetivos desse trabalho de pesquisa e, em seguida, será apresentada a organização dessa dissertação.

1.2 Objetivos

Por se tratar de uma área de pesquisa bastante recente, cujas publicações e trabalhos vêm sendo recentemente divulgados, foram traçados alguns objetivos para este trabalho:

1) Desenvolver uma Ferramenta de Apoio ao Teste de Regressão para dar suporte a estudos, discussões e experimentos sobre essa área, utilizando-se a família de Critérios Potenciais Usos.
2) Realizar estudos de casos, com o objetivo de validar a ferramenta e comprovar a aplicabilidade da seleção de casos de teste funcionais na satisfação dos requisitos da família de critérios estruturais Potenciais Usos.
3) Propor, através da análise dos resultados obtidos, um pequeno guia (*guideline*) cujo objetivo é orientar o programador de manutenção na modificação dos programas, de forma que os altos custos do teste de regressão sejam diminuídos, sem esquecer o objetivo de qualquer atividade de teste: a revelação de defeitos no software.

Além disso, a ferramenta desenvolvida neste trabalho tem como objetivo fornecer suporte para uma série de experimentos que poderão ser realizados (e.g., Modelos de Custo de Teste de Regressão para família Potenciais Usos). Esses experimentos estão descritos no último capítulo, na seção de Trabalhos Futuros.

1.3 Organização da Dissertação

Neste capítulo, foi apresentado o contexto no qual este trabalho se insere, algumas discussões introdutórias sobre Qualidade, Engenharia de Software, Testes e Teste de

Regressão, as motivações que justificaram o desenvolvimento desse trabalho, e os principais objetivos que foram traçados.

No Capitulo 2 é apresentado o atual estado da arte, através da discussão e comparação das técnicas e ferramentas publicadas na literatura especializada, bem como os conceitos básicos e fundamentais sobre Teste de Regressão.

No Capítulo 3 são apresentados os fundamentos e principais aspectos teóricos que foram considerados no desenvolvimento da ferramenta de apoio ao teste de regressão. Tais discussões descrevem os requisitos desejáveis de uma ferramenta de software dessa natureza.

No Capítulo 4 são apresentadas a Análise, Arquitetura, Projeto e Aspectos da Implementação da ferramenta, que apoiará os procedimentos de teste de regressão em unidades (funções ou módulo) que foram anteriormente testadas através da ferramenta de teste POKE-TOOL.

O Capítulo 5 apresentará alguns estudos de caso cujos objetivos são: subsidiar as discussões teóricas da dissertação e comprovar a eficiência da seleção de casos de teste, utilizando abordagem funcional como ponto de partida para a condução do teste de regressão estrutural. Adicionalmente, esses exemplos estudados servirão também para consolidar uma série de pequenas orientações ("guidelines") cujo objetivo será facilitar a atividade de manutenção, de forma a diminuir o conjunto de atributos do programa (elementos requeridos e casos de teste) que deverão ser reavaliados no teste de regressão, ainda sim considerando o maior objetivo do teste de regressão que é a revelação dos erros de regressão .

Finalmente, o Capítulo 6 apresenta as conclusões, considerações finais e os trabalhos futuros que deverão ser conduzidos. No único Apêndice é mostrado um exemplo de execução da ferramenta e são mostrados os arquivos gerados por essa execução.

Capítulo 2

Conceitos, Ferramentas e Técnicas Aplicadas ao Teste de Regressão

2.1 A Atividade de Manutenção de Software

Dentre as atividades relacionadas à Engenharia de Software, uma delas possui caraterísticas especiais: a manutenção. Definida por Pressmam [Pre92] como "Conjunto de atividades de Engenharia de Software que ocorrem após o software ter sido entregue ao cliente e colocado em operação" e por Schneidewind [Sch87] como "Modificação de um software após a detecção de defeitos, com objetivo de melhorar a performance e/ou outros atributos, ou ainda para adaptar o produto a um novo ambiente", essa atividade é destacada em qualquer paradigma de desenvolvimento de software [Pre92] (e.g., Modelo Cascata, Prototipação, Modelo Espiral), uma vez que um software sempre será modificado durante seu ciclo de vida. Mais que isso, a manutenção é uma atividade que, em geral, ocorre após a entrega do produto ao cliente, característica que lhe é peculiar. Estudos apontam que, durante todo o ciclo de vida de um produto de software, mais de 50% dos custos são gastos em atividades de manutenção [LW91].

O termo manutenção surgiu derivado da Engenharia de Produto. Por exemplo, quando um aparelho eletrônico apresenta problemas, o mesmo é entregue a um técnico habilitado para que seja consertado. Esse é o conceito clássico de manutenção, pois, em geral, os técnicos apenas identificam e substituem os componentes que apresentaram defeito. Quando aplicado na Engenharia de Software, esse termo ganha novas conotações. Além de ser corrigido (e não consertado), um software pode ser expandido e/ou melhorado, para atender novas necessidades do usuário. Conceitualmente, é mais correto afirmar que o software evolui durante seu ciclo de vida. Essa argumentação pode ser verificada através das definições do termo manutenção. Nos últimos anos, uma nova linha, denominada evolução de software, tem ganho força e vem sendo usada como sinônimo de manutenção.

A atividade de manutenção/evolução de um software ocorre devido a motivos distintos, como por exemplo detecção de falhas pelo usuário, adaptações a novas plataformas de hardware, necessidade de implementação de novas funções, etc. Com isso, essa atividade é freqüentemente dividida em quatro categorias. [Pre92]

(1) Manutenção Corretiva, que ocorre quando é detectado um defeito no software. Beizer classificou os defeitos [Bei84] através de uma escala de conseqüências que variam da Moderada (e.g., erro de alinhamento de relatório), até as Infecciosas que podem corromper outros sistemas, chegando a colocar vidas humanas em risco (e.g., falha no sistema de controle de um reator nuclear).

(2) Manutenção Adaptativa, que ocorre quando é necessário adaptar um determinado software a uma nova plataforma de hardware ou mesmo de software (e.g., migração para uma nova plataforma de hardware, mudança de sistema operacional).

(3) Manutenção Perfectiva (ou Evolutiva), que ocorre quando há necessidade de atender novos requisitos solicitados pelos usuários, o que implica na implementação de novas funções em um software.

(4) Manutenção Preventiva, que ocorre para melhorar as características de um software (e.g., portabilidade, manutenibilidade). Ocorre em geral com programas antigos (e.g., "código alienígena" [Pre92,Sch87]), e que sofrem constantes manutenções durante seu ciclo de vida. Na maioria das vezes inicia-se sem a intervenção do usuário, por iniciativa exclusiva do corpo técnico e gerencial. Mais recentemente, a introdução das técnicas de Engenharia Reversa e Reengenharia têm agregado novas características e modificado os conceitos de manutenção preventiva.

Seja qual for a categoria, a atividade de manutenção de software requer uma equipe especializada, que esteja preparada para trabalhar sob os rígidos métodos requeridos pela Engenharia de Software, e muitas vezes, contra o tempo (alguns softwares "parados" para manutenção por períodos de tempo, mesmo que pequenos, podem gerar altos prejuízos). Aprimorar os procedimentos e automatizar as atividades relacionadas à manutenção são os principais desafios para que um software possa ser modificado e/ou melhorado, preservando-se a qualidade e com o menor custo possível.

2.2 Conceitos e Fundamentos sobre Teste de Regressão

Assim como na fase de desenvolvimento de um software, o teste que ocorre após uma modificação é a atividade mais cara da manutenção. Esse novo procedimento de teste é a última das etapas a ser cumprida durante uma atividade de manutenção e, devido a isso, também possui características próprias, dentre elas o de ser denominado teste de regressão.

As pesquisas sobre teste de regressão são razoavelmente recentes. Em 1988, Ostrand [OW88] já argumentara que o teste de programas é atividade relacionada com a Engenharia de Software mais inadequadamente projetada dentre todas, por ser considerada maçante e não permitir o uso de criatividade. Por ter despertado maior interesse apenas nas últimas duas décadas, foi classificada como a "criança" da Engenharia de Software. De forma análoga, Ostrand classificou o teste de regressão como "bebê de colo", pois muitas vezes ainda é um ilustre desconhecido dos profissionais que desenvolvem software. As pesquisas avançaram bastante nesse período; porém, em ambientes de software industriais, essa realidade permanece praticamente inalterada nos dias de hoje.

A forma mais simples de definir o teste de regressão é afirmar que "Se trata do teste que ocorre após uma modificação ter sido feita em um software". Leung e White [LW91] completaram essa definição ao afirmar que "o teste de regressão envolve o teste do programa modificado, utilizando-se os casos de teste originais que forem

necessários para restabelecer a confiança de que o programa funciona de acordo com sua (possivelmente modificada) especificação". Essa definição, em conjunto com outros estudos conduzidos pelos mesmos autores [LW89], indica que existem dois tipos distintos de teste de regressão:

1. Teste de regressão corretivo, que acontece quando a modificação no software ocorreu para corrigir um defeito, ou seja, tipicamente identificada com as atividades de manutenção corretiva.
2. Teste de regressão progressivo, que ocorre quando a manutenção ocorreu devido a mudanças na especificação do software, ou seja, identificada com as atividades de manutenção adaptativa ou perfectiva.

A definição de Leung e White evidencia os principais objetivos do teste de regressão:

a) Garantir que não foram introduzidos novos erros no software durante o processo de modificação. Esse objetivo é especialmente importante na medida em que Hetzel [Het84] publicou um estudo indicando que, sob certa condições, a probabilidade de um defeito ser introduzido durante as atividades de manutenção varia entre 50% e 80%.

b) Verificar se e o programa modificado corrigiu o defeito encontrado ou funciona de acordo com a nova especificação.

c) Reduzir os custos de seleção dos elementos requeridos e dos casos de teste que poderão exercitá-los, na medida em que o teste de regressão reutiliza os casos de teste aplicados no programa original (antes da modificação).

d) Garantir, ou mesmo aumentar, a confiabilidade do software que sofreu manutenção.

Leung e White [LW91] também classificaram as duas estratégias que podem ser utilizadas para que os objetivos do teste de regressão sejam atingidos.

i. Estratégia "Retest-All", que consiste em aplicar novamente todos os casos de teste usados no programa original.

ii. Estratégia Seletiva, que consiste em selecionar e aplicar somente os casos de teste originais necessários, que exercitem as partes do código que foram afetadas pela modificação realizada.

Uma análise a primeira vista mostra que a primeira estratégia é de aplicação muito mais simples, porém mais cara. Seu uso é perfeitamente justificável quando uma manutenção muito complexa é realizada. A segunda estratégia possui a clara vantagem de ser mais eficiente, na medida em que procura selecionar apenas os elementos requeridos e casos de teste estritamente necessários para o teste de regressão.

O primeiro modelo de custo, que compara as duas estratégias de aplicação de testes de regressão, foi apresentado por Leung e White [LW91]. Esse modelo

apresenta uma forma de se concluir qual é a estratégia mais adequada a custos menores e baseia-se na cardinalidade do conjunto de casos de teste a serem reaplicados. Recentemente, Rosenblum e Weyuker [RW97] apresentaram um modelo de custo/eficiência das estratégias de teste de regressão baseado em cobertura, que ainda sim utiliza o modelo de Leung e White. Esses modelos de custo serão apresentados com mais detalhes na Seção 3.6, quando será discutida qual a melhor estratégia a ser aplicada em cada caso.

2.3 Aspectos de uma Ferramenta de Apoio ao Teste de Regressão

O desenvolvimento de software nos dias atuais exige uma diversidade muito grande de ferramentas de apoio automatizadas, visando aumentar a produtividade e a qualidade bem como a diminuição de custos. Uma parte dessas é destinada a facilitar a atividade de testes, porém, poucas podem ser aplicadas diretamente ao teste de regressão.

Algumas das ferramentas que se propõem a ser úteis no teste de regressão apenas provêm a capacidade de armazenar os testes originais e reutilizá-los após executada uma manutenção [DR88, RO87, Stu80, Stu84], sendo interessantes quando da aplicação de estratégias do tipo "retest-all". Porém, já surgiram ferramentas (vide Seção 2.4) que utilizam estratégia seletiva e são capazes de auxiliar tanto no processo de seleção dos elementos requeridos (caminhos, associações, classes de equivalência, etc., que cubram as exigências de um determinado critério de teste) bem como na seleção dos casos de teste reaplicáveis (teste armazenados após finalizados o teste original, que podem ser reaplicados no teste de regressão).

As ferramentas de teste de regressão existentes hoje são bastante específicas, provavelmente devido ao fato de serem razoavelmente recentes os esforços de pesquisa nessa área. A maioria provê capacidade de aplicação de apenas uma metodologia de testes (funcional ou estrutural), sendo que, em geral, é restrita ao teste de unidade, não abrangendo os testes de integração e de sistema. Outra característica observada é que as ferramentas trabalham obrigatoriamente em conjunto com ferramentas de testes, pois as técnicas envolvidas no teste tradicional são as mesmas aplicadas no teste de regressão.

As ferramentas de apoio ao teste de regressão que utilizam estratégia seletiva possuem, em geral, uma seqüência de procedimentos a serem seguidos. Considerando-se: P o programa original; S sua especificação; E seu conjunto de elementos requeridos segundo algum critério de teste; T o conjunto de casos de teste aplicados para cobrir E; P' o programa modificado; S' sua especificação; E' seu conjunto de elementos requeridos segundo algum critério de teste; E'' um subconjunto de E' que considera apenas as partes modificadas de P' e T' um subconjunto de T que, eventualmente, poderá cobrir os elementos de E'', essa seqüência é a seguinte:

1. Selecionar E'' $\subseteq$ E', um subconjunto de elementos requeridos para o teste de regressão, que estão relacionadas às modificações realizadas.
2. Selecionar T' $\subseteq$ T, um subconjunto do casos testes originais.

3. Aplicar T' sobre P', avaliando a correção de P' com respeito a S'.
4. Se necessário, criar o conjunto T'', um conjunto de casos de teste para verificar novas funcionalidade e/ou garantir cobertura desejada para P'.
5. Aplicar T'' sobre P', avaliando a correção de T'' com respeito a S'.
6. Criar o conjunto T''' , preferencialmente de forma automatizada, que representa o estado final (atualizado) da base de testes, após encerrado o teste de regressão sobre P', pois contém apenas os elementos relevantes dos conjuntos T, T' e T''.

Na próxima seção serão discutidas, com mais detalhes, as principais características de algumas ferramentas e técnicas de apoio ao testes de regressão, encontradas na literatura.

2.4 Ferramentas e Técnicas Relacionadas

Essa seção apresentará uma série de ferramentas que auxiliam as atividades relacionadas com o teste de regressão. Após uma breve apresentação das técnicas e ferramentas relacionadas, serão utilizados os critérios de avaliação para técnicas de teste de regressão, propostos por Rothermel e Harrold [RH96, Rot96]. A maioria das ferramentas apresentadas estão na forma de protótipo e outras ainda não foram implementadas.

2.4.1 Técnicas baseadas em Equações Lineares

2.4.1.1 Ferramenta de Fischer, Raji e Chruscicki

Utilizando-se técnicas baseadas na metodologia de teste estrutural, aplicadas ao teste de unidade de programa escritos em FORTRAN, a ferramenta de Fischer, Raji e Chruscicki [FRC81] utiliza programação inteira zero-um para encontrar um conjunto minimal de testes que cubram todos os segmentos modificados de um programa.

São usadas quatro tabelas para armazenar as relações de transferência de controle entre os comandos, alcance de comandos, definições de variáveis e segmentos. Uma série de restrições é descrita e, utilizando-se essas restrições e técnicas de programação linear, são selecionados os segmentos que precisam ser novamente testados.

2.4.1.2 Ferramenta de Hartmann e Robson

A ferramenta de Hartmann e Robson [HR90], estendeu as características já implementadas na ferramenta de Fischer, Raji e Chruscicki. Trabalha com programas

escritos em linguagem C e permite a integração de vários módulos e vários pontos de definição de variáveis. Essa ferramenta trabalha sobre um sistema de equações lineares mais complexo e, além de selecionar os segmentos que precisam ser reavaliados, também seleciona os testes armazenados após o final da atividade de teste que poderão ser reaplicados no teste de regressão.

2.4.2 Técnica baseada em Execução Simbólica

A ferramenta proposta por Yau e Kishimoto [YK87] é baseada na metodologia de teste funcional, aplicada ao teste de unidade. Consiste na divisão do domínio de entrada de um programa em classes, derivadas tanto da especificação como da implementação do módulo.

O objetivo dessa técnica de teste de regressão é exigir que cada classe modificada ou nova seja testada pelo menos uma vez. Para identificar as classes não modificadas e auxiliar na geração de dados de teste é utilizada execução simbólica.

Terminada essa identificação, a ferramenta seleciona os casos de teste a serem reaplicadas e sugere dados para as partições que exercitam o código modificado.

2.4.3 Técnica baseada em Caminhos

A ferramenta proposta por Benedusi, Cimitile e DeCarlini [BCC88] é baseada na metodologia de teste estrutural, considera apenas análise de caminhos, e também trabalha apenas com teste de unidade.

O objetivo da técnica aplicada na ferramenta é escolher um conjunto de casos de teste que exercitem um conjunto de caminhos selecionados, usando-se duas etapas:

1. Identificar caminhos adicionados, removidos e modificados.

2. Atualizar e reaplicar os casos de teste que exercitem os caminhos adicionados e modificados.

A representação dos programas original e modificado é feita através de expressões algébricas e, a classificação dos caminhos, através de operações algébricas razoavelmente simples.

2.4.4 Técnicas baseadas em Análise de Fluxo de Dados

2.4.4.1 Ferramenta ASSET

A ferramenta ASSET foi desenvolvida por Frankl e Weyuker [FW85, Fra87], com o objetivo de suportar a aplicação dos critérios de Rapps e Weyuker [RW82, RW85], trabalhando sobre programas escritos em linguagem Pascal.

ASSET é uma ferramenta de tratamento intra-procedural e sua preocupação está na análise de fluxo de dados de um módulo, através dos pares definição-uso. Ostrand e Weyuker [OW88] propuseram os melhoramentos necessários para que a ferramenta acima passasse a suportar o teste de regressão. São eles:

1. Prover a ferramenta de capacidade de armazenamento dos Casos de teste originais.
2. Construção das matrizes de Def-Use (Associações Definição-Uso), de Nós e de Arcos, para armazenar as características dos testes originais e do programa original.
3. Dividir o conjunto de Testes originais em dois outros subconjuntos: o primeiro com os dados de teste que exercitam as associações definição-uso alteradas e o segundo contendo os casos de teste que incluam os caminhos de todas as partes alteradas do código.
4. Alterar o subconjunto de dados de teste executados de acordo com as modificações feitas.
5. Definir novos casos de testes para as novas associações definição-uso identificadas no programa alterado.
6. Definir novos casos de teste para satisfazer novamente o critério de adequação, que deixou de ser satisfeito pelo conjunto original de casos de testes.

2.4.4.2 Ferramenta de Harrold, Soffa e Malloy

A ferramenta de Harrold e Soffa [HS88] é baseada na metodologia de teste estrutural baseada em análise de fluxo de dados, combinada com teste incremental. A ferramenta é aplicada tanto ao teste de unidade como ao teste de integração.

Utilizando-se o histórico armazenado durante o teste original, a ferramenta identifica as associações definição-uso afetadas pela modificação e os casos de teste originais são reutilizados sempre que possível. Na extensão para teste de integração [HM91], as associações definição-uso entre dois módulos são testadas (através das interfaces entre os módulos).

2.4.5 Técnica baseada no "Program Dependence Graph" (PDG)

Bates e Horwitz [BH93] lançaram as bases de utilização de uma ferramenta de teste de regressão utilizando-se o PDG (Program Dependence Graph). Esse tipo de grafo, mais complexo que o GFC (Grafo de Fluxo de Controle), armazena informações sobre o fluxo de controle e também sobre o fluxo de dados, usando arcos diferenciados.

A técnica de revalidação é baseada nos critérios de teste aplicados nos PDGs: (1) Todos os Nós do PDG e (2) Todos os Arcos de Fluxo do PDG; utiliza também as técnicas de "Slicing" para agrupar componentes (conjunto disjunto de comandos) tanto de P como de P'. Baseando-se nos critérios de teste aplicados aos PDGs, a ferramenta identifica os componentes que mostram diferente comportamento semântico em P e P', e sugere os casos de teste originais que cobrem esses componentes.

2.4.6 Técnica baseada no "System Dependence Graph" (SDG)

A técnica de diferenciação proposta por Binkley [Bin92], baseia-se na análise e identificação das diferenças semânticas de duas versões de um programa, denominadas versão certificada (P) e modificada (P'). Baseado nessa técnica, Binkley propôs uma ferramenta de apoio ao teste de regressão [Bin94], que utiliza o SDG ("System Dependence Graph"), uma extensão para representação inter-procedimental do PDG. Utilizando a técnica de "slice" baseada no contexto de chamadas, a ferramenta identifica as partes modificadas, preservadas, afetadas e removidas de P' em relação a P. Recentemente, foi proposta uma técnica [Bin96, Bin97] que seleciona um conjunto de casos de teste originais, os quais cobrem as modificações do novo programa, utilizando como guia as diferenças semânticas encontradas entre as duas versões do programa.

2.4.7 Técnica baseada na Identificação de Modificações

Sherlund e Korel [SK91] apresentaram uma técnica que utiliza as dependências estáticas de um programa para determinar os componentes intra-procedimental e estruturais que são dependentes das modificações de dados e/ou de controle em P'. A técnica identifica quais as partes de P' são influenciadas pela modificação, mas não provê facilidades para seleção de casos de teste do programa original (T') que possam ser reaplicados.

2.4.8 Técnica baseada em "Firewall"

A técnica de "Firewall", proposta por Leung e White [LW90], trabalha sobre teste inter-procedimental e aborda técnicas baseadas na especificação e baseadas no programa. A técnica identifica os módulos que circundam o módulo onde a modificação foi realizada. Esse conjunto é chamado de parede de fogo (ou "firewall"). A ferramenta auxilia na seleção de um conjunto T' intra-procedimental que atua na revalidação do módulo modificado ("intra-firewall"), bem como em outro conjunto T' que atravesse toda a "parede de fogo" ("inter-firewall").

2.4.9 Técnica baseada na identificação de "Clusters"

Laski e Szemer [LS92] apresentaram uma técnica que trabalha com subgrafos (denominados clusters). São identificados os subgrafos do GFC do programa modificado (P'), que estão afetados pela modificação. Utilizando-se da classificação dos "clusters" em preservados, modificados, novos e removidos, são sugeridos casos de teste que cubram os "clusters" modificados e novos.

2.4.10 Técnicas baseadas em "Slicing"

Agwall, Horgan, Krauser e London [AHKL93] definiram uma família de técnicas de seleção baseadas em "Slicing". Para cada t $\in$ T, são construídos "slices" [Wei84, GL91], que são partes de um programa que sofrem influência de uma determinada variável, dos programas original (P) e modificado (P') para cada uma das quatro técnicas apresentadas:

1. "Slice" de Execução
2. "Slice" Dinâmico
3. "Slice" Relevante
4. "Slice" aproximadamente Relevante

Utilizando informações dos "slices" obtidos para cada caso de teste, e comparando a equivalência entre os "slices" obtidos para P e P', os casos de teste a serem reaplicados (T') são selecionados.

2.4.11 Técnicas baseadas em Busca em Grafo

As técnicas baseadas em busca em grafo, propostas por Rothermell e Harrold [Har93, HMG93, RH93a, RH93b, RH93c, RH93d, RH94, Rot96], trabalham com a comparação do Grafo de Fluxo de Controle (GFC) dos programas P e P'. As modificações léxicas encontradas em cada nó equivalente dos GFC são anotadas e todos os casos de teste de T, que atravessam as partes modificadas, são selecionados para serem reavaliados (T'). A ferramenta que trabalha com essas técnicas baseadas em busca no grafo fazem tratamento para o teste de unidade e para o teste de integração.

2.4.12 Técnicas baseadas em Entidades Modificadas

Chen, Rosenblum e Vo [CRV94], propuseram uma ferramenta cujo objetivo é identificar as entidades modificadas do programa modificado (P'). Deve-se entender por entidades, as partes do código e funções executáveis que sofreram modificação. A ferramenta, denominada `TestTube`, identifica essas entidades e seleciona os casos de teste que exercitam essas partes modificadas.

2.5 Análise das Técnicas Apresentadas

2.5.1 Critérios de Avaliação das Ferramentas e Técnicas de Teste de Regressão

As técnicas e ferramentas apresentadas na seção 2.4 foram avaliadas e os resultados foram comparados por Rothermel e Harrold [RH96, Rot96]. Porém, para que esse trabalho pudesse ser realizado, os autores identificaram inicialmente quatro categorias (ou critérios) que, quando compostas e relacionadas, formam uma estrutura fundamental de suporte ("framework") utilizada para análise das ferramentas e técnicas de apoio ao teste de regressão, baseadas em abordagem seletiva.

2.5.1.1 Inclusão

A primeira categoria identificada é denominada Inclusão e mede o quanto uma técnica é capaz de escolher testes (T') dentre os originais (T) que, quando aplicados em P', produzam valores de saída diferentes dos que foram gerados pelo programa P e, conseqüentemente, possam revelar falhas.

2.5.1.2 Precisão

A segunda categoria identificada, denominada precisão, mede o quanto uma técnica é capaz de evitar a escolha de testes (T') dentre os originais (T) que, quando aplicados em P', produzam os mesmos valores de saída de P, excluindo-se o caso dos valores de saída coincidentes.

2.5.1.3 Eficiência

A terceira categoria, denominada eficiência, mede o custo computacional de aplicação de uma técnica, ou seja, a viabilidade de aplicação da técnica, considerando-se o seu pior caso.

2.5.1.4 Generalidade

A quarta e última categoria, denominada generalidade, mede a habilidade de uma técnica em manipular programas escritos em linguagens reais, com construções de programação diversificadas, com modificações arbitrariamente complexas e com ambientes de teste realísticos.

2.5.2 Comparação das Técnicas Apresentadas

Rothermel e Harrold [RH96, Rot96] apresentam todos os detalhes e argumentações sobre as categorias e sobre o processo de avaliação das técnicas de teste de regressão apresentadas na Seção 2.4. Para o escopo desse trabalho, basta a apresentação da Tabela 2.1, que sintetiza os resultados obtidos após a comparação das técnicas.

Os resultados apontados na Tabela 2.1 são bastante interessantes. A inclusão varia muito de uma técnica para outra. Muitas omitem testes que atravessam partes modificadas do código, pois procuram obter um conjunto minimal de casos de teste a serem reaplicados. Além de possuírem custo muito alto, as técnicas de minimização tendem a selecionar menos casos de teste que as técnicas seguras, fato que indica que a confiabilidade obtida na segunda técnica tende a ser maior. Estudos que relacionem o custo de minimização em função da confiabilidade poderão ser realizados para comprovar essa observação. Modelos de confiabilidade, baseados em cobertura de critérios estruturais baseados em fluxo de dados [Cre97], apontam que a confiabilidade aumenta na medida em que erros são removidos e essa detecção de erros aumenta na medida em que aumentam os casos de teste aplicados ou então na medida em que a cobertura aumenta. Tais modelos poderão servir como um ponto inicial para esses estudos. Na Seção 3.5, serão expostas com mais detalhes as três abordagens de seleção do conjunto T' (minimal, suficiente ou baseada em cobertura e segura). Adicionalmente, observa-se que nenhuma das três abordagens é 100% precisa. Rothermel e Harold [RH96] argumentam que a seleção de alguns casos de teste que não atravessam modificações é aceitável na prática. É bastante interessante notar que todas as técnicas de seleção trabalham com abordagem estrutural ou funcional. No Capítulo 5 serão apresentados estudos de casos que apontam a viabilidade da utilização de uma abordagem híbrida para seleção de casos de teste de regressão, utilizando estratégia funcional como passo inicial para essa seleção, visando-se obter um nível de cobertura desejado para algum critério de teste estrutural.

No quesito eficiência, as técnicas apresentam resultados muito diferentes. Assim mesmo, baseando-se na análise de pior caso, todas podem ser aplicadas em ambientes industriais de teste. Dentre os quatro critérios, o mais subjetivo de todos é a generalidade. Na última coluna da Tabela 2.1 são apresentados alguns pontos importantes a serem considerados nesse item, que são a abordagem de teste suportada, as classes de modificações tratadas e qual critério de teste é suportado por cada técnica apresentada.

Tabela 2.1 - Resultados das Comparações das Técnicas de Teste de Regressão [1]

Técnica	Categorias de Comparação			
	Inclusão	**Precisão**	**Eficiência (pior caso)**	**Generalidade**
Equações Lineares (minimização)	não segura (todas categorias)	Seleciona testes não-mt	exponencial |P|	Abordagem: unid Mod: ñ fluxo controle Critérios: FC/FD
Equações Lineares (s/ minimização)	segura em ambientes controlados	Seleciona testes não-mt	exponencial |P|	Abordagem: int Mod: todas Critérios: FC/FD
Execução Simbólica	não segura (para remoções)	Seleciona testes não-mt	exponencial |P|	Abordagem: unid e int Mod: ñ trata remoções Critérios: CE
Análise de Caminhos	não segura (para inserções e remoções)	Seleciona testes não-mt	exponencial |P|	Abordagem: unid Mod: ñ iserç e remoção Critérios: Caminhos
Análise Fluxo Dados incremental	não segura (todas categorias)	Seleciona testes não-mt	exponencial |P|	Abordagem: unid e int Mod: apenas pares DU Critérios: FD
Program Dependence Graph -PDG	não segura (para remoções)	Seleciona testes não-mt.Precisão menor que SDG	O (|T| * max(|P|, |P'|^3)	Abordagem: unid Mod: ñ para remoções Critérios: PDG
System Dependence Graph -SDG	não segura (para remoções)	Seleciona testes não-mt.Precisão maior que PDG	O (|T| * max(|P|, |P'|^3)	Abordagem: unid e int Mod: ñ para remoções Critérios: PDG
Baseda nas Modificações	não segura (todas categorias) minimização)	desconhecida	O (|T| * |P'|^2)	Abordagem: unid e int Mod: ñ trata todas Critérios: nenhum
FireWall	segura (ambientes controlados, se T for confiável	Seleciona testes não-mt e todos em alguns casos	O (|T| * max(|P| , |P'|)	Abordagem: unid e int Mod: manipula todas Critérios: nenhum
Identificação de Clusters	segura em ambientes controlados	Seleciona testes não-mt.	O (|T| * max(|P|, |P'|^3)	Abordagem: unid Mod: manipula todas Critérios: nenhum
Slicing	não segura (algumas categorias)	Seleciona testes não-mt.	O (|T| * |P'|^2)	Abordagem: unid Mod: não trata todas Critérios: nenhum
Busca no Grafo	segura em ambientes controlados	Seleciona testes não-mt	O (|T| * max(|P|, |P'|^2)	Abordagem: unid e int Mod: manipula todas Critérios: nenhum
Entidades Modificadas	segura em ambientes controlados	Seleciona testes não-mt.	O (|T| * |P|)	Abordagem: unid e int Mod: não trata todas Critérios: nenhum

[1] Legendas:
unid: unidade (intra-módulo); int: integração (inter-módulos)
Mod: tratamento de modificações
mt: que atravessa as modificações; não-mt: que não atravessa as modificações
FC: Fluxo Controle; FD: Baseados em Fluxo de Dados; PDG: Baseados em PDG;
CE: Classes Equivalência

Capítulo 3

Aspectos Teóricos da Ferramenta

Neste capítulo serão discutidos os principais aspectos teóricos abordados nas técnicas e ferramentas de teste de regressão. Essa discussão dar-se-á à luz dos conceitos utilizados pela família de critérios Potenciais Usos [Mal91] e pela ferramenta POKE-TOOL [Cha91], que implementa e automatiza esses critérios. Essa abordagem justifica-se na medida em que a ferramenta de teste de regressão, que será posteriormente discutida no Capítulo 4, será implementada para trabalhar em conjunto com a POKE-TOOL e, conseqüentemente, com os critérios Potenciais Usos.

3.1 Conceitos Básicos sobre a Família de Critérios Potenciais Usos e a Ferramenta de Teste POKE-TOOL

A família de Critérios Potenciais Usos [Mal91], insere-se na categoria de critérios de teste baseados em análise de fluxo de dados e é derivada da família de critérios apresentada por Rapps e Weyuker [RW82, RW85]. Ambas as famílias requerem, durante o teste de unidade, que caminhos e/ou associações (elementos requeridos), derivados do Grafo de Fluxo de Controle (GFC) de um programa, exercitem as relações entre as definições (atribuição de valor) e usos com respeito a (c.r.a) uma variável qualquer *x*. A diferença entre as duas famílias de critérios está no fato de que os Critérios de Rapps e Weyuker exigem apenas as associações e/ou caminhos entre a definição e o uso (referência explícita) de uma variável qualquer *x*; já a família de Critérios Potenciais Usos é mais exigente, na medida em que requer qualquer relação (caminho e/ou associação) entre a definição de uma variável e um potencial uso da mesma. Essa exigência contempla, portanto, todos os caminhos e/ou associações onde possa haver um uso dessa variável, ou seja, qualquer caminho/associação onde, possivelmente, tenham sido introduzidos erros em um programa, com relação a uma variável qualquer *x*. Trabalhar com a possibilidade de introdução de erros é a melhor característica dessa família de critérios (maior possibilidade de revelação de erros) e a chave para o entendimento do conceito de *potencial uso*. Para fundamentar as futuras discussões desse capítulo, serão definidos, a seguir, alguns conceitos e termos importantes, extraídos das referências citadas, onde maiores detalhes poderão ser obtidos.

Um Grafo de Fluxo de Controle (GFC) é um grafo dirigido, representado por *G(N,A,s)*, onde *N* é o conjunto de nós, *A* o conjunto de arcos e *s* o nó inicial. Define-se caminho como sendo uma seqüência finita de nós (n_i, n_{i+1}, ..., n_k), $k \geq 2$, tal que existe um arco que parte de n_i para n_{i+1}, i = 1,2,...,k-1; um caminho completo é um caminho onde o nó de entrada é o nó inicial s, e o último nó é um nó de saída de G. Caminho livre de laço é aquele cujos nós são distintos e caminho simples é um caminho com as mesmas características do livre de laço, porém, o primeiro e/ou último nó podem, eventualmente, repetir-se no caminho.

Um caminho $(i,n_i,..., n_m, j)$, $m \geq 0$ onde não ocorra definição de uma variável x, definida em i, nos nós $n_i,..., n_m$, é denominado caminho livre de definição com respeito a x, do nó i ao nó j, ou do nó i ao arco (n_m, j).

Um caminho livre de definição $(i,n_i,..., n_m , j)$ com respeito a uma variável x, onde o sub-caminho $(i,n_i,..., n_m)$ é livre de laço, é denominado potencial-du-caminho com respeito a x.

O conjunto de variáveis definidas em um nó i-ésimo do GFC é denominado defg(i). Para cada nó i de G, onde defg(i) $\neq \varnothing$, deriva-se outro grafo, denominado Grafo(i), cujos caminhos componentes partem de i e atingem nós terminais $j_{k1}, j_{k2}, ... ,$ j_{kn} $[(i, n_i, ..., n_m, j_{k1}), (i, n_i, ..., n_m, j_{k2}), ... ,(i, n_i, ..., n_m, j_{kn})]$, e são potenciais-du-caminhos.

A ferramenta de teste POKE-TOOL [Cha91] implementa alguns critérios definidos pela família Potenciais Usos. Supondo-se Π um conjunto de caminhos completos:

i) Critério Todos-Potenciais-Usos: Π satisfaz o critério todos-potenciais-usos se, para todo nó *i* do GFC e para toda variável *x* definida em *i*, Π inclui pelo menos um *caminho livre de definição* c.r.a *x*, do nó *i* para todo nó e para todo arco alcançável a partir de *i*.

ii) Critério Todos-Potenciais-Usos/DU: Π satisfaz o critério todos-potenciais-usos/du se, para todo nó *i* do GFC e para toda variável *x* definida em *i*, Π inclui pelo menos um *potencial-du-caminho* c.r.a *x*, do nó *i* para todo nó e para todo arco alcançável a partir de *i*.

iii) Critérios Todos-Potenciais-DU-Caminhos: Π satisfaz o critério todos-potenciais-du-caminhos se, para todo nó *i* do GFC e para toda variável *x* definida em *i*, Π inclui todos os *potenciais-du-caminho* c.r.a *x*, em relação ao nó *i*.

A ferramenta de apoio ao teste de regressão (vide Capítulo 4), utilizará os critérios citados anteriormente para nortear a seleção de elementos (caminhos/associações) requeridos para essa atividade de teste em particular. Para tal, serão utilizados arquivos gerados pela ferramenta de teste POKE-TOOL. Maiores detalhes sobre a arquitetura e sobre a implementação da ferramenta de teste POKE-TOOL podem ser obtidas na referência citada.

3.2 O Escopo de uma Manutenção

O conceito de escopo de uma manutenção, recentemente introduzido por Leung [Leu95] tem como objetivo identificar as partes de um programa que foram afetadas por uma manutenção, facilitando a localização dos atributos (elementos requeridos) do programa modificado, que devem ser submetidos ao teste de regressão, dado um determinado critério de teste.

Suponha a existência de um programa P que, após sofrer um processo de manutenção, foi transformado em um programa P', com o objetivo de atender uma

determinada especificação S' (que pode não ter mudado em relação a S). Suponha, também, que δ(P) seja o conjunto de nós que representam os comandos de P que devem ser modificados para que a nova versão desse programa (P') seja corretamente modificada para atender à especificação S', e que δ'(P') seja o conjunto de nós cujos comandos foram modificados em P' (comandos originais podem ser modificados e/ou removidos e novos comandos podem ser inseridos nesse conjunto). Considere, finalmente, que os comandos de um programa estão contidos em blocos disjuntos de comandos, representados pelo nós do GFC e, portanto, δ(P) e δ'(P') podem ser denotados em termos de nós de um GFC. O escopo de uma manutenção (escopo(P')) pode ser então representado como conjuntos de nós do GFC do programa P' que são dependentes da manutenção realizada, ou seja, nós que dela sofrem influência. É importante notar que o conceito de escopo é dependente do critério de teste que está sendo aplicado na regressão.

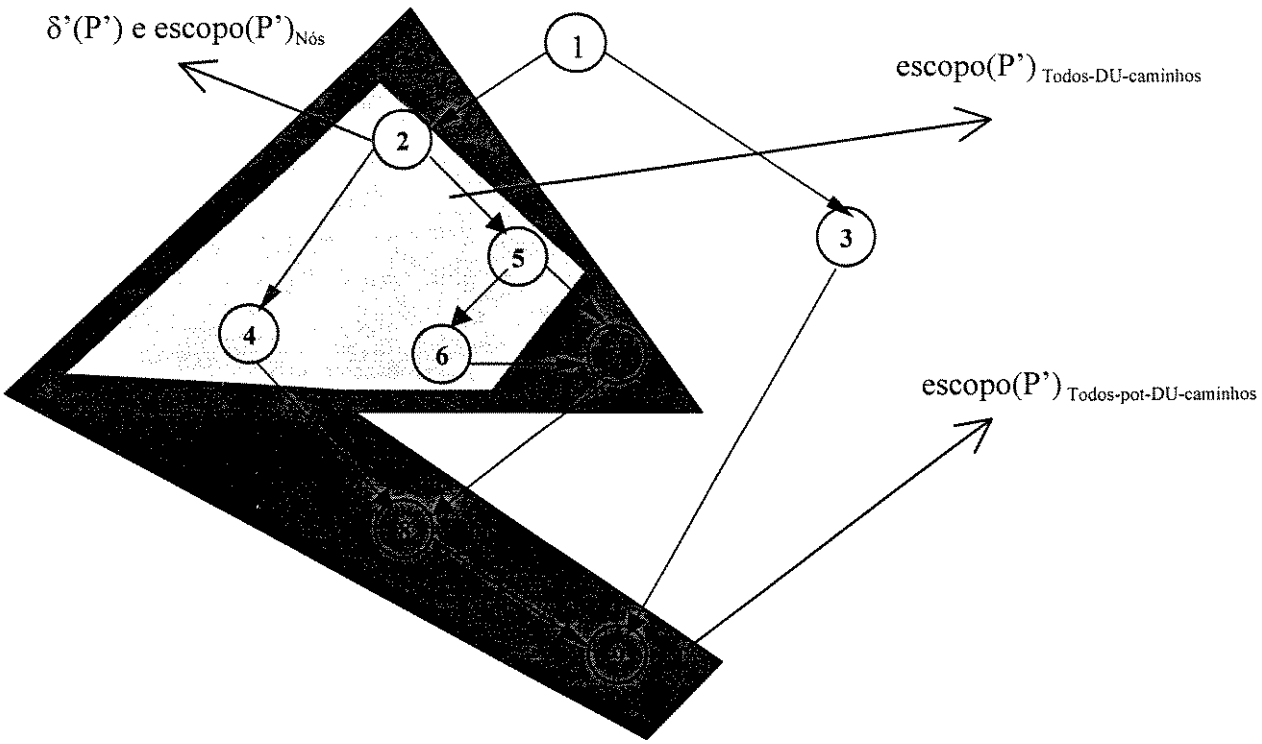


Figura 3.1 - Escopo de uma Modificação e o Conjunto δ'(P')

Para exemplificar melhor, observe o Grafo de Fluxo de Controle mostrado na Figura 3.1 e suponha, que após uma manutenção, foi modificada apenas a definição de uma variável qualquer *x* no nó 2 e suponha que existem referências (usos) a essa variável nos nós 4 e 6 e nova definição de *x* no nó 7. Usando-se as definições dadas anteriormente teríamos que δ(P) = {2} e δ'(P') = {2}. Se o critério de revalidação adotado fosse todos os nós (estrutural), escopo(P') = {2}. Se o critério utilizado fosse todos-du-caminhos [RW82, RW85], escopo(P') = { (2,4) e (2,5,6) }. Já fosse adotado o critério todos-potencias-du-caminhos [Mal91], seria obtido o escopo(P') = { (2,4,8,9), (2,5,6,7) e (2,5,7) } como resultado. É importante salientar que o escopo(P') sempre denota pelo menos um subgrafo S (e.g., um du-caminho, um arco, um nó) do GFC do programa modificado (P'). A utilização desse conceito é um importante guia na seleção dos elementos requeridos de um programa modificado, que

devem ser submetidos ao teste de regressão após uma manutenção. A próxima seção discutirá essa técnica de seleção com maior riqueza de detalhes.

3.3 Seleção dos Elementos Requeridos para o Teste de Regressão

A primeira tarefa a ser realizada pelo testador, durante a condução do teste de regressão, é selecionar qual o critério (ou quais critérios) de teste deverá ser utilizado e qual a estratégia de revalidação deve ser aplicada ao programa modificado. Se a estratégia de teste de regressão a ser aplicada for "retest-all", basta aplicar os casos de teste já existentes com uma ferramenta de teste que suporte o critério escolhido. Porém, se a estratégia escolhida for seletiva, a próxima etapa será, então, selecionar os elementos requeridos que devem ser retestados. Essa é a primeira das atividades típicas de uma abordagem seletiva para teste de regressão (vide Seção 2.1). Essa atividade da estratégia seletiva é dividida em alguns passos importantes:

1. Identificar as modificações que foram introduzidas no módulo.
2. Identificar o escopo dessa modificação.
3. Identificar os elementos requeridos que devem ser reavaliados.
4. Identificar as Potenciais Equivalências de cada elemento requerido do programa modificado com os elementos requeridos do programa original.

3.3.1 Identificação das Modificações Introduzidas

Identificar as modificações introduzidas em um programa (módulo) é uma tarefa bastante difícil, pois uma manutenção pode implicar em modificações no fluxo de dados (e.g., introdução de uma nova definição de uma variável) bem como no fluxo de controle desse módulo (e.g., introdução de um if-else). Esse segundo tipo de modificação, em geral, afeta também o fluxo de dados do programa. Existem duas abordagens para a identificação dessas modificações:

a) Automática

b) Semi-automática

A abordagem automática, apesar de ser a ideal, é muito difícil de ser realizada com total sucesso. Além de requerer a utilização de um "parser" para comparar os programas modificados, essa abordagem exige que sejam refeitas, sem qualquer intervenção do usuário (testador), a Análise de Fluxo de dados das porções modificadas do código e a Análise de Fluxo de Controle (se houver modificações no GFC), ou seja, definir a equivalência entre os nós do GFC dos programas original e modificado. Trata-se, pois, de um problema de isomorfismo de grafos, que é classificado como NP-Completo. Mesmo trabalhando com uma ferramenta de teste que possui uma Linguagem Intermediária independente do código (como é o caso da POKE-TOOL), muitas vezes seria necessário ter acesso ao código fonte do programa.

Isso exigiria a adaptação desse "parser" para cada uma das linguagens suportadas pela ferramenta.

A abordagem semi-automática é mais fácil de ser implementada. Consiste em "dirigir" o testador a identificar as mudanças realizadas. Primeiro, identificando as modificações realizadas no fluxo de controle do programa e as mudanças decorrentes que afetam o fluxo de dados. Em seguida, identificar também as modificações que afetam o fluxo de dados em pontos do programa onde não houve alteração no fluxo de controle (e.g., inicialização de novas variáveis). Finalmente, identificar pontos onde possa haver mudanças nos predicados (e.g., condições de um if), em comandos que referenciem (usem) variáveis que não afetem definições (e.g. `printf ("%d",u) `) e em comandos que não possuem uso e/ou definições (e.g., *printf* de uma mensagem). A abordagem semi-automática é mais barata de ser implementada; porém, requer que o testador conheça bem as modificações que foram realizadas no módulo. Como, em geral, as manutenções ocorrem sob pressão de prazos e custos, é muito comum que o "programador" da manutenção seja também responsável pelo teste de regressão, fato que viabiliza a aplicação dessa abordagem semi-automática.

Seja qual for a abordagem a ser utilizada, pode-se representar essas modificações através de conjuntos que qualificam o estado de cada nó do programa modificado. A qualificação que está sendo proposta neste trabalho (seções 3.3.1.1, 3.3.1.2 e 3.3.1.3) será fundamental para as próximas etapas a serem discutidas (identificação do escopo, seleção dos elementos requeridos e classificação da potencial equivalência entre elementos requeridos dos programas original e modificado).

3.3.1.1 Classificação das Modificações de Fluxo de Controle

Conforme já discutido na seção anterior, nem sempre acontecem mudanças no grafo de fluxo de controle de um módulo. Nesse caso, todos os nós do programa original permanecem, sob o ponto de vista de fluxo de controle, inalterados no programa modificado. Não é necessário, pois, estabelecer nenhuma equivalência entre os nós das duas versões do programa. Já quando houver alterações no fluxo de controle de um módulo, duas situações podem ocorrer:

a) Expansão do Grafo, que é caracterizada pela expansão de um nó ou arco em outros nós e arcos.

b) Contração do Grafo, que é caracterizada pela aglutinação de um conjunto de nós e arcos em um nó ou arco.

As contrações e expansões afetam uma parte dos nós de um programa. Outra parte desses nós permanecem inalterados, pois não sofrem influência da modificação. A Figura 3.2 mostra o grafo de um programa (a) antes da modificação e (b) depois da modificação. Note-se que no exemplo, o nó 3 foi expandido (através da inserção de um *if-else*) para os nós 3,4,5 e 6. Já os nós 1,2 e 7 (do programa modificado) não sofreram alterações.

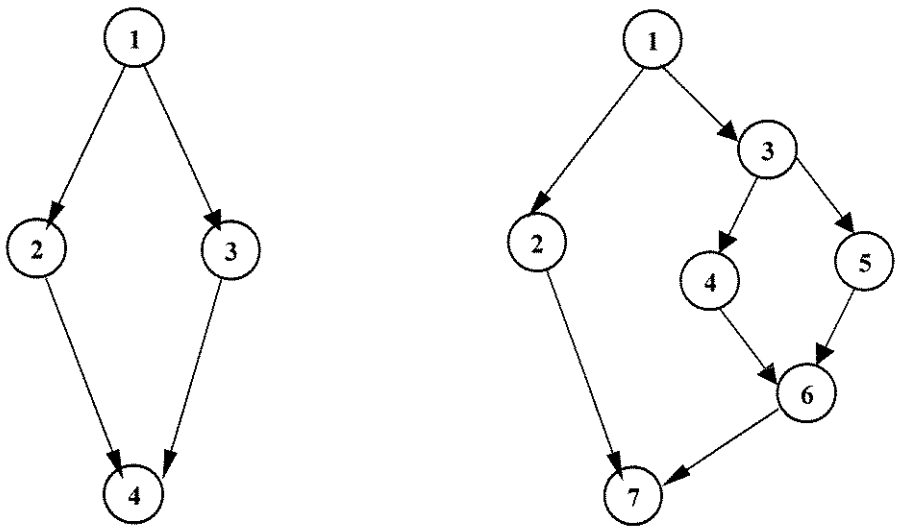
Observando-se mais uma vez a expansão ilustrada na Figura 3.2, é possível classificar os nós do grafo do programa modificado em quatro categorias (conjuntos):

a) Nós Preservados (NP_{fc}): Nós que não sofreram modificação no fluxo de controle. Cada nó desse conjunto deve possuir, obrigatoriamente, um equivalente (mesmo que a numeração seja distinta) no grafo original. Na Figura 3.2 NP_{fc} = {1,2,7} equivalentes, respectivamente, ao conjunto de nós {1,2,4} do programa original.

b) Nós com Fluxo de Entrada (Input) Preservado (NI_{fc}): Nós que sofreram influência da expansão, porém seu arco (fluxo) de entrada permanece inalterado em relação ao nó expandido. Cada nó desse conjunto é sempre semi-equivalente ao nó expandido no grafo original. Na Figura 3.2 NI_{fc} = {3}, semi-equivalente ao Nó 3 do programa original.

c) Nós com Fluxo de Saída (Output) Preservado (NO_{fc}): Nós que sofreram influência da expansão, porém seu arco (fluxo) de saída permanece inalterado em relação ao nó expandido. Cada nó desse conjunto é sempre semi-equivalente ao nó expandido no grafo original. Na Figura 3.2 NO_{fc} = {6}, semi-equivalente ao Nó 3 do programa original

d) Nós Novos (NN_{fc}): Nós que não existiam no programa original e foram introduzidos após a manutenção. Na Figura 3.2 NN_{fc} = {4,5}. Naturalmente, nós novos não possuem equivalência no grafo original.



(a) Versão 1 (b) Versão 2

Figura 3.2 - Expansão de um Grafo de Fluxo de Controle.

As expansões realizadas em um GFC geralmente implicam em mudanças no fluxo de dados. Por exemplo, suponha que no Nó 3 do programa original (vide Figura 3.2) exista definição das variáveis {a, b, c}. Após a expansão do grafo, esse conjunto de variáveis pode estar situado integralmente em algum dos nós gerados pela expansão {3, 4, 5 ou 6}, ou então distribuídas entre esses nós. Algumas dessas variáveis podem também ter sido removidas desse trecho do programa e outras

inseridas. Os conjuntos derivados das mudanças no fluxo de dados serão oportunamente discutidos na próxima seção.

De maneira similar à expansão, a contração afeta a estrutura dos nós do grafo de um programa. Observe mais uma vez a Figura 3.2, mas dessa vez considere o GFC (b) antes da modificação e (a) após a modificação. Nesse caso, os nós {1,2,4} permaneceram com o fluxo de controle inalterados e equivalentes aos nós {1,2,7}. Já os nós {3, 4, 5, 6} tiveram seu fluxo de controle contraído para {3}. As contrações também geram quatro conjuntos que qualificam os nós:

a) Nós Preservados (NP_{fc}): Conceito similar ao da expansão. Na Figura 3.2 NP_{fc} = {1,2,4} equivalentes, respectivamente, ao conjunto de nós {1,2,7} do programa original.

b) Nós com Fluxo de Entrada Preservado (NI_{fc}): Conceito similar ao da expansão. O nó que aglutinou a contração sempre será semi-equivalente ao primeiro nó do conjunto contraído. Na Figura 3.2 NI_{fc} = {3}, semi-equivalente ao Nó 3 do programa original.

c) Nós com Fluxo de Saída Preservado (NO_{fc}): Também similar a expansão. O nó que aglutinou a contração sempre será semi-equivalente ao último nó do conjunto contraído Na Figura 3.2 NO_{fc} = {3}, semi-equivalente ao nó 6 do programa original.

d) Nós Removidos (NR_{fc}): Nós que foram removidos da versão modificada, após a manutenção. Na Figura 3.2 NR_{fc} = {4,5}.

Observando mais uma vez os conjuntos definidos para as contrações e expansões, pode-se observar que os três primeiros conjuntos (NP_{fc} , NI_{fc} ,NO_{fc}) são comuns para ambas e que os conjunto NN_{fc} , NR_{fc} são particulares para as expansões e contrações, respectivamente. Esses cinco conjuntos armazenam, portanto, qualquer modificação realizada no fluxo de controle de um programa qualquer (P') em relação à sua versão anterior (P).

Assim como as expansões, as contrações também implicam, em geral, em mudanças no fluxo de dados. Porém, nesse caso, as variáveis definidas nos nós que foram aglutinados tendem a se concentrar em apenas um nó (que aglutinou a contração). Também é possível que algumas variáveis sejam eliminadas e outras inseridas.

As contrações e expansões podem ser expressas através de modelos. As Figuras 3.3 e 3.4 mostram, respectivamente, os modelo de expansão {nó → *if-else*} e contração {*if-else* → nó}. Todos os modelos de expansão/contração de fluxo de controle, implementados na ferramenta de apoio ao teste de regressão, estão descritos na Seção 4.3.

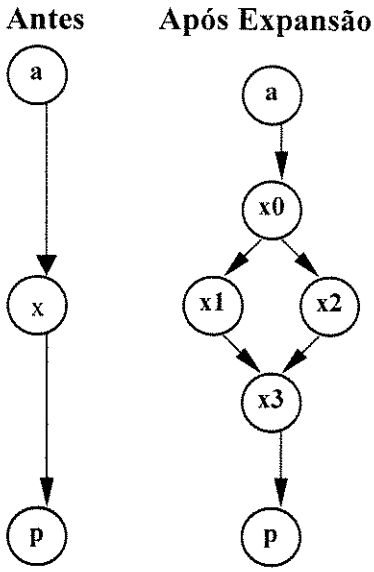


Tabela do Modelo de Expansão

Nó Expandido	Conjunto de Expansão	Semi Equivalência
x0	NI_{fc}	x
x1	NN_{fc}	-
x2	NN_{fc}	-
x3	NO_{fc}	x

Figura 3.3 - Modelo de Expansão: { nó → *if-else* } (de nó para *if-else*)

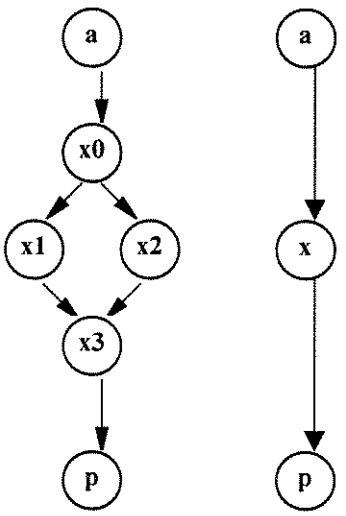


Tabela do Modelo de Contração

Nó Aglutinado	Conjunto de Contração	Semi Equivalência
x	NI_{fc}	x0
x	NO_{fc}	x3
-	NR_{fc}	x1
-	NR_{fc}	x2

Figura 3.4 - Modelo de Contração: { *if-else* → Nó } (de *if-else* para nó)

3.3.1.2 Classificação das Modificações que Afetam o Fluxo de Dados

Classificar as modificações que afetam o fluxo de dados de um programa é fundamental para a correta identificação dos elementos requeridos que deverão ser reavaliados em critérios baseados em fluxo de dados (como é o caso da família Potenciais Usos). Esse tipo de modificação é mais comum do que as de fluxo de controle, pois podem existir sem depender da primeira e, geralmente, são decorrentes

desta. As modificações que afetam o fluxo de dados de um programa podem ser divididas em três grandes tipos:

a) Nas Definições de Variáveis

b) Nos Predicados

c) Em Usos que não Implicam em Definição

Em cada um desses três tipos, um conjunto de variáveis pode estar preservado, removido, modificado e/ou inserido. Esses conjuntos de variáveis de um nó podem aparecer isoladamente ou combinados, dependendo da modificação realizada. Por exemplo, o conjunto de variáveis de um nó que não sofreu modificação possui total equivalência com o conjunto de variáveis de seu nó equivalente no programa original. Já o conjunto de variáveis de um nó que sofreu modificação, pode ser dividido nos quatro subconjuntos de seu equivalente no programa modificado. Em seguida, serão discutidos os conjunto derivados de cada tipo de modificação que afeta o fluxo de dados (Definições, Predicados e usos livres de definição), sob a luz dos conjuntos de variáveis.

Conforme já discutido, quando uma manutenção implica em uma mudança nas Definições de Variáveis de um determinado nó de um programa, essas definições podem ser removidas, inseridas, preservadas ou ainda modificadas. Qualquer variável que tenha sido inserida, removida, preservada ou modificada possui relação com algum nó do programa original (de onde ela foi removida, modificada, inserida ou preservada).

Apesar de tratarem de comandos que não existiam, ou deixaram de existir no programa modificado, as inserções e remoções de definições sempre ocorrem em um determinado nó do programa original. As definições preservadas sempre possuem equivalência direta no programa original. As modificações de definições referem-se a mudanças nos comandos que definem (atribuem) valores para essas variáveis, ou seja, as mudanças se localizam do lado direito de um comando de atribuição (modificações nas constantes, expressões e/ou usos afetam a definição dessa variável). Por se tratar de uma alteração em algum comando do programa original, as modificações nas definições de variáveis também possuem equivalência com esse programa. Observe os trechos de código listados na Figura 3.5 (a) programa original (b) programa modificado (modificações em itálico e negrito) (c) GFC de ambos os programas. Como indica a Figura, não houve mudança no fluxo de dados nesse trecho do programa, mas diversas mudanças afetarão o fluxo de dados. As modificações se enquadram nas categorias que afetam o fluxo de dados, listadas anteriormente.

- Modificação em Definição

Inicialmente, observem-se modificações realizadas nas definições do Nó 1 do programa original (Figura 3.5). A manutenção realizada não afetou a única definição de variável desse nó, `x = 1`. No caso do Nó 3, a definição `e = 1` não existia e, portanto, foi inserida. Já nas definições do Nó 2, é fácil notar que `x = x + 1` permanece inalterada. Portanto a definição da variável x do nó 2 pode ser classificada como equivalente à versão anterior. Já a definição da variável `t = 1` foi removida do

programa original e, em seu lugar, foi inserida a definição `i = 1`. A definição `k = x` foi modificada para `k = y`. Com base nessas observações, as modificações que afetam definições de variáveis, podem ser divididas em alguns conjuntos, a saber:

a) NP_{def}: Nós do programa modificado onde não ocorreram modificações na definição de variáveis em relação ao seu equivalente no programa original. Na Figura 3.5, $NP_{def} = \{1\}$ equivalente ao Nó 1 do programa original.

b) NN_{def}: Nós do programa modificado onde ocorreram, exclusivamente, inserções de definição de variáveis em relação ao seu nó equivalente do programa original, que não possuía definição de nenhuma variável. Na Figura 3.5, $NN_{def} = \{3\}$ equivalente ao Nó 3 do programa original.

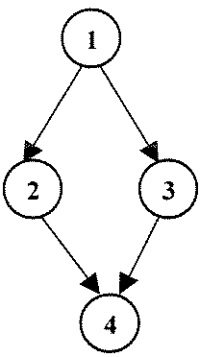
c) NM_{def}: Nós do programa modificado onde ocorreram modificações de qualquer tipo (inserções, remoções e/ou modificações) na definição de variáveis em relação ao seu equivalente no programa original, que poderia ter ou não definição de variáveis. Na Figura 3.5, $NM_{def} = \{2\}$ equivalente ao Nó 2 do programa original.

```
  x = 1;
1 if (a > 10)
2 {   x = x + 1;
      k = x;
      t = 1; }
  else
3,4 printf("Erro");
```

(a) Programa Original

```
    x = 1;
1   if (a >= 10)
2   {  x = x + 1;
       k = y;
       i = 1; }
3    else{e=1;
     printf("Erro %d",e);
4    }
```

(b) Programa Modificado



(c) GFC de ambos

Figura 3.5 - Programa com Modificação no Fluxo de Dados

Cada um dos conjuntos acima possui peculiaridades com respeito ao conjunto de variáveis a ele associado. As variáveis de cada nó do conjunto NP_{def} sempre serão preservadas; as do conjunto NN_{def} sempre serão inseridas (ou novas), pois no nó equivalente no programa original não havia definição de variáveis ($defg(i) \neq \varnothing$). Já as variáveis de cada nó do conjunto NM_{def} podem ser enquadradas em qualquer uma das quatro categorias (preservadas, novas, removidas ou modificadas). A seguir, estão definidos esses quatro conjuntos de variáveis:

a) VP(n): Variáveis do n-ésimo nó que permanecem preservadas (com a definição inalterada) no programa modificado. Na Figura 3.5, $VP(1) = \{x\}$; $VP(2) = \{x\}$ e $VP(3) = \varnothing$.

b) VN(n): Variáveis do n-ésimo nó que foram inseridas, ou seja, definidas no programa modificado e, portanto, são novas em relação ao programa original. Na Figura 3.5, $VN(1) = \varnothing$; $VN(2) = \{i\}$ e $VN(3) = \{e\}$.

c) VR(n): Variáveis do n-ésimo nó cujas definições foram removidas no programa modificado. Na Figura 3.5, VR(1) = ∅; VR(2)={y} e VR(3) = ∅.

d) VM(n): Variáveis do n-ésimo nó cuja definição foi alterada no programa modificado, ou seja, no lado direito da expressão de atribuição houve remoção seguida de inserção de usos de variáveis. Na Figura 3.5, VM(1) = ∅; VM(2) = {k} e VM(3) = ∅.

A Tabela 3.1 mostra as relações possíveis entre as modificações nas definições de variáveis e os conjuntos de variáveis.

Tabela 3.1 - Relações entre Modificações nas Definições e os Conjuntos de Variáveis

Conjunto de Variávies	**Conjuntos de Nós com Definições de Variáveis**		
	NP_{def}	NN_{def}	NM_{def}
VP	√		√
VN		√	√
VR			√
VM			√

Os conjuntos NP_{def} , NN_{def} e NM_{def} , bem como os conjuntos de variáveis definidos anteriormente funcionam da mesma forma quando alguma modificação nas definições acontecer devido a uma modificação no fluxo de controle do grafo. No caso da expansão, as variáveis definidas no nó original podem estar distribuídas pelos nós gerados e, assim, poderá haver vários nós do novo programa que se referem ao mesmo nó do programa original, independentemente de sua classificação NP_{def}, NN_{def} ou NM_{def}. Já no caso da contração, cada variável do conjunto do nó que aglutinou a contração deverá indicar a equivalência com algum dos nós que foram contraídos no programa original.

- Modificação em Predicado

O segundo tipo de modificação que afeta o fluxo de dados foi classificado como mudanças de predicados. Esse segundo tipo de modificação, como o próprio nome sugere, só pode acontecer em nós de predicados, que são os nós gerados por comandos de desvio condicional (e.g., *if, while*). Depois de realizada uma manutenção, um nó de predicado pode ser novo, removido, preservado ou modificado. Os dois primeiros tipos são situações derivadas de modificações de fluxo de controle, como inserção ou remoção de um *if-else*, respectivamente. Quando for realizada a análise e classificação dos elementos requeridos, esses dois tipos de alterações de predicados estarão incluídos na análise de fluxo de controle sendo, pois, desnecessário a definição de conjuntos que armazenem informações desses dois tipos de modificação em predicados. Os nós de predicado preservados são aqueles cujo

predicado não sofre influência da manutenção do programa original. Já os nós de predicado modificados são aqueles cujos predicados foram modificados em função da modificação realizada. Modificações em um predicado podem ser definidas através da inserção, remoção e/ou modificação de operadores relacionais, lógicos, constantes, expressões e/ou usos de variáveis. Então, nós de predicados de um programa deverão ser classificados em apenas dois conjuntos:

a) NP_{pre}: Nós de predicado do programa alterado onde não houve qualquer modificação. Na Figura 3.5, $NP_{pre} = \varnothing$.

b) NM_{pre}: Nós de predicado do programa original onde foram introduzidas modificações em seus respectivos predicados. Na Figura 3.5, $NM_{pre} = \{1\}$ equivalente ao nó 1 do programa original.

- Modificação em Usos que não Implicam em Definição

O terceiro tipo de modificação que afeta o fluxo de dados foi classificado como usos que não implicam em definição. Um comando que exibe na tela uma mensagem qualquer, composta por alguma variável do programa é um exemplo desse terceiro caso (e.g., `printf("Código de erro: %d",erro);`). Modificações em usos de comandos que implicam em definições (e.g., `a = y`) já foram discutidos nas Modificações em Definições e, portanto, não devem ser enquadrados nesse caso. Apesar de uma característica marcante da família de critérios potenciais usos ser a de não considerar explicitamente associações/caminhos do tipo < definição, uso >, a classificação desse tipo de informação é fundamental para a realização de um teste de regressão seguro. O comportamento dos conjuntos gerados por essa categoria de modificação é idêntico aos da categoria de Definição de variáveis. Porém, como os critérios potenciais usos não tratam explicitamente os usos de uma variável, a classificação do conjunto de variáveis não é necessária nesse caso, assim como nos predicados. Assim, as modificações realizadas em comandos do tipo usos independentes de definições são as seguintes:

a) NP_{uso}: Nós do programa alterado onde não ocorreram quaisquer modificações nos comandos que não afetam os usos independentes de definição, em relação ao seu equivalente no programa original. Na Figura 3.5, $NP_{uso} = \varnothing$.

b) NN_{uso}: Nós do programa modificado onde ocorreram, exclusivamente, inserções de comandos com usos independentes de definição de variáveis em relação ao seu equivalente no programa original. Na Figura 3.5, $NN_{uso} = \{3\}$ equivalente ao Nó 3 do programa original.

c) NM_{uso}: Nós do programa alterado onde ocorreram modificações (remoções seguidas de inserções) nos comandos com usos independentes de definição em relação ao equivalente no programa original. Na Figura 3.5, $NM_{uso} = \varnothing$.

d) NR_{uso}: Nós do programa modificado onde ocorreram remoções de comandos com usos independentes de definição, em relação ao seu equivalente no programa original. Na Figura 3.5, $NR_{uso} = \varnothing$.

Os usos independentes de definição serão considerados na revalidação de critérios estruturais como todos-nós, e também nos critérios potenciais usos. A inserção ou eliminação de um uso em um nó qualquer de um programa requer, para um teste de regressão seguro, que todos os Potenciais-DU-Caminhos, ou caminhos livres de definição que passem por esse nó sejam revalidados. A Seção 3.3.2 tratará desses pontos com mais detalhes.

3.3.1.3 Classificação das Modificações que não Afetam o Fluxo de Dados e o Fluxo de Controle

Existem comandos em um programa que podem não afetar o fluxo de dados ou o fluxo de controle do programa. Um exemplo desse tipo de comando é uma mensagem de erro que não possui nenhum uso em sua composição (e.g., `printf("Erro");`). Esse tipo de modificação está sendo classificada apenas para que os critérios estruturais todos-nós e todos-arcos, implementados pela POKE-TOOL, possam também ser revalidados. A classificação dos conjuntos dessa classe de modificação é a mesma utilizada pela classe usos independentes de definição:

a) $NP_{não}$: Nós do programa alterado onde não ocorreram modificações nos comandos que não afetam o fluxo de dados e/ou controle, em relação ao seu equivalente no programa original. Na Figura 3.5, $NP_{não} = \varnothing$.

b) $NN_{não}$: Nós do programa modificado onde ocorreram, exclusivamente, inserções de comandos que não afetam o fluxo de dados e/ou controle em relação ao seu equivalente no programa original. Na Figura 3.5, $NN_{não} = \varnothing$.

c) $NM_{não}$: Nós do programa alterado onde ocorreram modificações (remoções seguidas de inserções) nos comandos que não afetam o fluxo de dados e/ou controle, em relação ao equivalente no programa original. Na Figura 3.5, $NM_{não} = \varnothing$.

d) $NR_{não}$: Nós do programa modificado onde ocorreram remoções de comandos que não afetam o fluxo de dados e/ou controle, em relação ao seu equivalente no programa original. Na Figura 3.5, $NR_{não} = \{ 3 \}$.

3.3.2 Identificação do Escopo das Modificações e Seleção dos Elementos Requeridos a serem Reavaliados

Na Seção 3.2, foi apresentado o conceito de escopo de uma manutenção. Nesta Seção, será discutida a aplicação do conceito de escopo para cada um dos critérios implementados pela ferramenta POKE-TOOL. As regras descritas em cada um desses escopos serão utilizadas para selecionar os elementos requeridos que deverão ser reavaliados no teste de regressão.

Utilizando-se os conjuntos que armazenam as informações sobre as modificações realizadas em um determinado programa (discutidos na Seção 3.3.1), é possível determinar dois conjuntos fundamentais para a identificação do escopo da manutenção.

a) Conjunto de Nós Modificados (NOS_{mod}): Armazena todo e qualquer nó que sofreu alguma modificação no programa P' com relação a P.

b) Conjunto de Nós Preservados (NOS_{pres}): Armazena apenas os nós que não sofreram modificação.

Em termos mais formais pode-se definir esses conjuntos como:

$$NOS_{mod} = \{ \forall n \in N \mid n \in [NN_{fc} \cup NI_{fc} \cup NO_{fc} \cup NN_{def} \cup (NM_{def} \mid VN(n) \neq \varnothing \vee VM(n) \neq \varnothing \vee VR(n) \neq \varnothing) \cup NM_{pre} \cup NN_{uso} \cup NM_{uso} \cup NN_{não} \cup NM_{não}] \}$$

ou seja, NOS_{mod} deve incluir qualquer nó que:

a) Seja Novo, IN ou OUT no Fluxo de Controle;

b) Possua inserções ou modificações nas definições de variáveis, usos independentes de definição e/ou usos que não implicam em definição; e

c) Possua modificação no predicado.

Assim, o conjunto de Nós Preservados será:

$$NOS_{pres} = N - NOS_{mod}$$

onde N é o conjunto de Nós do Grafo de Fluxo de Controle G(N,A,s) do programa P'.

Observando-se bem o conjunto NOS_{mod}, será fácil notar que nele estão contidos todos os nós do programa modificado P' onde o comportamento léxico é diferente de alguma forma do seu equivalente no programa original (P). Essa observação leva à conclusão de que esse conjunto é o próprio escopo para o Critério Todos os Nós, uma vez que toda e qualquer modificação em algum comando de um nó, por menor que seja, exigirá nova execução desse.

- Escopo para o Critério Todos os Nós

A seguir, está definido mais formalmente o escopo do Critério Todos os Nós:

$$escopo(P')_{nós} = NOS_{mod}$$

- Escopo para o Critério Todos os Arcos

Para satisfazer o critério todos os arcos, basta incluir todos os arcos primitivos (j,l) de um programa [Chu87]. Um arco primitivo é definido como um arco que, quando executado, garante também a execução de um conjunto de arcos dependentes

hierarquicamente dele em um Grafo de Fluxo de Controle (arcos herdeiros). Assim qualquer arco primitivo (j,l) pertencerá ao escopo(P')$_{arcos}$, se e somente se, pelo menos um dos nós componentes de seus arcos herdeiros (g,h) pertencer ao conjunto de nós modificados (NOS_{mod}). Ou seja, se pelo menos um dos arcos herdeiros de um determinado arco primitivo sofreu alguma influência da manutenção realizada, a execução desse arco primitivo (j,l) será requerida no teste de regressão. Em termos mais formais temos o seguinte:

Seja Φ o conjunto de arcos herdeiros [(g_1,h_1), (g_2,h_2),..., (g_i,h_i)] de um arco primitivo (j,l).

$$(j,l) \in \text{escopo}(P')_{arcos}, \text{ sse } \{ \exists (g_i,h_i) \mid g_i \in NOS_{mod} \vee h_i \in NOS_{mod} \}$$

- Escopo para os Critérios da Família Potenciais Usos

Para definição do escopo dos critérios, Todos-Potenciais-Usos (PU), Todos-Potenciais-Usos/DU (PU/DU) e Todos-Potenciais-DU-Caminhos (PDU), deve-se utilizar como guia cada um dos Potenciais-DU-Caminhos exigidos para cada Grafo(i) do programa P'.

Para avaliar os critérios baseados em fluxo de dados (Todos Potenciais Usos, Todo Potenciais Usos/Du e Todos Potenciais DU-Caminhos) o conjunto NOS_{mod} deve ser reduzido. Os nós que devem ser retirados desse conjunto são aqueles que pertencem, exclusivamente, aos conjuntos $NN_{não}$ ou $NM_{não}$ ou então se ($\exists n \in NO_{fc} \mid (n \in NP_{def}) \vee (n \in NM_{def} \wedge VP(n) \neq \varnothing)$), onde NP_{def} e $(NM_{def} \wedge VP(n) \neq \varnothing)$ representam a equivalência (total ou parcial, respectivamente) entre as definições de variáveis de um nó do programa P' (apenas aqueles cujo fluxo de controle foi classificado como NO_{fc}) e as variáveis do nó que foi expandido ou dos nós que foram contraídos no programa P. A retirada desses nós dará origem ao conjunto NOS_{red}.

$$NOS_{red} = NOS_{mod} - (NN_{não} \cup NM_{não} \cup (n \in NO_{fc} \mid (n \in NP_{def}) \vee (n \in NM_{def} \wedge VP(n) \neq \varnothing)))$$

A retirada dos conjuntos $NN_{não}$ e $NM_{não}$ ocorrerá devido ao fato de que esses nós não interferem no fluxo de dados. No outro caso, qualquer nó que pertença ao conjunto NO_{fc} é um nó terminal de uma contração ou expansão do programa modificado P'. Esses nós devem ser retirados, apenas se contiverem pelo menos uma das variáveis do(s) nó(s) equivalente(s) do programa original P. Essa retirada justifica-se pois os Potenciais-DU-Caminhos que partem desse nó podem não ter sofrido nenhuma modificação de fluxo de controle e parte de suas variáveis está preservada. Se cumprir essas exigências esse Potencial-DU-Caminho possuirá um equivalente no programa modificado A seguir, serão definidos os escopos para os demais critérios da POKE-TOOL.

Para os critérios Todos-Potenciais-Usos e Todos-Potenciais-usos/DU, os elementos requeridos são associações do tipo **<i,(j,l) <vars> >**, onde i é o nó inicial do grafo(i) e (j,l) é um arco primitivo ou um arco que atinge o nó l, no qual pelo menos uma variável do conjunto defg(i) é redefinida, ou seja o sub-caminho $(i_1, i_2, \ldots i_n, j)$ é

livre de definição com respeito às variáveis definidas <vars> no nó i (conjunto defg(i)). A partir do nó l, a variável redefinida em l é retirada do conjunto <vars>, até que esse conjunto esteja vazio ou o nó de saída do programa for atingido (nó final de um du-caminho). Analisando-se cuidadosamente essas observações, pode-se notar que qualquer associação **< i,(j,l) <vars> >** pode ser um sub-caminho, ou mesmo um potencial-du-caminho do programa que está sendo testado.

Assim sendo, para os critérios PU e PU/DU, qualquer associação **i,(j,l) <vars>** que possua pelo menos um nó do subcaminho $(i_1, i_2, \ldots i_n, j)$ pertencente ao conjunto NOS_{red} pertencerá ao escopo(P')$_{pu,\ pu/du}$ ou, em termos mais formais.

Seja pdc um potencial-du-caminho $(i_1, i_2, i_3 \ldots i_n, j, l, \ldots, k)$.

Temos que:

$$i,(j,l) <vars> \in escopo(P')_{pu,\ pu/du},\ sse\ \{\exists\ n_i \in (i_1, i_2, \ldots i_n, j, l) \subseteq pdc \mid n_i \in NOS_{red}\}$$

Finalmente, para o critério PDU, qualquer Potencial-DU-Caminho que possua pelo menos um nó que atravesse o conjunto NOS_{red}, pertencerá ao escopo(P')$_{pdu}$ ou, em termos mais formais:

$$pdu \in escopo(P')_{pdu},\ sse\ \{\exists\ n_i \in pdu \mid n_i \in NOS_{red}\}$$

Baseados nos escopos definidos, pode-se definir os conjuntos de elementos requeridos para cada critério de teste ***c*** que permaneceram preservados (ERP(***c***)), aqueles que devem ser reavaliados no teste de regressão (ERR(***c***)) e também aqueles que pertenciam ao programa original mas foram removidos do programa modificado (ERRm(***c***)).

Sejam:

E(***c***) o conjunto de elementos requeridos para um determinado critério ***c***.

$(er_1, er_2, \ldots, er_n) \in E(c)$, os n elementos requeridos desse conjunto.

Temos que:

$er_i \in ERR(c)$, sse $\{\exists\ er_i \in escopo(P')_c\}$ e,

$$ERP(c) = E(c) - ERR(c)$$

Dado um critério ***c***, cada elemento requerido preservado do programa modificado P' possui um equivalente direto no programa original P, uma vez que o elemento é formado apenas por nós onde não foram realizadas modificações (preservados). Já os elementos requeridos selecionados para a regressão possuem

elementos potencialmente equivalentes no programa original, pois um caso de teste que exercita um elemento requerido do programa original pode, eventualmente, exercitar um elemento requerido modificado derivado desse original.

Por exemplo, um nó que foi expandido para um *if-else* durante a modificação, vai gerar para cada grafo(i) que alcançar essa parte do programa modificado, dois du-caminhos (p1 e p2) contra apenas um caminho (p) que havia no programa original. O caso de teste que exercitava o caminho original p exercitará apenas um desses dois novos caminhos (p1 ou p2); portanto, ambos os caminhos p1 e p2 de P' são potencialmente equivalentes ao caminho p de P. Somente quando esse caso de teste for reaplicado em P', será definido qual dos dois caminhos (p1 ou p2) será realmente exercitado. Nesse caso, a verdadeira equivalência (que só poderá ser definida dinamicamente), depende de um novo predicado (condição do if inserido) que não existia no programa original.

Existe ainda um terceiro conjunto de elementos requeridos que deve ser considerado. Esse conjunto, denominado ERRm(c), contém os elementos requeridos do programa original P que não são equivalentes, nem potencialmente equivalentes a qualquer elemento requerido do programa modificado P'. Esses elementos são qualificados como removidos do programa original.

Sejam:

> equiv(er_j,er'_j) e pot-equiv(er_j,er'_j), respectivamente, as relações de equivalência e potencial equivalência entre elementos requeridos dos programas original (er_j) e modificado (er'_j).

temos que:

> $er_j \in$ERRm(c), sse $\{ \exists\ er_j \mid \text{equiv}(er_j,\varnothing) \wedge (\text{pot-equiv}(er_j,\varnothing)\ ) \}$

Dos três conjuntos definidos conclui-se que, para um determinado critério de teste c, o conjunto de elementos requeridos a serem reavaliados no teste de regressão ($E'' \subseteq E'$) é o próprio conjunto ERR(c).

3.4 Seleção dos Casos de Teste a serem Reaplicados

Na seção anterior foi apresentado o método de seleção dos elementos requeridos (E'') que deverão ser novamente exercitados durante o teste de regressão (programa P'). A reavaliação desses elementos requeridos, segundo a estratégia de teste de regressão seletiva, deve ser realizada utilizando-se um subconjunto (*"regression testing suite"*) dos casos de teste aplicados no programa original ($T' \subseteq T$). Cada caso de teste original ($t \in T$) pode ser classificado de três formas [LW89]: reutilizável, reaplicável ou obsoleto.

O primeiro tipo, denominado reutilizável, identifica os casos de teste que não exercitam as partes do código que foram afetadas direta ou indiretamente pela manutenção. Em outras palavras, são casos de teste que, quando exercitados, não atravessam partes do código que pertençam ao escopo da modificação (escopo(P')$_c$),

ou seja, produzirão a mesma saída original quando aplicados ao programa modificado. Os casos de teste reutilizáveis não precisam ser reaplicados no teste de regressão.

Os casos de teste reaplicáveis são aqueles que exercitam as partes do código que foram afetadas direta ou indiretamente pela manutenção. Em outras palavras, são casos de teste que, quando executados, percorrem partes do código que pertencem ao escopo da modificação (escopo(P')$_c$). Esses casos de teste produzem, geralmente, saídas diferentes daquelas obtidas quando aplicados ao programa original (salvo o caso da saída coincidente). Conclui-se, pois, que o conjunto T', que deverá ser reaplicado ao teste de regressão, é o próprio conjunto de casos de teste reaplicáveis.

O terceiro tipo, denominado obsoleto, identifica casos de teste que exercitam apenas partes do código que foram removidas do programa modificado (P') e, portanto, podem ser eliminados do conjunto de casos de teste.

Existe, ainda, um quarto tipo denominado redundante. Um teste que exercita somente elementos requeridos já eliminados, ou seja, nada acrescente em termos de cobertura ou funcionalidades testadas em um programa pode ser descartado da base. É bastante comum que, após determinadas manutenções (e.g., mudanças em predicados), casos de teste anteriormente úteis tornem-se redundantes. A última seção deste capítulo tratará com mais detalhes algumas formas de eliminar da base casos de teste desse tipo.

Para cada caso de teste aplicado sobre o programa original (P), pode-se representar uma tripla <CT, ER(*c*), F> para cada um dos critérios da família potenciais usos. CT identifica o caso de teste (que é composto por outra tripla <entrada, caminho percorrido, saída produzida>), ER(*c*) identifica o conjunto de elementos requeridos (para um determinado critério de teste *c*) que foram eliminados pela execução de CT e F representa a funcionalidade que foi exercitada por CT. É importante salientar que uma mesma funcionalidade F pode ser testada por conjuntos de elementos requeridos completamente diferentes em um mesmo programa.

Utilizando-se a classificação descrita na seção anterior, sabe-se que o conjunto ERR(*c*) contém apenas os elementos requeridos do programa modificado que foram classificados como novos e modificados, dado um critério de teste c, e que estes precisam ser novamente exercitados no teste de regressão. Alguns elementos desse conjunto (apenas os classificados como modificados) possuem potenciais-equivalências com os elementos requeridos do programa original (P). O segundo conjunto, denominado ERP(*c*), aponta os elementos requeridos que não sofreram influência da modificação e, portanto, estão preservados em relação ao programa P. Todos os elementos desse conjunto possuem pelo menos um equivalente no programa original. O terceiro conjunto, denominado ERRm(*c*), contém os elementos requeridos do programa original (P) que foram removidos no programa modificado (não possuem equivalência com nenhum elemento requerido do programa modificado P').

Considerando-se cada uma das triplas <CT, ER(*c*), F>, pode-se afirmar que a funcionalidade F do programa original (P) estará preservada (portanto, computará o mesmo valor de saída) se e somente se todos os elementos requeridos do conjunto ER(*c*) pertencerem ao conjunto de elementos requeridos classificados como preservados no programa modificado (P'); de forma análoga, a funcionalidade F não estará preservada (provavelmente computará um valor de saída diferente) se existir

pelo menos um elemento requerido do conjunto ER(c) do programa original P que não é equivalente a nenhum elemento requerido preservado no programa modificado (ou seja, se o elemento equivalente em P' for novo ou modificado). Finalmente, a função F estará removida se e somente se, todos os elementos requeridos do Conjunto ER(c) estiverem classificados como removidos em relação ao programa original P. A Proposição 1, dada a seguir, coloca essas observações em termos mais formais:

Proposição 1:

Seja a tripla <CT, ER(c), F>, onde:

CT, identifica o caso de teste original

$(er_1, \ldots, er_n) \in ER(c)$, é o conjunto de elementos requeridos eliminados por CT em P, dado o critério de teste c.

F, identifica a funcionalidade exercitada por CT em P.

temos que:

$F \in FP$, sse $\{ \forall er_i \in ER(c) \mid er_i \in ERP(c) \}$,

$F \in FNP$, sse $\{ \exists er_i \in ER(c) \mid er_i \in ERR(c) \}$ e

$F \in FR$, sse $\{ \forall er_i \in ER(c) \mid er_i \in ERRm(c) \}$

onde:

FP é conjunto de funcionalidades preservadas após a manutenção,

FNP é o conjunto de funcionalidades não preservadas após a modificação e

FR é o conjunto de funcionalidades removidas pela manutenção

Utilizando-se a proposição acima, podemos afirmar que os casos de teste reutilizáveis serão aqueles cuja funcionalidade estiver preservada; os casos de teste reaplicáveis serão aqueles cuja funcionalidade estiver classificada como não preservada e os casos de teste obsoletos serão aqueles cuja funcionalidade foi removida do programa original. A Proposição 2, a seguir, denota em uma linguagem mais formal, as regras de formação desses conjuntos de casos de teste.

Proposição 2:

Temos que:

$CT \in R(T)$, sse $\{ \exists$ <CT, ER, F> $\mid F \in FP \}$,

ou

$CT \in Ra(T)$, sse $\{\exists$ <CT, ER, F> $\mid F \in FNP \}$,

ou

$CT \in O(T)$, sse $\{ \exists$ <CT, ER, F> $\mid F \in FR \}$.

onde:

R(T) o conjunto de Casos de teste reutilizáveis,

Ra(T) o conjunto de casos de teste reaplicáveis e

O(T) o conjunto de casos de teste obsoletos

3.5 Classificação dos Casos de Teste Selecionados para o Teste de Regressão

Segundo a classificação obtida na seção anterior através das Proposições 1 e 2, os casos de teste que estiverem preservados (conjunto R(T)) não precisarão ser novamente aplicados. Os casos de teste que forem classificados no conjunto O(T) devem ser eliminados da base de casos de teste do programa modificado (P'). Já os casos de teste que forem classificados como não preservados (conjunto Ra(T)) servirão como ponto de partida para a aplicação do teste de regressão. Esta seção discutirá com maiores detalhes as classificações dos testes selecionados para a regressão e também a forma de condução da aplicação dos casos de teste de regressão, uma vez adotada a estratégia de revalidação seletiva.

Uma vez selecionados todos os casos de teste que deverão ser reaplicados no programa modificado (conjunto Ra(T)), é possível, em seguida, aplicar esse conjunto sobre o programa modificado (P') e então avaliar os resultados obtidos. Esses resultados são importantes para que cada caso de teste t do conjunto Ra(T) possa ser classificado em termos de outros dois conjuntos definidos por Rothermel e Harrold [RH96], que são:

a) Testes Reveladores de Modificação: são dados de entrada do casos de teste t que, quando aplicados em P' (programa modificado), produzem um valor de saída diferente do obtido quando aplicado em P (programa original), excluindo-se o caso da saídas coincidentes.

b) Testes Reveladores de Falha: são aqueles dados de entrada do caso de teste t que, quando aplicados em P' (programa modificado), produzem um valor de saída incorreto de acordo com S' (especificação do programa modificado).

Resumidamente, um caso de teste revelador de modificação é aquele cuja execução em P' implicará na execução das partes modificadas desse programa (escopo(P')). Já um caso de teste revelador de falha é aquele cuja execução em P' resultará em um valor diferente do esperado de acordo com a especificação do programa modificado S'. É importante observar que os dois conjuntos podem ser independentes, se forem consideradas também partes do programa não pertencentes ao escopo da manutenção. Nesse caso, um determinado caso de teste t pode pertencer a ambos os conjuntos, apenas ao conjunto de Testes Reveladores de Falha, ou apenas ao conjunto de Testes Reveladores de Modificação.

Por exemplo, um caso de teste t que expõe um erro introduzido em decorrência de uma manutenção (erro de regressão), pertencerá a ambos os conjuntos. Um caso de teste t, que gera um valor de saída diferente do original, mas correto em relação a S', e atravessa o escopo da modificação (escopo(P')), pertence ao conjunto Revelador de

Modificação mas não ao conjunto Revelador de Falhas. Finalmente, um caso de teste t que expõe um erro não revelado durante o teste tradicional do programa P, e não atravessa o escopo da manutenção (escopo(P')), pertence ao conjunto de testes Reveladores de Falha, mas não ao conjunto de teste Reveladores de Modificação.

Essas três combinações dos dois tipos discutidos anteriormente são importantes pois os teste selecionados por uma estratégia de teste de regressão seletiva devem ser, em tese, capazes de revelar situações idênticas apenas às duas primeiras descritas anteriormente. Situações do terceiro tipo (apenas Revelador de Erro), só deveriam ser reveladas se a estratégia adotada for "retest-all", uma vez que a estratégia seletiva isola os elementos requeridos que não pertencem ao escopo da modificação. Em contrapartida, foge aos objetivos principais do teste de regressão revelar erros que não possuem relação com a modificação realizada, e que deveriam ter sido revelados durante o teste tradicional. Além disso, adotar uma estratégia "retest-all" simplesmente para, eventualmente, revelar erros fora do escopo da manutenção, pode aumentar muito a relação custo-benefício do teste de regressão. Na próxima seção serão apresentados e discutidos alguns modelos que auxiliam na escolha da estratégia correta a ser aplicada em cada caso.

Aproveitando o bojo dessa discussão, a adoção do teste de regressão seletivo, utilizando-se critérios da família Potenciais Usos, pode ser considerada uma abordagem intermediária entre o "retest-all" e seletiva usada com outras famílias baseadas em fluxo de dados (como os Critérios de Rapps e Weyuker [RW82, RW85]). A família de critérios Potenciais Usos exigirá que mais elementos requeridos sejam satisfeitos no teste de regressão (vide exemplo da Figura 3.1), pois deverão ser exercitados todos os caminhos/associações entre as definições e um potencial-uso de uma variável. Já os critérios de Rapps e Weyuker exigirão o reteste apenas dos elementos requeridos entre a definição e o uso explícito de uma variável. Resumidamente, a adoção da estratégia seletiva usando-se os critérios Potenciais Usos aumenta o escopo da manutenção e, conseqüentemente, tende a ser mais cara quando comparada com os critérios de Rapps e Weyuker; porém, implica em uma maior probabilidade na revelação de erros. A Figura 3.6 mostra a relação de inclusão entre os elementos requeridos selecionados para o teste de regressão utilizando-se a estratégia "retest-all" e estratégias seletivas (baseadas nas famílias de critérios Potenciais Usos [Mal91] e de Rapps e Weyuker [RW82, RW85]). Estudos empíricos (conduzidos através de experimentos) serão necessários para que essa observação importante sobre ambas as famílias de critérios possa ser comprovada.

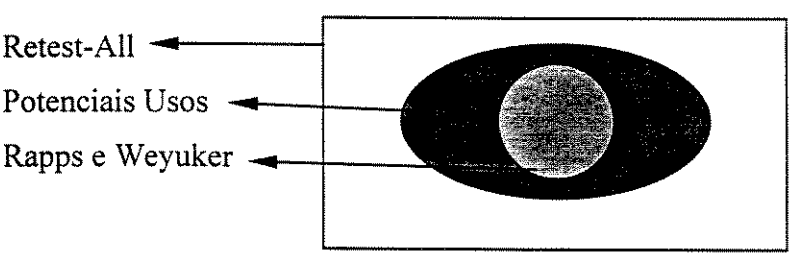


Figura 3.6 - Relação de Inclusão: Estratégias "Retest-All" e Seletivas (Critérios Potenciais Usos e Rapps e Weyuker)

A observação anterior desperta com clareza a essência de uma grande discussão cada vez mais presente entre os pesquisadores. Como conduzir o Teste de Regressão, para atingir seus objetivos? Existem, nos dias de hoje, três grande famílias de técnicas de teste de regressão seletivas que procuram responder essa questão.

A primeira é denominada Família de Técnicas de Minimização [RH96, RW97]. Sua abordagem visa a seleção do menor número de casos de teste de regressão (conjunto minimal) para exercitar as partes do software que sofreram modificação. Uma análise simplista indica que essa família de técnicas tendem a atender um objetivo secundário do teste de regressão, que é a minimização de custos (vide Seção 2.2). Porém, a aplicação de técnicas de minimização pode ser muito cara, devido à natureza combinatória e exponencial desse tipo de técnica.

A segunda família é conhecida como Baseada em Cobertura [RH96, RW97]. Sua abordagem procura selecionar os componentes que devem ser novamente exercitados e testá-los com um número suficiente de casos de teste (em geral maior que o minimal). Essa abordagem pode ser considerada intermediária entre as três famílias.

A terceira família é denominada Técnicas Baseadas em Abordagens Seguras [RH96, RW97]. Tem como objetivo central identificar as partes modificadas do código e selecionar qualquer caso de teste que, potencialmente, possa expor um erro nesse programa modificado. Essa estratégia vai ao encontro do principal objetivo do teste de regressão, que é revelar erros introduzidos durante o processo de manutenção. A maioria das técnicas e ferramentas apresentadas utilizam essa tendência, que vem sendo adotada freqüentemente nos dias de hoje, sendo também aquela que, em tese, mais se aproxima do ponto ótimo entre estratégias "retest-all" e seletivas.

A Figura 3.7 mostra um quadro onde cada uma das famílias discutidas acima são apresentadas, em comparação com a estratégia "retest-all".

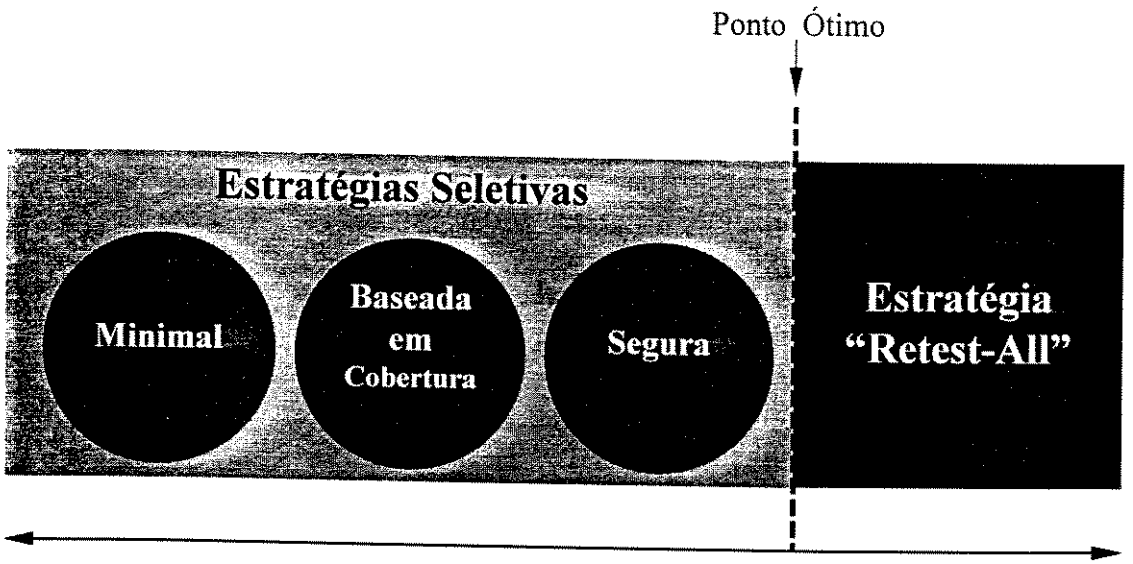


Figura 3.7 - Famílias de Técnicas/Ferramentas de Teste de Regressão (Seletivas x "Retest-All")

As pesquisas mais recentes [RW97, RH96] apontam a necessidade de selecionar o conjunto mais próximo do mínimo; porém a necessidade de selecionar sempre um conjunto minimal pode ser muito cara. O esforço para selecionar conjunto minimais pode não valer a pena, dependendo do escopo da manutenção que foi realizada. As pesquisas apontam que as estratégias de seleção e aplicação dos casos de teste de regressão deve considerar, prioritariamente, a maximização da revelação de erros. A tendência atual na área dos testes de regressão é selecionar um conjunto de testes que maximize a revelação de erros e que não ultrapasse o ponto ótimo entre as estratégias "retest-all" e minimal.

3.6 Modelos de Custo de Teste de Regressão: Um Apoio na Escolha da Estratégia de Revalidação

Conforme já discutido no Capítulo 2 deste trabalho, manutenção e teste de regressão são duas atividades da Engenharia de Software que demandam muitos recursos em sua correta condução. Uma questão de ordem prática, que pode ser decisiva na economia de recursos, deve ser tomada quando da escolha da estratégia de aplicação do teste de regressão a ser aplicada: "retest-all" ou seletiva? Cada uma dessas estratégias deve ser aplicada sob determinadas circunstâncias favoráveis. A determinação de um ponto ótimo entre as duas estratégias vem despertando grande interesse entre os pesquisadores, com o objetivo de se encontrar um modelo que, quando aplicado com precisão, possa auxiliar essa difícil decisão.

O primeiro modelo, proposto por Leung e White [LW91], é baseado na cardinalidade do conjunto de casos de teste a serem reaplicados (conjunto T'). O segundo modelo, proposto por Rosenblum e Weyuker [RW96], é fundamentado em informações de cobertura do programa e servem para ajustar o modelo básico inicialmente proposto por Leung e White.

3.6.1 Modelo de Custo de Leung e White

Leung e White [LW91] apontam que muitos fatores afetam o custo do teste de regressão. Alguns desses custos, de natureza indireta (e.g., Custo de armazenamento da base de dados de teste) são difíceis de mensurar. Por esse motivo, custos indiretos foram eliminados desse modelo. Os custos possíveis de mensurar, ou custos diretos, foram identificados em quatro grupos:

a) Custo de Análise de Sistemas, que incluem, por exemplo, o custo de tornar o grupo que está conduzindo a manutenção familiar com a especificação do sistema como um todo;

b) Custo de Seleção dos Testes, que incluem a análise e seleção dos testes a serem novamente aplicados durante a regressão;

c) Custo de Execução do Teste de Regressão, que inclui o custo de aplicação dos testes (de forma manual ou automatizada); e

d) Custo de Analise dos Resultados, que inclui a o comparação dos resultados obtidos com os resultados esperado.

Esses fatores são mensurados de forma diferente para cada estratégia. Por exemplo, o custo de Análise dos resultados é maior na estratégia "retest-all". Em contrapartida, o custo de Seleção do teste da estratégia seletiva é alto enquanto na estratégia "retest-all" é zero. Todos esses quatro componentes do modelo são relacionados entre si. Através de um seqüência de simplificações e deduções, que podem ser averiguados em detalhes na referência citada, Leung e White chegaram a uma inequação final , que representa o ponto ótimo de seu modelo:

$$s < (e+c) (1 - |T'| / |T|)$$

onde:

s,e,c são respectivamente o custo de seleção, execução e comparação de resultados,

$|T'|$ é a cardinalidade do conjunto de teste selecionados para a regressão e

$|T|$ é a cardinalidade do conjunto de testes original.

Segundo o modelo, o custo da seleção dos elementos requeridos deve ser menor que o custo de execução e comparação do resultado da inequação. Considerando-se s e (e+c) linearmente dependentes e retirando-os do modelo, a estratégia seletiva é vantajosa se o custo de seleção de T' for inferior à metade do custo de execução e comparação de metade do testes armazenado no conjunto T, ou $|T'| < 0.5 |T|$.

3.6.2 Modelo de Custo de Rosenblum e Weyuker

A motivação principal da definição do modelo de custo de Rosenblum e Weyuker [RW97] foram algumas observações que seus autores fizeram sobre o modelo de Leung e White. Os autores argumentam que simplificações realizadas por Leung e White podem afetar a precisão de algumas medidas importantes do modelo. A principal observação foi o fato de o modelo original considerar os custos de execução e avaliação constantes para todos os casos de teste. Para sistemas de maior porte, sabe-se que isso não é verdadeiro.

Baseado principalmente nessa observação, os autores definiram o conceito de "*predictor*" para o teste de regressão. Esse "*predictor*" nada mais é que uma medida (valor entre 0 e 1) que ajusta a fórmula do modelo de Leung e White, considerando que os casos de teste contribuem com pesos diferentes no ponto ótimo entre as estratégias seletiva e "restest-all", ou seja, substitui o valor constante 0.5 apresentado no modelo original, discutido na seção anterior.

A argumentação é bastante simples. Casos de teste que, no programa original, exercitaram um número maior de entidades (elementos requeridos) devem possuir um peso maior no cálculo do "*predictor*" (π), dada uma estratégia seletiva qualquer. Esse valor é calculado através da razão entre o Custo Cumulativo dos casos de teste (CC), que representa o somatório de todos os elementos requeridos eliminados por todos os

casos de teste (daí o aumento do peso dos casos de teste mais custosos), e produto da cardinalidade do conjunto T (testes originais) e E^C (elementos requeridos exercitados pelos casos de teste selecionados para regressão). A fórmula está descrita a seguir:

$$\pi = CC / |E^C| |T|$$

A estimativa apontada por π é mais precisa que o valor constante 0.5, pois nela o peso dos casos de teste estão medidos de acordo com os número de elementos requeridos que cada um elimina originalmente. Os autores comprovaram essas observações através de padrões que relacionam casos de teste e elementos requeridos, e também através de dois experimentos que foram conduzidos. A aplicação do "*predictor*" π aumenta muito a precisão do modelo de Leung e White.

$$|T'| < \pi |T|$$

3.7 Manutenção da Configuração do Ambiente: Última Atividade de uma Ferramenta de Teste de Regressão

O último ponto a ser discutido nesse capítulo trata da garantia da consistência da configuração de um ambiente de teste de regressão, em especial do conjunto de casos de teste ("*regression testing suite*"). Conforme discutido no decorrer deste capítulo, após a seleção dos elementos requeridos que serão novamente exercitados (E''), são selecionados os casos de teste originais que serão reaplicados (T') e, se necessário, novos testes serão definidos (T''). As ferramentas de teste de regressão possuem como característica o trabalho em conjunto com ferramentas de teste. Assim, o conjunto E' e os conjuntos T' e T'' representam subconjuntos que são definidos para que o teste seja realizado apenas considerando o escopo da modificação.

Sendo assim, quando for encerrado o teste de regressão, é necessário que os subconjuntos T' e T'' sejam novamente compostos com o restante dos casos de teste ((T-T') - To), que não foram selecionados para o teste de regressão (To representa o conjunto de casos de teste obsoletos do programa original). Esse conjunto completo de casos de teste (T''') deve ser mantido íntegro, de forma a possibilitar um novo procedimento de teste de regressão, se isso for necessário em uma nova oportunidade.

Antes de serem adicionados ao conjunto T''', devem ser identificados e eliminados dos conjuntos T' e T'' os casos de teste obsoletos. Assim que toda a base estiver recomposta é necessário, ainda, verificar a existência de casos de teste redundantes. Define-se como caso de teste redundante aquele que não provê nenhum aumento de cobertura (em termos de teste estrutural), nem exercita alguma funcionalidade não testada do programa (em termos de teste funcional). Se algum casos de teste dessa categoria for localizado, o mesmo deverá ser retirado da base. Esse processo requer um cuidado especial, pois em ferramentas que suportam vários critérios (multi-critérios), um caso de teste pode ser redundante para um determinado critério e não para outro.

A atividade de atualização da base de casos de teste é uma atividade fundamental que pode ser realizada após encerrado o teste de regressão (em geral é a

última tarefa a ser conduzida). A não execução dessa atividade poderá gerar bases de teste cada vez maiores, contendo testes que não possuem relação com elementos do estado atual do programa. Esse fato gera, consequentemente, dificuldade de manipulação dessa base (mesmo utilizando-se ferramentas automáticas). A base deve, pois, ser mantida com a menor cardinalidade possível (eliminado-se testes obsoletos e redundantes) e que represente o estado final do teste de regressão conduzido (T' $\cup$ T''), adicionada do estado anterior dos casos de teste que não foram selecionados para o teste de regressão ((T-T') - To), retirando-se, ainda os casos de teste redundantes (Tr) que forem localizados nessa base. Assim, temos que T''' é formado pela expressão dada a seguir.

$$T''' = ((((T-T') - To) \cup (T' \cup T'')) - Tr)$$

Capítulo 4

Análise, Projeto e Implementação da Ferramenta

Nesse capítulo serão apresentados os aspectos relevantes da implementação da ferramenta de apoio ao teste de regressão, denominada **RePoKe-Tool** (Regression Testing Support for Potencial-Uses Criteria Tool). Inicialmente, será apresentado o modelo funcional da ferramenta, cujo objetivo principal é automatizar, através do uso da estratégia seletiva e seleção segura de casos de teste, o procedimento de teste de regressão de programas (unidades) que já foram submetidos anteriormente à ferramenta de teste POKE-TOOL. Será utilizada como técnica de modelagem e representação o Diagrama de Fluxo de Dados. Em seguida, serão apresentados a Arquitetura utilizada na implementação da ferramenta, sua integração com a ferramenta POKE-TOOL e os modelos de implementação mais importantes. Os detalhes do projeto de implementação dos módulos da ferramenta, implementados em linguagem C ou através de "scripts", serão objeto das últimas considerações desse capítulo.

4.1 Modelagem Funcional

Encerrada a fase de estudos preliminares, levantamento e definição dos requisitos (discutidos nos capítulos anteriores), a próxima etapa do desenvolvimento de software, seguindo o modelo clássico [Pre92], é o desenvolvimento dos modelos que o representam. A técnica escolhida para esse projeto foi a da modelagem funcional, utilizando-se o Diagrama de Fluxo de Dados (DFD).

Utilizando a notação do Diagrama de Bolhas [Dem79], as Figuras 4.1 e 4.2 representam, o Diagrama de contexto (DFD nível 0) e o DFD nível 1 da ferramenta, respectivamente. Foram identificadas e modeladas as principais funções da ferramenta de apoio ao teste de regressão. A seguir, serão discutidos os principais aspectos de cada uma dessas macro funções.

i) **Identificação das Modificações**: Conforme já discutido, a atual versão da ferramenta suporta apenas o teste de unidade. Assim sendo, essa função tem como objetivo identificar a unidade que sofreu o processo de manutenção, e que será submetida ao teste de regressão. A identificação dessa unidade dará início a uma série de sub-funções: (a) Execução da Análise Estática (realizada pela ferramenta POKE-TOOL) da unidade modificada que foi identificada, para que possa ser aplicado o modelo de seleção do teste de regressão; (b) Identificação das modificações no fluxo de controle e de dados.

ii) **Qualificação dos Elementos Requeridos**: Essa segunda função do sistema é a maior de todas. Para que seu objetivo seja alcançado é necessária a execução de uma série de sub-funções importantes (discutidas e detalhadas uma a uma na Seção 3.3): (a) Calculo do Escopo da modificação para cada critério; (b) Seleção dos elementos requeridos que deverão ser reavaliados para cada critério; (c) Qualificação dos elementos requeridos da função modificada e relacionamento da equivalência de cada um com os elementos do programa original.

iii) **Qualificação dos Casos de Teste Originais**: A terceira grande função do software tem como principal objetivo qualificar e selecionar os casos de teste originais que serão reaplicados no teste de regressão (vide Seção 3.4).

iv) **Aplicação de um Modelo de Custo**: A quarta função do sistema permitirá a implantação de um (ou mais) modelos de custo que auxiliem na decisão da continuidade da estratégia seletiva ou se essa deve ser abandonada em detrimento da aplicação da estratégia "retest-all" (vide Seção 3.6).

v) **Ajuste da Sessão de Teste para Regressão**: Essa função tem como objetivo apenas preparar o ambiente reduzido para que o testador possa conduzir o teste de regressão, usando a ferramenta de teste POKE-TOOL.

vi) **Aplicação do Teste de Regressão**: Essa função é desempenhada pela ferramenta POKE-TOOL. Consiste da aplicação dos casos de teste selecionados para o teste de regressão, bem como verificação dos resultados obtidos. Além dos testes selecionados, caso haja necessidade, deverão ser aplicados novos testes, com objetivo de testar novas funcionalidades e/ou atingir um nível de cobertura desejado, dado algum critério de teste.

vii) **Reconfigura Sessão de Teste**: A última função do sistema será executada assim que o testador estiver satisfeito com os resultados obtidos no teste de regressão. Seu objetivo é atualizar a base de dados do teste de forma a torná-la consistente, ou seja, contendo apenas os casos de teste estritamente necessários, bem como os arquivos da sessão de teste atualizados, de forma a possibilitar a condução futura de um novo procedimento de teste de regressão.

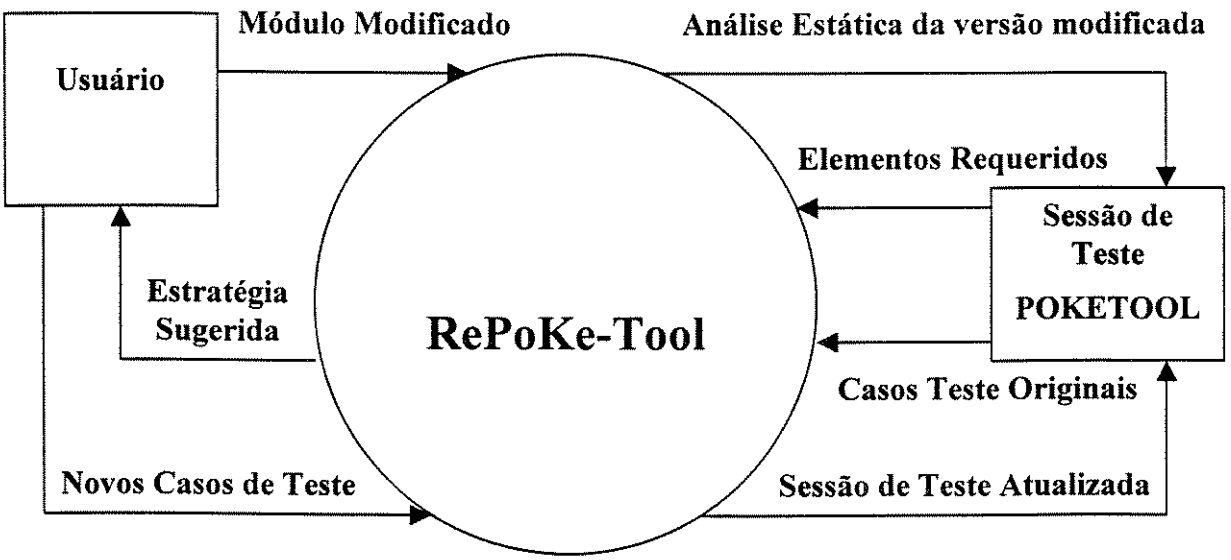


Figura 4.1 – Diagrama de Contexto (DFD Nível 0) da Ferramenta

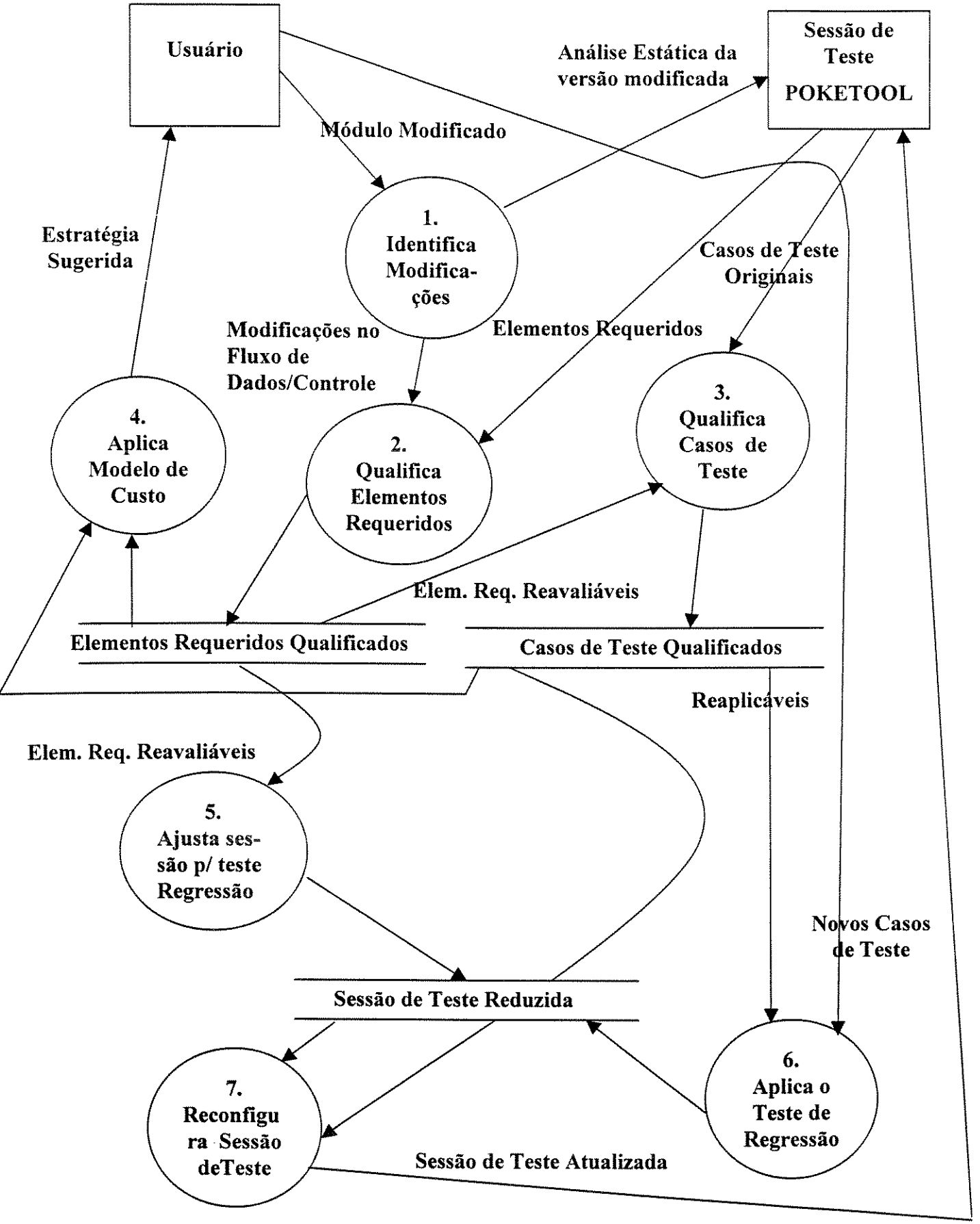


Figura 4.2 – Diagrama de Fluxo de Dados (Nível 1) da Ferramenta

A Figura 4.3 mostra a explosão das duas primeiras bolhas (funções) definidas no Diagrama de Fluxo de Dados, cujas características já foram detalhadas.

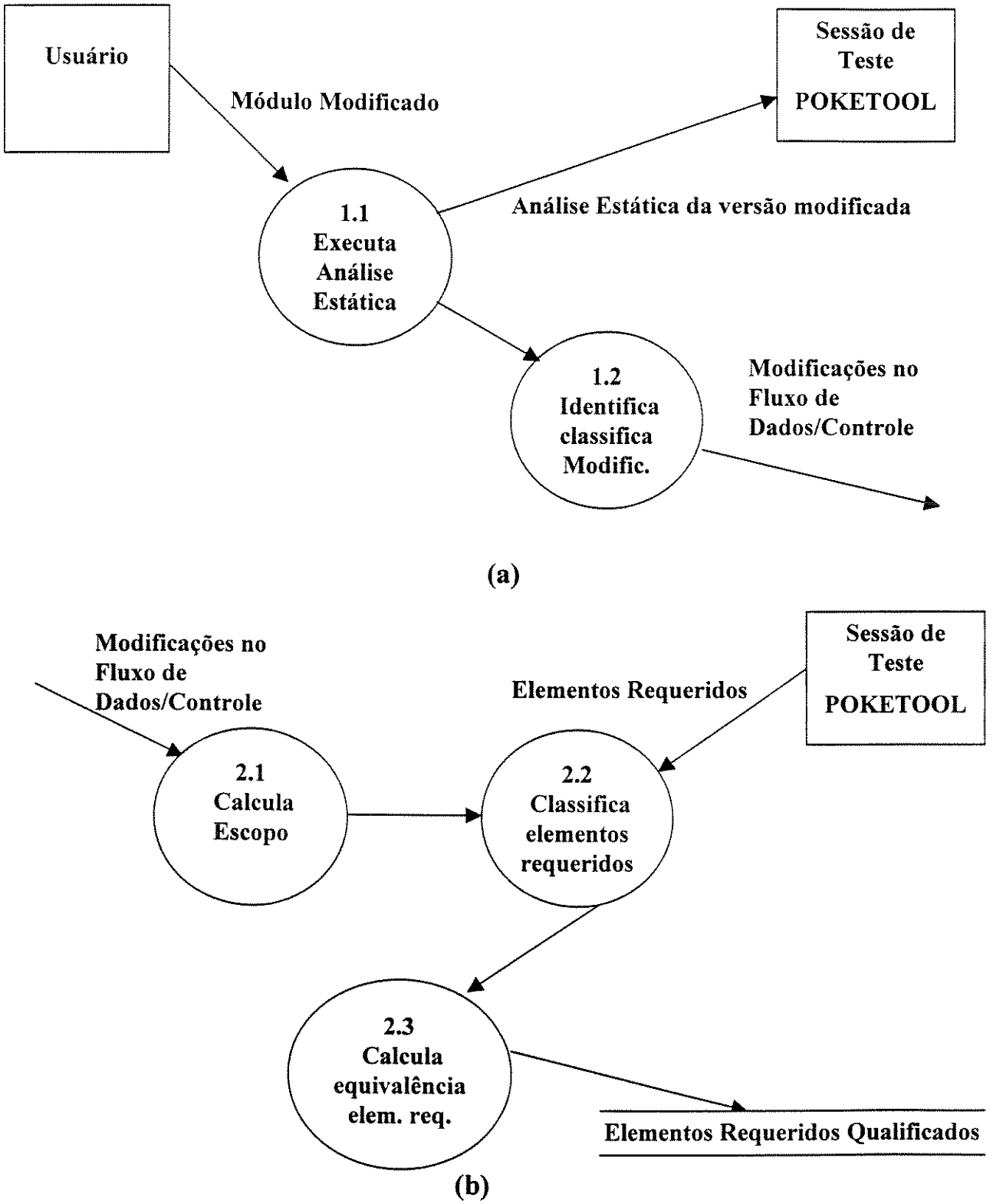


Figura 4.3 – (a) Explosão da Bolha Identifica Modificações

(b) Explosão da Bolha Qualifica Elementos Requeridos

4.2 Arquitetura da Ferramenta

Encerrada a especificação funcional, cabe agora a definição dos módulos que serão implementados para a ferramenta de apoio ao teste de regressão. Foram definidos quatro grandes módulos (que serão discutidos em detalhes na Seção 4.4):

Módulo I: Fase Estática - Nessa etapa todos as informações estáticas geradas pela POKE-TOOL (para as duas versões da função) são avaliadas e trabalhadas com o objetivo de identificar, qualificar e selecionar os elementos requeridos que devem ser aplicados no teste de regressão. Esse módulo é formado pelas duas primeiras funções descritas no Diagrama de Fluxo de Dados e, naturalmente, pelas funções derivadas da explosão de cada uma dessas bolhas.

Módulo II: Fase Dinâmica - Esse segundo módulo considera as informações dinâmicas recolhidas da sessão de teste aplicada na versão original da função modificada. Essas informações são avaliadas e trabalhadas de forma a identificar, qualificar e selecionar os casos de teste que deverão ser reaplicados durante a condução do teste de regressão. A terceira e a quarta bolhas, descritas no DFD, formam esse segundo módulo.

Módulo III: Ajusta a Sessão de Teste - Esse módulo, derivado exclusivamente da bolha número cinco do DFD, tem como objetivo ajustar o ambiente da POKE-TOOL para que seja conduzido teste de regressão. Essa sessão reduzida contém apenas os Casos de Teste e Elementos Requeridos selecionados pelo Módulo II.

Módulo IV: Configuração Final da Sessão de Teste - Encerrada a execução do teste de regressão pela POKE-TOOL (usando o ambiente reduzido), o último grande módulo da ferramenta tem como objetivo restabelecer a base de dados da sessão de teste, de forma a mantê-la consistente para uma eventual nova aplicação da RePoKe-Tool. Esse módulo é derivado, exclusivamente, da sétima e última bolha definida no Diagrama de Fluxo de Dados.

A Figura 4.4 mostra um modelo da arquitetura da ferramenta de apoio ao teste de regressão, e sua relação com a ferramenta de teste POKE-TOOL.

4.3 Modelos de Implementação

Conforme já discutido no capítulo anterior, modificações na estrutura de um programa podem resultar em contrações ou expansões no Grafo de Fluxo de Controle. Para ambos os casos, a modificação pode ocorrer em um nó ou mesmo em um arco.

Quando a modificação de fluxo de controle é realizada em um nó, esse poderá ser expandido para qualquer estrutura de programação suportada pelos modelos implementados, que são *if-then, if-else, while, for, do-while* ou combinação desses. A contração, ao contrário, transforma uma estrutura dessas em um único nó do grafo.

Por sua vez, a modificação em arcos acontece em apenas duas situações. A primeira, quando houver inserção de um *else* em uma estrutura *if-then*. Nesse caso, o

arco que representa a inexistência da opção else (realiza a transição diretamente do nó de predicado para o nó terminal do *if-then*), poderá ser expandido para qualquer tipo de estrutura de programação suportada pela ferramenta, até mesmo um nó simples. A segunda situação é exatamente inversa à primeira, pois trata-se da eliminação de qualquer estrutura de programação situada na opção else de um *if-else* gerando, pois, o arco que transita do nó de predicado para o nó de saída da estrutura *if-then*.

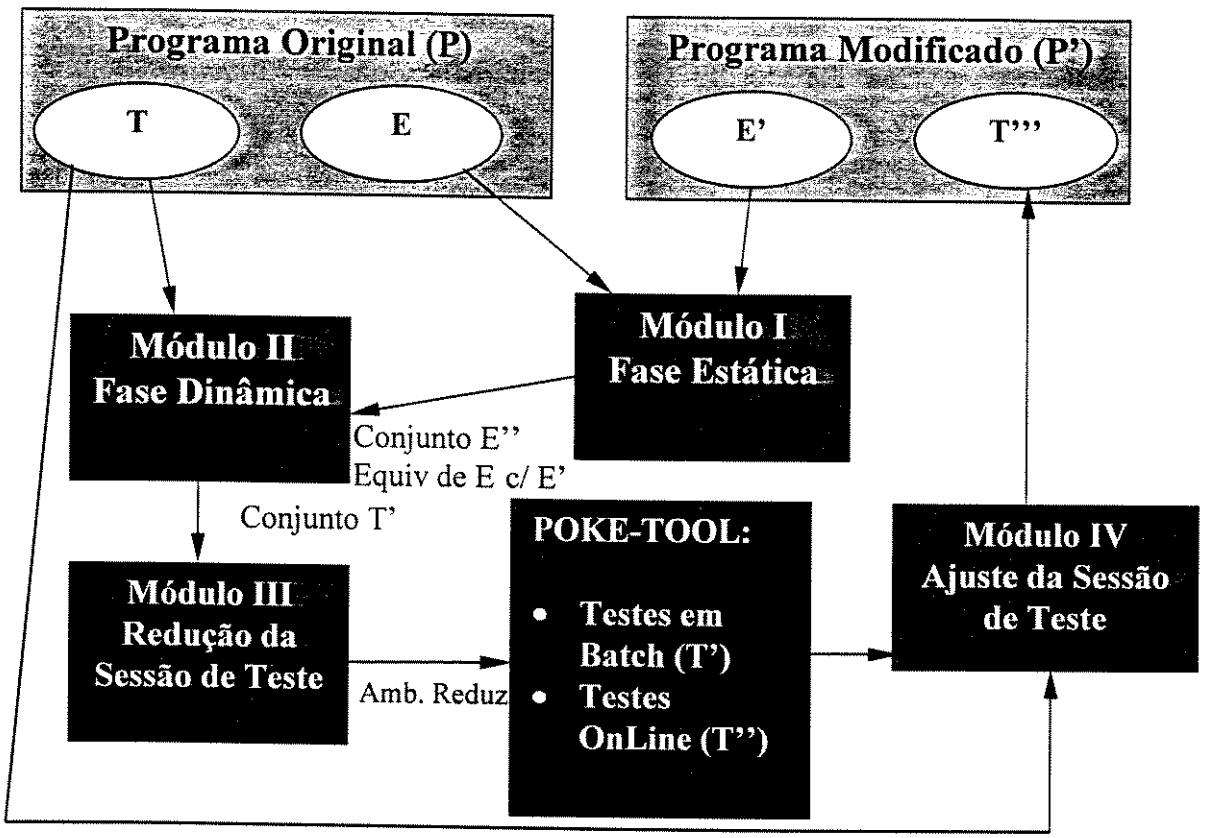


Figura 4.4 – Arquitetura da Ferramenta e sua relação com a POKE-TOOL

A seguir, serão apresentados os modelos de contração e expansão para cada uma dessa estruturas, com exceção dos modelos nó→ if-else que foram apresentados na Seção 3.3.1.1 (Figuras 3.3 e 3.4).

4.3.1 Modelos de Expansão do Grafo de Fluxo de Controle

Serão apresentados nesta seção os seis modelos básicos de expansão, tanto de arcos {*if-then→ if-else*} como de nós (demais modelos). Esses modelos básico, descritos a seguir, podem ser relacionados dando origem a estruturas combinadas (e.g. expansão de um nó para *if-else* com *do-while* na opção *then* e um *for* na opção *else*).

1. **Modelo de expansão { *if-then→if-else*} – Transformação de um *if-then* em um *if-else* com qualquer estrutura na opção *else***: Conforme já citado, esse é o único modelo de expansão de arco, pois a estrutura *then* já é composta por um nó (que também poderá ser expandido) e, no lugar do

arco *else*, que conduz o fluxo de controle do nó de predicado par o nó de saída do *if-then*, poderá ser inserido qualquer um dos modelos descritos nesta seção. A Figura 4.5 mostra a expansão desse arco para um nó *else* simples; qualquer um dos modelos poderia ser utilizado nesse *else*.

2. **Modelo de expansão { *Nó* → *if-then* } – Transformação de um *nó* em um *if-then*:** Esse modelo de expansão (Figura 4.6) descreve a transformação de um nó qualquer de um GFC em uma estrutura condicional *if-then*.

3. **Modelo de expansão { *Nó* → *if-else* } – Transformação de um *nó* em um *if-else*:** Esse modelo de expansão (Figura 3.3) descreve a transformação de um nó qualquer de um GFC em uma estrutura condicional *if-else*.

4. **Modelo de expansão { *Nó* → *while* } – Transformação de um *nó* em um *while*:** Esse modelo, que também representa expansões do tipo (***Nó* → *for***) (vide Figura 4.7), descreve a transformação de um nó qualquer de um GFC em uma estrutura iterativa *while*.

5. **Modelo de expansão { *Nó* → *do-while* } – Transformação de um *nó* em um *do-while*:** Esse modelo de expansão (Figura 4.8) descreve a transformação de um nó qualquer de um GFC em uma estrutura iterativa *do-while*.

6. **Modelo de expansão { *Nó* → *combinação* } – Transformação de um nó em uma estrutura combinada:** Esse modelo de expansão (vide um exemplo na Figura 4.9) descreve a transformação de um nó qualquer do GFC em uma estrutura que combina todas as que foram discutidas anteriormente. Torna-se útil na medida em que manutenções em ambiente reais nem sempre são bem comportadas como nos casos anteriores.

Antes **Após Expansão**

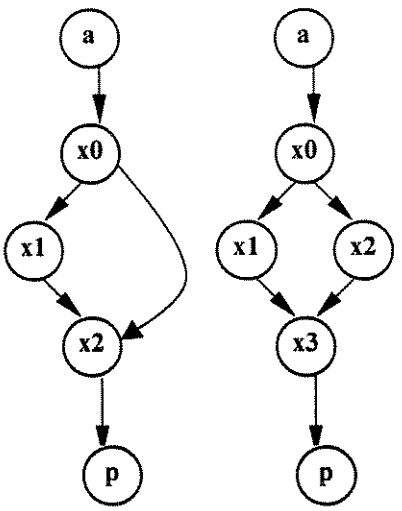


Tabela do Modelo de Expansão

Nó Expandido	Conjunto de Expansão	Equivalência
x0	NP_{fc}	x0
x1	NP_{fc}	x1
x2	NN_{fc}	-
x3	NP_{fc}	x2

Figura 4.5 - Modelo de Expansão: {*if-then*→*if-else*} (de *if-then* para *if-else*)

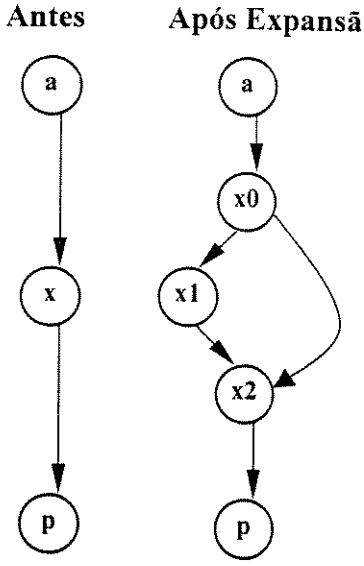


Tabela do Modelo de Expansão

Nó Expandido	Conjunto de Expansão	Semi Equivalência
x0	NI_{fc}	x0
x1	NN_{fc}	x1
x2	NO_{fc}	x0

Figura 4.6 - Modelo de Expansão: {*nó*→ *if-then*} (de *nó* para *if-then*)

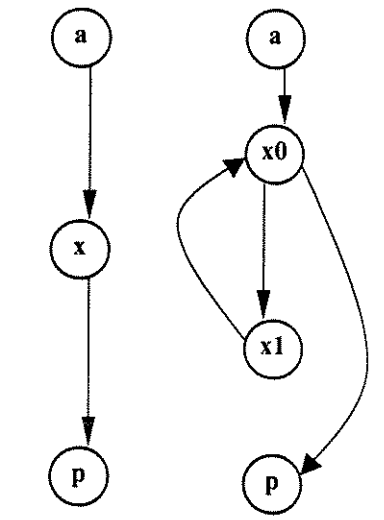


Tabela do Modelo de Expansão

Nó Expandido	Conjunto de Expansão	Semi Equivalência
x0	NI_{fc} e NO_{fc}	x
x1	NN_{fc}	x

Figura 4.7 - Modelo de Expansão: {*nó*→ *while*} (de *nó* para *while*)

Antes **Após Expansão**

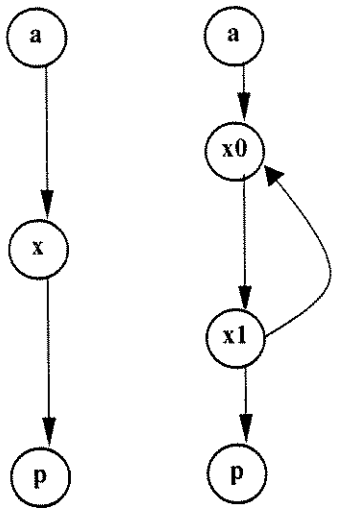


Tabela do Modelo de Expansão

Nó Expandido	Conjunto de Expansão	Semi Equivalência
x0	NI_{fc}	x
x1	NN_{fc}	-

Figura 4.8 - Modelo de Expansão: {*nó*→ do-*while*} (de *nó* para *do-while*)

Antes **Após Expansão**

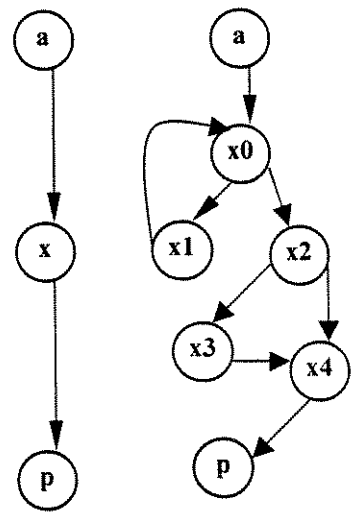


Tabela desse Modelo de Expansão

Nó Expandido	Conjunto de Expansão	Semi Equivalência
x0	NI_{fc}	x
x1	NN_{fc}	-
x2	NN_{fc}	-
x3	NN_{fc}	-
x4	NO_{fc}	X

Figura 4.9 - Modelo de Expansão: {*nó*→combinação} (de *nó* para *estrutura combinada*)

4.3.2 Modelos de Contração do Grafo de Fluxo de Controle

Serão apresentados nesta seção os seis modelos básicos de contração de grafo, tanto de arcos {*if-else*→ *if-then*} como de nós (demais modelos). Da mesma forma que na expansão, os modelos básicos descritos a seguir podem ser relacionados dando origem à contração de estruturas combinadas.

1. **Modelo de contração { *if-else*→*if-then*} – Transformação de um *if-else* com qualquer estrutura na opção *else* em um *if-then***: Esse é o único modelo de contração de nó(s) para arco, pois serão removidos todos os nós

hierarquicamente dependente desse *else* (que será eliminado). No lugar desses nós será introduzido um arco que conduz o fluxo de controle do nó de predicado para o nó de saída do *if-then* gerado. A Figura 4.10 mostra a contração de um nó simples para um arco *else*, porém, qualquer um dos modelos básicos poderia estar implementado no *else* que foi contraído.

2. **Modelo de contração { *if-then* → *Nó* } – Transformação de um *if-then* em um *nó*:** Esse modelo de contração (Figura 4.11) descreve a contração de uma estrutura condicional *if-then* em um nó simples de um GFC.
3. **Modelo de contração { *if-else*→ *Nó* } – Transformação de um *if-else* em um *nó*:** Esse modelo de contração (Figura 3.4) descreve a contração de uma estrutura condicional *if-else* em um nó simples de um GFC.
4. **Modelo de contração { *while* → *Nó* } – Transformação de um *while* em um *nó*:** Esse modelo, que também representa expansões do tipo (***for* → *nó***), (vide Figura 4.12) descreve a contração de uma estrutura iterativa *while* em um nó simples de um GFC.
5. **Modelo de contração { *do-while* → *Nó* } – Transformação de um *do-while* em um *nó*:** Esse modelo de contração (Figura 4.13) descreve a transformação de uma estrutura iterativa *do-while* em um nó simples de um GFC.
6. **Modelo de contração { *combinação* → *Nó* } – Transformação de uma *estrutura combinada* em um *nó*:** Esse modelo de contração (vide um exemplo na Figura 4.14) descreve a transformação de uma estrutura que combina todas as que foram discutidas anteriormente em um nó qualquer do GFC. Torna-se útil na medida em que manutenções em ambiente reais nem sempre são bem comportadas como nos modelos básicos.

Antes **Após Contração**

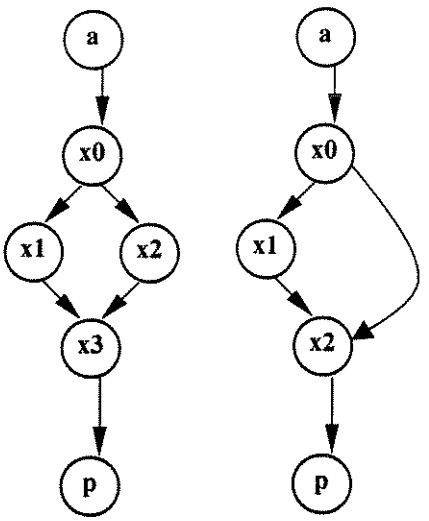


Tabela do Modelo de Contração

Nó Aglutinado	Conjunto de Contração	Equivalência
x0	NP_{fc}	x0
x1	NP_{fc}	x1
-	NR_{fc}	x2
x2	NP_{fc}	x3

Figura 4.10 - Modelo de Contração: {*if-else*→*if-then*} (de *if-else* para *if-then*)

Antes **Após Contração**

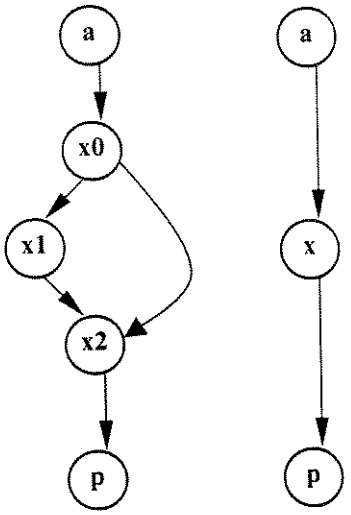


Tabela do Modelo de Contração

Nó Aglutinado	Conjunto de Contração	Semi Equivalência
x	NI_{fc}	x0
x	NO_{fc}	x2
-	NR_{fc}	x1

Figura 4.11 - Modelo de Contração: { *if-then* → *nó* } (de *if-then* para *nó*)

Antes **Após Contração**

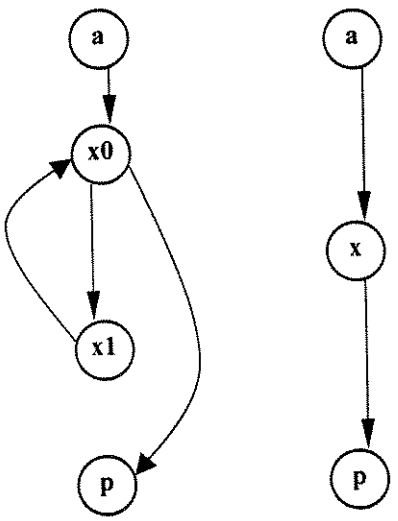


Tabela do Modelo de Contração

Nó Aglutinado	Conjunto de Contração	Semi Equivalência
x	NI_{fc} e NO_{fc}	x0
-	NR_{fc}	x1

Figura 4.12 - Modelo de contração: {*while* →*nó*} (de *while* para *nó*)

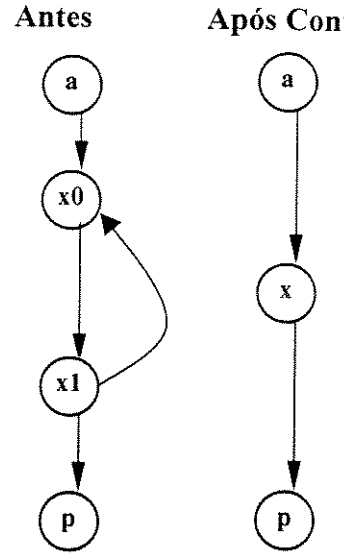


Tabela do Modelo de Contração

Nó Aglutinado	Conjunto de Contração	Semi Equivalência
x	NI_{fc}	x0
-	NR_{fc}	x1

Figura 4.13 - Modelo de Contração: {*do-while* → *nó* } (de *do-while* para *nó*)

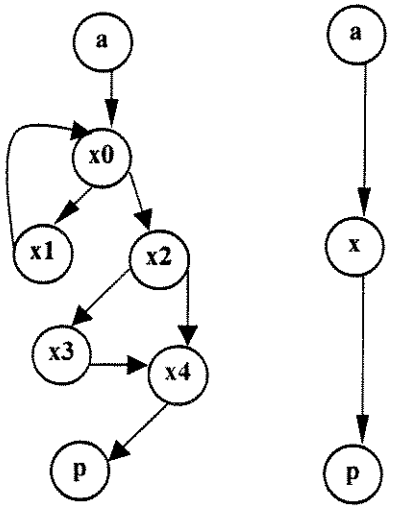


Tabela desse Modelo de Contração

Nó Aglutinado	Conjunto de Contração	Semi Equivalência
x	NI_{fc}	x0
x	NO_{fc}	x4
-	NR_{fc}	x1
-	NR_{fc}	x2
-	NR_{fc}	x3

Figura 4.14- Modelo de Contração:{combinação→*nó*} (de *estrutura combinada* para *nó*)

4.4 Projeto da Ferramenta

Na Seção 4.2 foram descritos e brevemente comentados os módulos componentes da RePoKe-Tool, bem como a relação dos mesmos com a ferramenta de teste POKE-TOOL. Nesta seção serão apresentadas as principais informações sobre o projeto da ferramenta, em especial a hierarquia dos módulos e os principais arquivos gerados por cada um deles.

A RePoKe-Tool foi projetada para funcionar em conjunto com a versão mais recente implementada da POKE-TOOL, que foi desenhada para ser executada em

estações de trabalho SUN[1] SPARC[2] sob o sistema operacional UNIX[3] e é denominada de POKE-TOOL versão Script[4]. Essa versão foi escolhida por ser a mais atualizada e também porque provê arquivos com informações mais detalhadas sobre a execução das fases estáticas e dinâmicas do teste (e.g., arquivos contendo a relação de elementos requeridos eliminados por cada caso de teste, arquivo de tabela de definição de variáveis "`tabvardef.tes`", etc.).

A Figura 4.15, mostra o projeto básico da ferramenta. Nessa ilustração, os retângulos representam os módulos que devem ser executados pela ferramenta; a lista de informações sob cada retângulo apresentam as principais informações geradas por cada um dos módulos[5]; os paralelogramos, as informações fornecidas pelos usuários e os cilindros arquivos da POKE-TOOL que são utilizados pela ferramenta. A ordem de execução dos módulos é dada pelas linhas pontilhadas (fluxo de controle da ferramenta) e as linhas cheias representam o fluxo de transição das informações entre esses diversos módulos. Nas próximas seções cada um desses módulos será discutido detalhadamente.

4.4.1 Módulo I – Fase Estática

Conforme o próprio nome sugere, o objetivo desse primeiro módulo é levantar as informações estáticas do programa modificado que deverão ser reavaliadas no teste de regressão. Resumidamente, essas informações são os elementos requeridos que deverão ser submetidos ao teste de regressão, para cada um dos critérios suportados pela POKE-TOOL. Para atingir esse objetivo, o módulo I foi dividido em cinco módulos menores, detalhados na Figura 4.16.

Para executar a ferramenta, o usuário deverá executar o script `repoketool`, quando será solicitado que sejam identificados a unidade (função) que sofreu manutenção e o nome do arquivo onde está armazenada. A primeira ação desse "script" será modificar o nome do diretório que contém as informações da seção de teste da unidade original; em seguida, será executado o script `poketool`, que realizará a análise estática da versão modificada da função. Terminada essa primeira etapa (Módulo I.1), estarão disponíveis a base de dados da versão original do programa e de sua versão modificada. Adicionalmente, será gerado o arquivo `diferenças`, que contém as diferenças entre os programas fonte original e modificado (lançando-se mão o utilitário *diff* do UNIX), para auxiliar o usuário na localização das alterações realizadas. Em seguida, será executado o restante da ferramenta, escrita em linguagem C.

Na segunda etapa (Módulo I.2), o usuário identificará as modificações de forma semi-automática (vide Seção 3.3.1). Com essas informações, serão calculados todos os conjuntos de equivalência descritos na Seções 3.3.1.1, 3.3.1.2 e 3.3.1.3

[1] SUN é Marca Registrada da SUN Microsystems.
[2] SPARC é Marca Registrada SPARC International Inc.
[3] UNIX é Marca Registrada, licenciada pela X/OPEN Company Ltd.
[4] Existe também a versão para MS-DOS, porém com menos recursos que a versão Script.
[5] Não há nenhum padrão de representação gráfica dessas informações, que podem estar armazenadas em listas ligadas ou lineares ou mesmo em arquivos.

(índice *fc* para as que afetam o Fluxo de Controle; ***def, pre, uso*** para as que afetam Fluxo de Dados; ***não*** para as que não afetam Fluxo de Controle e/ou Fluxo de Dados).

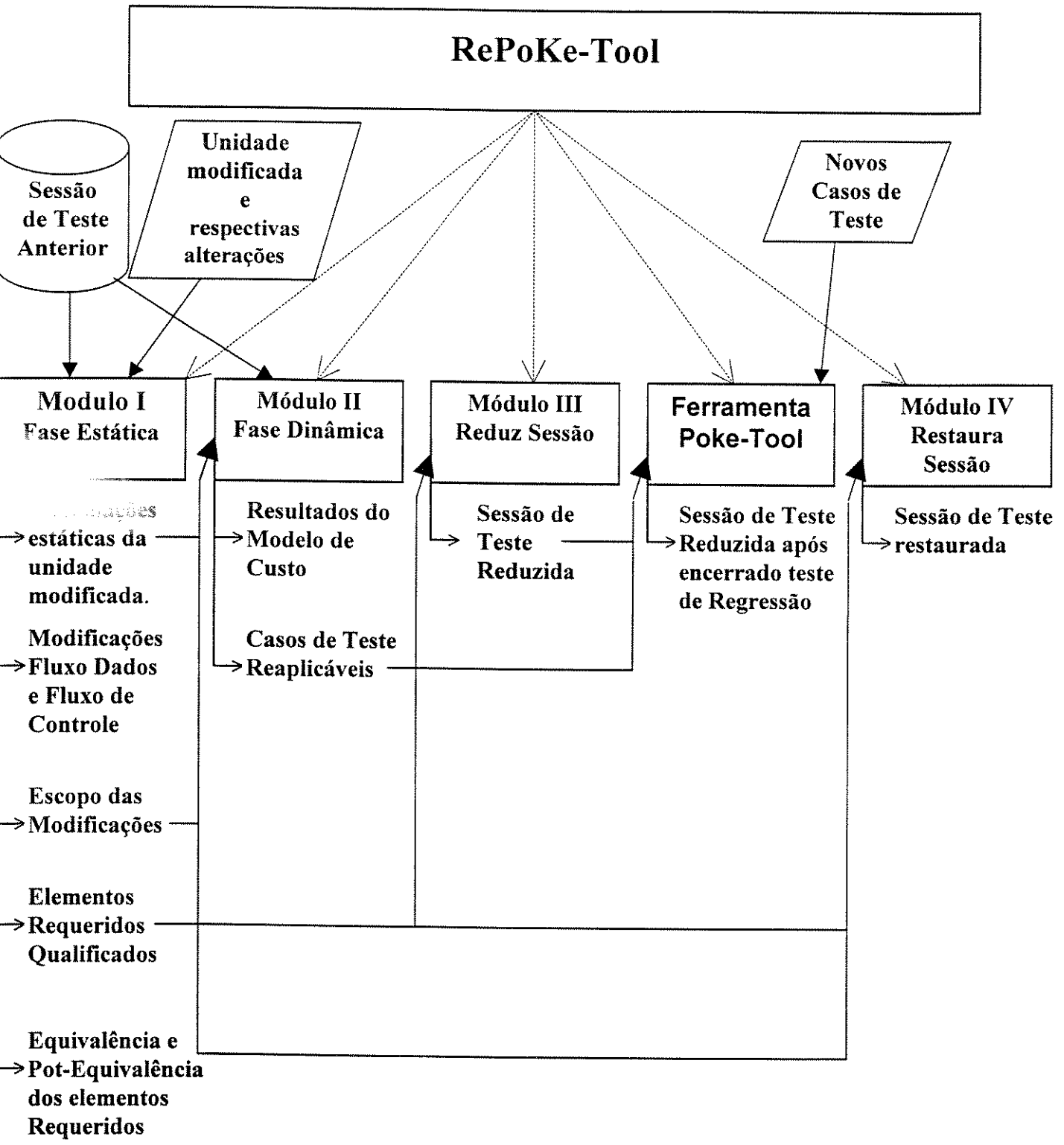


Figura 4.15 - Projeto da Ferramenta

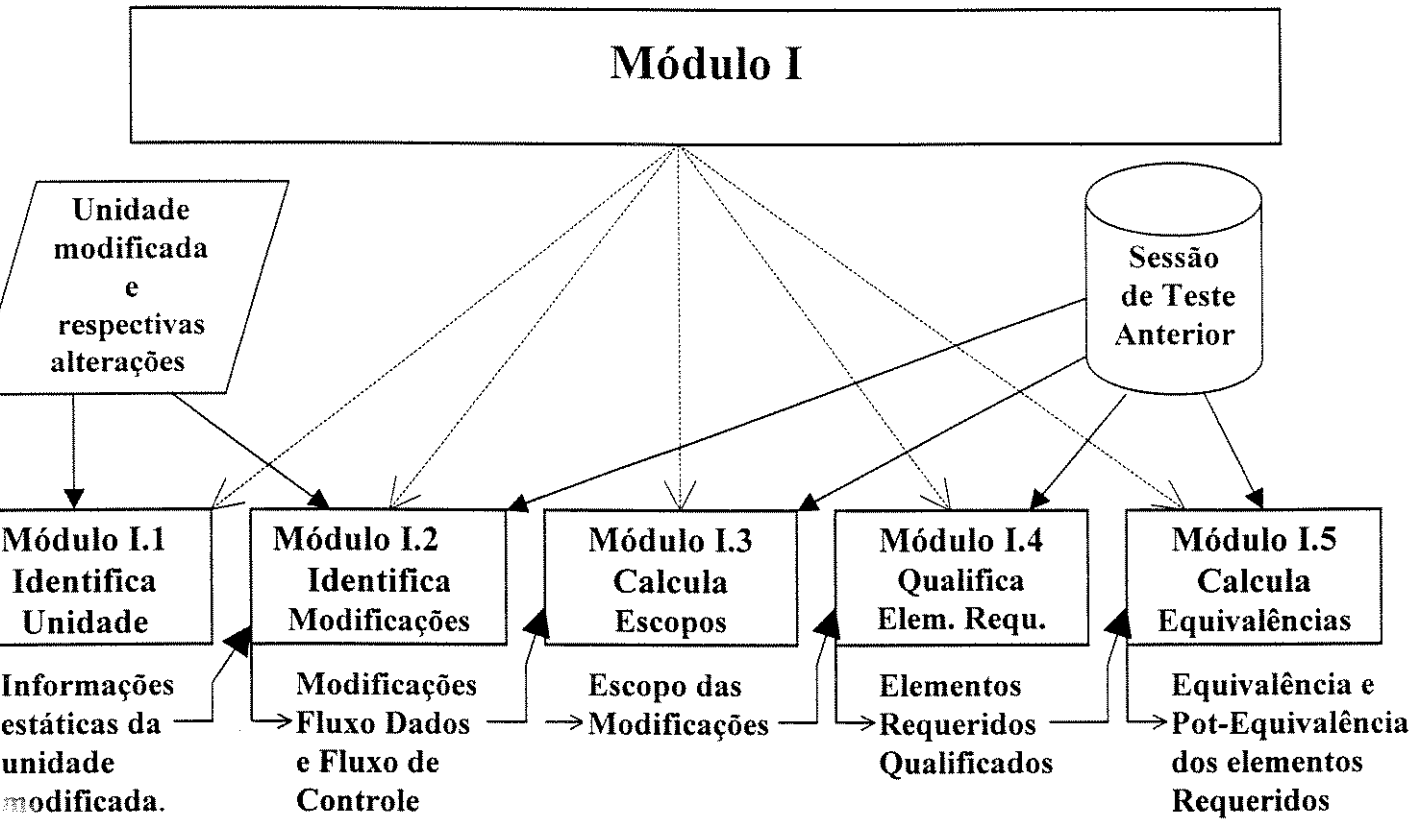


Figura 4.16 – Sub-Módulos do Módulo I da Ferramenta

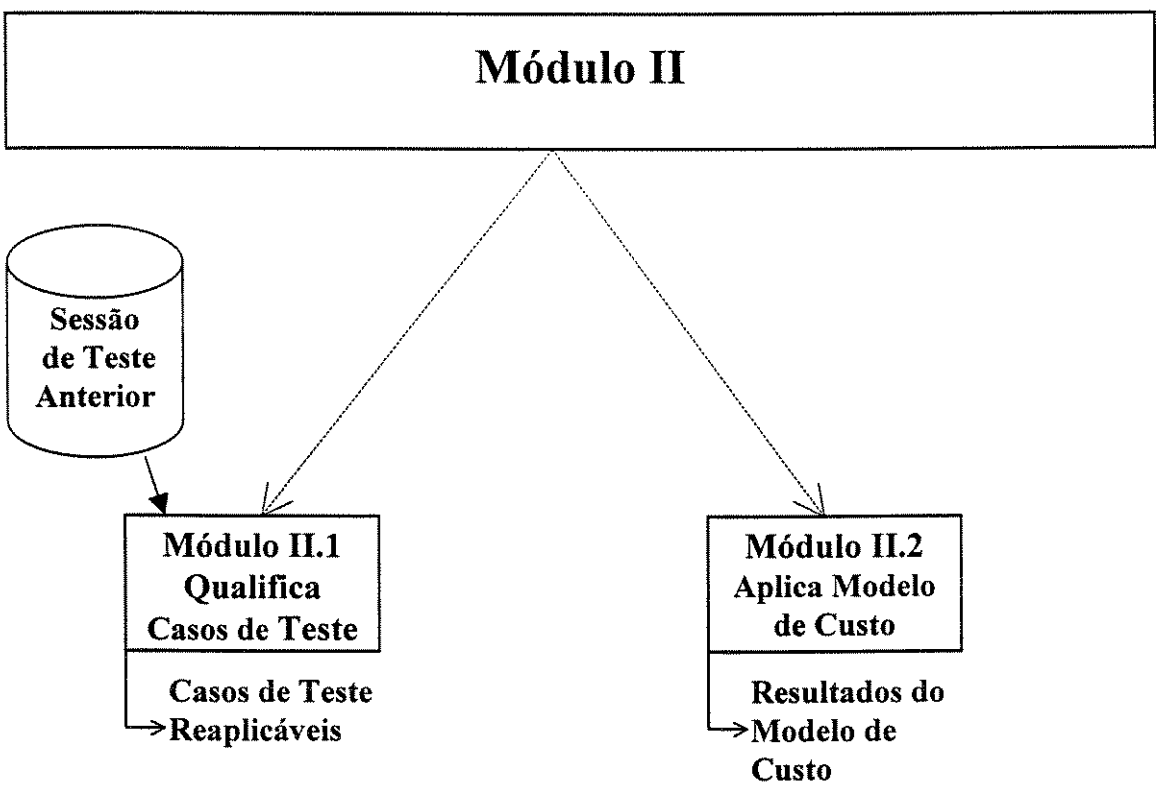


Figura 4.17 – Sub-Módulos do Módulo II da Ferramenta

As informações geradas pelo módulo I.2 estarão armazenadas através de listas ligadas, pois serão utilizadas em outros módulos da ferramenta, e também em arquivos, cujos nomes são `noseqfc.tes`, `noseqdef.tes`, `noseqpre.tes`, `nosequso.tes` e `noseqnao.tes`.

Na terceira etapa (Módulo I.3), o escopo da modificação é calculado através das regras de geração de escopo de cada um dos critérios implementados pela POKE-TOOL. É importante salientar que os escopos serão calculados para cada grafo(i) do programa modificado, pois se for utilizado o escopo genérico para o grafo do programa, alguns caminhos e associações poderão ser selecionados inadequadamente para o teste de regressão.

Para exemplificar uma situação dessas, suponha que há definição da variável x nos Nós 1 e 3, e que esses estejam ligados por um Nó 2, que após a manutenção passou a definir valor para uma variável y. Assim, o Potencial-DU-caminho 1,2,3 do Grafo(1) não deve ser selecionado pois, apesar de atravessar o Nó 2, que sofreu modificação no fluxo de dados, essa mudança não influencia nenhuma variável definida no grafo (1), visto que defg(1) = {x}. Assim, para o Grafo(1), o Nó 2 não pertence ao escopo do critério todos-potenciais-DU-caminhos. A definição do escopo genérico para o programa modificado exigiria a reavaliação desnecessária desse caminho, ou seja, se um programa sofreu apenas modificações no fluxo de dados, as associações/caminhos definidas em um grafo(i) qualquer não devem ser selecionadas para o teste de regressão se atravessarem apenas nós preservados, ou aqueles cujas modificações não influenciam as variáveis definidas no nó i (conjunto defg(i)). Os escopos dos grafo(i), para cada critério, serão armazenados em listas ligadas.

Terminada a terceira etapa, cada elemento requerido do programa modificado será avaliado de acordo com o escopo da modificação definido para o respectivo critério. Esses elementos requeridos serão então classificados, por essa quarta etapa (Módulo I.4), em duas categorias: Não Reaplicáveis (que não sofreram modificação) e Reaplicáveis (Modificados ou Novos).

Finalmente, a última etapa (Módulo I.5) encontrará a equivalência entre cada elemento requerido preservado do programa modificado com seu original, bem como a potencial-equivalência entre os elementos requeridos modificados e novos com os do programa original. Para armazenar essas informações, serão gerados dois arquivos para cada um dos critérios, contendo os elementos requeridos Reaplicáveis e Não Reaplicáveis, bem como as respectivas equivalências ou potencial-equivalências. A Tabela 4.1 mostra os arquivos gerados para cada um dos critérios.

Tabela 4.1 – Arquivos contendo os Elementos Requeridos Reaplicáveis e Não Reaplicáveis gerados para cada Critério

Critério	Nome dos Arquivos	
	Reaplicávies	**Não Reaplicáveis**
Todos-Nós (NOS)	`nosret.tes`	`nosnret.tes`
Todos-Arcos (ARCS)	`arcprimr.tes`	`arcprimn.tes`
Todos-Potenciais-Usos (PU) e Todos-Potenciais-Usos/DU (PUDU)	`puassocr.tes`	`puassocn.tes`
Todos-Potenciais-DU-Caminhos (PDU)	`pdupathr.tes`	`pdupathn.tes`

4.4.2 Módulo II – Fase Dinâmica

O objetivo desse módulo é selecionar os casos de teste (gerados durante a fase dinâmica da POKE-TOOL) que, potencialmente, exercitarão os elementos requeridos reaplicáveis selecionados no Módulo I, para um dado critério de teste. Assim, cada caso de teste aplicado originalmente será avaliado segundo a regras definidas na Seção 3.4. Essa primeira etapa (II.1) terminará com os casos de teste classificados em três categorias: Reutilizáveis (que não atravessam partes modificadas do programa); Reaplicáveis (que atravessam partes modificadas) e Obsoletos (que atravessam apenas partes removidas do programa modificado). A tabela 4.2 mostra os nomes arquivos gerados pela ferramenta onde serão armazenados os casos de teste reaplicáveis, reutilizáveis e obsoletos para cada critério.

Tabela 4.2 – Arquivos contendo a Qualificação dos Casos de Teste para cada Critério.

Critério	Nome dos Arquivos		
	Reutilizáveis	**Reaplicáveis**	**Obsoletos**
Todos-Nós (NOS)	`nosctreu.tes`	`nosctrea.tes`	`nosctrem.tes`
Todos-Arcos (ARCS)	`arcctreu.tes`	`arcctrea.tes`	`arcctrem.tes`
Todos-Potenciais-Usos (PU)	`puctreu.tes`	`puctrea.tes`	`puctrem.tes`
Todos-Potenciais-Usos/DU (PUDU)	`puductreu.tes`	`puductrea.tes`	`puductrem.tes`
Todos-Potenciais-DU-Caminhos (PDU)	`pductreu.tes`	`pductrea.tes`	`Pductrem.tes`

Obtida essa classificação dos casos de teste, é possível executar uma segunda etapa do Módulo II (módulo II.2), que calcula e apresenta ao usuário o resultado de um (ou mais) modelo(s) de custo de teste de regressão, conforme discutido na Seção 3.6. Encerrado o Módulo II, dar-se-á início ao teste de regressão propriamente dito.

4.4.3 Módulo III – Reduz Sessão de Teste

Realizada a seleção dos elementos requeridos a serem reavaliados (Conjunto E'') e dos casos de teste que deverão ser reaplicados (Conjunto T'), é necessário configurar os arquivos da POKE-TOOL para que essa possa ser executada em um ambiente reduzido. Esse módulo de geração da configuração reduzida da POKE-TOOL altera alguns arquivos gerados pela análise estática do módulo modificado.

Cada elemento requerido pelos critérios de teste implementados pela POKE-TOOL está armazenado no formato de um descritor, em arquivos gerados durante a fase estática. Esses arquivos são denominados `des_nos`, `des_arc`, `des_pu`, `des_pudu`, `des_pdu`, e contêm os descritores dos critérios todos-nós, todos-arcos, todos-potenciais-usos, todos-potenciais-usos/DU, todos-potenciais-DU-Caminhos, respectivamente. De cada um desses arquivos, serão removidos os elementos requeridos que não precisam ser reavaliados no teste de regressão e que foram eliminados durante a última atividade de teste realizada através da POKE-TOOL. Isso é fundamental, pois serão eliminados da base de teste apenas os

elementos requeridos que não sofreram modificação. É bastante comum que elementos requeridos preservados após a manutenção não tenham sido eliminados no último teste que foi conduzido, pois podem ser não executáveis ou mesmo por decisão do testador, quando a cobertura desejada foi atingida. Esses elementos devem ser novamente requeridos no teste de regressão, pois poderão ser executáveis após a manutenção e então poderão ser eliminados. A mesma observação é valida para os elementos requeridos que não foram eliminados devido à cobertura atingida anteriormente.

A única ação do módulo III é, portanto, recuperar as informações produzidas pelo Módulo I (classificação e equivalência dos elementos requeridos) e remover dos arquivos citados anteriormente os elementos requeridos que não precisam ser reavaliados. Antes disso, é necessário mudar o nome dos arquivos gerados originalmente pela fase estática, pois no módulo IV serão recompostas as informações do último teste realizado.

4.4.4 Execução do Teste de Regressão

Conforme já discutido nesse trabalho, uma ferramenta de teste de regressão deve trabalhar em conjunto com uma ferramenta de teste, preparando o ambiente para que esta possa executar o teste sob um ambiente reduzido.

A primeira ação do testador, assim que for encerrada a execução do script `repoketool` (módulos 1,2 e 3), deverá ser a compilação da nova versão do programa `testeprog.c`, que contém a nova unidade instrumentada na fase estática da POKE-TOOL. Encerrado esse passo inicial, deverá ser conduzida então, a execução do teste de regressão (através da POKE-TOOL). Esse procedimento inicia-se com a execução em batch dos casos de teste selecionados para a regressão, dado um determinado critério suportado pela POKE-TOOL. Essa execução deve ser conduzida através de um "script" denominado `repoketool-exec`, que executa subconjuntos de casos de teste informados como parâmetros. Através de outro "script", já implementado pela POKE-TOOL, denominado `pokeaval`[6], a cobertura obtida por esse mesmo conjunto de casos de teste é calculada, desde que esse conjunto de casos de teste não contínuos seja informado através de parâmetros, dado um dos critérios da família Potenciais Usos.

Após encerrada a execução e avaliação dos testes em batch, caberá ao testador averiguar a cobertura atingida contra o estipulado para a regressão bem como se todos os requisitos funcionais (novos ou modificados) já foram verificados. Caso seja necessário preparar mais casos de teste (conjunto T''), os mesmos deverão ser conduzidos na POKE-TOOL através dos "scripts" de execução e de avaliação de casos de teste já implementados `pokeexec e pokeaval`, respectivamente. Se a manutenção conduzida foi adaptativa ou perfectiva, maior será a probabilidade de que o conjunto T'' seja necessário para atingir um nível de cobertura razoável, ou mesmo para testar todas as funcionalidades do módulo modificado. O Capítulo 5 mostrará

[6] Para que a RePoKe-Tool fosse executada corretamente, foi necessária uma pequena modificação nesse "script" (vide Seção 4.6).

alguns estudos de casos que apontam relações entre o teste funcional e estrutural sob a ótica do teste de regressão.

Tabela 4.3 – Nomes e Informações armazenadas nos arquivos atualizados pelo módulo IV da RePoKe-Tool.

Nome Arquivo	O que armazena	O que foi modificado
xxxx`output.tes`[7]	Exibe Elementos Requeridos não exercitados e a cobertura obtida para o referido critério.	Elementos Requeridos não exercitados e a cobertura obtida.
`exec_`***`xxxx`***`.tes`	Exibe Elementos Requeridos exercitados e a cobertura obtida para o referido critério.	Elementos Requeridos exercitados e a cobertura obtida.
xxxx`his.tes`	Número dos Elementos Requeridos exercitados para o referido critério	Atualizar o número dos elementos requeridos que foram exercitados
xxxx`his`***kk***`.tes`[8]	Número dos Elementos Requeridos exercitados pelo referido caso de teste para o referido critério	Atualizar o número dos elementos requeridos que foram exercitados
xxxx`dif`***mm-nn***`.tes`[9]	Número de todos os elementos requeridos eliminados pelo m-ésimo caso de teste em relação ao n-ésimo, dado o referido critério. (cobertura incremental de mm em relação à nn)	Atualizar o número dos elementos requeridos que foram exercitados
`path`***kk***`.tes`	Descrição do caminho exercitado pelo referido caso de teste	Atualização dos caminhos, segundo a Tabela de Equivalência dos Nós

4.4.5 Módulo IV – Restaura Sessão de Teste

A condução do teste de regressão nada mais é do que executar novamente a fase dinâmica da POKE-TOOL, sob um ambiente menor e controlado, uma vez que a fase estática da ferramenta de teste já foi executada pelo Módulo I da RePoKe-Tool (que invoca a execução do "script" `poketool`). Encerrada essa fase, a execução do script `repoketool-end` dará início à etapa de finalização da RePoKe-Tool, que recomporá o ambiente de teste adicionando as informações obtidas pelo teste de regressão com as que foram reutilizadas da versão anterior. Esse ambiente armazenará a configuração final do teste de regressão, deixando-o pronto para um nova necessidade de manutenção.

Cabe a esse módulo, portanto, automatizar o trabalho da Gerência de Configuração de Software, uma vez que o teste é um dos itens de configuração de um software [Pre92]. Encerrada a execução desse módulo, serão atualizados os vários arquivos da POKE-TOOL, cujos nomes e descrições estão contidos na Tabela 4.3. O

[7] ***xxxx*** representa um dos critérios da família Potenciais-Usos: **`nos`** (todos-nós), **`arcs`** (todos-arcos), **`pu`** (todos-potenciais-usos), **`pudu`** (todos-potenciais-usos/DU) ou **`pdu`** (todos-potenciais-DU-Caminhos).

[8] ***kk*** indica o número do caso de teste.

[9] ***mm-nn*** indica que a cobertura incremental é do caso de teste ***mm*** em relação ao caso de teste ***nn***.

diretório que contém a configuração do teste da versão anterior do módulo modificado terá seu nome modificado para `nome_função`***XX***, onde ***XX*** indicará o número da última versão. O diretório que contém a configuração da versão do módulo que sofreu a manutenção, e que foi atualizada pelo módulo IV, permanecerá com o nome dessa função, de acordo com o padrão estabelecido pela POKE-TOOL.

4.5 Modificações na POKE-TOOL Requeridas pela Ferramenta

Para que a RePoKe-Tool fosse desenvolvida, pequenas modificações na POKE-TOOL foram requeridas. Essas modificações são restritas a um "script" da POKE-TOOL (`pokeaval`) e não foram necessárias modificações nos programas escritos em linguagem C. Também foi necessária a criação de dois novos "scripts", (denominado `pokesave` e `repoke-exec`), que armazena as informações geradas pela POKE-TOOL, uma vez que o início de um novo teste (mesmo de outro módulo), pode eliminar informações importantes para o teste de regressão, dentre elas as entradas dos casos de teste (diretório `keyboard`), entradas das linhas de comando (diretório `input`) e as saídas geradas (diretório `output`). Essa seção apresentará mais detalhes sobre essas modificações que foram necessárias.

4.5.1 "Script" de Execução de Casos de Teste de Regressão (`repoketool-exec`)

O referido "script" foi necessário para atualizar o caminho percorrido pelos casos de teste selecionados para o teste de regressão (reaplicáveis), pois, certamente, percorrerão as partes modificadas do programa. Os casos de teste reutilizáveis não precisaram ser novamente executados, pois percorrerão o mesmo caminho do programa original. Caberá ao módulo IV, que reconfigura o ambiente original da POKE-TOOL, atualizar os caminhos percorridos por esses casos de teste, uma vez que existe um arquivo que contém a equivalência entre os nós dos programas original e modificado.

Para executar esse "script", é fundamental que antes o usuário compile a unidade `testeprog.c`, gerando o arquivo executável instrumentado correspondente à versão modificada do programa. Esse script exige 4 parâmetros que são: nome do programa executável; nome da função testada; padrão de entrada de dados dos casos de teste originais e o número dos casos de teste que deverão ser novamente executados. Como saída, esse "script" atualizará o arquivos de saída (`outputxx.tes`) e o caminho percorrido por cada caso de teste (`pathxx.tes`).

4.5.2 "Script" de Avaliação (`pokeaval`)

O "script" `pokeaval`, executa a avaliação da cobertura obtida dos elementos requeridos de algum critério de teste da família Potenciais-Usos. Essa avaliação pode ser feita usando-se como entrada todos os casos de teste ou de um subconjunto desses, que foram introduzidos na POKE-TOOL através de outro "script" (`pokeexec`). A

avaliação implementada (programa escrito em linguagem C) consiste da verificação dos elementos requeridos que foram eliminados pelo caminho percorrido por cada caso de teste. As saídas geradas por esse "script" são as seguintes:

a) **Relatórios (gravados em arquivos texto) :**

- Porcentagem de Cobertura Obtida, armazenado nos arquivos ***xxxx***`output.tes` e `exec_`***xxxx***`.tes` (e.g `nosoutput.tes`, `exec_pudu`).
- Elementos Requeridos Executados, armazenado no arquivo `exec_`***xxxx***`.tes` (e.g `exec_pdu.tes`).
- Elementos Requeridos não Executados, armazenado no arquivo ***xxxx***`output.tes` (e.g `puoutput.tes`).

b) **Arquivos de Elementos Requeridos Cobertos:** São gerados os arquivos que armazenam o histórico do teste. Esse arquivo possui formato genérico ***xxxx***`his.tes` (e.g `puhis.tes`, `pduhis.tes`), onde ficam armazenados o número de todos os elementos requeridos eliminados para o referido critério.

c) **Arquivo de Cobertura Incremental**: São gerados os arquivos que armazenam o histórico incremental do teste. Esse arquivo possui formato genérico ***xxxx***`dif`***mm-nn***`.tes` (e.g. `pudif1-0`, `nosdif4-2`), onde estão armazenados o número de todos os elementos requeridos eliminados pelo m-ésimo caso de teste em relação ao n-ésimo (no primeiro caso de teste, o anterior é numerado como zero). Assim, o número que identifica cada elemento requerido só aparecerá no arquivo cujo ***mm*** é o número do primeiro caso de teste que o exercitou e, portanto, não será eliminado por nenhum outro caso de teste .

De acordo com as discussões levantadas na Seção 3.4, a RePoKe-Tool necessita identificar as funcionalidades preservadas, modificadas e removidas. Para que essa avaliação possa ser conduzida, é necessário que todos os elementos requeridos eliminados por cada caso de teste estejam armazenados, informação que não é provida pelo arquivos ***xxxx***`his.tes` nem por ***xxxx***`dif`***mm-nn***`.tes`. Assim, fez-se necessário adicionar essa informação à POKE-TOOL, modificando-se o "script" `pokeaval`. Um novo padrão de arquivo ***xxxx***`his`***kk***`.tes` passou a ser gerado por este "script". Além dessa modificação, o script `pokeaval` permitirá também a execução de subconjuntos de casos de teste, ou seja, realizará a execução dos casos de teste "em batch" durante o teste de regressão.

4.5.3 "Script" para salvar a Sessão do Teste (`pokesave`)

A atual versão da POKE-TOOL não foi projetada para suportar o teste de regressão. Assim, os diretórios `output`, `input` e `keyboard` são gerados como sub-diretórios do diretório onde foi invocada a execução do "script" `poketool` (fase estática). Se uma nova execução da POKE-TOOL for realizada nesse mesmo diretório (inclusive de outra função completamente diferente), as informações armazenadas nos

referidos sub-diretórios serão removidas assim que iniciar-se a execução de casos de teste dessa nova sessão da POKE-TOOL.

Com isso, fez-se necessário o desenvolvimento do "script" `pokesave`, que armazena esses diretórios dentro do diretório da função que acaba de ser testada. Outras informações também são armazenadas nesse diretório, dentre elas uma cópia da versão do arquivo onde o módulo recém testado se encontra, bem como o módulo instrumentado.

A execução do módulo `pokesave` deve ser de iniciativa do testador apenas no primeiro teste conduzido para uma determinada função (módulo). A partir da segunda versão, o script `repoketool-end` (que invoca a execução do módulo IV escrito em C), se encarrega da execução de `pokesave`, uma vez que o módulo IV só deve ser executado quando o teste de regressão for encerrado.

Capítulo 5

Uma Metodologia para o Teste de Regressão: Aplicar Estratégia Funcional na Seleção dos Casos de Teste

Este capítulo apresentará alguns resultados obtidos através de estudos de casos que foram conduzidos utilizando-se a ferramenta RePoKe-Tool. Esse estudo concentra-se na seleção dos teste de regressão, através de identificação das funcionalidades do software que foram afetadas pela modificação.

5.1 Aplicação de Casos de Teste Funcionais: Um Bom Começo para o Teste Estrutural

Dentre as abordagens utilizadas para o teste de programas, duas são as mais tradicionais: funcional (ou caixa preta) e estrutural (ou caixa branca) [Mey79, Pre92]. Existe ainda uma terceira abordagem, cuja utilização é mais recente: baseada em erros [DMLS78]. No contexto do teste de unidade, a primeira abordagem tem como características a revelação de erros nas interfaces e na implementação das funcionalidades do módulo; a segunda revela erros de detalhes internos da unidade sob teste (e.g., incialização de variáveis, erros em estruturas de dados); a terceira revela, através do uso de operadores de mutação que geram programas mutantes, erros gerados por desvios sintáticos introduzidos nos programas. Meyers e Pressman [Mey79, Pre92] já enfatizara as características das duas primeiras abordagens, argumentando que cada estratégia revela erros de classes diferentes e, devido a isso, devem ser utilizadas como abordagens complementares. Em momento algum, porém, sugeriu-se qual ordem deveria ser adotada na aplicação das estratégias.

Analisando-se as características das duas abordagens tradicionais, é bastante intuitivo sugerir que o teste funcional seja realizado ***a priori***. Para comprovar essa argumentação, basta analisar e comparar as características dos elementos requeridos em cada estratégia. No teste estrutural (mesmo no baseado em fluxo de dados) esses elementos são caminhos e/ou associações que são extraídos da representação do programa através de um Grafo de Fluxo de Controle, enquanto que no teste funcional esses elementos descrevem as funcionalidades (e.g., partições do domínio de entrada, subdomínios, particionamentos de equivalência) que são extraídas da especificação do programa.

Uma análise superficial indica que existe uma relação entre elementos requeridos para critérios caixa preta e os elementos requeridos por critérios caixa branca pois, uma funcionalidade testada também elimina um conjunto de Caminhos/Associações do programa, ou seja, uma determinada funcionalidade de um módulo é exercitada por um conjunto de caminhos executados em uma determinada

ordem. Porém, a recíproca nem sempre é verdadeira, pois um único caminho requerido por um critério estrutural não necessariamente exercita alguma funcionalidade especificada para um programa. Com base nessas observações, fica claro que a abordagem funcional deve ter prioridade de aplicação sobre a estrutural, uma vez que o teste de um elemento requerido funcional tende a cobrir muitos elementos requeridos estruturais. Aplicando-se um único caso de teste para cada elemento requerido funcional, obtêm-se, geralmente, uma cobertura razoável do conjunto de elementos requeridos para o teste estrutural. Em seguida, o teste estrutural tradicional poderá ser aplicado, para que seja obtido um nível de cobertura estabelecido pelo testador. Deve-se ressaltar que esse nível de cobertura atingida inicialmente varia muito de acordo com a complexidade do programa em teste com a técnica funcional utilizada para descrição do domínio do programa (que serve de base para a derivação dos casos de teste funcionais). Por exemplo, quanto mais classes de equivalência forem definidas, mais casos de teste funcionais serão gerados e, conseqüentemente, maior deverá ser a cobertura inicial atingida.

No entanto, deve-se lembrar que essa abordagem "híbrida" pode ser um pouco arriscada, pois uma determinada seqüência de caminhos que exercita uma funcionalidade pode não expor um erro que seria revelado se a abordagem utilizada fosse estrutural. Um exemplo importante é o caso da saída coincidente (correção coincidente), que pode ser gerada por dois erros introduzidos seguidamente em um caminho que exercita uma determinada função. Nesse caso, a função gera um valor de saída correto, porém um erro mascarou o outro (um erro "consertou" o outro, gerando uma saída que coincide com a esperada). Como os caminhos requeridos pelo critério estrutural já foram exercitados pelo teste funcional (não farão parte dos elementos requeridos estruturais que ainda deverão ser cobertos), esse erro pode não ser descoberto pelo teste estrutural que será conduzido logo em seguida. Apesar de existirem riscos como esse, alguns autores eliminam a possibilidade da correção coincidente em suas técnicas e metodologias, uma vez que sua ocorrência é muito rara [DO91]. Recentemente, Vergílio [Ver97] realizou estudos baseados em diversas técnicas de teste (funcional, estrutural e baseadas em erros), que serviram de subsídio para a introdução de uma família de critérios de teste estruturais restritos, onde são combinadas características do teste estrutural com técnicas de geração de dados de teste sensíveis a erros. A realização desse estudo comprova a tendência de utilizar, de forma complementar, as melhores características de diferentes técnicas e metodologias de teste.

5.2 Seleção dos Casos de Teste que poderão exercitar Funcionalidades Modificadas ou Novas: Um bom começo para o Teste de Regressão Estrutural

Seguindo a mesma linha de raciocínio da última seção, a aplicação de casos de teste funcionais pode ser um bom começo para o teste de regressão. Porém, nesse caso, é importante identificar, inicialmente, as funcionalidades que foram modificadas e/ou inseridas e então identificar dentre os casos de teste utilizados originalmente, todos aqueles que poderão exercitá-las. Nesse contexto de teste de regressão seguro, quanto

mais casos de teste capazes exercitar uma funcionalidade forem aplicados, maior será a probabilidade de que um erro de regressão seja revelado.

Assim como no teste original, a execução de um conjunto de casos de teste funcionais reaplicáveis tende a garantir uma boa cobertura inicial dos elementos requeridos para o teste estrutural. Além disso, quando o teste de regressão for corretivo (motivado por uma manutenção corretiva), maior será a probabilidade de que seja obtida a mesma cobertura do teste original, desconsiderando-se os elementos requeridos estruturais não executáveis.

Conforme já discutido em detalhes no Capítulo 3, a classificação dos casos de teste depende da correta classificação dos elementos requeridos da versão modificada de um programa, e também do cálculo correto da sua equivalência (ou potencial-equivalência) com os elementos requeridos pelo programa original. Considerando que os casos de testes foram derivados usando-se, originalmente, alguma técnica funcional, esses poderão ser classificados em três categorias principais no teste de regressão: Reutilizável (aquele cujo caminho completo original exercitou apenas elementos requeridos estruturais preservados no programa modificado); Reaplicável (aquele cujo caminho completo original exercitou pelo menos um elemento requerido estrutural modificado no programa alterado) e Removido ou Obsoleto (aquele cujo caminho completo original exercitou apenas elementos requeridos estruturais que foram removidos do programa modificado). Os casos de teste do primeiro tipo não precisam ser novamente aplicados, pois computarão o mesmo valor de saída. Já os do segundo tipo, serão sugeridos para o teste de regressão, pois poderão exercitar funcionalidades modificadas ou mesmo novas. Aqueles que forem classificados como obsoletos (último tipo) deverão ser retirados da base de casos de teste, pois exercitam uma funcionalidade que não existe mais na atual representação do programa. Na próxima sessão, serão mostrados estudos de casos aplicados à ferramenta RePoKe-Tool com o objetivo de mostrar a aplicabilidade desse método.

5.3 Estudos de Casos – Condução e Organização

A condução dos estudos de casos foi realizada com três objetivos: (1) Para dar subsídio prático às discussões dos capítulos anteriores; (2) para validar a RePoKe-Tool e, principalmente; (3) para comprovar a aplicabilidade do método de seleção de casos de teste de regressão, proposto na seção anterior. Adicionalmente, através da análise desses estudos de casos, será apresentado um pequeno guia (*guideline*) para orientar os programadores a fazer manutenção em programas visando facilitar a condução e diminuir o custo do teste de regressão.

Serão apresentados quatro estudos de casos. Para cada um deles foi executada a seguinte seqüência de passos:

1. Definição dos elementos requeridos funcionais (e.g., partições, classes de equivalência), segundo algum critério funcional.
2. Geração de um caso de teste para cada elemento requerido funcional.
3. Aplicação dos casos de teste à ferramenta POKE-TOOL.

4. Se necessário, cobrir o restante dos elementos requeridos estruturais (gerados pela POKE-TOOL) até que seja atingida a cobertura máxima, eliminando-se os elementos requeridos não executáveis (critério de parada dos estudos de casos).
5. Implementar uma manutenção no módulo testado (corretiva, adaptativa, perfectiva ou motivada por reuso de componentes).
6. Executar a RePoKe-Tool, selecionando os elementos requeridos que devem ser novamente avaliados (ambiente POKE-TOOL reduzido) e os casos de teste que poderão cobri-los (conjunto T').
7. Aplicar os casos de teste selecionados (T') na POKE-TOOL.
8. Se necessário, derivar novos casos de testes para cobrir as novas funcionalidades do módulo.
9. Se necessário, cobrir também os elementos requeridos estruturais ainda não eliminados (critério de parada do teste de regressão).

No primeiro estudo de caso (Seção 3.5.1) serão mostradas várias tabelas com resultados intermediários, até que seja atingido o resultado final. Nos demais estudos serão apresentados apenas os resultados finais obtidos. Em todos os casos, serão listadas a versão original da função, sua especificação e seu respectivo GFC, sua versão modificada, seu GFC e sua nova especificação (se houver). As análises serão realizadas utilizando-se dois dos critérios estruturais implementados pela POKE-TOOL (PU – Todos Potenciais Usos e PDU - Todos Potenciais-DU-Caminhos).

5.3.1 Estudo de Caso 1 - Função `calc_seg`

O módulo escolhido para representar esse caso, denominado `calc_seg` calcula o valor do seguro de um automóvel, utilizando como entradas o valor do automóvel, o número de anos de prêmio do seguro e o sexo do segurado. Esse módulo foi escolhido porque sua especificação funcional é simples e os casos de teste funcionais podem ser facilmente derivados através da técnica de particionamento do domínio de entrada em classes de equivalência [Pre92]. A Figura 5.1 mostra o módulo `calc_seg`, e seu grafo de fluxo de controle. Logo a seguir está descrita a especificação do módulo e as tabelas 5.1, 5.2 e 5.3 detalham, respectivamente, as partições do domínio de entrada em classes de equivalência, o detalhamento dessas classes segundo a especificação e os elementos requeridos (combinação das classes).

Especificação do módulo `calc_seg`:

1. Objetivo: Módulo que calcula o Valor do seguro de um automóvel
2. Entradas:
   a) Valor do Automóvel
   b) Anos de Seguro não Usados (Prêmio)
   c) Sexo (1 - Mulher ou 0 - Homem)
3. Processamento:
   - Cálculo Base: Valor Seguro = 10 % Valor carro - Descontos

- Descontos:
    - Iniciando-se com 1% de desconto no primeiro ano, a cada novo ano de seguro não usado esse incremento aumenta 1%, e soma-se esse incremento com o valor acumulado até ano anterior (Ex: 3 anos = 1% + 2% + 3% = 6% desconto total)
    - Mais 2% de Seguro se o cliente for do sexo Feminino

4. Restrição: Automóveis c/ valor inferior a R$ 3.000,00 não Interessam à seguradora
5. Saídas Esperadas (Valor de Retorno):
    a) Valor de Seguro, segundo processamento especificado
    b) -1, se o carro Não Interessa à Seguradora
    c) -2, -3, -4, se forem detectados erros nas Classes de Entrada de entrada Valor Automóvel, Prêmio e Sexo, respectivamente.

Tabela 5.1 - Particionamento de Equivalência das entradas em Classes de Equivalência Base

Variáveis de Entrada	Classes Base Válidas	Classes Base Inválidas
Valor	>= 0	< 0
Mulher	[0,1]	< 0 ou > 1
Premio	>= 0	< 0

Tabela 5.2 - Detalhamento das Classes Base

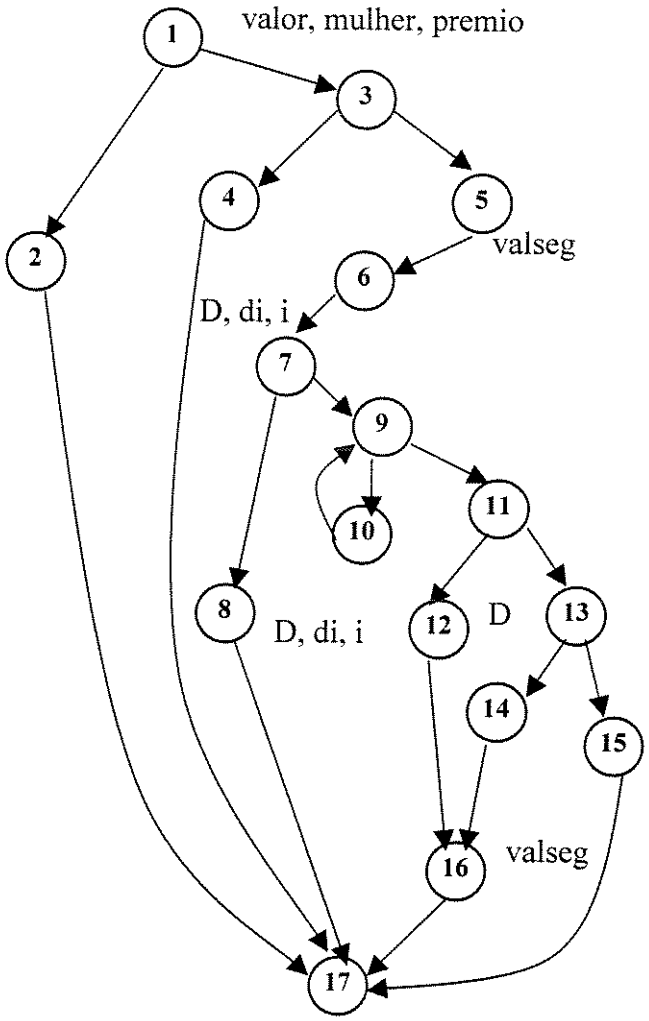
Variáveis de Entrada	Classes Válidas		Classes Inválidas	
	I	II	I	II
Valor	[0 .. 3000)	>= 3000	< 0	-
Mulher	1	0	< 0	> 1
Premio	>= 0	-	< 0	-

Tabela 5.3 - Elementos Requeridos para o Teste Funcional (Combinação de Classes Detalhadas)

Elem. Requ.	**Valor**	**Mulher**	**Premio**
1	Inválida I	-	-
2	Válida I	-	-
3	Válida II	-	Inválida I
4	Válida II	Válida I	Válida I
5	Válida II	Válida II	Válida I
6	Válida II	Inválida I	Válida I

```
#include <stdio.h>
float calc_seg(float valor,int mulher,int premio)
{  float valseg,D,di,i;
1  if (valor <= 0)
2    return -2;
3    else if (valor < 3000)
4          return -1;
5,6 else valseg = valor * 0.10;
7   D = 0;
7   di = 0.01;
7   i = 0;
7,8 if (premio < 0) return -3;
9   while (i < premio)
10        {
10         D = D + (valseg*di);
10         di = di + 0.01;
10         i++;
10        }

11 if (mulher==1)
12    D = D + (valseg * 0.02);
13    else if (mulher !=0 )
14,15          return -4;
16 valseg = valseg - D;
17 return (valseg);
}
```



(a) Função `calc_seg` (b) GFC gerado

Figura 5.1 – Informações do arquivo `seguros.c` (versão original de `calc_seg`)

Para cada elemento requerido, foi derivado um caso de teste funcional, que resultou na cobertura descrita na Tabela 5.4, quando aplicados à POKE-TOOL.

Tabela 5.4 – Cobertura Inicial Obtida para os Critérios Estruturais, pelo Conjunto de Casos de Teste Funcionais

Critério Estrutural	Total de Elementos Requeridos	Total de Elementos Eliminados	Cobertura Obtida
PU	27	24	88,88 %
PDU	23	14	60,86 %

Para atingir a cobertura máxima para ambos os critérios (100% pois, nesse caso, não há caminhos/associações não executáveis), foram preparados mais 3 casos de teste, totalizando-se assim 9 casos de teste. Mesmo com objetivo de aumentar a cobertura estrutural (atingir a cobertura exigida pelo critério de parada), cada um desses 3 casos de teste adicionais, também exercita uma das funcionalidades básicas (elementos requeridos funcionais) do programa. Nesse exemplo, os casos de teste 7, 8 e 9 também exercitam as funcionalidades descritas pelos elementos requeridos funcionais 4, 5 e 6, respectivamente. A Tabela 5.5 mostra todos os elementos requeridos pelo critério Todos-Potenciais-Usos, e quais casos de teste exercitaram cada elemento requerido. É importante lembrar que o elemento requerido estrutural precisa ser eliminado apenas uma vez, portanto, o primeiro caso de teste a exercitá-lo é o responsável por sua eliminação no teste estrutural.

Tabela 5.5 – Elementos Requeridos pelo Critério Todos-Potenciais-Usos Eliminados pelos Casos de Teste

Elementos Requeridos pelo Critério Todos-Potenciais-Usos	**Casos de Teste**								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Associacoes requeridas pelo ***Grafo(1)***									
1) <1,(13,15),{ valor, mulher, premio }>					√		√		
2) <1,(13,14),{ valor, mulher, premio }>						√		√	
3) <1,(11,12),{ valor, mulher, premio }>				√					√
4) <1,(10,9),{ valor, mulher, premio }>				√	√	√			
5) <1,(9,10),{ valor, mulher, premio }>				√	√	√			
6) <1,(7,8),{ valor, mulher, premio }>			√						
7) <1,(3,4),{ valor, mulher, premio }>		√							
8) <1,(1,2),{ valor, mulher, premio }>	√								
Associacoes requeridas pelo ***Grafo(5)***									
9) <5,(15,16),{ valseg }>					√		√		
10) <5,(13,15),{ valseg }>					√	√	√		
11) <5,(13,14),{ valseg }>								√	
12) <5,(12,16),{ valseg }>				√					√
13) <5,(11,12),{ valseg }>				√					√
14) <5,(10,9),{ valseg }>				√	√	√			
15) <5,(9,10),{ valseg }>				√	√	√			
16) <5,(7,8),{ valseg }>			√						
Associacoes requeridas pelo ***Grafo(7)***									
17) <7,(13,15),{ D, di, i }>							√		
18) <7,(13,14),{ D, di, i }>								√	
19) <7,(11,12),{ D, di, i }>									√
20) <7,(9,10),{ D, di, i }>				√	√	√			
21) <7,(7,8),{ D, di, i }>			√						
Associacoes requeridas pelo ***Grafo(10)***									
22) <10,(13,15),{ D, di, i }>					√				
23) <10,(13,14),{ D, di, i }>						√			
24) <10,(11,12),{ D, di, i }>				√					
25) <10,(9,10),{ D, di, i }>				√	√				
Associacoes requeridas pelo ***Grafo(12)***									
26) <12,(,),{ D }>				√					√
Associacoes requeridas pelo ***Grafo(16)***									
27) <16,(,),{ valseg }>					√		√		√

Ao final do teste desse programa, foi proposta uma modificação no mesmo. Nesse primeiro caso, uma manutenção perfectiva ocasionou uma expansão no Grafo de Fluxo de Controle. A modificação consiste, resumidamente, da alteração do valor do incremento que deve ser adicionado a cada ano de seguro não utilizado. Até o terceiro ano, esse incremento permanece em 1%. A partir do quarto ano, esse incremento diminui e passa a ser 0,5%. A Figura 5.2 mostra a versão modificada da função `calc_seg` (as modificações estão marcadas em negrito) e seu respectivo grafo de fluxo de controle. Abaixo, estão descritas as modificações realizadas na especificação do módulo. Não houve modificações nas classes base (não há modificações nas entradas), porém o detalhamento dessas classes foi afetado pelas modificações na especificação. A Tabela 5.6 mostra (em negrito) essas modificações no detalhamento das classes base. A Tabela 5.7 mostra (em negrito) os novos elementos requeridos funcionais.

Especificação do módulo `calc_seg` (modificado):

1. Objetivo: Módulo que calcula o Valor do seguro de um automóvel
2. Entradas: Não há modificações
3. Processamento:
    - Calculo Base: O mesmo
    - Descontos:
        - Iniciando-se com 1% de desconto no primeiro ano, nos três primeiros anos esse incremento aumenta 1%, e a partir dos demais anos aumenta 0,5%. Soma-se esse incremento com o valor acumulado até ano anterior (Ex: 4 anos = 1% + 2% + 3% + 3,5% = 9,5% desconto total)
        - Mantém-se o desconto adicional de 2% para o sexo feminino
4. Restrição: Mantida
5. Saídas Esperadas: Não há modificações.

Tabela 5.6 – Modificações do Detalhamento das Classes Base

Variáveis de Entrada	Classes Válidas		Classes Inválidas	
	I	II	I	II
`Valor`	[0 .. 3000)	>= 3000	< 0	-
`Mulher`	1	0	< 0	> 1
`Premio`	**[0..3]**	**> 3**	**< 0**	**-**

Tabela 5.7 – Modificação dos Elementos Requeridos para o Teste Funcional (Combinação de Classes Detalhadas)

Caso de Teste	**`Valor`**	**`Mulher`**	**`Premio`**
1	Inválida I	-	-
2	Válida I	-	-
3	Válida II	-	Inválida I
4	Válida II	Válida I	Válida I
5	Válida II	Válida II	Válida I
6	Válida II	Inválida I	Válida I
7	**Válida II**	**Válida I**	**Válida II**
8	**Válida II**	**Válida II**	**Válida II**
9	**Válida II**	**Inválida I**	**Válida II**

Terminada a identificação das modificações, a ferramenta RePoKe-Tool foi executada. Para ambos os critérios foram selecionados 3 casos de teste (conjunto T') de um total de 9 usados originalmente. A Tabela 5.8 mostra as informações mais importantes geradas após a execução da RePoKe-Tool (cardinalidade dos conjuntos E', E'' e T'), e a Tabela 5.9 mostra os resultados obtidos após esse subconjunto T' ter sido aplicado na POKE-TOOL (sessão reduzida para execução do teste de regressão).

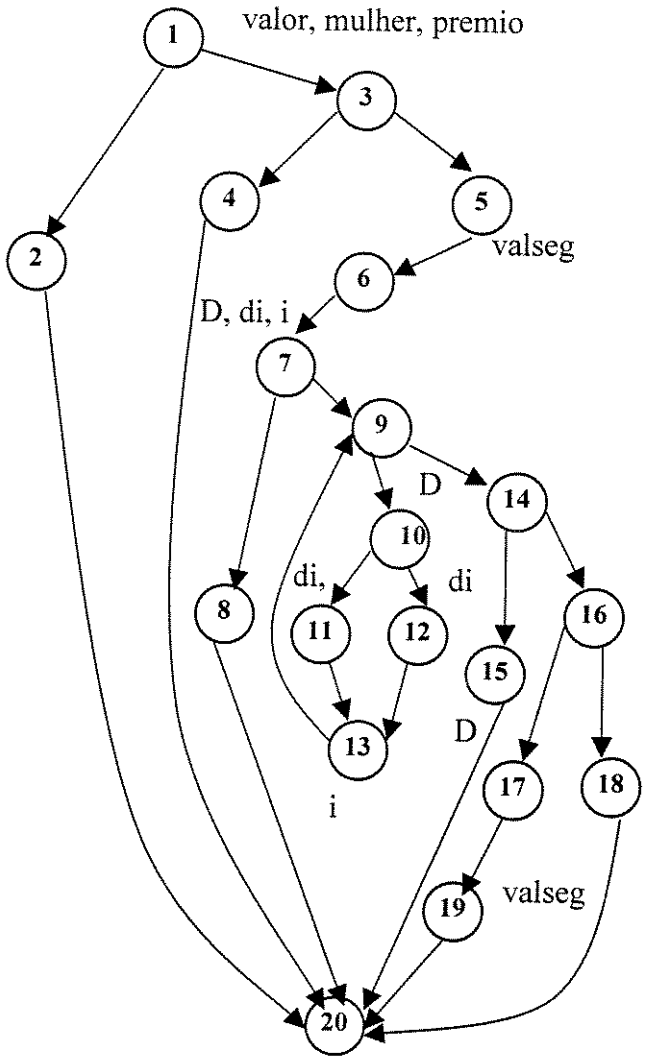
Tabela 5.8 – Informações da Sessão de Teste da versão modificada (`calc_seg`)

Critérios	Nº Total de Elementos Requeridos (E')	Nº de Elementos Selecionados para o Teste de Regressão (E'')	Nº Total de Elementos Requeridos não executáveis	Nº de Casos de Teste Selecionados p/ Regressão (T')
PU	52	30	3	3
PDU	45	26	4	3

Tabela 5.9 – Cobertura Inicial Obtida pelo Conjunto de Casos de Teste Reaplicáveis (T') para os Critérios Estruturais

Critério Estrutural	Nº de Elementos Requeridos Selecionados	Total de Elementos Eliminados	Cobertura Obtida
PU	30	17	56,66 %
PDU	26	12	46,15 %

```
#include <stdio.h>
float calc_seg(float valor,int mulher,int premio)
{  float valseg,D,di,i;
1  if (valor <= 0)
2    return -2;
3    else if (valor < 3000)
4         return -1;
5,6 else valseg = valor * 0.10;
7   D = 0;
7   di = 0.01;
7   i = 0;
7,8 if (premio < 0) return -3;
9   while (i < premio)
10       {
10        D = D + (valseg*di);
10        if (i < 3)
11             di = di + 0.01;
12        else di = di + 0.005;
13        i++;
13       }
14 if (mulher==1)
15    D = D + (valseg * 0.02);
16    else if (mulher !=0 )
17,18          return -4;
19 valseg = valseg - D;
20 return (valseg);
}
```



(a) Função `calc_seg` (b) GFC gerado

Figura 5.2–Informações do arquivo `seguros.c` (versão modificada de `calc_seg`)

Os três casos de teste selecionados exercitavam as funcionalidades representadas pelos elementos requeridos funcionais 4, 5 e 6 do programa original (vide Tabela 5.3). Esse resultado era esperado, pois apenas esses 3 casos de teste exercitam o ponto do programa que representa a única classe válida da variável *prêmio* do programa original.

Nas Tabelas 5.2 e 5.6 observa-se que a referida classe de equivalência foi dividida em duas. Assim, os casos de teste que atravessam o ponto do programa que representam esse sub-domínio deverão ser novamente testados pois, dependendo do valor de entrada para a variável *premio*, qualquer uma das duas classes geradas pela modificação poderá ser exercitada pelo teste de regressão.

Sob o ponto de vista funcional, deverão ser reavaliados os elementos requeridos funcionais 4, 5, 6, 7, 8 e 9 da Tabela 5.7 (programa modificado). Além disso pode-se afirmar que, nesse exemplo, os 3 casos de teste selecionados para o teste

de regressão exercitarão pelo menos 3 desses elementos requeridos funcionais do programa modificado. No exemplo utilizado, foram exercitados os elementos requeridos funcionais 4, 5 e 6 (vide Tabela 5.7) pois todos os dados de teste selecionados da variável *premio* possuem valores inferiores ou iguais a três (limite entre as duas classes válidas dessa variável).

Assim como no teste tradicional, o passo seguinte será a definição de novos casos de teste (conjunto T'') para essas funcionalidades que ainda não foram testadas na versão modificada. Foram preparados, portanto, mais 3 casos de teste para cobrir os elementos requeridos funcionais 7, 8 e 9 (vide Tabela 5.7) que, aplicados no teste de regressão, atingiram o patamar de 90% e 84,61% de cobertura para os critérios Todos-Potenciais-Usos (PU) e Todos-Potenciais-DU-Caminhos (PDU), respectivamente. A cobertura ideal a ser atingida seria 100% para ambos os critérios, porém, se observarmos a Tabela 5.8, notaremos que isso não acontecerá nesse caso, devido à existência de elementos requeridos não executáveis.

Caso ainda existissem elementos requeridos executáveis não exercitados, os mesmos deveriam ser eliminados através da derivação e aplicação de novos casos de teste estruturais suficientes para cobri-los e satisfazer o critério de parada (seriam adicionados outros casos de teste ao conjunto T'').

A Tabela 5.10 mostra os elementos requeridos do critérios Todos-Potenciais-Usos que não necessitaram ser reavaliados no teste de regressão e os casos de teste que os eliminaram. É importante observar que, entre parênteses, à frente de cada associação, estão listadas associações equivalentes no programa original, e que apenas os casos de teste que exercitaram somente elementos requeridos do programa original, cujos equivalentes são preservados no programa modificado, estão listados nessa tabela (casos de teste reutilizáveis).

Também poderá ser observado que algumas associações que não precisam ser reavaliadas não foram exercitadas pelos casos de teste selecionados (26, 44, 45 e 46). Isso ocorre porque a manutenção consistiu da expansão de um nó em um conjunto de quatro nós que representam um *if-else*. A Figura 5.2 mostra que a variável *D* permaneceu no nó de entrada do *if-else* (nó 10 antes da condição inserida) e, portanto, as associações que partem de qualquer nó e terminam no nó 10, sem cruzar partes modificadas, estão preservadas (são associações que atingem o limite superior da modificação, denominado nó **IN**). De forma similar, a variável *i* permaneceu no nó de saída do *if-else* (nó 13, que encerra o *if-else* inserido) e, portanto, as associações que partem desse nó estarão preservadas se não passarem por regiões modificadas do programa (associações que partem do limite inferior da modificação, denominado nó **OUT**).

Partindo-se do princípio que todos os casos de teste de regressão selecionados serão aplicados, cada uma dessas associações, certamente, será eliminada no teste de regressão por algum desses casos de teste; portanto, é desnecessário testá-las novamente.

A Tabela 5.11 mostra os elementos requeridos que precisaram ser reavaliados e os casos de teste que os eliminaram. Mais uma vez, pode ser observado que estão listados apenas os casos de teste novos (conjunto T'') e aqueles que, no programa original, eliminaram algum elemento requerido, cujo potencial-equivalente está

classificado como novo ou modificado na versão modificada de função (casos de teste reaplicáveis, conjunto T').

Tabela 5.10 – Elementos Requeridos Não Selecionados para o Teste de Regressão, Eliminados pelo Teste Original

Elementos Requeridos Não Selecionados para o Teste de Regressão pelo Critério Todos-Potenciais-Usos	**Casos de Teste Não Selecionados (T-T')**					
	1	**2**	**3**	**7**	**8**	**9**
Associacoes requeridas pelo ***Grafo(1)***						
1) <1,(16,18),{ valor, mulher, premio }> (1)				√		
2) <1,(16,17),{ valor, mulher, premio }> (2)					√	
3) <1,(14,15),{ valor, mulher, premio }> (3)						√
7) <1,(7,8),{ valor, mulher, premio }> (6)			√			
8) <1,(3,4),{ valor, mulher, premio }> (7)		√				
9) <1,(1,2),{ valor, mulher, premio }> (8)	√					
Associacoes requeridas pelo ***Grafo(5)***						
10) <5,(18,19),{ valseg }> (9)				√		
11) <5,(16,18),{ valseg }> (10)				√		
12) <5,(16,17),{ valseg }> (11)					√	
13) <5,(15,19),{ valseg }> (12)						√
14) <5,(14,15),{ valseg }> (13)						√
18) <5,(7,8),{ valseg }> (16)			√			
Associacoes requeridas pelo ***Grafo(7)***						
19) <7,(16,18),{ D, di, i }> (17)				√		
20) <7,(16,17),{ D, di, i }> (18)					√	
21) <7,(14,15),{ D, di, i }> (19)						√
26) <7,(9,10),{ D, di, i }> (20)						
27) <7,(7,8),{ D, di, i }> (21)			√			
Associacoes requeridas pelo ***Grafo(13)***						
44) <13,(16,18),{ i }> (22)						
45) <13,(16,17),{ i }> (23)						
46) <13,(14,15),{ i }> (24)						
Associacoes requeridas pelo ***Grafo(15)***						
51) <15,(,),{ D }> (26)						√
Associacoes requeridas pelo ***Grafo(19)***						
52) <19,(,),{ valseg }> (27)				√		√

Também observa-se na Tabela 5.11 que existem algumas associações que não foram eliminadas pelo teste de regressão (22, 23 e 43), pois não existe conjunto de valores de entrada que possam exercitá-las (são associações não executáveis).

As Tabelas 5.12 e 5.13 mostram resumos dos resultados obtidos para as versões original e modificada da função `calc_seg`. A partir do próximo estudo de caso, serão mostradas apenas essas tabelas finais, que sintetizam os resultados obtidos pela metodologia de seleção de casos de teste de regressão que está sendo proposta.

Nessas tabelas, para cada critério de teste estrutural avaliado, é mostrada a cobertura provida por cada caso de teste, em número absolutos (número de casos de teste eliminados) e relativos (porcentagem de cobertura). Na última coluna são apresentados os resultados finais obtidos (cobertura obtida) e desejáveis (cobertura máxima, eliminado-se os elementos requeridos não executáveis).

Tabela 5.11 – Elementos Requeridos Selecionados para o Teste de Regressão Eliminados pelos Casos de Teste Reaplicáveis e Novos

Elementos Requeridos Selecionados para o Teste de Regressão pelo Critério Todos-Potenciais-Usos	**Casos de Teste**					
	Selecionados T'			**Novos T''**		
	4	**5**	**6**	**10**	**11**	**12**
Associacoes requeridas pelo ***Grafo(1)***						
4) <1,(10,12),{ valor, mulher, premio }> (5)				√	√	√
5) <1,(13,9),{ valor, mulher, premio }> (4)	√	√	√	√	√	√
6) <1,(10,11),{ valor, mulher, premio }> (5)	√	√	√	√	√	√
Associacoes requeridas pelo ***Grafo(5)***						
15) <5,(10,12),{ valseg }> (15)				√	√	√
16) <5,(13,9),{ valseg }> (14)	√	√	√	√	√	√
17) <5,(10,11),{ valseg }> (15)	√	√	√	√	√	√
Associacoes requeridas pelo ***Grafo(7)***						
22) <7,(12,13),{ i }> (20)						
23) <7,(10,12),{ di, i }> (20)						
24) <7,(11,13),{ i }> (20)	√	√	√	√	√	√
25) <7,(10,11),{ di, i }> (20)	√	√	√	√	√	√
Associacoes requeridas pelo ***Grafo(10)***						
28) <10,(10,12),{ D }> (25)				√	√	√
29) <10,(16,18),{ D }> (22)		√			√	√
30) <10,(16,17),{ D }> (23)			√			√
31) <10,(14,15),{ D }> (24)	√			√		
32) <10,(9,10),{ D }> (25)	√	√		√	√	√
33) <10,(10,11),{ D }> (25)	√	√	√		√	√
Associacoes requeridas pelo ***Grafo(11)***						
34) <11,(16,18),{ di }> (N)		√				
35) <11,(16,17),{ di }> (N)			√			
36) <11,(14,15),{ di }> (N)	√					
37) <11,(10,12),{ di }> (N)				√	√	√
38) <11,(10,11),{ di }> (N)	√	√		√	√	√
Associacoes requeridas pelo ***Grafo(12)***						
39) <12,(16,18),{ di }> (N)					√	√
40) <12,(16,17),{ di }> (N)						√
41) <12,(14,15),{ di }> (N)				√		
42) <12,(10,12),{ di }> (N)				√	√	
43) <12,(10,11),{ di }> (N)						
Associacoes requeridas pelo **Grafo(13)**						
47) <13,(12,13),{ i }> (25)				√	√	√
48) <13,(10,12),{ i }> (25)				√	√	√
49) <13,(11,13),{ i }> (25)	√	√	√	√	√	√
50) <13,(10,11),{ i }> (25)	√	√	√	√	√	√

Tabela 5.12 – Cobertura Obtida pela função `calc_seg` (versão original)

Critério	Casos de Teste que cobrem Elementos Requeridos Funcionais (Funcionalidades)							Casos de Teste que provêm aumento de Cobertura				Cobertura Final	
	1	**2**	**3**	**4**	**5**	**6**	**Total**	**7**	**8**	**9**	**Total**	**Obtida**	**Máx.**
PU	1 3,7	1 3,7	3 11,1	12 44,5	4 14,8	3 11,1	**24** **88,9%**	1 3,7	1 3,7	1 3,7	**3** **11,1 %**	**27** **100%**	**27/27** **100%**
PDU	1 4,35	1 4,35	3 13,04	7 30,44	1 4,35	1 4,35	**14** **69,88%**	3 13,04	3 13,04	3 13,04	**9** **39,12%**	**23** **100%**	**23/23** **100%**

Tabela 5.13 – Cobertura Obtida pela função `calc_seg` (versão modificada)

Critério	Casos de Teste Selecionados para o Teste de Regressão (T')				Novos Casos de Teste (T'')				Cobertura Final	
	4	**5**	**6**	**Total**	**10**	**11**	**12**	**Total**	**Obtida**	**Máx.**
PU	13 43,34	2 6,66	2 6,66	**17** **56,66%**	8 26,66	1 3,34	1 3,34	**10** **33,34 %**	**27** **90%**	**27/30** **90%**
PDU	8 30,77	2 7,69	2 7,69	**12** **46,15%**	6 23,08	2 7,69	2 7,69	**10** **38,46%**	**22** **84,61%**	**22/26** **84,61%**

5.3.2 Estudo de Caso 2 – Função `merge`

Nesse segundo estudo de caso, foi escolhido o programa `merge`. Logo abaixo, está descrita a especificação desse módulo, que intercala números inteiros armazenados em dois vetores, gerando um terceiro vetor (que permite repetições). A Figura 5.3 mostra o módulo `merge`, e seu grafo de fluxo de controle e a Tabela 5.14 descreve os resultados finais obtidos.

Especificação do módulo `merge`:

1. Objetivo: Intercalar, permitindo repetição, dois vetores de entrada.
2. Entradas:
   a) v1: Vetor de números inteiros (sem repetições)
   b) v2: Vetor de números inteiros (sem repetições)
   c) v3: Vetor Vazio (receberá o vetor intercalado como saída)
   d) t1: Tamanho do vetor V1
   e) t2: tamanho do vetor V2
3. Processamento:

   Algoritmo de intercalação de dois vetores, gerando vetor com repetições.
4. Restrição: Vetores de entrada classificados em ordem crescente.
5. Saídas Esperadas:

a) Parâmetro de saída: (passados por Endereço)

V3: vetor intercalado

b) Retornos Válidos da Função:

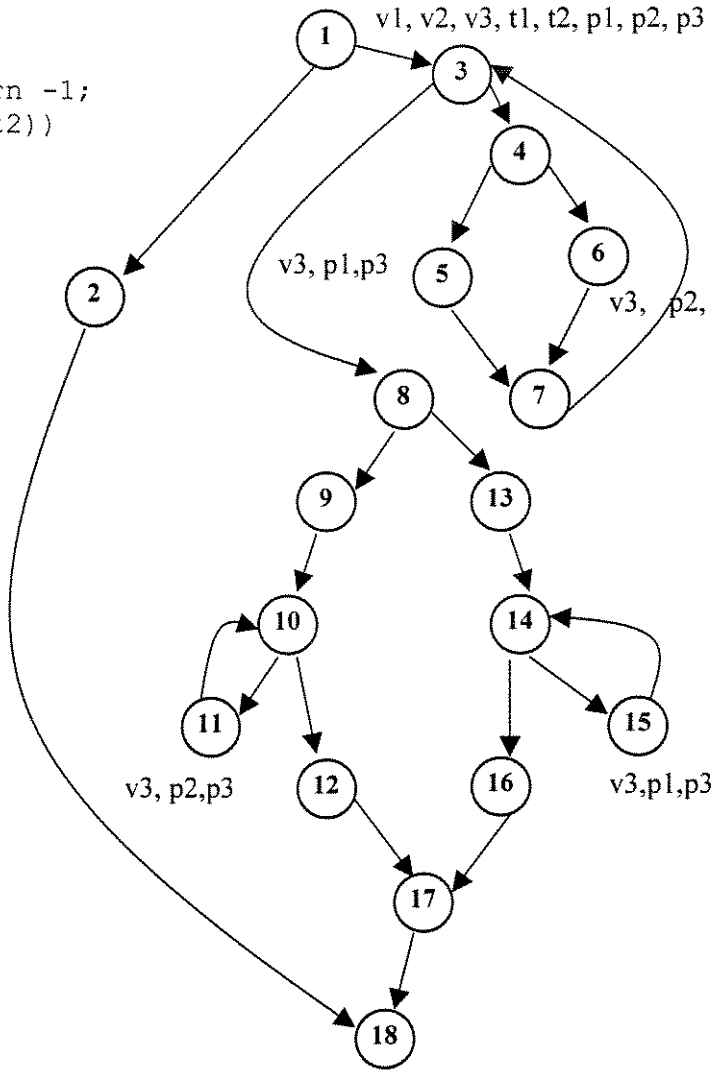
-1: Erro no tamanho em um dos vetores entrada

0: Vetor resultante vazio

> 0: Tamanho do vetor de saída

```
#include <stdio.h>

int merge(int v1[],int v2[],int v3[],int t1,int t2)
{
    int p1,p2,p3;
1   p1 = p2 = p3 = 0;
1,2 if (t1 < 0 || t2 < 0) return -1;
3   while ((p1 < t1) && (p2 < t2))
4   { if (v1[p1] < v2[p2])
5       { v3[p3] = v1[p1];
5         p1++;
5         p3++;
5       }
6     else { v3[p3] = v2[p2];
6            p2++;
6            p3++;
6          }
7   }
8   if (p1 >= t1)
9       for (;p2<t2;p2++)
10       {  v3[p3] = v2 [p2];
11          p3++;
11,12    }
13  else
14      for (;p1<t1;p1++)
15      {  v3[p3] = v1 [p1];
15         p3++;
15,16    }
17   return (p3);
18
}
```



(a) Função `merge` (b) GFC gerado

Figura 5.3 – Informações do arquivo `merge.c` (versão original de `merge`)

Tabela 5.14 – Cobertura Obtida pela função `merge` (versão original)

Cri-tério	Casos de Teste que cobrem Elementos Requeridos Funcionais (Funcionalidades)										
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Total**
PU	1 3,23	1 3,23	3 9,67	1 3,23	3 9,67	4 12,9	1 3,23	1 3,23	4 12,9	2 6,45	**21** **67,74 %**
PDU	1 4,35	1 4,35	2 8,69	1 4,35	2 8,69	2 8,69	1 4,35	1 4,35	2 8,69	1 4,35	**14** **60,86 %**

Tabela 5.14 – Continuação

Critério	Aumento de Cobertura	Cobertura Final	
	11	**Total**	**Máxima**
PU	3 12,9	24 77,41%	**24/31** **77,41%**
PDU	2 8,69	16 69,55%	**16/23** **69,55%**

A manutenção realizada nesse módulo, também perfectiva, ocasionou uma expansão no Grafo de Fluxo de Controle. Consiste, resumidamente, da alteração do algoritmo de intercalação para gerar o vetor resultante sem repetição de valores. As modificações na especificação do módulo estão descritas a seguir e a Figura 5.4 mostra a versão modificada e seu respectivo GFC. Na Tabela 5.14 estão listadas as informações que foram fornecidas pela RePoKe-Tool, para a orientar a condução do teste de regressão. Na Tabela 5.16 estão listados os resultados obtidos após a execução do teste de regressão.

Especificação do módulo `merge` (modificado):

1. Objetivo: Intercalar, permitindo repetição, dois vetores de entrada.
2. Entradas: Não há modificações.
3. Processamento:

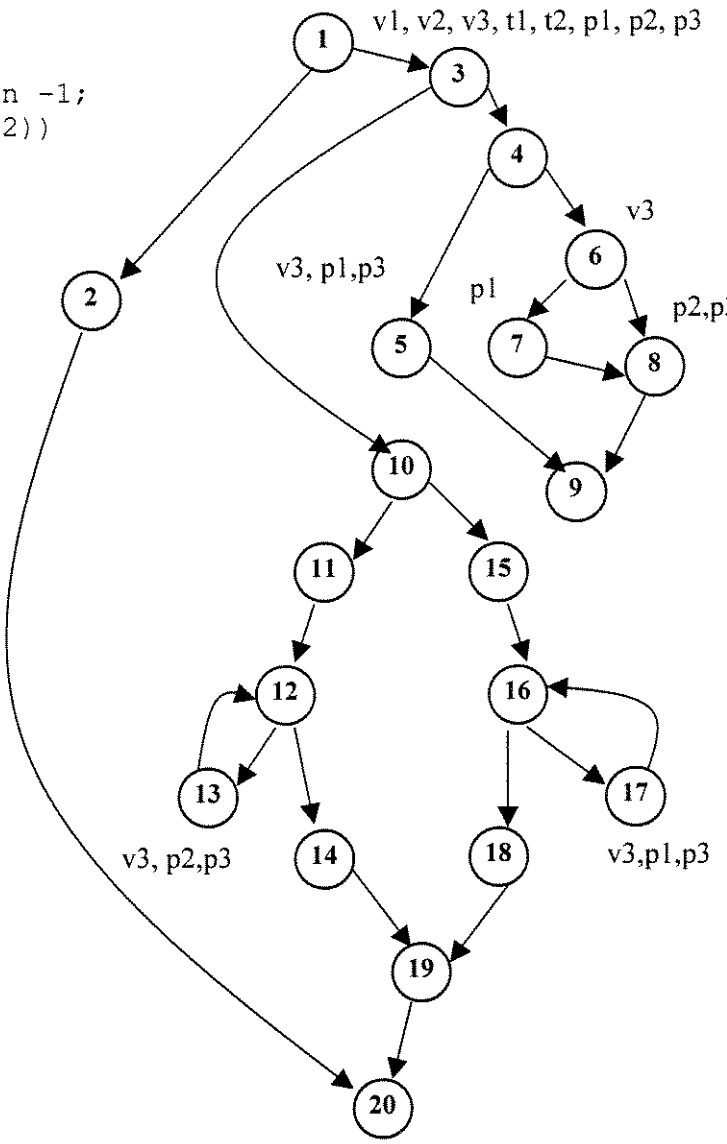
   Algoritmo de intercalação de dois vetores, gerando vetor sem repetições.
4. Restrição: Mantida
5. Saídas Esperadas: Não há modificações.

Tabela 5.15 – Informações da Sessão de Teste da Versão Modificada (`merge`)

Critérios	Nº Total de Elementos Requeridos (E')	Nº de Elementos Selecionados para o Teste de Regressão (E'')	Nº Total de Elementos Requeridos não executáveis	Nº de Casos de Teste Selecionados p/ Regressão (T')
PU	56	35	7	4
PDU	45	32	10	5

```
#include <stdio.h>

int merge(int v1[],int v2[],intv3[],int t1,int t2)
{
    int p1,p2,p3;
1   p1 = p2 = p3 = 0;
1,2 if (t1 < 0 || t2 < 0) return -1;
3   while ((p1 < t1) && (p2 < t2))
4   { if (v1[p1] < v2[p2])
5       { v3[p3] = v1[p1];
5         p1++;
5         p3++;
5       }
6     else { v3[p3] = v2[p2];
6            if v2[p2]==v1[p1]
7              p1++
8            p2++;
8            p3++;
8         }
9   }
10   if (p1 >= t1)
11,12    for (;p2<t2;p2++)
13       {  v3[p3] = v2 [p2];
13          p3++;
13,14    }
15  else
15,16    for (;p1<t1;p1++)
17       {  v3[p3] = v1 [p1];
17          p3++;
17,18    }
19   return (p3);
20
}
```



(a) Função `merge` (b) GFC gerado

Figura 5.4 – Informações do arquivo `merge.c` (versão modificada de `merge`)

Tabela 5.16 – Cobertura Obtida pela função `merge` (versão modificada)

Cri-tério	Casos de Teste Selecionados para o Teste de Regressão (T')					
	6	**7**	**9**	**10**	**11**	**Total**
PU	7 20,0	7 20,0	4 11,42	1 2,86	0 0,00	**19** **54,28 %**
PDU	4 12,5	4 12,5	3 9,375	1 3,125	0 0,00	**12** **37,5 %**

Tabela 5.16 – Continuação

Cri-tério	Novos Casos de Teste (T'')						Cobertura Final	
	12	**13**	**14**	**15**	**16**	**Total**	**Obtida**	**Máxima**
PU	1 2,86	2 5,72	4 11,42	1 2,86	1 2,86	**9** **25,72**	**28** **80%**	**28/35** **80%**
PDU	2 6,25	1 3,125	3 9,375	2 6,25	2 6,25	**10** **31,25**	**22** **68,75%**	**22/32** **68,75 %**

5.3.3 Estudo de Caso 3 – Função `triangulo`

Para este estudo de caso, foi escolhido o programa `triangulo`, que classifica um triângulo de acordo com o valor de seus lados. A Figura 5.5 mostra a versão original do módulo, seu grafo de fluxo de controle. A Tabela 5.17 descreve os resultados finais obtidos e, logo abaixo, está descrita a especificação desse módulo.

Especificação do módulo `triangulo`:

1. Objetivo: Classificar um triângulo de acordo com o valor de seus lados
2. Entradas (Parâmetros):
   a) a, primeiro lado
   b) b, segundo lado
   c) c, terceiro lado
3. Processamento:

   Classificar o triângulo de acordo com seus lados. Retornar códigos de erro se as entradas não estiverem no formato adequado ou se os valores não formarem um triângulo.
4. Restrição: Formato da Entrada: em ordem crescente (a >= b >= c)
5. Saídas Esperadas (Valor de Retorno):
   a) 0, Formato de entrada Inválido
   b) 1, Valores lidos não formam um triângulo

c) 2, Equilátero

d) 3, Isósceles

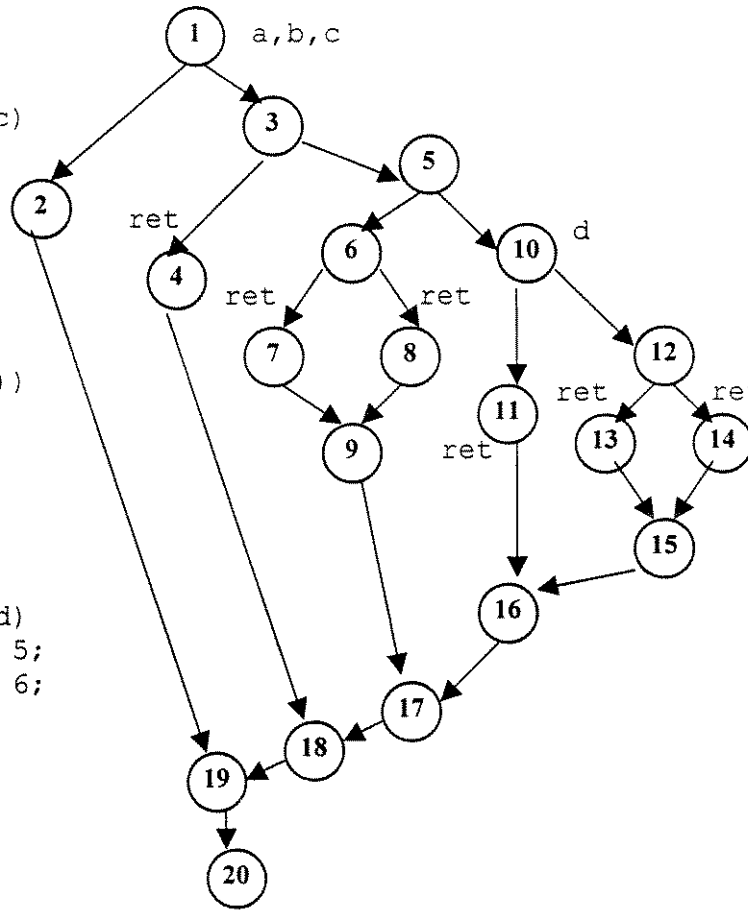
e) 4, Escaleno Reto

f) 5, Escaleno Agudo

g) 6, Escaleno Obtuso

```
#include <stdio.h>

int triangulo(int a,int b,int c)
{ int ret, d;
1   if (a<b || b<c)
      ret = 0;
3   else
3   if (a>=b+c)
4       ret = 1;
5   else if ((a==b) || (b==c))
6           if ((a==b) && (b==c))
7                 ret = 2;
8,9         else ret = 3;
10        else
10        { d = (b*b)+(c*c);
10          if ((a*a)==d)
11                ret = 4;
12            else if ((a*a) < d)
13                        ret = 5;
14                    else ret = 6;
15,16     }
17,18
19   return (ret);
20
}
```



(a) Função `triangulo` (b) GFC gerado

Figura 5.5 – Informações do arquivo `triang.c` (versão original de `triangulo`)

Tabela 5.17 – Cobertura Obtida pela função `triangulo` (versão original)

Cri-tério	Casos de Teste que cobrem Elementos. Requeridos Funcionais (Funcionalidades)							Cobertura Final	
	1	**2**	**3**	**4**	**5**	**6**	**7**	**Obtida**	**Máxima**
PU	2 11,77	2 11,77	2 11,77	2 11,77	3 17,64	3 17,64	3 17,64	**17** **100%**	**17/17** **100%**
PDU	2 11,77	2 11,77	2 11,77	2 11,77	3 17,64	3 17,64	3 17,64	**17** **100%**	**17/17** **100%**

A manutenção proposta poderia ser realizada para que esse módulo fosse reutilizado como componente em algum outro sistema (reuso de software). Ocasionou uma contração no Grafo de Fluxo de Controle, pois na especificação da versão modificada retirou-se a classificação diferenciada dos triângulos escalenos. As modificações na especificação do módulo estão descritas a seguir e a Figura 5.6 mostra a versão modificada e seu respectivo GFC. Na Tabela 5.18 estão listadas as informações que foram fornecidas pela RePoKe-Tool, para a orientar a condução do teste de regressão.

Especificação do módulo `triangulo` (modificada):

1. Objetivo: Classificar um triângulo de acordo com o valor de seus lados
2. Entradas: Não houve modificações
3. Processamento: Suprimir a classificação diferenciada dos triângulos escaleno.
4. Restrição: Não houve modificação
5. Saídas Esperadas (Valor de Retorno):
    a) 0, Formato de entrada Inválido
    b) 1, Valores lido não formam um triângulo
    c) 2, Equilátero
    d) 3, Isósceles
    e) 4, Escaleno

Tabela 5.18 – Informações da Sessão de Teste da versão modificada (`triangulo`)

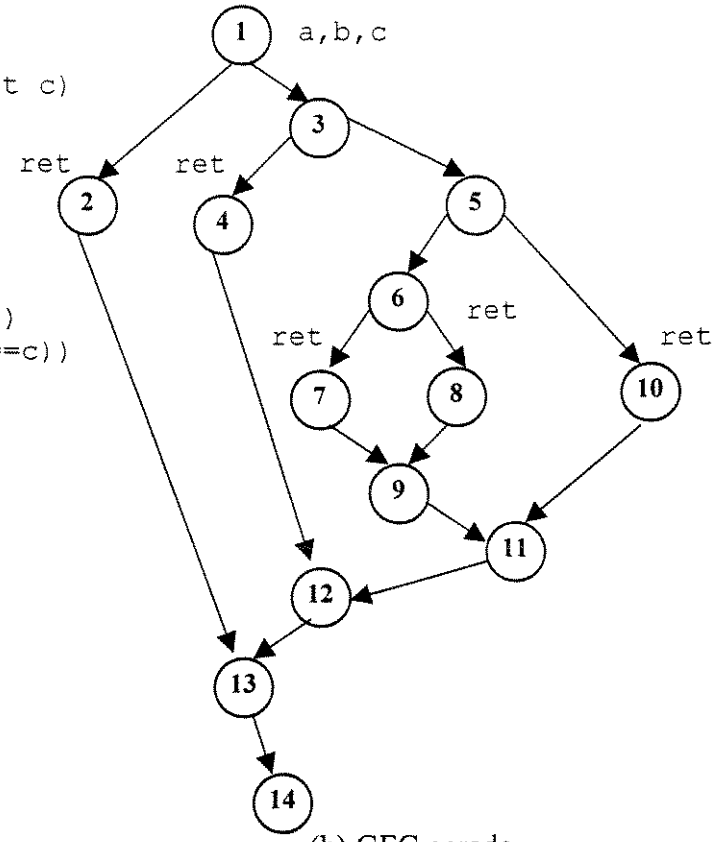
Critérios	Nº Total de Elementos Requeridos (E')	Nº de Elementos Selecionados para o Teste de Regressão (E'')	Nº Total de Elementos Requeridos não executáveis	Nº de Casos de Teste Selecionados p/ Regressão (T')
PU	10	2	-	3
PDU	10	2	-	3

O estudo de caso da função `triangulo` mostra dois exemplos de caso de teste redundantes. Observando-se a cobertura obtida por cada caso de teste de regressão (vide Tabela 5.19), notar-se-á que os casos de teste 6 e 7 (selecionados para o teste de regressão) não provêm aumento de cobertura na versão modificada, para ambos os critérios de teste que estão sendo usados. Contudo, na versão original, contribuíam para o aumento da cobertura para ambos os critérios. Casos de teste que não fornecem aumento de cobertura no programa modificado e todos os elementos requeridos que foram eliminados por ele no programa original são relacionados como potencial-equivalente a elementos requeridos selecionados para o teste de regressão, podem ser retirados da base de teste, pois são redundantes. É importante lembrar que essa situação é diferente do caso de teste obsoleto, que ocorre quando um caso de teste original exercitava apenas associações que foram removidas da nova versão do

programa (e não associações potencial-equivalentes que não provêm aumento de cobertura).

```
#include <stdio.h>
int triangulo(int a,int b,int c)
{ int ret, d;
1  if (a<b || b<c)
2   ret = 0;
3  else
3  if (a>=b+c)
4       ret = 1;
5  else if ((a==b) || (b==c))
6            if ((a==b) && (b==c))
7                 ret = 2;
8,9          else ret = 3;
10,11,12 else ret = 4;
13  return (ret);
14
}
```

(a) Função `triangulo`



(b) GFC gerado

Figura 5.6 –Informações do arquivo `triang.c` (versão modificada de `triangulo`)

Tabela 5.19 – Cobertura Obtida pela função `triangulo` (versão modificada)

Critério	Casos de Teste Selecionados para o Teste de Regressão (T')			Cobertura Final	
	5	**6**	**7**	**Obtida**	**Máxima**
PU	2 100%	0 0,00	0 0,00	**2** **100%**	**2/2** **100%**
PDU	2 100%	3 0,00	0 0,00	**2** **100%**	**2/2** **100%**

5.3.4 Estudo de Caso 4 – Função `contagem`

Para este estudo de caso, foi escolhido o programa `contagem`, que conta e imprime o número de vogais e seqüências de vogais (qualquer encontro vocálico) existentes em uma cadeia de caracteres recebida como parâmetro. A Figura 5.7 mostra a versão original do módulo `contagem`, seu grafo de fluxo de controle e uma função "stub", denominada `isvogal` cujo valor de retorno é fundamental para a função `contagem`. A Tabela 5.20 descreve os resultados finais obtidos e logo abaixo, está descrita a especificação desse módulo.

Especificação do módulo `contagem`:

1. Objetivo: Contar o número de vogais e seqüências de vogais (qualquer encontro 2 ou mais vogais) de uma string recebida como parâmetro

2. Entrada: s, uma cadeia de caracteres qualquer

3. Processamento:

   Contar o número de vogais e seqüências de vogais dessa string .

4. Restrição: Não há

5. Saídas Esperadas:

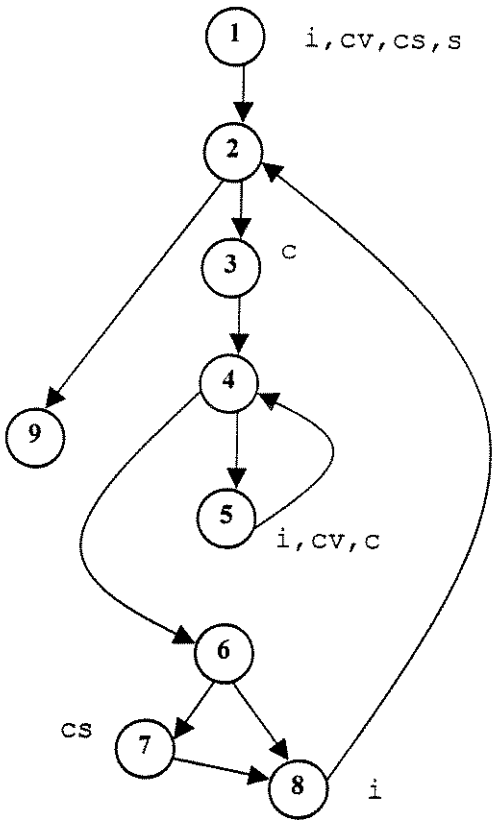
   a) Retornos: Não há

   b) Parâmetros de Saída: Não há

   c) Mensagem: Imprimir os dois valores obtidos na função

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
int isvogal(char car)
{   int ret;
    switch (tolower(car))
    {
     case 'a':
     case 'e':
     case 'i':
     case 'o':
     case 'u': ret = 1;
               break;
     default: ret = 0;
    }
     return(ret);
}


void contagem(char s[])
{
1   int cv = 0, cs = 0, i = 0, c;
2   while (i<strlen(s))
3   {  c = 0;
4      while (isvogal(s[i]))
5      { i++;
5        c++;
5        cv++;
5      }
6,7    if (c > 2) cs++;
8,     i++;
8   }
9   printf ("String: %s possui %d vogais e %d sequencias de
vogais",s,cv,cs);}
```



(a) Função `contagem` (b) GFC gerado

Figura 5.7 – Informações do arquivo `vog.c` (versão original de `contagem`)

Tabela 5.20 – Cobertura Obtida pela função `contagem` (versão original)

Cri-tério	Casos de Teste que cobrem Elem. Req. Funcionais (Funcionalidades)			Casos de Teste que provêm aumento de Cobertura			Cobertura Final	
	1	**2**	**Total**	**3**	**4**	**Total**	**Obtida**	**Máxima**
PU	3 8,1	21 56,76	24 64,86	6 16,21	2 5,41	**8** **21,62**	**32** **88,48 %**	**32/37** **88,48 %**
PDU	3 11,11	11 40,71	14 51,85	5 18,51	2 7,41	7 **25,92**	**21** **77,77%**	**21/27** **77,77 %**

A manutenção realizada nesse módulo foi corretiva e não ocasionou modificações no Grafo de Fluxo de Controle. Observando-se o programa implementado na Figura 5.7, pode-se verificar que para que aconteça o menor encontro vocálico possível (2 vogais), basta que o laço *while* mais interno (nó 4) seja executado 2 vezes em seguida, pois em cada passagem, a variável *c* será incrementada em uma unidade. Portanto, no pior caso de encontro vocálico, essa variável sairá desse laço valendo 2, pois é inicializada fora do laço (nó 3) com valor 0. Logo após esse laço, o comando de seleção `if (c > 2)` (nó 6) tomará a decisão se a variável que acumula a quantidade de encontros vocálicos deve ou não ser incrementada em função do valor da variável *c*. Essa decisão será tomada erradamente, se acontecer o caso do menor encontro vocálico e corretamente para os demais casos.

Além disso, deve ser observado que nenhum dos quatro casos de teste (que foram suficientes para atingir o critério de parada estabelecido) revelou esse erro. Assim, a manutenção necessária para corrigir essa função é modificar o predicado do ***if*** do nó 6 para `(c >= 2)`.

Por se tratar de uma manutenção corretiva e não haver mudanças na especificação do módulo, no grafo de fluxo de controle e em definições de variáveis, a quantidade de elementos requeridos também permanecerá a mesma. A Figura 5.8 mostra a versão modificada do programa e seu respectivo GFC. Na Tabela 5.21 estão listadas as informações que foram fornecidas pela RePoKe-Tool, para auxiliar a condução do teste de regressão.

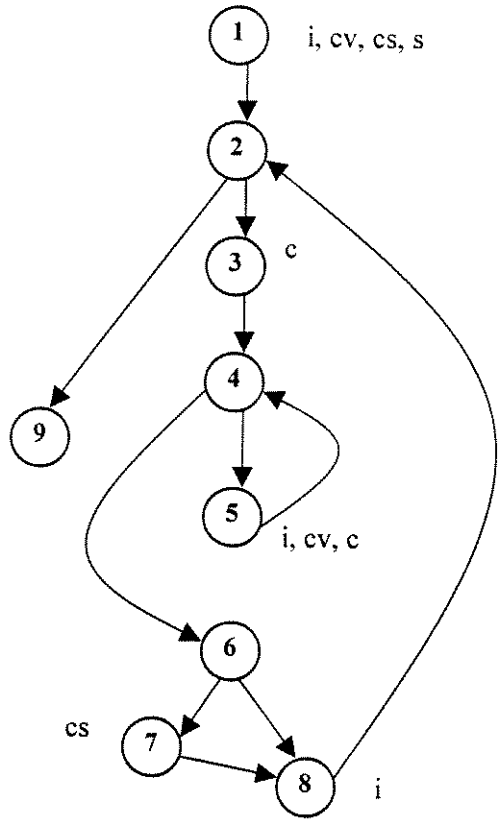
Tabela 5.21 – Informações da Sessão de Teste da Versão Modificada (`contagem`)

Critérios	Nº Total de Elementos Requeridos (E')	Nº de Elementos Selecionados para o Teste de Regressão (E'')	Nº Total de Elementos Requeridos não executáveis	Nº de Casos de Teste Selecionados p/ Regressão (T')
PU	37	20	5	3
PDU	27	16	6	3

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

int isvogal(char car)
{   int ret;
    switch (tolower(car))
    {
     case 'a':
     case 'e':
     case 'i':
     case 'o':
     case 'u': ret = 1;
               break;
     default: ret = 0;
    }
     return(ret);
}


void contagem(char s[])
{
1   int cv = 0, cs = 0, i = 0, c;
2   while (i<strlen(s))
3   {  c = 0;
4      while (isvogal(s[i]))
5      { i++;
5        c++;
5        cv++;
5      }
6,7    if (c >= 2) cs++;
8,     i++;
8   }
9   printf ("String: %s possui %d vogais e %d sequencias de
vogais",s,cv,cs);
}
```



(a) Função `contagem` (b) GFC gerado

Figura 5.8 – Informações do arquivo `vog.c` (versão modificada de `contagem`)

Tabela 5.22 – Cobertura Obtida pela função `contagem` (versão modificada)

Critério	Casos de Teste Selecionados para o Teste de Regressão (T')				Cobertura Final	
	2	3	4	Total	Obtida	Máxima
PDU	7 35	6 30	2 10	**15** **75,0**	**15** **75%**	**15/20** **75%**
PU	3 18,75	5 31,25	2 12,5	**10** **62,5**	**10** **62,5 %**	**10/16** **62,5 %**

O estudo de caso da função `contagem` mostra um exemplo de manutenção corretiva sem alteração do GFC. Observando-se a cobertura final obtida pelo teste de regressão (vide Tabela 5.22), notar-se-á que esses foram suficientes para atingir o

critério de parada, para ambos os critérios de teste que estão sendo usados. Esse resultado comprova que, nesse tipo de manutenção, a probabilidade do conjunto de casos de teste selecionados (T') ser suficiente é maior que em outros casos.

Isso poderia não acontecer se a modificação afetasse um predicado, cuja avaliação gerada (em tempo de execução) para cada um desses casos de teste não mais implicasse na execução de um de seus ramos. Supondo-se que houvesse 2 casos de teste que exercitavam caminhos que atravessam um determinado predicado modificado (um para cada ramo dessa condição), e que a modificação fez com que ambos exercitassem o mesmo ramo (e por conseqüência o mesmo caminho), um dos casos de teste passaria a ser redundante e outro teria que ser preparado para substituí-lo, a fim de exercitar o caminho que era coberto no teste original e não foi coberto no teste de regressão.

5.3.5 Análise e Comparação dos Resultados Obtidos

Nessa seção, os resultados obtidos através dos estudos de casos serão analisados e comparados com o principal objetivo traçado (estudar a viabilidade da utilização de técnicas funcionais para orientar a seleção dos casos de teste que devem ser reaplicados no teste de regressão, com objetivo de obter uma boa cobertura estrutural inicial). Para cada estudo de caso, os resultados relevantes foram mostrados em algumas tabelas. As principais informações dessas tabelas foram corrigidas, de forma a desconsiderar os elementos requeridos não executáveis. A Tabela 5.23, apresenta esses resultados já ajustados.

Tabela 5.23 – Resumo da Cobertura Obtidas nos Estudos de Casos [1]

Programa	Critério	Cobertura Inicial Obtida pelos Casos de Teste Funcionais					
		Versão Original (Todos os Elementos Requeridos) (E)		Versão Modificada (Apenas Elementos Requer. Selecionados) (Sessão Reduzida E'')		Versão Modificada (Todos Elementos Requeridos) (E')	
		Eliminados	% Obtida	Eliminados	% Obtida	Eliminados	% Obtida
calc_seg	PU	24/27	88,90%	17/27	62,96%	39/49	79,59%
	PDU	14/23	69,88%	12/22	54,54%	31/41	75,60%
merge	PU	21/24	87,50%	19/28	67,85%	40/49	81,63%
	PDU	14/16	87,50%	12/22	54,54%	25/35	71,42%
triang	PU	17/17	100%	2/2	100%	10/10	100%
	PDU	17/17	100%	2/2	100%	10/10	100%
Contagem	PU	24/32	75%	15/15	100%	32/32	100%
	PDU	14/21	66,66%	10/10	100%	21/21	100%

[1] Já Eliminados os Elementos Requeridos Não Executáveis (Escala de 0 a 100%)

A primeira coluna mostra a cobertura obtida pela **aplicação dos casos de teste funcionais** nas versões originais dos programas (critérios PU e PDU). As demais colunas mostram os resultados obtidos pela aplicação de **todos os casos de teste** que foram **selecionados** para o teste de regressão sendo que, na segunda coluna, esses resultados estão calculados em função apenas dos elementos requeridos que foram selecionados para o teste de regressão (conjunto reduzido ou E'') e na terceira coluna esses dados foram corrigidos considerando-se todo o conjunto (como se os casos de teste reutilizáveis também fossem aplicados e todos os elementos requeridos considerados). A seguir, estão listadas as principais conclusões, divididas por grupo de programas com características afins.

a) **Programas com Expansão do Grafo de Fluxo de Controle:** Em geral, manutenções desse tipo são perfectivas, ou seja, há modificação da especificação do programa (introdução de novas funcionalidades e conseqüentemente novos elementos requeridos estruturais). Nesses casos, a tendência é que os casos de teste selecionados pelo teste de regressão (T') não sejam suficientes para cobrir todas as novas funcionalidade e novos casos de teste deverão ser preparados (T''). Mesmo assim, a aplicação do conjunto T' tende a prover uma boa cobertura inicial desses conjuntos. Os resultados obtidos pelos programas `calc_seg` e `merge` , mostrados na Tabela 5.21 comprovam essa argumentação. Para o primeiro programa (`calc_seg`) foram obtidas pelo teste de regressão coberturas iniciais de 79,59% (Todos-Potenciais-Usos) e 75,60% (Todos-Potenciais-Du-Caminhos), valores muito próximos aos obtidos no teste original (88,90% e 69,88% para os critérios PU e PDU, respectivamente). O aumento da cobertura para o critérios Todos-Potenciais-Du-Caminhos é um dado interessante a ser considerado. Sua ocorrência é perfeitamente normal (apesar de pouco freqüente), pois casos de testes reutilizados podem cobrir uma relação maior de elementos requeridos na versão modificada do que na versão original. Os resultados obtidos pela versão modificada da função `merge` (81,63% para PU e 71,42% para PDU) estão dentro da normalidade (a cobertura obtida está próxima e abaixo da obtida originalmente).

b) **Programas com Contração do Grafo de Fluxo de Controle:** Assim como no caso anterior, as contrações também são caracterizadas por modificações na especificação do programa. Porém, nesse casos, o comum é que algumas funcionalidades do programa sejam suprimidas (e conseqüentemente, elementos requeridos estruturais), o que pode implicar na existência de casos de teste obsoletos ou redundantes dentre os selecionados para o teste de regressão. Nesses casos, a aplicação de um subconjunto de T' pode ser suficiente para que o critério de parada seja satisfeito. O programa `triangulo` ilustra bem esse caso, pois somente um caso de teste selecionado para o teste de regressão foi suficiente para obter 100% de cobertura da versão modificada. Os demais casos de teste passaram a ser redundantes e foram eliminados da base de testes.

c) **Programas sem Modificações no Grafo de Fluxo de Controle:** Nesse caso, o conjunto T' selecionado para o teste de regressão tende a ter o tamanho exato para garantir a cobertura esperada pelo critério de parada (a quantidade de elementos requeridos tende a ser a mesma, ou então um pouco maior ou menor que na versão original). O programa `contagem` é um exemplo típico desse caso, pois o conjunto T' foi suficiente para cobrir 100% dos elementos requeridos selecionados para o teste de regressão. Casos de teste adicionais somente serão necessários se houver ocorrência de casos de teste redundantes, motivados, por exemplo, por modificações em predicados.

Os dados obtidos e analisados anteriormente comprovam que a adoção de uma metodologia de seleção de casos de teste segura, como é a que está sendo discutida, tende a prover um bom índice de cobertura dos elementos requeridos para o teste estrutural e também aumentam a probabilidade de que erros de regressão sejam revelados (a confiabilidade aumenta à medida que erros são descobertos; essa revelação de defeitos é maior se forem usados mais casos de teste). A utilização de uma estratégia Minimal tenderia a reduzir o número de casos de teste selecionados para o teste de regressão, economizaria recursos, mas tenderia a revelar menos erros de regressão (vide Figura 3.7). Estratégias desse tipo atendem prioritariamente um objetivo secundário do teste de regressão (diminuição dos custos) em detrimento ao principal (revelação de erros).

As conclusões aqui apresentadas fornecem subsídios que convergem para a aplicabilidade da metodologia de seleção proposta. Experimentos mais abrangentes e melhor controlados serão necessários para comprovar empiricamente essa conclusão. No último capítulo serão descritos alguns trabalhos futuros que deverão ser realizados, dentre os quais serão destacados esses experimentos.

5.4 Um Pequeno Guia para Orientar Programadores de Manutenção

Escrever programas de computadores é uma "arte" que requer domínio da linguagem de programação que está sendo usada, das técnicas de programação, experiência e uma boa dose de criatividade. Modificar programas (geralmente escritos por outras pessoas) requer, além de tudo isso, uma boa dose de paciência, pois esse tipo de atividade de programação geralmente é conduzida sob muita pressão, dentre as quais destacam-se o prazo apertadíssimo, custos altos (manutenção e teste de regressão) e, pior de todos, a freqüente falta de uma configuração adequada do software que está sendo mantido pode resultar em modificações "às cegas" em um programa cuja legibilidade é muito baixa (também conhecido como código alienígena). Mesmo para os mais experimentados programadores, métricas e guias de referência sempre são bons companheiros nessas situações.

O objetivo desta seção é propor, baseando-se em observações extraídas dos estudos de caso apresentados, um pequeno guia (guideline) que poderá servir como referência para programadores de manutenção, com o objetivo de facilitar o teste de

regressão das unidades (funções ou módulos) modificadas. Nesse contexto, facilitar significa modificar um programa de forma a diminuir o número de elementos requeridos que devem ser reavaliados e o número de casos de teste selecionados para a regressão (usando estratégia de seleção segura), o que vai ao encontro dos objetivos primários e secundários do teste de regressão (revelar erros e diminuir os custos). Deve-se lembrar também que em software estáveis, as manutenções corretivas tendem, em tese, a ser pequenas modificações em algumas unidades, o que justifica a apresentação desse guia. Já no caso de manutenções perfectivas ou então no reuso de componentes de software que exijam modificações, essas regras tendem a ser menos úteis, devido à grande complexidade das modificações que serão levadas a efeito nesses casos.

Antes dessas poucas "regras" serem apresentadas, cabe ressaltar que as mesmas devem ser aplicadas com ponderação, bom senso e, principalmente, caso haja possibilidade. É mais importante manter a legibilidade de um programa amanhã, do que facilitar o teste de regressão hoje (a famosa frase que diz "o barato sai caro" se encaixa perfeitamente nesse contexto). Poderá ser observado que em algumas situações, as regras não poderão ser consideradas.

Regra 1 - Evitar Modificações na Interface de um Módulo

Antes mesmo do programador, essa regra deve ser considerada pelo projetista da manutenção. A modificação da interface de um módulo, ou seja modificar os parâmetros de entrada dessa unidade ou introduzir novas entradas de dados (e.g., leitura via teclado) inviabilizará a reutilização de casos de teste. Para reutilizar-se os teste originais, dados de teste para essas novas entradas/parâmetros deverão ser criados de forma aleatória, constante (e.g., atribuir zero para os dados numéricos), ou então através de geração de dados de teste apenas para essa variável (e.g., execução simbólica). O mais importante a ser colocado é que essa regra pode diminuir muito o custo com geração adicional de dados de teste.

Regra 2 - Evitar a Introdução ou Modificação de Definições de Variáveis, Predicados ou Usos que não Implicam em definições

Evitar a desnecessária modificação de definições, predicados e usos que não implicam em definições pode evitar a seleção de muitos caminhos/associações que deverão ser reavaliados no teste de regressão. Essa regra deve ser considerada especialmente para a família de critérios Potenciais-Usos, que não considera referências explícitas a variáveis, mas sim, a potencialidade de existir essa referência (uso). A modificação de uma definição, causada por alteração de uma constante ou do uso de uma variável à direita do comando de atribuição, exigirá novo teste para todos os Potenciais-DU-Caminhos que partem do nó onde está essa definição. Mudanças em predicados e usos que não implicam em definição (usos predicativos ou **p-uso** e uso computacional ou **c-uso**, respectivamente), exigirão que qualquer Potencial-Du-Caminho que atravesse esse ponto do programa modificado seja reavaliado.

Regra 3 - Evitar a Distribuição das Variáveis Definidas em um Nó que foi Expandido

A adoção dessa regra poderá gerar uma grande economia no teste de regressão. Suponha que um nó que possui **n** variáveis seja expandido para um ***if-else***. Se todas as definições de variáveis permanecerem no nó de entrada (nó IN) desse ***if-else*** todos os Potenciais-DU-Caminhos que terminavam no nó explodido da versão anterior continuarão preservados, uma vez que o predicado foi inserido após todas essas definições de variáveis. De forma análoga, se as variáveis ficarem no nó de saída da expansão (nó OUT), todos os Potenciais-DU-Caminhos que partem desse nó e não passem pela modificação (pode haver um laço) não precisarão ser reavaliados. Em contrapartida, se as variáveis forem distribuídas pelos dois novos nós gerados pela manutenção (vide o modelo de expansão correspondente) ambos os tipos de Potenciais-DU-Caminhos (os que atingem e os que partem desse nó) deverão ser revalidados.

Regra 4 - Evitar Modificações na "Espinha Dorsal" de um Programa

Dentre todas a regras aqui apresentadas, essa talvez seja a mais importante a ser considerada. Todo programa tem uma "espinha dorsal", que é representada pela seqüência de estruturas de programação que se seguem desde o nó de entrada até o nó de saída do programa. As estruturas que se ramificam dessas estruturas que compõe a "espinha dorsal" são estruturas secundárias de um programa (algumas só são exercitadas por conjuntos de entrada de dados muito particulares). Com base nesse conceito, quanto mais longe da "espinha dorsal" for realizada uma modificação, menos elementos requeridos serão reavaliados no teste de regressão. Por exemplo, se for necessária a introdução de mais um *else* em um conjunto de comandos *if-else* encadeados, e essa modificação puder ser feita tanto no primeiro nível desse *if-else* encadeado como no último, a segunda opção deverá ser escolhida.

Regra 5 – Se não for Possível Evitar Modificações na "Espinha Dorsal" do Programa, faze-las o mais Próximo do Nó de Saída do Programa

Essa regra deve ser aplicada caso a anterior não possa ser considerada. Quanto mais próximo do nó de saída de um programa, mais elementos requeridos (associações ou caminhos) cujos nós terminais estão situados antes do ponto onde foi realizada a modificação, não serão afetados pela modificação, ou seja, essas associações não chegarão a sofrer influência da manutenção, uma vez que essa está estrategicamente colocada no final da estrutura de controle do programa ("espinha dorsal").

Apesar de não poder ser adotada integralmente em algumas situações práticas, as regras desse pequeno guia podem ser fundamentais para a minimizar os altos custos do teste de regressão. Estudos empíricos poderão ser conduzidos para avaliar a eficácia dessas regras em diferentes metodologias de teste de regressão, baseadas em diferentes critérios de teste.

Capítulo 6

Conclusões e Trabalhos Futuros

Este trabalho apresenta inicialmente os principais requisitos de uma ferramenta de apoio ao teste de regressão, que é uma das áreas de pesquisa mais recentes da Engenharia de Software. Esses requisitos serviram como suporte para o desenvolvimento de uma ferramenta de apoio ao Teste de Regressão Estrutural, denominada RePoKe-Tool. Neste capítulo são apresentadas as conclusões desse trabalho, trabalhos futuros que poderão ser conduzidos utilizando-se a atual versão da ferramenta e melhorias que ainda deverão ser adicionadas à ferramenta.

6.1 Considerações Finais

Antes de apresentar as conclusões finais, a ferramenta RePoKe-Tool (descrita nos Capítulos 3 e 4) e a metodologia de seleção de casos de teste nela aplicada (descrita no Capítulo 5) são avaliadas. Usando-se os critérios propostos por Rothermell e Harrold [Rot96, RH96] para avaliação de estratégias e técnicas de Teste de Regressão (descritos no Capítulo 2), a avaliação levou às seguintes observações:

a) **Inclusão:** Esse primeiro quesito mede em que extensão a técnica é capaz de escolher casos de teste que, aplicados ao programa modificado, vão gerar valores de saída diferentes; portanto, aplica-se à metodologia proposta que está implementada na ferramenta. A inclusão mede a capacidade de gerar casos de teste reveladores de modificação. Sob essa ótica do teste estrutural, o método de seleção não pode ser classificado como seguro, pois existe pelo menos um caso (mudança em um predicado, que faça com que os dados de teste exercitem o mesmo caminho) onde esse quesito não poderá ser assim qualificado. Apesar disso, os casos de teste que forem selecionados por essa metodologia tendem a possuir altíssima probabilidade de atravessar partes modificadas do código (seleção de alguns casos de teste que não atravessem as modificações é aceitável na prática [RH96]). Usando a mesma terminologia usada por Rothermell e Harold [RH96], pode-se definir a abordagem como segura considerando-se as restrições levantadas, ou seja segura em ambientes controlados.

b) **Precisão:** Ao contrário da inclusão, a precisão mede a capacidade de uma técnica em evitar a escolha de testes que não atravessarão partes modificadas do código. Mais uma vez, sob o ponto de vista do teste estrutural, a técnica é precisa, pois não serão selecionados testes que não atravessam partes modificadas. O fato da Precisão e da Inclusão serem altas deve-se, principalmente, ao escopo da modificação maior que é requerido pela família de Critérios Potenciais Usos, relativamente ao que é requerido por outras famílias de teste estrutural.

c) **Eficiência:** A eficiência da ferramenta não será apresentada nos moldes da Tabela 2.1, uma vez que não foi analisado, para os critérios Potenciais Usos, o pior caso de programa que poderia ser submetido à ferramenta (programas cuja complexidade de caminhos é 2^n, onde n é o número de nós de predicados do programa [Mal91]). Mesmo assim, nos estudos de caso que foram submetidos à ferramenta, o tempo de máquina que foi exigido se enquadrou nos mesmos patamares de ferramentas similares. Análises de complexidade, tanto para a técnica de classificação e seleção de elementos requeridos como da seleção de casos de teste deverão ser realizadas para medir precisamente esse requisito da avaliação.

d) **Generalidade:** Nesse requisito, a ferramenta mostrou-se bastante poderosa. A única estrutura de programação que não é suportada na atual versão é o comando de seleção múltipla (e.g., *switch...case*). Além disso, devido à implementação dos modelos de contração e expansão, qualquer combinação de modelos básicos pode ser implementada (vide Seção 4.3). Além disso, podem ser realizadas quantas mudanças forem necessárias, de cada tipo (fluxo controle, definição de variáveis, predicados, usos que não implicam em definição e as que não afetam fluxo de controle ou fluxo de dados). O exemplo da ferramenta em execução, mostrado no Apêndice A, ilustra bem essa qualidade, uma vez que apenas modificações em predicados não foram realizadas no referido exemplo.

6.2 Conclusões

Para que as conclusões deste trabalho sejam melhor analisadas, foram divididas em duas categorias: conclusões sobre a ferramenta e sobre a metodologia proposta.

6.2.1 Conclusões sobre a Ferramenta RePoKe-Tool

Sobre a ferramenta **RePoKe-Tool** pode-se concluir, baseando-se nas observações anteriormente descritas, que a mesma pode ser classificada, sob a ótica do Teste de Regressão Estrutural baseado em Análise de Fluxo de Dados para a família de Critérios Potenciais-Usos, como genérica, precisa, segura e com eficiência dentro dos parâmetros aceitáveis para essa classe de software.

Além disso, é importante salientar que a RePoKe-TOOL é apenas mais um dentre outros componentes já desenvolvidos que trabalham em conjunto com a POKE-TOOL (e.g., *pokepadrão* [Ver97], *ViewGraph* [VMJ97]) e outros que estão em desenvolvimento (e.g., *pokepaths*). Nesse momento, faz-se necessária uma integração cuidadosa de todos esses componentes, para que o ambiente de teste formado pela POKE-TOOL, suas extensões, componentes adicionais e ferramentas de apoio (como é o caso da RePoKe-TOOL), torne-se cada vez mais abrangente.

6.2.2 Conclusões sobre a Metodologia de Seleção de Casos de Teste de Regressão

Sobre a metodologia de seleção de Casos de Teste de Regressão, pode-se afirmar que os resultados obtidos pelos estudos de casos apresentam fortes indícios da eficiência da aplicação de uma estratégia de seleção de casos de teste utilizando-se técnicas funcionais e abordagem segura, pois os níveis de cobertura obtidos sempre estiveram próximos aos patamares obtidos no teste original.

Destacam-se, dentre os resultados obtidos, a cobertura dos elementos requeridos pelo Teste de Regressão sempre superior de 50% quando houve expansão do Grafo de Fluxo de Controle do programa (tanto para o critério Todos-Potenciais-Usos como para o Critério Todos-Potenciais-DU-Caminhos), e 100% quando não houve modificações ou então contrações do Grafo de Fluxo de Controle (também para ambos os critérios).

Também utilizando os resultados obtidos e as observações anotadas durante os estudos de caso, é proposto um guia de referência que poderá ser muito útil para que projetistas e programadores de manutenção possam projetar modificações inteligíveis, com o menor custo de teste possível e boa probabilidade de revelação de defeitos introduzidos durante esse processo (erros de regressão).

6.3 Trabalhos Futuros

Assim como as conclusões, os trabalhos futuros foram divididos em duas categorias: Melhorias na Ferramenta e Experimentos a serem conduzidos futuramente.

6.3.1 Melhorias na Ferramenta

Apesar de ter sido classificado como um ambiente genérico no contexto do teste de unidade, muitas extensões podem ser feitas na ferramenta. Dentre elas destacam-se:

a) Integrar a ferramenta com o ambiente gráfico que está sendo desenvolvido para a POKE-TOOL.

b) Expandir a ferramenta para tratar variáveis apontadoras (ponteiros) e estruturas de dados. Essa extensão requer que a ferramenta POKE-TOOL seja estendida e passe a considerar de forma mais abrangente esses tipos de estruturas de dados na Análise de Fluxo de Dados para os critérios Potenciais-Usos.

c) Expandir a ferramenta para dar suporte ao Teste de Regressão em Nível de Integração Estrutural. Esse trabalho depende do término do desenvolvimento de outra ferramenta, que seleciona elementos requeridos estruturais para integração entre dois módulos [Vil97].

d) Estender a ferramenta para suportar a seleção de Elementos Requeridos que deverão ser reavaliados e a escolha de Casos de Teste de Regressão Reaplicáveis, para a família de Critérios de Teste Restritos [Ver97].

e) Estender a ferramenta para dar suporte ao teste de Regressão de Programas escritos no Paradigma de Orientação a Objetos (e.g., C++). Essa extensão depende da agregação de critérios de teste para esse tipo de programa à ferramenta de teste POKE-TOOL.

6.3.2 Experimentos Futuros

Conforme já comentado nos Capítulos 3 e 5, novos experimentos deverão ser conduzidos. Dentre esses experimentos destacamos os seguintes:

a) Experimentos que comprovem empiricamente os Resultados obtidos pelos Estudos de Casos conduzidos no Capítulo 5.

b) Experimentos que verifiquem a Revelação de defeitos (antigos e de regressão) em programas que sofreram manutenções (adaptativas e/ou perfectivas) e comparem os resultados obtidos quando o teste de regressão for aplicado à RePoKe-TOOL (Critérios Potenciais-usos) com os que forem obtidos através de outras ferramentas que suportem o teste de Regressão Estrutural (e.g, `TestTube` e ASSET).

c) Experimentos que comprovem empiricamente as regras (guidelines) definidas pelo Guia de Referência para Programadores de Manutenção, proposto na Seção 5.4.

d) Experimento que auxilie a obtenção de um modelo de custo de Teste de Regressão adequado para a definição do ponto ótimo entre as estratégias seletiva e "retest-all" , sob a ótica da família de Critérios Potenciais Usos.

Referências Bibliográficas

[AHKL93] H. Agrawal, J. Horgan, E. Krause and S. London, "Incremental Regression Testing", *Proc. of the IEEE Conference on Software Maintenance*, pp. 348-357, September 1993.

[BCC88] P. Benedusi, A. Cimitile and U. De Carlini, "Post-Maintenance Testing on Path Change Analysis", *Proc. of the IEEE Conference on Software Maintenance*, pp 352-361, 1988.

[Bei84] B. Beizer, *Software System Testing and Quality Assurance*, Van Nostrand and Reinhold, 1984.

[BH93] S. Bates and S. Horwitz, "Incremental Program Testing Using Program Dependence Graphs", *Proc of. 20th ACM Symposium Principles of Programming Languages*, pp. 384-396, January 1993.

[Bin92] D.Binkley, "Using Semantic Difference to Reduce the Cost of Regression Testing", *Proc of the IEEE Conference on Software Maintenance*, pp 41-50, November 1992.

[Bin94] D.Binkley, "Interprocedural Constant Propagation using Dependence Graph and a Data-Flow Model", Technical Report, Department of Computer Science, Loyola College, Maryland, 1994.

[Bin96] D.Binkley, "Reducing the Cost of Regression Testing by Semantic Guided Test Case Selection", Technical Report, Department of Computer Science, Loyola College, Maryland, 1996.

[Bin97] D.Binkley, "Semantic Guided Regression Test Cost Reduction", *IEEE Transactions on Software Engineering*, Vol. SE-23(8), pp 498-516, August 1997.

[Cha91] M. L. Chaim, "POKE-TOOL - Uma Ferramenta para Suporte ao Teste Estrutural Baseado em Análise de Fluxo de Dados", Dissertação de Mestrado, DCA/FEEC/Unicamp, Campinas, SP, Abril de 1991.

[Chu87] T. Chusho, "Test Data Selection and Quality Estimation Based on the Concept of Essential Branches for Path Testing", *IEEE Transactions on Software Engineering*, Vol. SE-13,(5), pp. 509-517, May 1987.

[Cre97] A.N.Crespo, "Modelos de Confiabilidade de Software Baseados em Cobertura de Critérios Estruturais de Teste", Tese de Doutorado, DCA/FEEC/Unicamp, Campinas, SP, 1997 (em andamento).

[CRV94] Y.F. Chen, D.S. Rosenblum and K.P. Vo, "TestTube: A System for Selective Regression Testing", *Proc. of the IEEE Conference on Software Maintenance,* pp 241-260, May 1994.

[Dem79] T. DeMarco, *Structured Analysis and System Specification,* Prentice-Hall, 1979.

[DMLS78] R.A. DeMillo, R.J. Lipton and F.G. Sayward. "Hints os test data selection: Help for the practicing programmer". *IEEE Computer*, VOL. C-11, pp. 34-41, April 1978.

[DO91] R.A. DeMillo and A.J.Offutt, "Constraint-based Automatic Test Data Generation", *IEEE Transactions on Software Engineering,* SE-17(9), pp.900-910, September 1991.

[DR88] T. Dogsa and I. Rozman, "CAMOTE - Computer Aided Module Testing and Design Environment", *Proc. of the IEEE Conference on Software Maintenance*, pp 404-408, 1988.

[Fra87] F. G. Frankl , *The Use of Data Flow Information for the selection and evaluation of software test data.* PhD Thesis, Department of Computer Science, New York University, October 1987.

[FRC81] K.F.Fischer, F. Raji and A. Chruscicki, "A Methodology for Retesting Modified Software", *Proc. Conf. National Telecommunications*, pp. B-6-3(1-6), New Orleans, November, 1981.

[FW85] F. G. Frankl and E.J. Weyuker, "Data Flow Testing Tool", Proc. Softfair II, San Francisco, CA, pp. 46-53, December. 1985.

[GL91] K.B. Gallagher and R. Lyle "Using Program Slicing in Software Maintenance", *IEEE Transactions on Software Engineering,* Vol SE-17(8), August 1991.

[Har93] M.J. Harrold, "Using Data Flow Analysis for Testing", Technical Report, Department of Computer Science, Clemson University, 1993.

[Het84] W.Hetzel, *The Complete Guide to Software Testing*, QED Information Sciences, Wellesley, Massachusetts, 1984.

[HM91] M.J. Harrold and Brian Malloy, "A Unified Interprocedural Program Representation for a Maintenance Environment", *Proc. of the IEEE Conf. Software Maintenance*, pp. 138-147, 1991.

[HMG93] M.J.Harrold, B.Malloy and G.Rothermel, "Constructing Program Dependence Graph using a Parser", Technical Report, Department of Computer Science, Clemson University, 1993.

[HR90] J. Hartmann and D.J. Robson, "Techniques for Selective Revalidation", *IEEE Software*, pp 31-38, January 1990.

[HS88] M.J. Harrold and M.L. Soffa, "An Incremental Approach to Unit Testing During Maintenance", *Proc. of the IEEE Conference on Software Maintenance*", pp 362-367, 1988.

[Leu95] H.K.N Leung, "Selective Regression Testing Assumptions and Fault Detecting Ability", *Information and Software Technology*, pp 531-537, Vol 37(10), 1995.

[LS92] J.W. Laski and W. Szermer, "Identification of Program Modifications and Its Application in Software Maintenance", *Proc.of the IEEE Conference of Software Maintenance*, pp. 282-290, November 1997.

[LW89] H.K.N. Leung and L. White, "Insights into Regression Testing" *Proc.of the IEEE Conference on Software Maintenance*, pp. 60-69, Miami, October 1989.

[LW90] H.K.N. Leung and L. White, "A Study of Integration Testing and Software Regression at the Integration Level", *Proc. of the IEEE Conference on Software Maintenance*, pp. 290-301, San Diego, November 1990.

[LW91] H.K.N. Leung and L. White, "A Cost Model to Compare Regression Test Strategies", *Proc. of IEEE Conference on Software Maintenance*, pp. 201-207, Italy, 1991.

[Mal91] J.C. Maldonado, "Critérios Potenciais Usos: uma Contribuição ao Teste Estrutural de Software", Tese de Doutorado, DCA/FEEC/Unicamp, Campinas, SP, Julho de 1991.

[Mey79] G.J. Meyers, *The Art of Software Testing*, Willey, New York, 1979.

[OW88] T.J. Ostrand and E.J. Weyker, "Using Dataflow Analysis for Regression Testing", *Proc. of The Sixth Annual Pacific Northwest Software Quality Conference*, pp. 233-247, September 1988.

[Pre92] R.S. Pressman, *Software Engineering: a Practitioner's Approach*, Third Edition, New York, McGraw-Hill, 1992.

[RH93a] G.Rothermel and M.J.Harrold, "A System Testing and Analaysis for C Programs", Technical Report, Department of Computer Science, Clemson University, 1993.

[RH93b] G.Rothermel and M.J.Harrold, "Using Data Flow Analysis for Testing", Technical Report, Department of Computer Science, Clemson University, 1993.

[RH93c] G.Rothermel and M.J.Harrold, "A system Testing and Analysis of C Programs", Technical Report, Department of Computer Science Clemson, University, 1993.

[RH93d] G.Rothermel and M.J.Harrold, "A Safe, Efficient Algorithm for Regression Test Selection ", *Proc of the IEEE Conference on Software Maintenance*, pp. 358-367, September 1993.

[RH94] G. Rothermel and M.J.Harrold, "Selecting Test and Indentifying Test Coverage Requirements for Modified Software", Technical Report, Department of Computer Science, Clemson University, 1994.

[RH96] G.Rothermel and M.J.Harrold, "Analyzing Regression Test Selection Techniques", *IEEE Transactions on Software Engineering*, Vol. SE-22(8), pp 529-551, August 1996.

[RO87] B. Raither and L. Osterweil, "TRICS: a Testing Tool for C", *Proc. First European Software Engineering Conf.* pp 254-262, France, September 1987.

[Rot96] G,Rothemel, *Efficient, Effective Regression Testing Using Safe Test Selection*, PhD Thesis, Department of Computer Science, Clemson University, May 1996.

[RW82] S. Rapps and E. J. Weyuker "Data Flow Analysis Techniques for Test Data Selection", *IEEE Proc. of Int. Conference of Software Engineering*, pp 272-278, Tokio Japan, April 1985.

[RW85] S. Rapps and E. J. Weyuker "Selecting Software Test Data Using Data Flow Information", *IEEE Transactions on Software Engineering*, Vol. SE-11(4), April 1985, pp 367-375.

[RW97] D.S. Rosenblum and E.J.Weyuker, "Using Coverage Information to Predict the Cost-Effectiveness of Regression Testing Strategies", *IEEE Transactions on Software Engineering*, Vol. SE-23(3), pp 498-516, March 1997.

[Sch87] N.F. Schneidewind, "The State of Software Maintenance", *IEEE Transactions on Software Engineering*, SE-13(13), pp. 303-310, March 1987.

[SK91] B. Sherlund and B. Korel, "Modification Oriented Software Testing", *Proc. of the Conference Quality Week 1991,* pp. 1-17, 1991.

[Stu80] H.G. Stuebing, "A Modern Facility for Software Production and Maintenance", Proc. *of the COMPSAC 80*, pp. 407-418, 1980.

[Stu84] H.G. Stuebing, "A Software Engineering Environment (SEE) for Weapon System Software", *IEEE Transactions on Software Engineering,* SE-10(4) pp. 384-397, July 1984.

[Ver97] S.R.Vergílio, "Critérios Restritos de Teste de Software: Uma Contribuição para Gerar Dados de Teste Mais Eficazes", Tese de Doutorado, DCA/FEEC/Unicamp, Campinas, SP, 1997.

[Vil97] P.R.S.Vilela, "Análise dos Critérios Potenciais Usos e sua Extensão para o Teste de Integração", Tese de Doutorado, DCA/FEEC/Unicamp, Campinas, SP, 1997 (em andamento).

[VMJ97] P.R.S.Vilela, J.C. Maldonado and M. Jino, "Program Graph Visualization", *Software Practice & Experience,* vol. 27, issue 11, November 1997.

[Wei84] M. Weiser, "Program Slicing", *IEEE Transactions on Software Engineering,* SE-10(4), pp. 352-357, July 1984.

[YK87] S.S.Yau and Z. Kishimoto, "A Method for Revalidating Modified Software Programs in the Software Maintenance Phase", *in Proc of the.COMPSAC 87,* pp. 272-277, 1987.

Apêndice A

Um Exemplo Completo – `moda.c`

Neste apêndice será mostrado um exemplo completo da execução da ferramenta **RePoKe-Tool**. Assim, esse apêndice foi dividido em 2 partes. Na primeira será mostrado o programa original e os principais arquivos gerados pela **POKE-TOOL** durante a execução do teste. Na segunda parte, será mostrada a ferramenta em execução e serão mostrados os arquivos gerados pela RePoKe-Tool.

A.1 Principais Arquivos Gerados pelo Teste do Programa `moda.c` na Ferramenta POKE-TOOL

O programa exemplo calcula e imprima a moda de um vetor de números inteiros, classificado em ordem crescente, recebido como parâmetro. Caso o vetor de entrada esteja vazio, o programa retornará valor –1 e se o vetor não estiver ordenado, valor –2. A execução normal do programa retornará valor 1.

Arquivo `testeprog.c`

```
#define ponta_de_prova(num) if(++printed_nodes % 10) fprintf(path,"
%2d ",num);\
else fprintf(path," %2d\n",num);

#include <stdio.h>

int moda(int v[],int n)
/*  1 */        {
                   FILE * path = fopen("moda/path.tes","a");
                   static int printed_nodes = 0;
/*  1 */           int i,valor_moda,t,freq_max,valor;
                   ponta_de_prova(1);
/*  1 */           if(n < 1)
/*  2 */           {
                      ponta_de_prova(2);
                      ponta_de_prova(15);
                      fclose(path);
/*  2 */              return -1;
/*  2 */           }
/*  3 */           else
/*  3 */           {
                      ponta_de_prova(3);
/*  3 */              i = 0;
/*  3 */              freq_max = 0;
/*  4 */              while(i < n)
/*  5 */              {
                         ponta_de_prova(4);
                         ponta_de_prova(5);
/*  5 */                 valor = v[i];
```

```
/*  5 */                    t=0;
/*  6 */                    while(valor==v[i] && i<n)
/*  7 */                    {
                               ponta_de_prova(6);
                               ponta_de_prova(7);
/*  7 */                       t++;
/*  7 */                       i++;
/*  7 */                    }
                           ponta_de_prova(6);
                           ponta_de_prova(8);
/*  8 */                   if(valor>v[i] && i<n)
/*  9 */                   {
                               ponta_de_prova(9);
                               ponta_de_prova(15);
                               fclose(path);
/*  9 */                       return -2;
/*  9 */                   }
/* 10 */                   else
/* 10 */                   {
                               ponta_de_prova(10);
/* 10 */                       if(t > freq_max)
/* 11 */                       {
                                   ponta_de_prova(11);
/* 11 */                           freq_max = t;
/* 11 */                           valor_moda = valor;
/* 11 */                       }
                               ponta_de_prova(12);
/* 12 */                   }
                           ponta_de_prova(13);
/* 13 */               }
                       ponta_de_prova(4);
                       ponta_de_prova(14);
/* 14 */               printf ("\nValor Moda: %d",valor_moda);
                       ponta_de_prova(15);
                       fclose(path);
/* 14 */               return (1);
/* 14 */           }
                   ponta_de_prova(15);
                   fclose(path);
/* 15 */       }
```

Arquivo nos_grf.tes

```
NO'S DO MODULO moda

  1   2   3   4   5   6   7   8   9  10
 11  12  13  14  15
```

Arquivo arcprim.tes

```
ARCOS PRIMITIVOS DO MODULO moda

  1) arco ( 1, 2)
  2) arco ( 4,14)
  3) arco ( 6, 7)
  4) arco ( 8, 9)
  5) arco (10,11)
  6) arco (10,12)
```

Arquivo puassoc.tes

```
ASSOCIACOES REQUERIDAS PELOS CRITERIOS TODOS POT-USOS E POT-USOS/DU


Associacoes requeridas pelo Grafo( 1)

1) <1,(4,14),{ v, n }>
2) <1,(10,12),{ v, n }>
3) <1,(13,4),{ v, n }>
4) <1,(10,11),{ v, n }>
5) <1,(8,9),{ v, n }>
6) <1,(7,6),{ v, n }>
7) <1,(6,7),{ v, n }>
8) <1,(1,2),{ v, n }>


Associacoes requeridas pelo Grafo( 3)

9) <3,(4,14),{ i, freq_max }>
10) <3,(10,12),{ i, freq_max }>
11) <3,(13,4),{ i, freq_max }>
12) <3,(13,4),{ i }>
13) <3,(10,11),{ i, freq_max }>
14) <3,(8,9),{ i, freq_max }>
15) <3,(7,6),{ freq_max }>
16) <3,(6,7),{ i, freq_max }>


Associacoes requeridas pelo Grafo( 5)

17) <5,(10,12),{ t, valor }>
18) <5,(4,14),{ t, valor }>
19) <5,(4,5),{ t, valor }>
20) <5,(10,11),{ t, valor }>
21) <5,(8,9),{ t, valor }>
22) <5,(7,6),{ valor }>
23) <5,(6,7),{ t, valor }>


Associacoes requeridas pelo Grafo( 7)

24) <7,(10,12),{ i, t }>
25) <7,(4,14),{ i, t }>
26) <7,(5,6),{ i }>
```

```
27) <7,(4,5),{ i, t }>
28) <7,(10,11),{ i, t }>
29) <7,(8,9),{ i, t }>
30) <7,(6,7),{ i, t }>


Associacoes requeridas pelo Grafo(11)

31) <11,(4,14),{ valor_moda, freq_max }>
32) <11,(10,12),{ valor_moda, freq_max }>
33) <11,(10,11),{ valor_moda, freq_max }>
34) <11,(8,9),{ valor_moda, freq_max }>
35) <11,(7,6),{ valor_moda, freq_max }>
36) <11,(6,7),{ valor_moda, freq_max }>
```

Arquivo pdupaths.tes

```
CAMINHOS REQUERIDOS PELO CRITERIO TODOS POT-DU-CAMINHOS


Caminhos requeridos pelo Grafo( 1)

 1) 1 3 4 14 15
 2) 1 3 4 5 6 8 10 12 13 4
 3) 1 3 4 5 6 8 10 11 12 13 4
 4) 1 3 4 5 6 8 9 15
 5) 1 3 4 5 6 7 6
 6) 1 2 15


Caminhos requeridos pelo Grafo( 3)

 7) 3 4 14 15
 8) 3 4 5 6 8 10 12 13 4
 9) 3 4 5 6 8 10 11 12 13 4
10) 3 4 5 6 8 9 15
11) 3 4 5 6 7 6


Caminhos requeridos pelo Grafo( 5)

12) 5 6 8 10 12 13 4 14 15
13) 5 6 8 10 12 13 4 5
14) 5 6 8 10 11 12 13 4 14 15
15) 5 6 8 10 11 12 13 4 5
16) 5 6 8 9 15
17) 5 6 7 6


Caminhos requeridos pelo Grafo( 7)

18) 7 6 8 10 12 13 4 14 15
19) 7 6 8 10 12 13 4 5 6
20) 7 6 8 10 11 12 13 4 14 15
21) 7 6 8 10 11 12 13 4 5 6
22) 7 6 8 9 15
```

```
23) 7 6 7


Caminhos requeridos pelo Grafo(11)

24) 11 12 13 4 14 15
25) 11 12 13 4 5 6 8 10 12
26) 11 12 13 4 5 6 8 10 11
27) 11 12 13 4 5 6 8 9 15
28) 11 12 13 4 5 6 7 6
```

Casos de Teste Aplicados

Arquivos keyboard*xx*.tes, path*xx*.tes e output*xx*.tes

#CT	**keyboard*xx*.tes**	**Path*xx*.tes**	**output*xx*.tes**
1	0	1 2 15	Retorno: -1
2	1 -1	1 3 4 5 6 7 6 8 10 11 12 13 4 14 15	Valor Moda: -1 Retorno: 1
3	10 1 1 1 4 4 4 4 8 8 9	1 3 4 5 6 7 6 7 6 7 6 8 10 11 12 13 4 5 6 7 6 7 6 7 6 7 6 8 10 11 12 13 4 5 6 7 6 8 10 12 13 4 5 6 7 6 8 10 12 13 4 14 15	Valor Moda: 4 Retorno: 1
4	3 2 1 0	1 3 4 5 6 7 6 8 9 15	Retorno: -2
5	5 1 1 2 -4 0	1 3 4 5 6 7 6 7 6 8 10 11 12 13 4 5 6 7 6 8 9 15	Retorno: -2

Arquivo nosoutput.tes

```
NOS DO CRITERIO TODOS-NOS nao executados:


Cobertura Total = 100.000000
```

Arquivo arcoutput.tes

```
ARCOS DO CRITERIO TODOS-ARCOS nao executados:


Cobertura Total = 100.000000
```

Arquivo puoutput.tes

```
ASSOCIACOES DO CRITERIO TODOS POT-USOS nao executadas:

<3,(4,14),{ i, freq_max }>
<3,(10,12),{ i, freq_max }>
<3,(13,4),{ i, freq_max }>
<3,(13,4),{ i }>
<3,(10,11),{ i, freq_max }>
<3,(8,9),{ i, freq_max }>
<5,(10,12),{ t, valor }>
<5,(4,14),{ t, valor }>
<5,(4,5),{ t, valor }>
<5,(10,11),{ t, valor }>
<5,(8,9),{ t, valor }>

Cobertura Total = 69.444444

Media da Cobertura dos Grafo(i) = 70.714283
```

Arquivo puduoutput.tes

```
ASSOCIACOES DO CRITERIO TODOS POT-USOS/DU nao executadas:
<1,(4,14),{ v, n }>
<1,(10,12),{ v, n }>
<1,(13,4),{ v, n }>
<1,(10,11),{ v, n }>
<1,(8,9),{ v, n }>
<3,(4,14),{ i, freq_max }>
<3,(10,12),{ i, freq_max }>
<3,(13,4),{ i, freq_max }>
<3,(13,4),{ i }>
<3,(10,11),{ i, freq_max }>
<3,(8,9),{ i, freq_max }>
<5,(10,12),{ t, valor }>
<5,(4,14),{ t, valor }>
<5,(4,5),{ t, valor }>
<5,(10,11),{ t, valor }>
<5,(8,9),{ t, valor }>
<11,(10,12),{ valor_moda, freq_max }>
<11,(10,11),{ valor_moda, freq_max }>
<11,(8,9),{ valor_moda, freq_max }>

Cobertura Total = 47.222222

Media da Cobertura dos Grafo(i) = 48.214283
```

Arquivo pduoutput.tes

```
POTENCIAIS-DU-CAMINHOS que nao foram executados:
Caminhos:
1 3 4 14 15
1 3 4 5 6 8 10 12 13 4
1 3 4 5 6 8 10 11 12 13 4
1 3 4 5 6 8 9 15
3 4 14 15
3 4 5 6 8 10 12 13 4
3 4 5 6 8 10 11 12 13 4
3 4 5 6 8 9 15
5 6 8 10 12 13 4 14 15
5 6 8 10 12 13 4 5
5 6 8 10 11 12 13 4 14 15
5 6 8 10 11 12 13 4 5
5 6 8 9 15
11 12 13 4 5 6 8 10 12
11 12 13 4 5 6 8 10 11
11 12 13 4 5 6 8 9 15

Cobertura Total = 42.857143

Media da Cobertura dos Grafo(i) = 42.000003
```

Elementos Requeridos Eliminados – Critério Todos os Arcos

Arquivo arcshis.tes

```
1
2
3
4
5
6
```

Elementos Requeridos Eliminados e Cobertura Incremental – Todos-Arcos

arcshis*XX*.tes					arcsdif*XX-YY*.tes				
1	**2**	**3**	**4**	**5**	**1-0**	**2-1**	**3-2**	**4-3**	**5-4**
1	2 3 5	2 3 5 6	3 4	3 4 5	1	2 3 5	6	4	

Elementos Requeridos Eliminados – Critério Todos os Nós

Arquivo `noshis.tes`

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Elementos Requeridos Eliminados e Cobertura Incremental – Todos-Nós

`NoshisXX.tes`					`NosdifXX-YY.tes`				
1	**2**	**3**	**4**	**5**	**1-0**	**2-1**	**3-2**	**4-3**	**5-4**
1	1	1	1	1	1	3		9	
2	3	3	3	3	2	4			
15	4	4	4	4	15	5			
	5	5	5	5		6			
	6	6	6	6		7			
	7	7	7	7		8			
	8	8	8	8		10			
	10	10	9	9		11			
	11	11	15	10		12			
	12	12		11		13			
	13	13		12		14			
	14	14		13					
	15	15		15					

Elementos Requeridos Eliminados – Critério Todos os Potenciais-Usos

Arquivo `puhis.tes`

1
2
3
4
5
6
7
8
15
16
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36

Elementos Requeridos Eliminados e Cobertura Incremental – Todos-PU

`puhisXX.tes`					`pudifXX-YY.tes`				
1	**2**	**3**	**4**	**5**	**1-0**	**2-1**	**3-2**	**4-3**	**5-4**
8	1	1	5	3	8	1	2	5	34
	3	2	6	4		3	24	6	
	4	3	7	5		4	26	7	
	6	4	15	6		6	27	15	
	7	6	16	7		7	30	16	
	15	7	22	15		15	32	22	
	16	15	23	16		16	33	23	
	22	16	29	22		22	35	29	
	23	22		23		23	36		
	25	23		26		25			
	28	24		27		28			
	31	25		28		31			
		26		29					
		27		30					
		28		34					
		30		35					
		31		36					
		32							
		33							
		35							
		36							

Elementos Requeridos Eliminados – Critério Todos os Potenciais-Usos

Arquivo `puduhis.tes`

6
7
8
15
16
22
23
24
25
26
27
28
29
30
31
35
36

Elementos Requeridos Eliminados e Cobertura Incremental – Todos-PUDU

`puduhis`*XX*`.tes`					`pududif`*XX*`-`*YY*`.tes`				
1	**2**	**3**	**4**	**5**	**1-0**	**2-1**	**3-2**	**4-3**	**5-4**
8	6	6	6	6	8	6	24	29	
	7	7	7	7		7	26		
	15	15	15	15		15	27		
	16	16	16	16		16	30		
	22	22	22	22		22	35		
	23	23	23	23		23	36		
	25	24	29	26		25			
	28	25		27		28			
	31	26		28		31			
		27		29					
		28		30					
		30		35					
		35		36					
		36							

Elementos Requeridos Eliminados – Critério Todos os Potenciais-Usos

Arquivo `pduhis.tes`

5
6
11
17
18
19
20
21
22
23
24
28

Elementos Requeridos Eliminados e Cobertura Incremental – Todos-PDU

`pduhisXX.tes`					`pdudifXX-YY.tes`				
1	**2**	**3**	**4**	**5**	**1-0**	**2-1**	**3-2**	**4-3**	**5-4**
6	5 11 17 20 24	5 11 17 18 19 21 23 28	5 11 17 22	5 11 17 21 22 23 28	6	5 11 17 20 24	18 19 21 23 28	22	

A.2 Execução da Ferramenta RePoKe-Tool com a versão modificada de `moda.c` e os principais arquivos gerados

A modificação proposta para o programa exemplo `moda.c` consiste da correção do programa de forma a torná-lo apto a imprimir uma mensagem de erro quando existir uma distribuição plurimodal no vetor de entrada, e nesse caso, retornar valor 0. A Execução do teste de regressão foi dividia em duas etapas. Na primeira, será executada a ferramenta RePoKe-Tool. Terminada a execução serão mostrados os principais arquivos gerados e, em seguida, mostra o procedimento de execução do teste de regressão (usando o ambiente POKE-TOOL reduzido), apenas para o critério Todos-Potenciais-Usos. Nesse teste de regressão serão reaplicados os testes selecionados e gerado mais um caso de teste adicional.

Arquivo `testeprog.c`

```
#define ponta_de_prova(num) if(++printed_nodes % 10) fprintf(path,"
%2d ",num);\
else fprintf(path," %2d\n",num);

#include <stdio.h>

int moda(int v[],int n)
/*  1 */        {
                   FILE * path = fopen("moda/path.tes","a");
                   static int printed_nodes = 0;
/*  1 */          int i,valor_moda,t,freq_max,valor,flag_moda;
                   ponta_de_prova(1);
/*  1 */           if(n < 1)
/*  2 */           {
                      ponta_de_prova(2);
                      ponta_de_prova(21);
                      fclose(path);
/*  2 */              return -1;
/*  2 */           }
/*  3 */           else
/*  3 */           {
                      ponta_de_prova(3);
/*  3 */              i = 0;
/*  3 */              freq_max = 0;
/*  3 */              flag_moda = 1;
/*  4 */              while(i < n)
/*  5 */              {
                         ponta_de_prova(4);
                         ponta_de_prova(5);
/*  5 */                 valor = v[i];
/*  5 */                 t=0;
/*  6 */                 while(valor==v[i] && i<n)
/*  7 */                 {
                            ponta_de_prova(6);
                            ponta_de_prova(7);
/*  7 */                    t++;
/*  7 */                    i++;
/*  7 */                 }
                         ponta_de_prova(6);
```

```
                        ponta_de_prova(8);
/*   8 */               if(valor>v[i] && i<n)
/*   9 */               {
                           ponta_de_prova(9);
                           ponta_de_prova(21);
                           fclose(path);
/*   9 */                  return -2;
/*   9 */               }
/*  10 */               else
/*  10 */               {
                           ponta_de_prova(10);
/*  10 */                  if(t > freq_max)
/*  11 */                  {
                              ponta_de_prova(11);
/*  11 */                     freq_max = t;
/*  11 */                     valor_moda = valor;
/*  11 */                     flag_moda = 1;
/*  11 */                  }
/*  12 */                  else
/*  12 */                  {
                              ponta_de_prova(12);
/*  12 */                     if(t==freq_max)
/*  13 */                     {
                                 ponta_de_prova(13);
/*  13 */                        flag_moda = 0;
/*  13 */                     }
                              ponta_de_prova(14);
/*  14 */                  }
                           ponta_de_prova(15);
/*  15 */               }
                        ponta_de_prova(16);
/*  16 */            }
                     ponta_de_prova(4);
                     ponta_de_prova(17);
/*  17 */            if(!flag_moda)
/*  18 */            {
                        ponta_de_prova(18);
/*  18 */               printf ("\nDistribuicao Plurimodal");
/*  18 */            }
/*  19 */            else
/*  19 */            {
                        ponta_de_prova(19);
/*  19 */               printf("\nValor Moda:%d",valor_moda);
/*  19 */            }
                     ponta_de_prova(20);
                     ponta_de_prova(21);
                     fclose(path);
/*  20 */            return (flag_moda);
/*  20 */         }
                  ponta_de_prova(21);
                  fclose(path);
/*  21 */      }
```

1ª Etapa: Execução da RePoKe-Tool

```
afrodite% repoketool moda.c moda
RePoKe-TOOL - Ferramenta de Apoio ao Teste de Regressao p/ambiente
Poke-Tool

* *
* * Inicializacao - AGUARDE
* *
* *
* * Instrumentado a Unidade Modificada * *
* *
poketool - Analisador Estatico para Teste de Programas escritos em
Linguagem C
Arquivo em teste: moda.c
* * Carregando Tabela de transicao lexica... * *
* * Carregando Tabela de palavras chaves... * *
* * Carregando Parser especifico da linguagem fonte... * *
* * Fazendo a analise sintatica do codigo fonte... * *
* * Nucleo POKETOOL foi bem sucedido * *

Gerando os descritores para a funcao moda ...
* * Calculando os arcos primitivos... * *
* * Gerando descritores... * *
* * Geracao de Descritores da POKETOOL foi bem sucedida * *
* * Gerados os descritores da funcao moda * *

* *
* * Restaurando as informacoes da ultima sessao de teste
* *

Deseja Visualizar as diferencas entre os arquivos da versoes de
moda.c?[n]
s
* *
* * Modificacoes encontradas na funcao pelo utilitario Diff
* * e armazemadas no arquivo diferencas
* *
4c4
< {  int i,valor_moda,t,freq_max,valor;
---
> {  int i,valor_moda,t,freq_max,valor,flag_moda;
11a12
>       flag_moda = 1;
24a26
>             flag_moda = 1;
25a28,29
>          else if (t==freq_max)
>                flag_moda = 0;
27,28c31,35
<       printf ("\nValor Moda: %d",valor_moda);
<       return (1);
---
>       if (!flag_moda)
>          printf ("\nDistribuicao Plurimodal");
>       else
>          printf ("\nValor Moda: %d",valor_moda);
>       return (flag_moda);
```

```
** Identifique as Modificacoes realizadas no modulo moda.c **

Houve modificacoes no Fluxo de Controle..?<S/N>: s

     Modificacoes no Fluxo de Controle

ATENCAO: As modificacoes devem ser informadas em ordem crescente de
nos

Qual o tipo de modificacao realizada no GFC?
<C> para contracao
<E> para expansao
Sua Escolha: e

                  <TIPOS DE EXPANSAO>

            [1] Insecao de else em If-Then
            [2] de No' para If_then
            [3] de No' para If_then_else
            [4] de No' para For
            [5] de No' para While
            [6] de No' para Do_while
            [7] Combinacao dos modelos Basicos

     Entre com o tipo de EXPANSAO..:1

Indique o arco else do GFC onde sera' inserida uma estrutura (Ni
Nf)..:10 12

<Estruturas de Programacao que poderao ser inseridas no If-Else>

            [1] No simples
            [2] If_then
            [3] If_then_else
            [4] For
            [5] While
            [6] Do_while
            [7] Combinacao

     Sua opcao..:2

No' 12 do programa modificado nao possui variaveis

Ajuste das Definicoes de variaveis do no 13 do programa modificado
Indique a situacao das variaveis deste no
<P> para todas preservadas
<N> para todas novas
<M> para modificadas (Combinacao de Modificadas, Novas e Preservadas)
Sua opcao...:n

No' 14 do programa modificado nao possui variaveis

Deseja continuar modificando o Fluxo de Controle <S>im <N>ao..: s

Qual o tipo de modificacao realizada no GFC?
<C> para contracao
<E> para expansao
Sua Escolha: e
```

```
                    <TIPOS DE EXPANSAO>

                [1] Insecao de else em If-Then
                [2] de No' para If_then
                [3] de No' para If_then_else
                [4] de No' para For
                [5] de No' para While
                [6] de No' para Do_while
                [7] Combinacao dos modelos Basicos

      Entre com o tipo de EXPANSAO..:3

Indique o numero do no do GFC que foi expandido:14

No' 17 do programa modificado nao possui variaveis
No' 18 do programa modificado nao possui variaveis
No' 19 do programa modificado nao possui variaveis
No' 20 do programa modificado nao possui variaveis

Deseja continuar modificando o Fluxo de Controle <S>im <N>ao..: n


Houve modificacoes exclusivamente nas Definicoes de
variaveis..?<S/N>: s



      Modificacoes exclusivamente em Definices de Variaveis

Entre com o No' que possui modificacao nas definicoes de
variaveis..:11
Indique o numero do no do programa original que se relaciona c/
as modificacoes nas definicoes de variaveis do no' 11 ...:11

Indique a situacao das variaveis do no' 11
<N> para todas novas
<M> para modificadas (Combinacao de Modificadas, Novas e/ou
Preservadas)
Sua opcao...:m


Indique a situacao de cada variavel do no modificado
<N> para variavel nova
<P> para variavel preservada
<M> para variavel modificada

Informe a situacao da variavel (valor_moda)...:p
Informe a situacao da variavel (freq_max)...:p
Informe a situacao da variavel (flag_moda)...:n

Deseja continuar modificando exclusivamente o Fluxo de Dados <S>im
<N>ao..: n

Houve modificacoes em predicado(s)..?<S/N>: n

Houve modificacoes nos usos de variaveis..?<S/N>: s
```

```
     Modificacoes nos Usos

Entre com o numero do No' onde houve modificacao de Uso
que nao implicam em definicao ...: 19
Entre com a situacao do uso que foi modificado:
<M> para Modificado
<R> para Removido
<N> para novo
Sua opcao...:m

Deseja continuar modificando os Usos dos no's <S>im <N>ao..: s

Entre com o numero do No' onde houve modificacao de Uso
que nao implicam em definicao ...: 20
Entre com a situacao do uso que foi modificado:
<M> para Modificado
<R> para Removido
<N> para novo
Sua opcao...:n

Deseja continuar modificando os Usos dos no's <S>im <N>ao..: n

Houve modificacoes que nao afetam Fluxo de controle e Fluxo de
dados.?<S/N>: s



     Modificacoes nos No's NAO influenciados


Entre com o numero do No' onde houve modificacao que NAO possui
influencia em Fluxo de controle e Fluxo de dados...: 18
Entre com a situacao atual deste no'
<M> para Modificado
<R> para Removido
<N> para novo
Sua opcao...:n

     Deseja continuar modificando os no's sem influencia em FC/FD
<S>im <N>ao..: n


Existem CT's no programa original <S>im <N>ao..:s


Qual a quantidade de casos de teste usados no programa original:5

**
** Aguarde a Execucao da RePoKe-TOOL.
**
** Sessao Reduzida da POKE-TOOL gerada com Sucesso
**
** Atencao: Nao esqueca de recompilar o programa testeprog.c para
** exucutar e avaliar na POKE-TOOL os casos de teste selecionados
**
** Fim! Execucao da RePoKe-TOOL foi bem sucedida
```

Arquivo `nos_grf.tes` (gerado pela Análise Estática da Versão Modificada)

```
NO'S DO MODULO moda

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

Arquivo `nosret.tes`

```
Nos que devem ser reavaliados:

  11   12   13   14   17   18   19   20
```

Arquivo `nosnret.tes`

```
Nos que nao precisam ser reavaliados:

   1    2    3    4    5    6    7    8    9   10   15   16   21
```

Arquivo `arcprim.tes` (gerado pela Análise Estática da Versão Modificada)

```
ARCOS PRIMITIVOS DO MODULO moda

 1) arco ( 1, 2)
 2) arco ( 6, 7)
 3) arco ( 8, 9)
 4) arco (10,11)
 5) arco (12,13)
 6) arco (12,14)
 7) arco (17,18)
 8) arco (17,19)
```

Arquivo `arcprimr.tes`

```
   ARCOS PRIMITIVOS

      Arcos que podem ser reavaliados:

 2) arco (   6    7) (   3)
 3) arco (   8    9) (   4)
 4) arco (  10   11) (   5)
 5) arco (  12   13) (   6)
 6) arco (  12   14) (   6)
 7) arco (  17   18) (   2)
 8) arco (  17   19) (   2)
```

Arquivo `arcprimn.tes`

```
   ARCOS PRIMITIVOS

      Arcos primitivos que nao precisam ser reavaliados:

 1) arco (   1    2) (   1)
```

Arquivo puassoc.tes (gerado pela Análise Estática da Versão Modificada)

```
ASSOCIACOES REQUERIDAS PELOS CRITERIOS TODOS POT-USOS E POT-USOS/DU


Associacoes requeridas pelo Grafo( 1)

1) <1,(17,19),{ v, n }>
2) <1,(17,18),{ v, n }>
3) <1,(12,14),{ v, n }>
4) <1,(12,13),{ v, n }>
5) <1,(16,4),{ v, n }>
6) <1,(10,11),{ v, n }>
7) <1,(8,9),{ v, n }>
8) <1,(7,6),{ v, n }>
9) <1,(6,7),{ v, n }>
10) <1,(1,2),{ v, n }>


Associacoes requeridas pelo Grafo( 3)

11) <3,(17,19),{ i, freq_max, flag_moda }>
12) <3,(17,18),{ i, freq_max, flag_moda }>
13) <3,(12,14),{ i, freq_max, flag_moda }>
14) <3,(12,13),{ i, freq_max, flag_moda }>
15) <3,(16,4),{ i, freq_max, flag_moda }>
16) <3,(16,4),{ i, freq_max }>
17) <3,(16,4),{ i }>
18) <3,(10,11),{ i, freq_max, flag_moda }>
19) <3,(8,9),{ i, freq_max, flag_moda }>
20) <3,(7,6),{ freq_max, flag_moda }>
21) <3,(6,7),{ i, freq_max, flag_moda }>


Associacoes requeridas pelo Grafo( 5)

22) <5,(12,14),{ t, valor }>
23) <5,(12,13),{ t, valor }>
24) <5,(17,19),{ t, valor }>
25) <5,(17,18),{ t, valor }>
26) <5,(4,5),{ t, valor }>
27) <5,(10,11),{ t, valor }>
28) <5,(8,9),{ t, valor }>
29) <5,(7,6),{ valor }>
30) <5,(6,7),{ t, valor }>


Associacoes requeridas pelo Grafo( 7)

31) <7,(12,14),{ i, t }>
32) <7,(12,13),{ i, t }>
33) <7,(17,19),{ i, t }>
34) <7,(17,18),{ i, t }>
35) <7,(5,6),{ i }>
36) <7,(4,5),{ i, t }>
37) <7,(10,11),{ i, t }>
```

```
38) <7,(8,9),{ i, t }>
39) <7,(6,7),{ i, t }>


Associacoes requeridas pelo Grafo(11)

40) <11,(17,19),{ valor_moda, freq_max, flag_moda }>
41) <11,(17,18),{ valor_moda, freq_max, flag_moda }>
42) <11,(12,14),{ valor_moda, freq_max, flag_moda }>
43) <11,(14,15),{ valor_moda, freq_max, flag_moda }>
44) <11,(14,15),{ valor_moda, freq_max }>
45) <11,(12,13),{ valor_moda, freq_max, flag_moda }>
46) <11,(10,11),{ valor_moda, freq_max, flag_moda }>
47) <11,(8,9),{ valor_moda, freq_max, flag_moda }>
48) <11,(7,6),{ valor_moda, freq_max, flag_moda }>
49) <11,(6,7),{ valor_moda, freq_max, flag_moda }>


Associacoes requeridas pelo Grafo(13)

50) <13,(17,19),{ flag_moda }>
51) <13,(17,18),{ flag_moda }>
52) <13,(12,14),{ flag_moda }>
53) <13,(12,13),{ flag_moda }>
54) <13,(10,11),{ flag_moda }>
55) <13,(8,9),{ flag_moda }>
56) <13,(7,6),{ flag_moda }>
57) <13,(6,7),{ flag_moda }>
```

Arquivo puassocr.tes

```
ASSOCIACOES REQUERIDAS PELOS CRITERIOS TODOS POT-USOS E POT-USOS/DU

      Associacoes que devem ser reavaliados:


Associacoes requeridas pelo Grafo(   1)

1) <1,(17,19),{ v, n }> (   1)
2) <1,(17,18),{ v, n }> (   1)
3) <1,(12,14),{ v, n }> (   2)
4) <1,(12,13),{ v, n }> (   2)
5) <1,(16,4),{ v, n }> (   3)


Associacoes requeridas pelo Grafo(   3)

11) <3,(17,19),{ i, freq_max, flag_moda }> (   9)
12) <3,(17,18),{ i, freq_max, flag_moda }> (   9)
13) <3,(12,14),{ i, freq_max, flag_moda }> (  10)
14) <3,(12,13),{ i, freq_max, flag_moda }> (  10)
15) <3,(16,4),{ i, freq_max, flag_moda }> (  11  12)
16) <3,(16,4),{ i, freq_max }> (  11)
17) <3,(16,4),{ i }> (  11  12)
18) <3,(10,11),{ i, freq_max, flag_moda }> (  13)
```

```
Associacoes requeridas pelo Grafo(   5)

22) <5,(12,14),{ t, valor }> (  17)
23) <5,(12,13),{ t, valor }> (  17)
24) <5,(17,19),{ t, valor }> (  18)
25) <5,(17,18),{ t, valor }> (  18)
26) <5,(4,5),{ t, valor }> (  19)


Associacoes requeridas pelo Grafo(   7)

31) <7,(12,14),{ i, t }> (  24)
32) <7,(12,13),{ i, t }> (  24)
33) <7,(17,19),{ i, t }> (  25)
34) <7,(17,18),{ i, t }> (  25)
35) <7,(5,6),{ i }> (  26)
36) <7,(4,5),{ i, t }> (  27)


Associacoes requeridas pelo Grafo(  11)

40) <11,(17,19),{ valor_moda, freq_max, flag_moda }> (  31)
41) <11,(17,18),{ valor_moda, freq_max, flag_moda }> (  31)
42) <11,(12,14),{ valor_moda, freq_max, flag_moda }> (  32)
43) <11,(14,15),{ valor_moda, freq_max, flag_moda }> (  32)
44) <11,(14,15),{ valor_moda, freq_max }> (  32)
45) <11,(12,13),{ valor_moda, freq_max, flag_moda }> (  32)
46) <11,(10,11),{ valor_moda, freq_max, flag_moda }> (  33)
47) <11,(8,9),{ valor_moda, freq_max, flag_moda }> (  34)
48) <11,(7,6),{ valor_moda, freq_max, flag_moda }> (  35)
49) <11,(6,7),{ valor_moda, freq_max, flag_moda }> (  36)


Associacoes requeridas pelo Grafo(  13)

50) <13,(17,19),{ flag_moda }> ( N )
51) <13,(17,18),{ flag_moda }> ( N )
52) <13,(12,14),{ flag_moda }> ( N )
53) <13,(12,13),{ flag_moda }> ( N )
54) <13,(10,11),{ flag_moda }> ( N )
55) <13,(8,9),{ flag_moda }> ( N )
56) <13,(7,6),{ flag_moda }> ( N )
57) <13,(6,7),{ flag_moda }> ( N )
```

Arquivo puassocn.tes

```
ASSOCIACOES REQUERIDAS PELOS CRITERIOS TODOS POT-USOS E POT-USOS/DU

      Associacoes que nao precisam ser reavaliados:


Associacoes requeridas pelo Grafo(   1)

6) <1,(10,11),{ v, n }> (   4)
7) <1,(8,9),{ v, n }> (   5)
8) <1,(7,6),{ v, n }> (   6)
9) <1,(6,7),{ v, n }> (   7)
10) <1,(1,2),{ v, n }> (   8)
```

```
Associacoes requeridas pelo Grafo(   3)

19) <3,(8,9),{ i, freq_max, flag_moda }> (  14)
20) <3,(7,6),{ freq_max, flag_moda }> (  15)
21) <3,(6,7),{ i, freq_max, flag_moda }> (  16)


Associacoes requeridas pelo Grafo(   5)

27) <5,(10,11),{ t, valor }> (  20)
28) <5,(8,9),{ t, valor }> (  21)
29) <5,(7,6),{ valor }> (  22)
30) <5,(6,7),{ t, valor }> (  23)



Associacoes requeridas pelo Grafo(   7)

37) <7,(10,11),{ i, t }> (  28)
38) <7,(8,9),{ i, t }> (  29)
39) <7,(6,7),{ i, t }> (  30)
```

Arquivo `pdupaths.tes` (gerado pela Análise Estática da Versão Modificada)

```
CAMINHOS REQUERIDOS PELO CRITERIO TODOS POT-DU-CAMINHOS


Caminhos requeridos pelo Grafo( 1)

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


Caminhos requeridos pelo Grafo( 3)

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


Caminhos requeridos pelo Grafo( 5)

16) 5 6 8 10 12 14 15 16 4 17 19 20 21
```

```
17) 5 6 8 10 12 14 15 16 4 17 18 20 21
18) 5 6 8 10 12 14 15 16 4 5
19) 5 6 8 10 12 13 14 15 16 4 17 19 20 21
20) 5 6 8 10 12 13 14 15 16 4 17 18 20 21
21) 5 6 8 10 12 13 14 15 16 4 5
22) 5 6 8 10 11 15 16 4 17 19 20 21
23) 5 6 8 10 11 15 16 4 17 18 20 21
24) 5 6 8 10 11 15 16 4 5
25) 5 6 8 9 21
26) 5 6 7 6


Caminhos requeridos pelo Grafo( 7)

27) 7 6 8 10 12 14 15 16 4 17 19 20 21
28) 7 6 8 10 12 14 15 16 4 17 18 20 21
29) 7 6 8 10 12 14 15 16 4 5 6
30) 7 6 8 10 12 13 14 15 16 4 17 19 20 21
31) 7 6 8 10 12 13 14 15 16 4 17 18 20 21
32) 7 6 8 10 12 13 14 15 16 4 5 6
33) 7 6 8 10 11 15 16 4 17 19 20 21
34) 7 6 8 10 11 15 16 4 17 18 20 21
35) 7 6 8 10 11 15 16 4 5 6
36) 7 6 8 9 21
37) 7 6 7


Caminhos requeridos pelo Grafo(11)

38) 11 15 16 4 17 19 20 21
39) 11 15 16 4 17 18 20 21
40) 11 15 16 4 5 6 8 10 12 14 15
41) 11 15 16 4 5 6 8 10 12 13 14 15
42) 11 15 16 4 5 6 8 10 11
43) 11 15 16 4 5 6 8 9 21
44) 11 15 16 4 5 6 7 6


Caminhos requeridos pelo Grafo(13)

45) 13 14 15 16 4 17 19 20 21
46) 13 14 15 16 4 17 18 20 21
47) 13 14 15 16 4 5 6 8 10 12 14
48) 13 14 15 16 4 5 6 8 10 12 13
49) 13 14 15 16 4 5 6 8 10 11
50) 13 14 15 16 4 5 6 8 9 21
51) 13 14 15 16 4 5 6 7 6
```

Arquivo pdupathr.tes

```
CAMINHOS REQUERIDOS PELO CRITERIO TODOS POT-DU-CAMINHOS

      Potenciais-DU-Caminhos que devem ser reavaliados:


Caminhos requeridos pelo Grafo(   1)

  1)   1   3   4  17  19  20  21 (   1)
  2)   1   3   4  17  18  20  21 (   1)
  3)   1   3   4   5   6   8  10  12  14  15  16   4 (   2)
  4)   1   3   4   5   6   8  10  12  13  14  15  16   4 (   2)


Caminhos requeridos pelo Grafo(   3)

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


Caminhos requeridos pelo Grafo(   5)

 16)   5   6   8  10  12  14  15  16   4  17  19  20  21 (  12)
 17)   5   6   8  10  12  14  15  16   4  17  18  20  21 (  12)
 18)   5   6   8  10  12  14  15  16   4   5 (  13)
 19)   5   6   8  10  12  13  14  15  16   4  17  19  20  21 (  12)
 20)   5   6   8  10  12  13  14  15  16   4  17  18  20  21 (  12)
 21)   5   6   8  10  12  13  14  15  16   4   5 (  13)
 22)   5   6   8  10  11  15  16   4  17  19  20  21 (  14)
 23)   5   6   8  10  11  15  16   4  17  18  20  21 (  14)


Caminhos requeridos pelo Grafo(   7)

 27)   7   6   8  10  12  14  15  16   4  17  19  20  21 (  18)
 28)   7   6   8  10  12  14  15  16   4  17  18  20  21 (  18)
 29)   7   6   8  10  12  14  15  16   4   5   6 (  19)
 30)   7   6   8  10  12  13  14  15  16   4  17  19  20  21 (  18)
 31)   7   6   8  10  12  13  14  15  16   4  17  18  20  21 (  18)
 32)   7   6   8  10  12  13  14  15  16   4   5   6 (  19)
 33)   7   6   8  10  11  15  16   4  17  19  20  21 (  20)
 34)   7   6   8  10  11  15  16   4  17  18  20  21 (  20)


Caminhos requeridos pelo Grafo(  11)

 38)  11  15  16   4  17  19  20  21 (  24)
 39)  11  15  16   4  17  18  20  21 (  24)
 40)  11  15  16   4   5   6   8  10  12  14  15 (  25)
 41)  11  15  16   4   5   6   8  10  12  13  14  15 (  25)
 42)  11  15  16   4   5   6   8  10  11 (  26)
 43)  11  15  16   4   5   6   8   9  21 (  27)
 44)  11  15  16   4   5   6   7   6 (  28)
```

```
Caminhos requeridos pelo Grafo(  13)

  45)  13  14  15  16   4  17  19  20  21 ( N )
  46)  13  14  15  16   4  17  18  20  21 ( N )
  47)  13  14  15  16   4   5   6   8  10  12  14 ( N )
  48)  13  14  15  16   4   5   6   8  10  12  13 ( N )
  49)  13  14  15  16   4   5   6   8  10  11 ( N )
  50)  13  14  15  16   4   5   6   8   9  21 ( N )
  51)  13  14  15  16   4   5   6   7   6 ( N )
```

Arquivo pdupathn.tes

```
CAMINHOS REQUERIDOS PELO CRITERIO TODOS POT-DU-CAMINHOS

      Potenciais-DU-Caminhos que nao precisam ser reavaliados:


Caminhos requeridos pelo Grafo(   1)

   5)   1   3   4   5   6   8  10  11  15  16   4 (   3)
   6)   1   3   4   5   6   8   9  21 (   4)
   7)   1   3   4   5   6   7   6 (   5)
   8)   1   2  21 (   6)


Caminhos requeridos pelo Grafo(   3)

  14)   3   4   5   6   8   9  21 (  10)
  15)   3   4   5   6   7   6 (  11)


Caminhos requeridos pelo Grafo(   5)

  24)   5   6   8  10  11  15  16   4   5 (  15)
  25)   5   6   8   9  21 (  16)
  26)   5   6   7   6 (  17)


Caminhos requeridos pelo Grafo(   7)

  35)   7   6   8  10  11  15  16   4   5   6 (  21)
  36)   7   6   8   9  21 (  22)
  37)   7   6   7 (  23)
```

Classificação dos Caso de Teste – Critério Todos-Nós

Arquivo ctreanos.tes

```
CASOS DE TESTE REAPLICAVEIS para todos-nos
2   3
```

Arquivo ctreunos.tes

```
CASOS DE TESTE REUTILIZAVEIS para todos-nos
1   4   5
```

Arquivo ctremnos.tes

```
Vazio
```

Classificação dos Caso de Teste – Critério Todos-Arcos

Arquivo ctreaarc.tes

```
CASOS DE TESTE REAPLICAVEIS para todos-arcs
2   3   4   5
```

Arquivo ctreuarc.tes

```
CASOS DE TESTE REUTILIZAVEIS para todos-arcs
1
```

Arquivo ctremarc.tes

```
Vazio
```

Classificação dos Caso de Teste – Critério Todos-PU

Arquivo ctreapu.tes

```
CASOS DE TESTE REAPLICAVEIS para todos-pu
2   3   5
```

Arquivo ctreupu.tes

```
CASOS DE TESTE REUTILIZAVEIS para todos-pu
1   4
```

Arquivo ctrempu.tes

```
Vazio
```

Classificação dos Caso de Teste – Critério Todos-PUDU

Arquivo ctreapudu.tes

```
 CASOS DE TESTE REAPLICAVEIS para todos-pudu
 2    3    5
```

Arquivo ctreupudu.tes

```
CASOS DE TESTE REUTILIZAVEIS para todos-pudu
 1    4
```

Arquivo ctrempudu.tes

```
Vazio
```

Classificação dos Caso de Teste – Critério Todos-PDU

Arquivo ctreapdu.tes

```
 CASOS DE TESTE REAPLICAVEIS para todos-pdu
 2    3    5
```

Arquivo ctreupdu.tes

```
CASOS DE TESTE REUTILIZAVEIS para todos-pdu
 1    4
```

Arquivo ctrempdu.tes

```
Vazio
```

Arquivo des_pu.tes (Contém os descritores que deverão ser reavaliados e os que não precisam ser eliminados mas seu equivalente não foi eliminado no teste original)

```
DESCRITORES PARA O CRITERIO TODOS POT-USOS

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


Descritores para o Grafo( 1)

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

1) N* 1 Nnv* 17 [ Nnv* 17 ]* 19
Nnv = 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

2) N* 1 Nnv* 17 [ Nnv* 17 ]* 18
Nnv = 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

3) N* 1 Nnv* 12 [ Nnv* 12 ]* 14
Nnv = 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

4) N* 1 Nnv* 12 [ Nnv* 12 ]* 13
Nnv = 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
```

```
5) N* 1 Nnv* 16 [ Nnv* 16 ]* 4
Nnv = 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21


Descritores para o Grafo( 3)

Ni = 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
Nt = 4 6 21

11) N* 3 Nnv* 17 [ Nnv* 17 ]* 19
Nnv = 4 5 6 8 9 10 12 14 15 16 17 18 19 20 21

12) N* 3 Nnv* 17 [ Nnv* 17 ]* 18
Nnv = 4 5 6 8 9 10 12 14 15 16 17 18 19 20 21

13) N* 3 Nnv* 12 [ Nnv* 12 ]* 14
Nnv = 4 5 6 8 9 10 12 14 15 16 17 18 19 20 21

14) N* 3 Nnv* 12 [ Nnv* 12 ]* 13
Nnv = 4 5 6 8 9 10 12 14 15 16 17 18 19 20 21

15) N* 3 Nnv* 16 [ Nnv* 16 ]* 4
Nnv = 4 5 6 8 9 10 12 14 15 16 17 18 19 20 21

16) N* 3 Nnv* 16 [ Nnv* 16 ]* 4
Nnv = 4 5 6 8 9 10 12 13 14 15 16 17 18 19 20 21

17) N* 3 Nnv* 16 [ Nnv* 16 ]* 4
Nnv = 4 5 6 8 9 10 11 12 13 14 15 16 17 18 19 20 21

18) N* 3 Nnv* 10 [ Nnv* 10 ]* 11
Nnv = 4 5 6 8 9 10 12 14 15 16 17 18 19 20 21

19) N* 3 Nnv* 8 [ Nnv* 8 ]* 9
Nnv = 4 5 6 8 9 10 12 14 15 16 17 18 19 20 21


Descritores para o Grafo( 5)

Ni = 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
Nt = 5 6 21

22) N* 5 Nnv* 12 [ Nnv* 12 ]* 14
Nnv = 4 6 8 9 10 11 12 13 14 15 16 17 18 19 20 21

23) N* 5 Nnv* 12 [ Nnv* 12 ]* 13
Nnv = 4 6 8 9 10 11 12 13 14 15 16 17 18 19 20 21

24) N* 5 Nnv* 17 [ Nnv* 17 ]* 19
Nnv = 4 6 8 9 10 11 12 13 14 15 16 17 18 19 20 21

25) N* 5 Nnv* 17 [ Nnv* 17 ]* 18
Nnv = 4 6 8 9 10 11 12 13 14 15 16 17 18 19 20 21

26) N* 5 Nnv* 4 [ Nnv* 4 ]* 5
```

```
Nnv = 4 6 8 9 10 11 12 13 14 15 16 17 18 19 20 21

27) N* 5 Nnv* 10 [ Nnv* 10 ]* 11
Nnv = 4 6 8 9 10 11 12 13 14 15 16 17 18 19 20 21

28) N* 5 Nnv* 8 [ Nnv* 8 ]* 9
Nnv = 4 6 8 9 10 11 12 13 14 15 16 17 18 19 20 21


Descritores para o Grafo( 7)

Ni = 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
Nt = 6 7 21

31) N* 7 Nnv* 12 [ Nnv* 12 ]* 14
Nnv = 4 6 8 9 10 11 12 13 14 15 16 17 18 19 20 21

32) N* 7 Nnv* 12 [ Nnv* 12 ]* 13
Nnv = 4 6 8 9 10 11 12 13 14 15 16 17 18 19 20 21

33) N* 7 Nnv* 17 [ Nnv* 17 ]* 19
Nnv = 4 6 8 9 10 11 12 13 14 15 16 17 18 19 20 21

34) N* 7 Nnv* 17 [ Nnv* 17 ]* 18
Nnv = 4 6 8 9 10 11 12 13 14 15 16 17 18 19 20 21

35) N* 7 Nnv* 5 [ Nnv* 5 ]* 6
Nnv = 4 5 6 8 9 10 11 12 13 14 15 16 17 18 19 20 21

36) N* 7 Nnv* 4 [ Nnv* 4 ]* 5
Nnv = 4 6 8 9 10 11 12 13 14 15 16 17 18 19 20 21


Descritores para o Grafo(11)

Ni = 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
Nt = 6 11 15 21

40) N* 11 Nnv* 17 [ Nnv* 17 ]* 19
Nnv = 4 5 6 7 8 9 10 12 14 15 16 17 18 19 20 21

41) N* 11 Nnv* 17 [ Nnv* 17 ]* 18
Nnv = 4 5 6 7 8 9 10 12 14 15 16 17 18 19 20 21

42) N* 11 Nnv* 12 [ Nnv* 12 ]* 14
Nnv = 4 5 6 7 8 9 10 12 14 15 16 17 18 19 20 21

43) N* 11 Nnv* 14 [ Nnv* 14 ]* 15
Nnv = 4 5 6 7 8 9 10 12 14 15 16 17 18 19 20 21

44) N* 11 Nnv* 14 [ Nnv* 14 ]* 15
Nnv = 4 5 6 7 8 9 10 12 13 14 15 16 17 18 19 20 21

45) N* 11 Nnv* 12 [ Nnv* 12 ]* 13
Nnv = 4 5 6 7 8 9 10 12 14 15 16 17 18 19 20 21
```

46) N* 11 Nnv* 10 [Nnv* 10]* 11
Nnv = 4 5 6 7 8 9 10 12 14 15 16 17 18 19 20 21

47) N* 11 Nnv* 8 [Nnv* 8]* 9
Nnv = 4 5 6 7 8 9 10 12 14 15 16 17 18 19 20 21

48) N* 11 Nnv* 7 [Nnv* 7]* 6
Nnv = 4 5 6 7 8 9 10 12 14 15 16 17 18 19 20 21

49) N* 11 Nnv* 6 [Nnv* 6]* 7
Nnv = 4 5 6 7 8 9 10 12 14 15 16 17 18 19 20 21

Descritores para o Grafo(13)

Ni = 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
Nt = 6 11 13 14 21

50) N* 13 Nnv* 17 [Nnv* 17]* 19
Nnv = 4 5 6 7 8 9 10 12 14 15 16 17 18 19 20 21

51) N* 13 Nnv* 17 [Nnv* 17]* 18
Nnv = 4 5 6 7 8 9 10 12 14 15 16 17 18 19 20 21

52) N* 13 Nnv* 12 [Nnv* 12]* 14
Nnv = 4 5 6 7 8 9 10 12 14 15 16 17 18 19 20 21

53) N* 13 Nnv* 12 [Nnv* 12]* 13
Nnv = 4 5 6 7 8 9 10 12 14 15 16 17 18 19 20 21

54) N* 13 Nnv* 10 [Nnv* 10]* 11
Nnv = 4 5 6 7 8 9 10 12 14 15 16 17 18 19 20 21

55) N* 13 Nnv* 8 [Nnv* 8]* 9
Nnv = 4 5 6 7 8 9 10 12 14 15 16 17 18 19 20 21

56) N* 13 Nnv* 7 [Nnv* 7]* 6
Nnv = 4 5 6 7 8 9 10 12 14 15 16 17 18 19 20 21

57) N* 13 Nnv* 6 [Nnv* 6]* 7
Nnv = 4 5 6 7 8 9 10 12 14 15 16 17 18 19 20 21

Numero Total de Descritores = 45

2ª Etapa: Execução do teste de Regressão (Sessão POKE-TOOL) reduzida

```
afrodite% gcc testeprog.c -otesteprog.exe
afrodite%
afrodite% repoketool-exec testeprog.exe moda -t 2 to 3 5
repoketool-exec - Executa Casos de Teste em batch
Executando Novamente o caso de teste '2'
Executando Novamente o caso de teste '3'
Executando Novamente o caso de teste '5'
afrodite%
afrodite% pokeaval2 -dmoda -pu 2 to 3 5
pokeaval - Avaliador de Casos de Teste da POKE-TOOL


## Avalicao: Cobertura do Conjunto de Casos de Teste ##


          **** Avaliando Caso de Teste Numero 2 ****

* * Realizando a avaliacao do caso de teste * *

* * Avaliacao do caso de teste foi bem sucedida * *

* * Realizando a avaliacao do caso de teste separadamente * *

* * Avaliacao do caso de teste foi bem sucedida * *

          **** Avaliando Caso de Teste Numero 3 ****

* * Realizando a avaliacao do caso de teste * *

* * Avaliacao do caso de teste foi bem sucedida * *

* * Realizando a avaliacao do caso de teste separadamente * *

* * Avaliacao do caso de teste foi bem sucedida * *

          **** Avaliando Caso de Teste Numero 5 ****

* * Realizando a avaliacao do caso de teste * *

* * Avaliacao do caso de teste foi bem sucedida * *

* * Realizando a avaliacao do caso de teste separadamente * *

* * Avaliacao do caso de teste foi bem sucedida * *


ASSOCIACOES DO CRITERIO TODOS POT-USOS nao executadas:

<1,(17,18),{ v, n }>
<1,(12,13),{ v, n }>
<3,(17,19),{ i, freq_max, flag_moda }>
<3,(17,18),{ i, freq_max, flag_moda }>
<3,(12,14),{ i, freq_max, flag_moda }>
<3,(12,13),{ i, freq_max, flag_moda }>
```

```
<3,(16,4),{ i, freq_max, flag_moda }>
<3,(16,4),{ i, freq_max }>
<3,(16,4),{ i }>
<3,(10,11),{ i, freq_max, flag_moda }>
<3,(8,9),{ i, freq_max, flag_moda }>
<5,(12,14),{ t, valor }>
<5,(12,13),{ t, valor }>
<5,(17,19),{ t, valor }>
<5,(17,18),{ t, valor }>
<5,(4,5),{ t, valor }>
<5,(10,11),{ t, valor }>
<5,(8,9),{ t, valor }>
<7,(12,13),{ i, t }>
<7,(17,18),{ i, t }>
<11,(17,18),{ valor_moda, freq_max, flag_moda }>
<11,(12,13),{ valor_moda, freq_max, flag_moda }>
<13,(17,19),{ flag_moda }>
<13,(17,18),{ flag_moda }>
<13,(12,14),{ flag_moda }>
<13,(12,13),{ flag_moda }>
<13,(10,11),{ flag_moda }>
<13,(8,9),{ flag_moda }>
<13,(7,6),{ flag_moda }>
<13,(6,7),{ flag_moda }>

Cobertura Total = 33.333333

Media da Cobertura dos Grafo(i) = 34.444443

afrodite% pokeexec testeprog.exe -t -fmoda -r6
pokeexec mensagem: Comecando sessao de teste para o programa
"testeprog.exe" ...

pokeexec: "testeprog.exe" aceita parametros em linha de comando ?
(s/n) [s]
n
pokeexec:  Executando caso de teste numero 6.
5
1
1
2
3
3

Distribuicao Plurimodal
retorno: 0
pokeexec: Voce quer continuar a executar casos de teste? (s/n)[s]
n
n
pokeexec mensagem: Fim da Execucao de Casos de Teste para o
"testeprog.exe"

afrodite% pokeaval2 -dmoda -pu 2 to 3 5 to 6
pokeaval - Avaliador de Casos de Teste da POKE-TOOL
```

```
## Avalicao: Cobertura do Conjunto de Casos de Teste ##

          **** Avaliando Caso de Teste Numero 2 ****
* * Realizando a avaliacao do caso de teste * *
* * Avaliacao do caso de teste foi bem sucedida * *
* * Realizando a avaliacao do caso de teste separadamente * *
* * Avaliacao do caso de teste foi bem sucedida * *
          **** Avaliando Caso de Teste Numero 3 ****
* * Realizando a avaliacao do caso de teste * *
* * Avaliacao do caso de teste foi bem sucedida * *
* * Realizando a avaliacao do caso de teste separadamente * *
* * Avaliacao do caso de teste foi bem sucedida * *
          **** Avaliando Caso de Teste Numero 5 ****
* * Realizando a avaliacao do caso de teste * *
* * Avaliacao do caso de teste foi bem sucedida * *
* * Realizando a avaliacao do caso de teste separadamente * *
* * Avaliacao do caso de teste foi bem sucedida * *
          **** Avaliando Caso de Teste Numero 6 ****
* * Realizando a avaliacao do caso de teste * *
* * Avaliacao do caso de teste foi bem sucedida * *
* * Realizando a avaliacao do caso de teste separadamente * *
* * Avaliacao do caso de teste foi bem sucedida * *

ASSOCIACOES DO CRITERIO TODOS POT-USOS nao executadas:

<3,(17,19),{ i, freq_max, flag_moda }>
<3,(17,18),{ i, freq_max, flag_moda }>
<3,(12,14),{ i, freq_max, flag_moda }>
<3,(12,13),{ i, freq_max, flag_moda }>
<3,(16,4),{ i, freq_max, flag_moda }>
<3,(16,4),{ i, freq_max }>
<3,(16,4),{ i }>
<3,(10,11),{ i, freq_max, flag_moda }>
<3,(8,9),{ i, freq_max, flag_moda }>
<5,(12,14),{ t, valor }>
```

```
<5,(12,13),{ t, valor }>
<5,(17,19),{ t, valor }>
<5,(17,18),{ t, valor }>
<5,(4,5),{ t, valor }>
<5,(10,11),{ t, valor }>
<5,(8,9),{ t, valor }>
<11,(17,18),{ valor_moda, freq_max, flag_moda }>
<13,(17,19),{ flag_moda }>
<13,(12,14),{ flag_moda }>
<13,(12,13),{ flag_moda }>
<13,(10,11),{ flag_moda }>
<13,(8,9),{ flag_moda }>
<13,(7,6),{ flag_moda }>
<13,(6,7),{ flag_moda }>

Cobertura Total = 46.666667

Media da Cobertura dos Grafo(i) = 50.416668

afrodite% repoketool-end moda.c moda testeprog.exe

afrodite%
```